AN EXTENSIBLE COMPUTING ARCHITECTURE DESIGN FOR CONNECTED AUTONOMOUS VEHICLE SYSTEM

Jacob Daniel Hochstetler

Dissertation Prepared for the Degree of

DOCTOR OF PHILOSOPHY

UNIVERSITY OF NORTH TEXAS

May 2021

Hochstetler, Jacob Daniel. *An Extensible Computing Architecture Design for Connected Autonomous Vehicle System*. Doctor of Philosophy (Computer Science and Engineering), May 2021, 278 pp., 26 tables, 80 figures, 255 numbered references.

Autonomous vehicles have made milestone strides within the past decade. Advances up the autonomy ladder have come lock-step with the advances in machine learning, namely deep-learning algorithms and huge, open training sets. And while advances in CPUs have slowed, GPUs have edged into the previous decade's TOP 500 supercomputer territory. This new class of GPUs include novel deep-learning hardware that has essentially side-stepped Moore's law, outpacing the doubling observation by a factor of ten. While GPUs have make record progress, networks do not follow Moore's law and are restricted by several bottlenecks, from protocol-based latency lower bounds to the very laws of physics. In a way, the bottlenecks that plague modern networks gave rise to Edge computing, a key component of the Connected Autonomous Vehicle system, as the need for low-latency in some domains eclipsed the need for massive processing farms. The Connected Autonomous Vehicle ecosystem is one of the most complicated environments in all of computing. Not only is the hardware scaled all the way from 16 and 32-bit microcontrollers, to multi-CPU Edge nodes, and multi-GPU Cloud servers, but the networking also encompasses the gamut of modern communication transports. I propose a framework for negotiating, encapsulating and transferring data between vehicles ensuring efficient bandwidth utilization and respecting real-time privacy levels.

ACKNOWLEDGMENTS

Thank you to Lauren & Alfie.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Autonomous vehicles have made milestone strides within the past decade. Google's own Self-Driving Car Project was founded just 10 years ago on January 17, 2009. Since then, several production vehicles has made it to SAE Level 2 for autonomy, including notable names like Tesla, Mercedes-Benz and Cadillac. These advances up the autonomy ladder have come lock-step with the advances in machine learning, namely deep-learning algorithms and huge, open training sets. And while advances in CPUs have slowed, GPUs have edged into the previous decade's TOP 500 supercomputer territory, using just the power of an incandescent light-bulb. This new class of GPUs include novel deep-learning hardware that has essentially side-stepped Moore's law, outpacing the doubling observation by a factor of ten.

While GPU and software advances have enabled the A in Connected and Autonomous Vehicles (CAV), the network hardware and protocols on the vehicle are responsible for the C. As opposed to CPUs, networks do not follow Moore's law and are restricted by several bottlenecks, from protocol-based latency lower bounds to the very laws of physics (Shannon limit). In a way, the bottlenecks that plague modern networks gave rise to Edge computing, as the need for low-latency in some domains eclipsed the need for massive processing farms. The need for efficient network utilization is exacerbated by the fact autonomous vehicles (AVs) are producing terabytes of data per day. On a modern Tesla vehicle, one of the eight cameras (HW2.5) alone can produce over 1.2 terabytes of compressed video per day (H.264 720p @ 60fps). While video data is the bulk of the on-board sensor production, this does not include the radar, GPS, and ancillary sensor data. So equipped, the uncompressed data production on a single AV (9.92 Gbps) would encompass almost half of the available bandwidth for the future 5G communications standard (20 Gbps). Unfortunately, that is for a single vehicle-to-vehicle session, so a duplex exchange would consume the entire bandwidth, far from the goal of multi-vehicle/vehicle-to-edge communication.

At the same time software and hardware have accelerated, everyday end-consumers have become increasingly aware of data privacy issues. Some of these issues include super-cookies, social media selling, big Data mining, and EU's recent General Data Protection Regulation 2016/679 (GDPR). Even the FBI recently issued a warning against Smart TVs spying on customers [204].

I propose a framework for negotiating, encapsulating and transferring data between vehicles ensuring efficient bandwidth utilization and respecting real-time privacy levels.

1.1. Scope of Research

In order to enhance decision making at several levels, autonomous vehicles must become *connected*. While a generic term, it contains an expanse of problems, both technical and civil.

For the in-vehicle only computing model, the applications running on the platform must contend with each other for limited resources on the platform. For instance, assume two latency-sensitive applications require execution on the GPU at the same time. If there is only one GPU on the vehicle, the second scheduled application might not produce a *timely* decision. Similarly, even though processing of sensor data produced on the vehicle might be within latency constraints, full perception may still not be possible due to sensor limitations. But processing data that originated outside of the vehicle brings in several problems, falling into two categories of issues:

- *Internal* (technical issues related to hardware or software)
- *External* (privacy/regulatory or trust issues)

To create a framework that is possible, I will limit the scope of my proposal to vehicles already equipped with network connectivity hardware and that have established an ad hoc mesh network between two or more vehicles.

1.2. Major Contributions

The major contribution of this dissertation is my proposal for a standard architecture for CAV services. This framework includes a data format, protocol, and application orchestration layer, and is designed to be both extensible for future development and backwards compatible for legacy clients. Additionally is it technically superior, solving both resource -constrained communication problems and actual software development life cycle concerns. This work is separated into chapters, with the problems and challenges of a CAV architecture in Chapter 2 and Chapter 3 details the approach to these problems and challenges. Chapters 4, 5 and 6 detail the proposed framework, and provide comparisons to competing standards and systems. Chapter 7 details my prior work in the CAV ecosystem, and an overview of that work is given in Figure 1.1 below. Cloud-related work includes proactive storage-system data protection, and SSD drive reliability. Cloud services form the backbone of machine learning for autonomous vehicles and storage systems are an integral part of training the ML models. Additionally, with the reduction in SSD cost, flash storage is being deployed as both the caching layer fronting larger storage systems, and as the primary storage for Edge systems. My Edge-related work includes optimal police patrol planning based on information entropy, and a mutation testing tool for application pipelines. The application of optimal police patrols is important step for "smart cities", and can be feed data received from both far Edge nodes or near Edge police vehicles. The mutation testing tool is important for Edge systems as Golang is gaining ground in essential backend services for infrastructure. Lastly, my CAV-related research includes neural computing stick benchmarking for embedded systems, and a low-latency data sharing scheme for connected and autonomous vehicles. Together this research has provided insights into how data is both collected from vehicles, and transformed into useful products, either through Edge-nodes or Cloud services.

1.3. Background and Related Research

In this section I will explore four background concepts needed for the function of *C*onnected *A*utonomous *V*ehicles (CAVs), namely:

- Decentralized computing
- In-Vehicle services and systems
- Vehicle d producers/sensors
- Modern privacy concerns

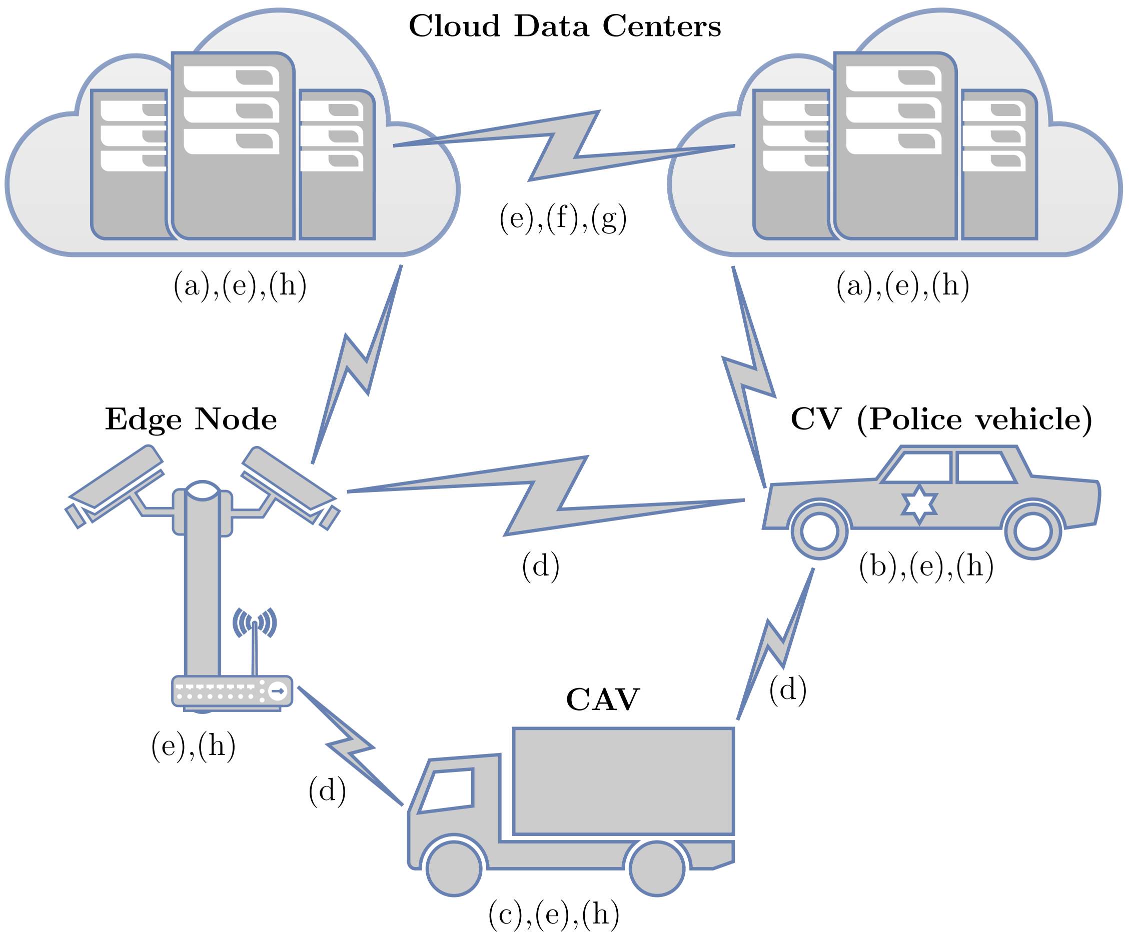


FIGURE 1.1. Overview of CAV ecosystem with prior work annotated.

(a) Sec. 7.5 *Reliability Characterization of Solid State Drives in a Scalable Production Data Center*
2018 IEEE Int. Conf. on Big Data (Big Data)

(b) Sec. 7.1 *Optimal Police Patrol Planning Strategy for Smart City Safety*
2016 IEEE 14th Int. Conf. on Smart City

(c) Sec. 7.3 *Embedded Deep Learning for Vehicular Edge Computing*
2018 IEEE/ACM Symposium on Edge Computing (SEC)

(d) Sec. 7.4 *Low-Latency High-Level Data Sharing for Connected and Autonomous Vehicular Networks*
2019 IEEE Int. Conf. on Industrial Internet (ICII)

(e) Sec. 7.2 *TuranGo: Mutation Testing a Language*

(f) Sec. 7.6 *Incorporate Proactive Data Protection in ZFS Towards Reliable Storage Systems*
2018 IEEE 16th Int. Conf. on Dependable, Autonomic and Secure Computing

(g) Sec. 7.7 *Developing Cost-Effective Data Rescue Schemes to Tackle Disk Failures in Data Centers*
2018 BigData, 7th Intl. Congress

(h) Sec. 7.8 *An Empirical Study of Quad-Level Cell (QLC) NAND Flash SSDs for Big Data Applications*
2019 IEEE Int. Conf. on Big Data (Big Data)

1.3.1. Decentralized Computing

Since then, connected computing has been categorized by different attributes based around their computing power and data locality (resource/power/latency), and these attributes can be mapped into a continuum. A simple hierarchy of this continuum is shown in Figure 1.2. Telecoms describe "the Edge" a bit differently from traditional IT operations. From their perspective, there are actually three different edges. First, there's the "cloud edge," where the *c*ontent *d*elivery *n*etworks (CDNs) work. The only job of the CDN is to place servers physically close to end users to distribute traffic and reduce latency. A CDN's edge server is normally just a caching web server and is just a single piece in an application's architecture.

Additionally, telecoms also label the "near edge" and the "far edge" (together called the "broad edge"). The near edge includes servers located at cell towers or local telecom's points of presence. Such locations are typically well-suited to host small data centers with all the essential trimmings: power, cooling, racks of equipment, and sometimes a bare-bones staff. The near edge may also represent a facility that hosts Internet of Things (IoT) gateways, a sensor center at a factory, in a city's traffic switching office (sometimes called a traffic management office), or in the security command center of a large building/campus such as a stadium, airport, or office park. Finally the far edge is the devices themselves, either handheld smartphones, IoT sensors/actuators or CAVs.

While the architecture and capacity that powers data centers differs from edge devices, all immediate storage has moved from magnetic spinning hard disks to solid state flash drives. This has moved all magnetic storage to serving roles as secondary (remote) storage for larger, long-term storage. For many platforms this is a concerted effort in conjunction with Hierarchical Storage Management (HSM), a data storage technique that automatically moves data between high-cost/high-performance and low-cost/low-performance storage media. For a typical data center that means moving data from a VM's flash SSD (shared multi-tenant with other VMs on the hypervisor) to a network-attached storage device consisting of spinning hard drives (fronted by SSD cache), and then finally to silos of magnetic

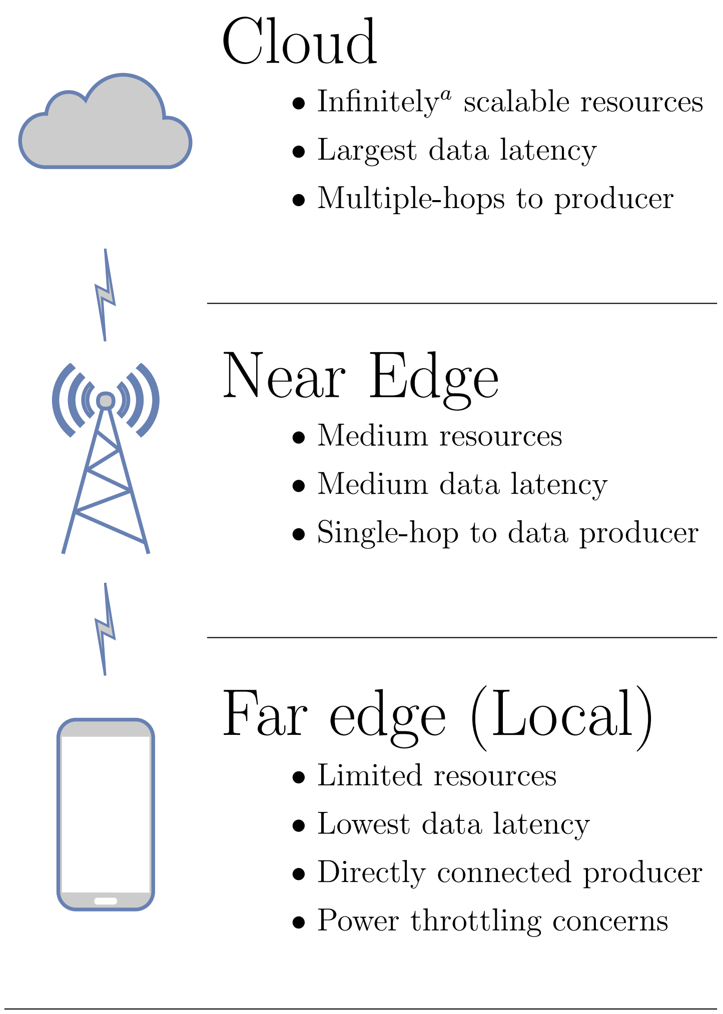

Cloud
• Infinitely[a] scalable resources
• Largest data latency
• Multiple-hops to producer
Near Edge
• Medium resources
• Medium data latency
• Single-hop to data producer
Far edge (Local)
• Limited resources
• Lowest data latency
• Directly connected producer
• Power throttling concerns

FIGURE 1.2. Connected computing hierarchy.
[a]Obviously not really infinite, but *workload-infinite*.

tape (fronted by more flash cache). Within a data center, this two-way data transition can take place frequently due to the data locality and a focus on data gravity. For an Edge node, the path from local SSD to a tape silo could take weeks, since the cost of moving data that far away from a given producer/consumer does not justify the benefits. More specifically, the cost could be actual money spent for network bandwidth, or the computational cost of tying up local edge resources.

1.3.1.1. Cloud Computing

Cloud computing is the name of services that offload traditional, on-premise computing to a remote set of internet-connected servers. These services normally involve at a

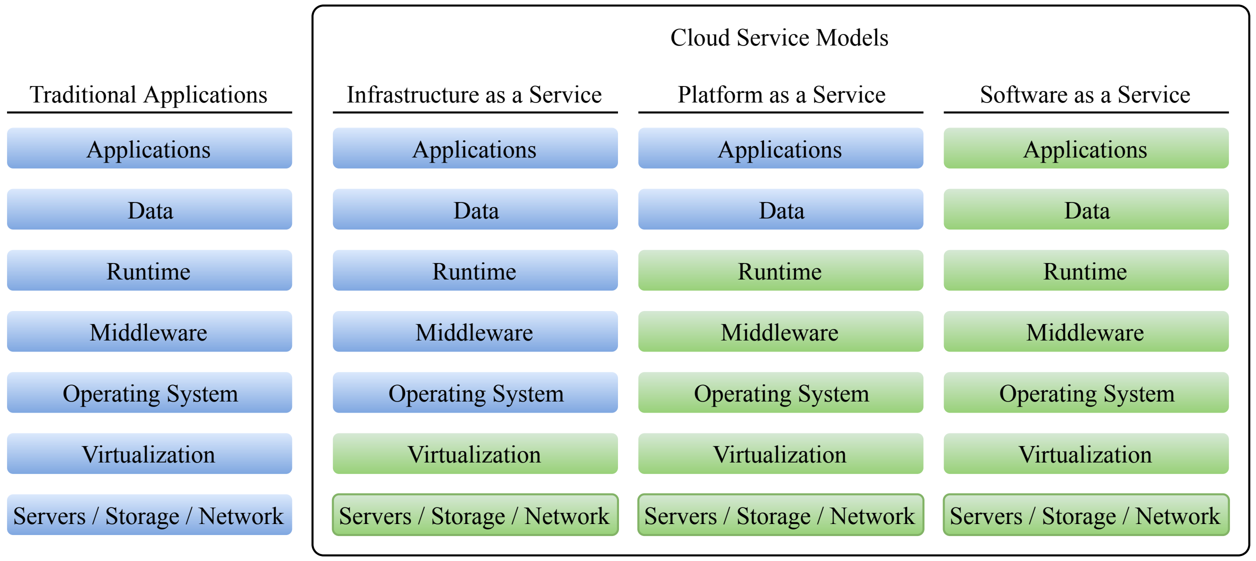


FIGURE 1.3. Cloud service models. Blue blocks are end user managed, green blocks are managed by your service provider.

basic level virtualization technologies in several areas, namely: compute, storage and networking. Cloud computing can also be utilized in an on-premise capacity, as long as the same characteristics are met when the service is delivered from the data center.

The National Institute of Standards and Technology [NIST] (U.S. Department of Commerce) defines five essential characteristics of cloud computing [146]:

(1) on-demand self-service
(2) broad network access
(3) resource pooling
(4) rapid elasticity/expansion
(5) measured service

In the same publication, NIST also lists three "service models" and what the end user manages:

- IaaS: Infrastructure as a Service (OS, application & data)
- PaaS: Platform as a Service (application & data)
- SaaS: Software as a Service (only data)

These service models and levels of end user-management are visualized in Figure 1.3 below.

1.3.1.2. Ubiquitous Computing

Ubiquitous computing is the 30-year trend of embedding computational capability into everyday objects to make them effectively communicate and perform useful tasks. Ubiquitous computing is also referred to as pervasive computing, as the differences between the two are mostly academic [145]. The main contrast with traditional computing is that ubiquitous computing minimizes the end user's need to interact with computers as computers. Ubiquitous/pervasive computing devices are network-connected and constantly available. Recent examples of ubiquitous computing includes connected-home devices like Amazon's Alexa or Google's Assistant, where the primary human-computer interaction is accomplished through voice commands.

Satyanarayanan's seminal 2001 paper "Pervasive Computing: Vision and Challenges" describes the flow from the agenda of distributed system applications to mobile computing and then to pervasive computing. Each set is composed of it's own new issues, and the problems/issues of the superset are multiplied. This is shown with original caption in Figure 1.4.

1.3.1.3. Connected Vehicles

The group tasked to create the first Wireless Access in Vehicular Environments (WAVE) standard was formed in November 2004. This was the beginning of the IEEE 802.11p amendment to the 802.11 wireless local area network (WLAN) standard. This was the start of vehicle-to-everything (V2X) communications. There are several categories of connected vehicle communications as shown in Figure 1.5:

- V2I: Vehicle-to-infrastructure
- V2V: Vehicle-to-vehicle
- V2N: Vehicle-to-network
- V2D: Vehicle-to-device
- V2G: Vehicle-to-grid
- V2P: Vehicle-to-pedestrian

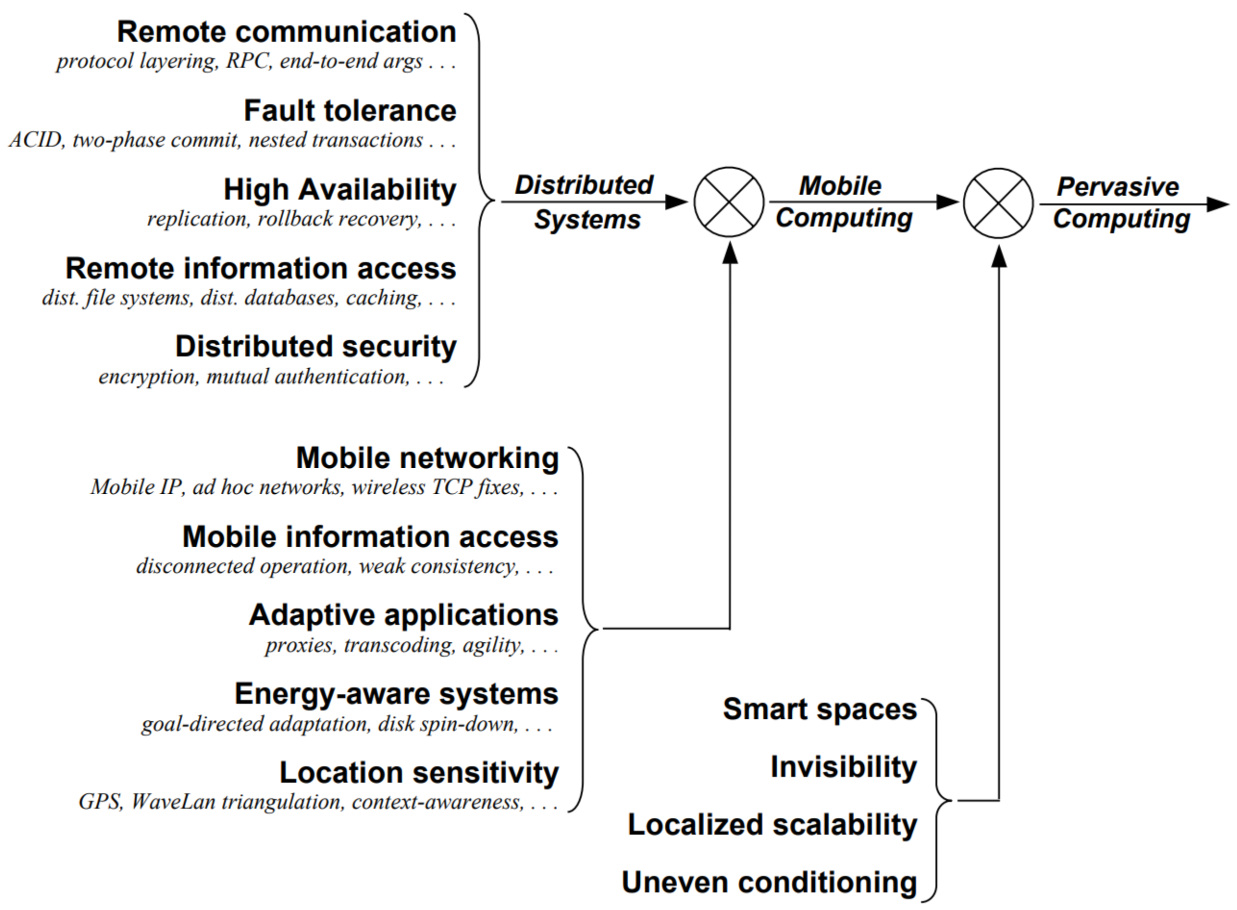

Remote communication
protocol layering, RPC, end-to-end args . . .
Fault tolerance
ACID, two-phase commit, nested transactions . . .
High Availability
replication, rollback recovery, . . .
Remote information access
dist. file systems, dist. databases, caching, . . .
Distributed security
encryption, mutual authentication, . . .
Distributed Systems
Mobile Computing
Pervasive Computing
Mobile networking
Mobile IP, ad hoc networks, wireless TCP fixes, . . .
Mobile information access
disconnected operation, weak consistency, . . .
Adaptive applications
proxies, transcoding, agility, . . .
Energy-aware systems
goal-directed adaptation, disk spin-down, . . .
Location sensitivity
GPS, WaveLan triangulation, context-awareness, . . .
Smart spaces
Invisibility
Localized scalability
Uneven conditioning

FIGURE 1.4. This figure shows how research problems in pervasive computing relate to those in mobile computing and distributed systems. New problems are encountered as one moves from left to right in this figure. In addition, the solution of many previously-encountered problems becomes more complex. As the modulation symbols suggest, this increase in complexity is multiplicative rather than additive — it is very much more difficult to design and implement a pervasive computing system than a simple distributed system of comparable robustness and maturity. Note that this figure describes logical relationships, not temporal ones. Although the evolution of research effort over time has loosely followed this picture, there have been cases where research effort on some aspect of pervasive computing began relatively early. For example, work on smart spaces began in the early 1990's and proceeded relatively independently of work in mobile computing. [193]

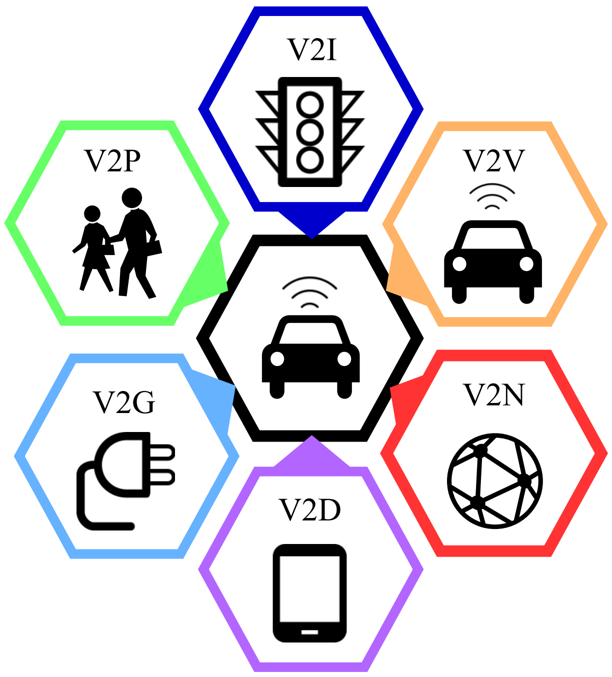


Figure 1.5. Types of Connected Vehicle communication types.

There are two main types of vehicular communications: 1) WLAN-based, and 2) cellular-based. The Society of Automotive Engineers (SAE) refers to the technology using 802.11p as Dedicated Short-Range Communications (DSRC), which is a held-over term from the 1999 FCC program opening up 75 MHz of spectrum in the 5.9 GHz band for intelligent transportation systems (ITS). The telephony standards group 3GPP refers to their technology as C-V2X (cellular V2X) to differentiate itself from the 802.11p V2X specifications. Bluetooth and satellite were also considered for V2X, but their high-latency has all but eliminated them.

DSRC is designed to be effective up to 1000 m, while actual the real-world range is less than 300 m [239]. While DSRC's strength lies in its low-latency, the 6 to 27 Mbps available bandwidth is a huge negative for anything more than command/control messaging. For this reason cellular-based V2X has seen increased interest in the last few years as the constant evolution in cell-based communications has pushed beyond DSRC capabilities.

Following the established by standards bodies, several original equipment manufacturers (OEM) have integrated V2X technologies into their vehicles. The adoption has been slow though, as there isn't much direct benefit for an OEM to spend the money, time and resources. The lament over the lack of adoption has been expressed since 2013 [248]. As of 2019, both Toyota and General Motors have sold vehicles equipped with DSRC, while several government-sponsored programs have utilized DSRC during demonstrations.

Once a majority of vehicles are equipped with V2X equipment, the applications and level of processing will increase exponentially. Many of the applications that V2X enables are safety related and all vehicles on the road must participate to achieve their full potential. Some these applications include [239]:

- Emergency warning system for vehicles
- Cooperative forward collision warning
- Cooperative adaptive cruise control
- Approaching emergency vehicle warning (Blue Waves)
- Vehicle safety inspection
- Transit or emergency vehicle signal priority
- Electronic parking payments
- Commercial vehicle clearance and safety inspections
- In-vehicle signing
- Rollover warning
- Probe data collection
- Highway-rail intersection warning
- Electronic toll collection
- Convoys

1.3.2. In-Vehicle Services and Systems

In this section, I briefly discuss four types of services that will be available on CAVs. Conventionally, these services on current vehicles could be classified into three groups according to their functionality: real-time diagnostics, advanced driver-assistant systems, and in-vehicle infotainment. In addition, there is an emerging type of services, 3rd party applications from various vendors, which will be prevalent on CAVs as the vehicle data in the future will not be exclusive to the automakers.

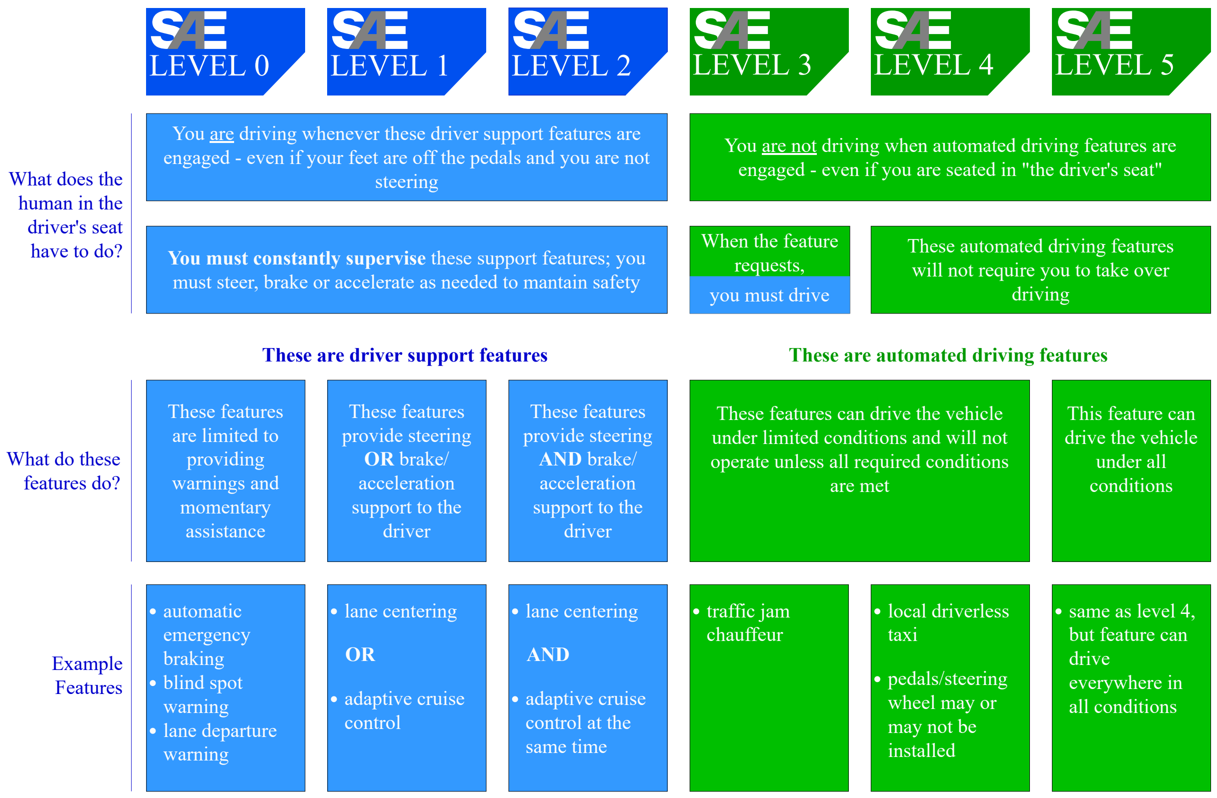
1.3.2.1. Real-time Diagnostics

This type of service usually refers to the On-board diagnostics (OBD) system, which allows the vehicle to have the capability of self-diagnosis and reporting. The OBD system appeared on the vehicle in the 1980s and has evolved from the early simple ”idiot light” to a modern version that can provide real-time vehicle data (e.g., the engine’s revolution from the engine control unit) and a standardized series of diagnostic trouble codes. Such codes are useful for vehicle troubleshooting and repair. The device reading the real-time data is actually an additional device to the vehicle called an OBD reader. The maintainer can leverage the OBD reader to obtain information about the fault, e.g., the diagnostic trouble code. This usually will not consume any resources of the vehicle, since it is just reading data already produced by the engine computer, however, it is not an in-vehicle system. In future CAVs, this type of service should be built in the vehicle, which collects the related vehicle data, including real-time data and historical data, and would be analyzed to predict faults. This proactive system could remind the owner of maintenance issues to ensure the vehicle is operating in good (and legal) condition.

As an important side-note, all Tesla models and some other modern vehicles utilize Over-the-Air programming (OTA). This effectively side-steps logistically cumbersome dealership-based firmware and operating system updates by delivering new features and value to existing customers in an agile manner. This is akin to Android/Apple upgrading operating system for their mobile phones. Normally the same hardware and infrastructure that supports OTA also enables bi-directional communication from the vehicle, which has a variety of uses from fleet-based deep learning (and inference optimization), route-learning and even automatic emergency service response (in the event of a detected collision).

1.3.2.2. Advanced Driver-Assistant Systems (ADAS)

Nowadays, more and more vehicles are equipped with the ADAS that can detect some objects, complete basic classification, and alert the driver of unsafe driving behaviors. It may



	SAE LEVEL 0	SAE LEVEL 1	SAE LEVEL 2	SAE LEVEL 3	SAE LEVEL 4	SAE LEVEL 5
What does the human in the driver's seat have to do?	You <u>are</u> driving whenever these driver support features are engaged - even if your feet are off the pedals and you are not steering			You <u>are not</u> driving when automated driving features are engaged - even if you are seated in "the driver's seat"		
	You must constantly supervise these support features; you must steer, brake or accelerate as needed to mantain safety			When the feature requests, you must drive	These automated driving features will not require you to take over driving	
	These are driver support features			**These are automated driving features**		
What do these features do?	These features are limited to providing warnings and momentary assistance	These features provide steering **OR** brake/ acceleration support to the driver	These features provide steering **AND** brake/ acceleration support to the driver	These features can drive the vehicle under limited conditions and will not operate unless all required conditions are met		This feature can drive the vehicle under all conditions
Example Features	• automatic emergency braking • blind spot warning • lane departure warning	• lane centering **OR** • adaptive cruise control	• lane centering **AND** • adaptive cruise control at the same time	• traffic jam chauffeur	• local driverless taxi • pedals/steering wheel may or may not be installed	• same as level 4, but feature can drive everywhere in all conditions

FIGURE 1.6. SAE J3016 levels of driving automation, adapted from [192].

also slow or stop the vehicle. For example, the steering wheel will vibrate when the vehicle is close to a traffic line without illuminating the turn light. The Society of Automotive Engineers (SAE) International grades autonomous driving at six levels [192], in which the level 0 means the vehicle is without any assistant, and the level 5 means the vehicle is in full control by an autonomous driving system. A vehicle with ADAS is usually rated the SAE level 1 or level 2 based on provided functions. Usually, level 3 means the human driver can safely turn their attention away from driving tasks. As the level ascends, the autonomous driving system takes over more controls from human drivers and needs more computation resources to run computational intensive algorithms. The SAE levels are shown below in Figure 1.6.

The most important element of ADAS is the real-time object detection that is based on either computer vision or deep learning technology. Two of the most representative detection algorithms for ADAS are Lane Detection and Vehicle Detection. The former

Table 1.1. ADAS Response Times. [251]

Name	**Latency**
Lane Detection	13.57 ms
Vehicle Detection (Haar)	269.46 ms
Vehicle Detection (TensorFlow)	13 971.98 ms

relies on the computer vision technology. Regarding the latter, the underlying technologies are Haar-based image processing and TensorFlow-based deep learning algorithm. As of December 2019 from [251], as shown in Table 1.1, the vehicle detection shows that the latency of Haar-based algorithm significantly outperforms (around 51x faster) the TensorFlow-based. But in the future CAVs, deep learning based algorithms will dominate since they can detect multiple types of objects at once, as shown in Microsoft Research Asia's *Region Proposal Networks* [185].

1.3.2.3. In-Vehicle Infotainment

In-Vehicle infotainment includes a wide range of services that provide audio or video entertainment. For example, the driver uses the radio to listen to music, including cloud services from the Internet such as Pandora, and the passengers sitting in the backseat can relax by watching online videos or news using the device embedded on the seat. This means these services might involve large-scale Internet data transmission. Most mainstream auto vendors have Internet supported infotainment services, e.g., Uconnect for Chrysler, Blue Link for Honda, and iDrive for BMW. Another trend of on-board infotainment is the Android-based system that has been implemented by many auto vendors including Honda, Hyundai, Audi, and Volvo. Tesla even includes a 17 inch tablet running Ubuntu for it's GPS mapping, Internet streaming radio and built-in web browser. For these services, video or audio data must be downloaded from the Internet and then decoded locally on the vehicle, and eventually delivered to the passengers. During negotiation, the infotainment system could scale up or down the quality, or even pre-fetch/cache data for a better user-experience. It means these applications not only require compute resources but also require an elastic high requirement for the network bandwidth.

1.3.2.4. Third-Party Applications

An in-vehicle, third-party application provided by a vendor other than the car maker is used to enhance the user experiences or provide other add-on services (e.g., finding a missing car for law enforcement). The trend of openness of vehicle data will make it easier for third-party vendors to develop and deploy different kinds of applications on the vehicle. In addition, with the rapid development of in-vehicle processor processing, vehicle to vehicle communications and autonomous driving technologies, future CAVs could be viewed as a sophisticated computer on wheels with a variety of third-party applications. Several projects including these types of applications have been initiated. For example, Kar *et al.* [120] proposed an application to enhance the vehicle safety by detecting whether the driver is registered or not through analyzing their operation features (e.g., the time duration of door open and close). Another example is to leverage the on-board camera to recognize and track a targeted vehicle, which is a mobile version for their *AMBER alert assistant* [252], promising to enhance the AMBER alert system. Generally, the core of such types of applications is either vision-related or machine learning based data processing algorithm fed by the same on-board sensor data (e.g., dash camera). Since the core algorithms are computationally intensive, these types of third-party applications need access to powerful computing hardware as well.

1.3.3. Vehicle Data Producers/Sensors

In this section I will briefly describe some of the main data producers available on modern CAVs:

(1) Positional tracking systems (GPS/IMUs)
(2) LiDAR
(3) Radar
(4) Imagery (camera/video)
(5) Convolutional Feature Maps

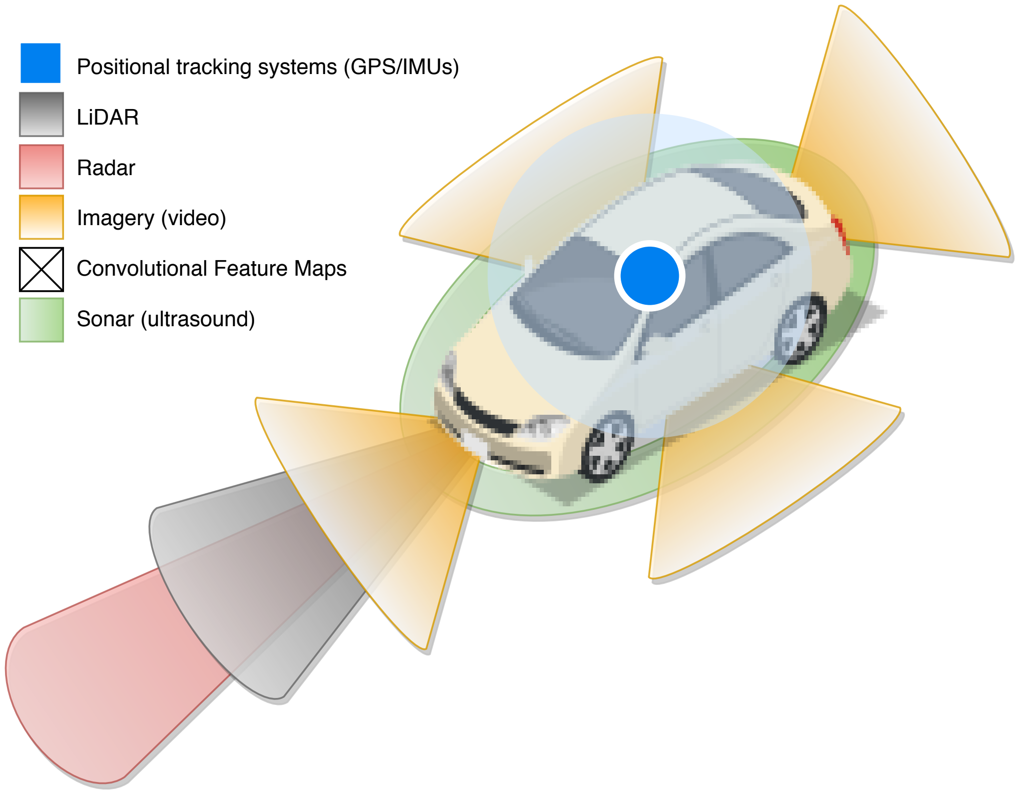

Positional tracking systems (GPS/IMUs)
LiDAR
Radar
Imagery (video)
Convolutional Feature Maps
Sonar (ultrasound)

FIGURE 1.7. Overlay showing vehicle sensors discussed and their approximate ranges/placement. Sensor data characteristics are listed in Table 1.2.

(6) Sonar (ultrasound)

The data formats and approximate bandwidth of each sensor class is shown in Table 1.2 below, and an overlay of these sensors on an example vehicle is shown in Figure 1.7.

Currently there are two separate factions within the AV community: 1) those that believe LiDAR is needed for full SAE Level 5, and 2) those that believe a combination of radar and imagery can provide enough sensor information instead. AV manufacturers/integrators can normally be categorized into one of these two groups with General Motors, Ford Motor Company, Uber and Alphabet Inc's Waymo (a Google subsidiary) relying upon LiDAR while Nissan and Tesla utilize multiple cameras with radar [207]. Waymo specifically has heavily invested in LiDAR technologies [94] and [93], while newer research has produced "LiDAR-like" results using only cameras [224]

1.3.3.1. Positional Tracking Systems

The Global Positioning System (GPS) is an American created, satellite-based radio-navigation system. Currently there are 31 satellites in medium-earth orbit (MEO), each containing a synchronized atomic clock. Each satellite continuously transmits a radio signal containing the current time and position data. Since the speed of radio waves is constant and independent of the satellite speed, the time delay between when the satellite transmits a signal and the receiver receives it is proportional to the distance from the satellite to the receiver. A GPS receiver monitors multiple satellites and solves equations to determine the precise position of the receiver and its deviation from true time [107]. At a minimum, four satellites must be in view of the receiver for it to compute four unknown quantities: three position coordinates on a spheroid and clock deviation from satellite time. With modern single-frequency GPS receivers, locations can be fixed to $\leq 1.6\,\mathrm{m}$ (horizontal) 95% of the time [67]. Dual-frequency receivers can achieve $\leq 0.3\,\mathrm{m}$ accuracy.

Inertial measurement units (IMUs) can be coupled, or integrated into GPS units to provide both extra precision, and coverage during GPS blackouts. They generate this information by detecting linear acceleration using one or more accelerometers and rotational rate using one or more gyroscopes. IMUs are important since GPS signals may be obscured in places like cities with high buildings or valleys between mountains. The downside of IMUs is accumulated error, but this drift is corrected through Kalman filters when GPS data becomes available again [34].

1.3.3.2. LiDAR

*Li*ght *D*etection *A*nd *R*anging is a type of sensor based on the emission and detected reflection of a pulsed laser. The duration of these laser pulses are precisely measured by a light detector, and combined with a synchronized GPS feed, produces a 3D point cloud originating at the sensor. Modern automotive LiDAR sensors like the Velodyne VLS-128, produce millions of laser pulses per second, and mechanically spin at up to 20 Hz [222].

Since these sensors operate around humans, to be sold commercially they must be certified eye-safe. This is a legal requirement in the U.S., from standards established by the

U.S. Food & Drug Administration in 1992. This standard sets limits for wavelengths, and provides classification levels for powers [218]. Automotive LiDAR systems operate at one of two wavelengths, either ~905 nm or 1550 nm (331.3 THz or 193.4 THz).

1.3.3.3. Radar

Similar to LiDAR, radar (*ra*dio *d*etection *a*nd *r*anging) is based on the transmission and subsequent detection of reflected radio waves. If an object is moving either towards or away from the transmitter/receiver, then there will also be a frequency shift due to the Doppler effect, and the velocity of the target object can be computed. Radar has a long technological history, spanning over 100 years (1904 *Telemobiloscope* patent), and is the standard technique used when the range/position or speed of an object is needed in an application. This includes industries ranging from aviation, to weather forecasting and even civil traffic enforcement.

Since the radio waves used for radar are much lower frequency than the narrow wavelength pulses used in LiDAR, radar can penetrate weather effects (rain/snow/fog/dust), and is effective at longer distances. Unlike LiDAR, radar can also provide relative target speed and is effective at ranging dirty vehicles, while LiDAR has problems getting a good reflection return. LiDAR is also much more expensive than radar and has many more moving parts. This is why most manufacturers base their adaptive cruise control on one or more radar(s) possibly in conjunction with cameras for reliability.

Most Autonomous Vehicle radar uses 77 GHz in the W band (75 to 110 GHz / 4.0 to 2.7 mm wavelength). This is classified as millimeter radar and gives higher resolution than radar packages in other lower-frequency bands. Additionally, as shown in Table 1.2, radar has bandwidth requirements orders of magnitude lower than LiDAR.

1.3.3.4. Imagery

Imagery is the capture of visual data (attenuation of light waves) through mechanical or electronic means. In my research I will use “image sensor” and “camera” interchangeably, unless specifically mentioned, to mean the same thing (a visual information recorder). In reality one or more image sensors are merely parts of a camera, which also includes an image signal processor, color processors/filters and other discrete signal processors which together produce an image that is human consumable.

Image sensors themselves fall into two categories: either a "charge-coupled device" (*CCD*) or a "complementary metal-oxide semiconductor" (*CMOS*) sensor. Either sensor is composed of a 2-D array of thousands/millions of miniature solar cells, each of which transforms the photons received from one small portion of the image into electrons. Next, the values are read from this array (accumulated charge) and packetized frame by frame and line by line for reading. CCD and CMOS differ on both the manufacturing process and how the values are collected, with CCD being more expensive and precise, while CMOS is cheaper and has lower power-consumption. CMOS sensors are sometimes referred to as "active-pixel sensors" since each single unit cell contains a photodetector and one or more transistors.

There are a number of ways to detect color with an image sensor, either using a filter in front of the detectors or increasing the number of CCDs in the package. Most normal cameras use RGGB (red-green-green-blue) Bayer-filter, which doubles-up on the green filter to enhance luminance as it's in the middle of the visible color spectrum. Many AV integrators utilize special filters to tailor the specific camera to it's application. For instance, an RCCC filter (red-clear-clear-clear) removes three color filters, except for the red filter used in the detection of red traffic lights and taillights. In this application detection of blue or green is not important, just the increased monochrome sensitivity for use in line detection (lane markings) and stereoscopic vision (object distance calculations).

At a hardware level, there are four main sensor package attributes that affect the final image product.

- Pixel count (width by height, e.g. 1920x1080)
- Bit depth (number of bits per pixel, e.g. 12-bit)
- Frame rate (usually in frames per second as *fps*)
- Filter type (e.g. Bayer RGGB filter)

Additionally, most imagery sensors insert metadata at a hardware level, either in

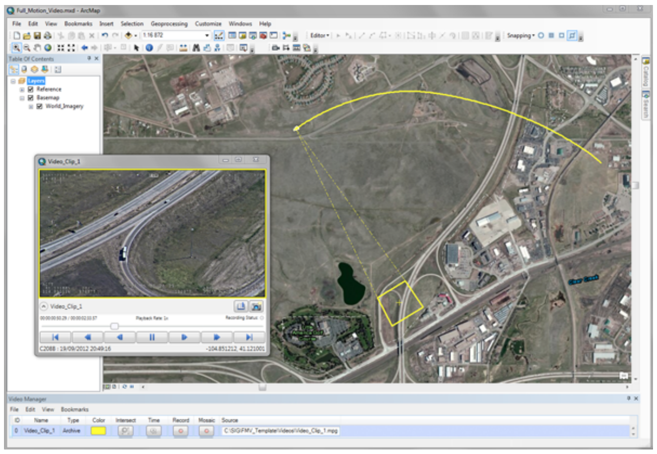


Figure 1.8. ArcGIS FMV [25] overlaying a UAV track (yellow spline) with the viewport coordinates (yellow rectangle) and the video (child window on left).

overlay or as extra rows above/below the visual array. For example The popular ON Semiconductor (formerly Aptina) AR0132 has a 1280x960 resolution CMOS, but adds four rows of embedded stats and data to make each frame 1280x964.

The imagery category encompasses all the visual recording sensors involved with AVs. This includes everything from CMOS sensors used for end user backup cameras, to color-shifted long-range CCD sensors, to even ultraviolet or infrared sensors used for night vision in some high-end luxury vehicles.

A specific subset of imagery that has applications in CAV is called full-motion video (*FMV*). FMV contains video that is either overlaid or embedded with other fused or derived metadata, e.g. geospatial metadata consisting of GPS coordinates, current viewport/zoom, and a directional heading. This is shown in Figure 1.8.

1.3.3.5. Convolutional Feature Maps

Convolutional Feature Maps are listed as a separate item because every manufacturer will have "hand-engineered" *features*, specific to their ADAS platform needs. Deep feature maps and even networks of convolutional feature maps are of particular importance for object detection [186]. These features, once mapped to a specific spatial area, can be transferred as vectors to another platform and translated to their coordinate origin. Since these feature maps are much smaller byte-wise than raw sensor data, but do not contain inferred (and possibly untrustworthy) information, they may provide a good-balance between transferring high-bandwidth sensor data and low-bandwidth fully-mapped 3D scenes.

1.3.3.6. Sonar

Sonar, as used in electronics, means having an ultrasonic emitter and and ultrasonic receiver. There are also some sonar systems with a single emitter/receiver. However, they all use ultrasonic sound waves in order to sense the distance to an object. This is achieved through measuring the timing between the emission of the ultrasonic wave and the reception of it, which is proportional to the distance to the object. The word *sonar* originally was an acronym for **so**und **na**vigation and **ra**nging.

Some animals have evolved to use ultrasound for target ranging and navigation (echo-location), and humans finally caught up to bats and dolphins in 1912 with a British Patent by English meteorologist Lewis Fry Richardson. Ultrasound has several characteristics which make it so useful and that have led to its use in many electronics applications. First, it is inaudible to humans and therefore undetectable by the user, which is important for parking sensors that may operate at 100 dB. Second, ultrasound waves can be produced with high directivity, which is needed for multi-sensor applications. Finally, they have a lower propagation speed than light, which allows measurement using low speed signal processing.

Currently, almost all vehicles manufacturers use ultrasound for their OEM parking sensors, as opposed to electro-magnetic sensors. These ultrasonic sensors are found as the small, ~15 mm wide, flat discs in front and rear bumpers. For parking applications most manufacturers install eight sensors (four front and back), while for AV applications, 12 sensors are often used for 360° coverage. Modern automotive ultrasonic sensors, such as those made by Murata, operate at > 58 kHz frequency, and provide detection and ranging from 10 cm up to 5 m [102].

Like Radar, sonar produces a sparse matrix of ranging information, which can be compressed easily but the information must first be GPS-tagged/fused so spatial-translation can take place at the receiver. Additionally, with 12 or more sensors producing data, combined with the fact that ultrasound is relatively short-range, other sensor data would commonly take priority for transmission between CAVs.

Table 1.2. Typical Vehicle Sensor Formats and Approximate Bandwidth.

Type	**Format**	**Bandwidth (Mbs)**
LiDAR	Binary [sparse matrix]	1280 @ 20 Hz
Video	1280x960/12-bit color [various containers]	864 @ 60 fps
Feature Maps	Binary [encoded vector]	9.4 @ 1 Hz
Sonar	Various	0.63 @ 10 Hz
Radar	Binary [sparse matrix]	0.63 @ 20 Hz
GPS/IMU	Various	0.31 @ 10 Hz

1.3.4. Modern Privacy Concerns

Modern privacy revolves around what data is collected, who is collecting the data and how it is being collected. According to Sheehan, there are four distinct groups of end-users [196]:

- unconcerned: exhibit minimal concern with privacy online
- circumspect: have minimal concern with privacy online overall, although some situations may cause them to have higher levels of concern.
- wary: have a moderate level of concern with their privacy in many situations, and several situation s cause them to experience higher than average concern with privacy.
- alarmed: are highly concerned about their privacy online.

The study focused on 1st party data collection (and spam email). Defined, here are the types of data by origin to the business consumer:

- 1st party: directly collected data from customers
- 2nd party: purchased from the data owner
- 3rd party: purchased from non-original owner

Sheehan's study was conducted in 1999, a time before constant geotracking of users and massive 3rd party data management services like Lotame existed. Back then, a consumer only had to be concerned with the 1st party implications of their choices. Consumer's role in online privacy has deteriorated so far today, that even a citizen's own government is acting as a non-consenting privacy-broker gift-giving data to 2nd and 3rd parties [235].

Broadly, there exists five kinds of data that is collected from consumers, ordered by veracity:

- Passively collected data: geolocation tracking, Google Analytics, sensor data, etc.
- Breadcrumbs: behaviors, actions or interests demonstrated across the product (website, application, etc.)
- Data supplied by the customer through a customer relationship management (CRM) or other similar system
- Directly-sourced: surveys, customer feedback, etc.
- Inferred data: Social media (linked through a key, e.g. through the CRM email address), purchased 3rd party data, etc.

CHAPTER 2

PROBLEMS AND CHALLENGES

2.1. Challenge 1: Extensibility and Compatibility

The first challenge in creating a flexible computing architecture for CAVs is one of abstraction and standards. In other words, how much abstraction is needed for different manufacturers' applications to work together. But more layering of abstractions create real-world consequences including increased latency and leaking interfaces [203].

As given in *Beautiful Code* [162], David Wheeler's famous aphorism "all problems in computer science can be solved by another level of indirection" has an important corollary: "But that usually will create another problem". This sometimes repeated as the "fundamental theorem of software engineering".

2.1.1. Subproblem 1-1: Data Format/Protocol

One of the main issues with creating a standard or protocol to encapsulate data, is how to extend the standard with new features/attributes while still providing service to older clients. This is normally referred to as "backward" and "forward" compatibility. This applies to both the formats used to model data (down to the byte level) and to the services (APIs) that interfaces those models.

Google provides some guidance on this through their API Design guidelines [88]. Backwards-compatible (non-breaking) changes include:

- Adding an API interface to an API service definition
- Adding a method to an API interface
- Adding an HTTP binding to a method
- Adding a field to a request message
- Adding a field to a response message
- Adding a value to an enum
- Adding an output-only resource field

Backwards-incompatible (breaking) changes include:

- Removing/renaming a service, field, method or enum
- Changing an HTTP binding
- Changing the type of a field
- Changing a resource name format
- Changing visible behavior of existing requests
- Changing the URL format in the HTTP definition
- Adding a read/write field to a resource message

Forward-compatibility is accomplished through loose coupling of data models and processes, i.e. to maintain forward-compatibility, a new field in a model should not cause a older version process to crash. This is different from extensibility, which refers to how easily new processes can be created without changes to other processes or the data model.

2.1.2. Subproblem 1-2: 1st Through 3rd-Party Integration

With so many different OEMs working together (and in some regards against each other), integration is understandable brittle between applications deployed on different operating systems, middleware and hardware. The application brittleness arises from tight coupling down the stack. In other words, relying upon a specific feature (in an operating system, middleware, or library) has detrimental effects when those layers are swapped out for different ones.

While subproblem 1 can be thought of a data problem, subproblem 2 can be thought of an architecture problem.

2.2. Challenge 2: Data Sharing

The second challenge is directed at the ramifications of sharing data between CAVs. The two aspects I will focus on are end-user privacy, and the actual trust between vehicles. Managing privacy in an AV is especially important since there's not an "opt-out" toggle built into the platform like there is with web browsers. The worst that would happen by disabling cookies or JavaScript on a modern website would be reduced functionality, while this would cripple the capabilities of the AV. The end result could be a fatal due to poor inference from faulty/incomplete data.

The second ramification of data sharing is trust. Since sharing of data between CAVs will most likely be a one-time interaction, the fundamental question becomes *with no prior communication, how can each vehicle trust the data originating outside of itself?* Data arriving from the other vehicle could be produced by a different sensors, software and assembled by a different OEM. And that just includes *rational* aberrations, not including exploitative or illegal participation. Only considering the two vehicles involved, there is no way to initially know that this data has not been unintentionally malformed or maliciously attacked. *A priori* knowledge may increase trust, but ultimately *a posteriori* knowledge gained during the vehicle interaction could determine the basis of trust and reputation.

2.2.1. Subproblem 2-1: Privacy (Data Ownership and Retention)

Modern data privacy has become such a normalized topic, the issue has made it on to U.S. Presidential candidates platforms. For other countries, data privacy has been a politicized subject for several years, followed by actual legislation. The European Union's GDPR of 2018 is one such regulation that has had wide ramifications for the data industry. Two specific GDPR principles had new significance for the legal, academic and business world: the reintroduced concept of *consent* along with its *revocation* as well as the right to be forgotten (*RtbF*) [170]. The GDPR both defines and separates as distinct rights, privacy and data protection. Privacy generally refers to the protection of an individual's "personal space," while data protection refers to limitations or conditions on the processing of data relating to an identifiable individual. In Article 4 of the GDPR [53], "personal data" is defined as

> any information relating to an identified or identifiable natural person ("data subject"); an identifiable natural person is one who can be identified, directly or indirectly, in particular by reference to an identifier such as a name, an identification number, location data, an online identifier or to one or more factors specific to the physical, physiological, genetic, mental,

economic, cultural or social identity of that natural person

As the GDPR is a regulation and not a directive, it identifies specific penalties for mishandling of an individual's personal data. For severe violations, this is €20 million or up to 4% of the annual worldwide turnover [revenue] of the preceding financial year, whichever is higher. As of December 2019, the largest fine levied has been €204 million against British Airways.

With the huge number of vehicles in circulation, mishandling of "personal data" by an OEM could result in disastrous financial consequences. Using Ford Motor Company as an example, they made $3.7 billion profit on $160.3 billion revenue in 2018[213]. During that year they sold 1,014,037 vehicles in Europe. If they experienced a data violation on all 1,014,037 2018 cars sold in Europe (categorically labeled as a severe "personal data" breach), the penalty could be up to $6.4 billion. Since their cash reserves in 2018 amounted to $33.9 billion, this fine would erase their global profit and decimate their cash holdings.

2.2.2. Subproblem 2-2: Trustworthiness of Non-Self Platforms

In order to utilize incoming data from other CAVs or edge nodes, trust will have to be established. This is necessary to avoid two issues. First, unintentional bad data from broken sensors or faulted software. Secondly, intentional bad data from bad actors or compromised systems. In the last 25 years, *T*rust and *R*eputation *S*ystems (TRSs) have become a mature research topic in the field of computational trust systems. Early work in the 1990's was merely based around reputation systems, but now there are entire conferences dedicated to trust computing, such as *The International Federation for Information Processing International Conference on Trust Management* (IFIPTM). Commercial implementations of TRSs are now part of mainstream client-server technology which has resulted in books on how to build TRSs in real world applications [66]. However, the literature specifically focusing on the robustness of TRSs is much more limited and still in an early stage. It should be noted that publications on TRSs usually analyze robustness to a certain extent, but typically only consider a very limited set of attacks.

Recent work has produced a taxonomy and analysis frame-work for TRSs [106]. They analyzed 24 of the most prominent TRSs based on 25 different attributes. Out of the 24 TRSs, six were analyzed further because their characteristics were representative of other TRSs. In other work, general challenges for the building robustness into TRSs described, and specific categorizing of attacks [116]. These typical attacks against TRSs are given below in Table 2.1.

Table 2.1. Various strategies for attacking trust and reputation systems, adapted from [116].

Attack type	**Short description**
Playbooks	Planned sequence of actions in order to manipulate and deceive
Unfair Ratings	Ratings that do not correctly reflect the actual experience
Review Spam	False reviews, often in conjunction with unfair, i.e. opinion spam
Discrimination	Deliberately providing different quality services to specific relying parties
Collusion	Coordinated actions among participants in order to manipulate and deceive
Proliferation	Multiple offerings of the same service in order to obscure competing services
Reputation Lag	Abuse multiple buyers before the TRS reacts to their negative feedbacks
Re-entry	Take new identity, in order to eliminate bad reputation of old identity
Value Imbalance	Exploit reputation from many low value services, for one high value fraud
The Sybil Attack	Take on multiple identities in order to generate rating and review spam

CHAPTER 3

APPROACH

3.1. Subproblem 1-1: Data Format/Protocol

They are quite a number of data formats and protocols in use in industry. Everything from Simple Object Access Protocol (SOAP) XML with Attachments to human-readable JavaScript Object Notation (JSON) over HTTP, and proprietary language binary formats (e.g. `.NET BinaryFormatter`). As computing has grown more distributed, both physically and logically, the protocols used for inter-machine communication have evolved to become both simpler and more extensible than the preceding protocols (SOAP, XML-RPC, CORBA, etc.). Abstract Syntax Notation One (ASN.1) and Protocol Buffers are two such interface description languages used to define data structures for cross-platform use.

In order to model all the data needed in a CAV, a registry of structures can be implemented to both introduce standardization and deduplication. Google's protocol buffers (protobuf) provide a way to enable serialization and deserialization of structured data between disparate services. Protocol buffers are a flexible, efficient, automated mechanism for serializing structured data, like XML, but smaller, faster, and simpler. How the data is structured is defined once, then special generated source code can easily write and read this structured data to and from a variety of data streams and using a variety of languages. This formal data definition is self-describing and can even be updated without breaking deployed programs that are compiled against the "old" format. An example of a Protobuf `.proto` definition file is shown in Listing 3.1 below.

Protobufs have some advantages over both text-based formats (JSON, XML) and even binary formats like *ASN.1* [15]:

(1) Schemas Are Awesome: By encoding the semantics of the business objects once in `proto` format, it helps "ensure that the signal doesn't get lost between applications, and that the boundaries you create enforce your business rules".

(2) Backward Compatibility for Free: JSON doesn't use number fields whereas Protocol

Buffers does. This obviates the need for version checks and avoids the need for "ugly code", making backward compatibility less of a challenge.

(3) Less Boilerplate Code: Protocol Buffers allows the `proto` generated classes evolve along with the schema whereas JSON endpoints in HTTP based services tend to rely on hand-written boilerplate code to handle the encoding and decoding of objects to and from JSON.

(4) Validations and Extensibility: The keywords in Protocol Buffers are powerful, allowing you to encode the shape of your data structure and how the classes will work in each language.

(5) Easy Language Interoperability: The variety of languages that are used to implement Protocol Buffers makes "interoperability between polyglot applications in your architecture that much simpler".

Listing 3.1. Example Protocol Buffers message declaration `person.proto`

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum TelType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message TelNumber {
    required string number = 1;
    optional TelType type = 2 [default = HOME];
  }

  repeated TelNumber phone = 4;
}
```

Official Protobuf bindings are available for Java, JavaScript, PHP, Python 2/3, Objective-C, C++, Dart, Go, Ruby and C#, with an example of a C++ program shown in Listing 3.2. With Proto3 (protobuf version 3), there is a canonical mapping between the proto message and a JSON-encoded value. This makes it easier to share data between systems and bridges the gap between legacy systems and modern microservices. Additionally, the protobuf specification includes supports remote procedure calls (RPC) through `gRPC` (or other RPC implementations). `gRPC` takes advantage of HTTP/2 for transport, so it avoids the overhead handshaking of JSON/XML over HTTP and may other network improvements, which are detailed in depth in Section 5.2.1.

Listing 3.2. Example Protocol Buffers C++

```cpp
// "Writing"
// Create new Person
Person person;

// Populate Person
person.set_name("John Doe");
person.set_id(1234);
person.set_email("jdoe@example.com");

// Serialize Person to stream
fstream output("file", ios::out | ios::binary);
person.SerializeToOstream(&output);

// "Reading"
// Read message from steam
fstream input("file", ios::in | ios::binary);

// Deserialize Person
Person person;
person.ParseFromIstream(&input);

// Read back Person attributes and print
cout << "Name: " << person.name() << endl;
cout << "E-mail: " << person.email() << endl;
```

Protobufs do have some drawbacks over equivalent JSON or XML formats:

- Smaller community: Probably the root cause of the first disadvantage. On Stack Overflow, for example, you will find about 5,000 questions tagged with `protobuf`. While there are more than 275,000 questions tagged `JSON` and 185,000 tagged `XML`.
- Resources: There isn't much detailed documentation, or technical writing online about using and developing with protobuf.
- Support: Google does not provide support for other programming languages like Swift, R, Scala and etc. There are third-party libraries, like Swift protobuf provided by Apple.

- Non-human readability: JSON, as exchanged on text format and with simple structure, is easy to be read and analyzed by humans. This is not the case with a binary format, and is makes debugging more in-depth.

For circumstances where a schema-less format is needed or encoding a text-based document is required, MessagePack (`msgpack`) is a good substitute for JSON. MessagePack is an efficient binary serialization format and is supported by over 50 programming languages and environments. Small integers are encoded into a single byte, and typical short strings require only one extra byte in addition to the strings themselves. Many of the language implementations also include RPC client/server codecs. An example of a Go program using `msgpack` is given in Listing 3.3 below.

Services over TCP/HTTP endpoints come in many shapes and sizes. How the service is exposed should flow out of the problem that the provider is trying to solve. Normally in distributed systems, this is either an RPC or a *RE*presentational *S*tate *T*ransfer (REST) API. RPC calls are directly tied to, and usually named as, functions on the server. RPC is also normally stateful, while REST is stateless [69], i.e. sessions are not persisted between connections. RPC has advantages as it can be encapsulated for passing through message queues or other distributed buses. Either style can be used over HTTP, and can serve multiple formats (JSON, XML, etc.). Servers can also provide both an RPC and a REST API, for reasons ranging from maintaining backwards-compatibility, supporting languages or clients not well supported by RPC, to simply maintaining the aesthetics and tooling involved with a RESTful architecture. An example of this is shown in Figure 3.1 below.

Other considerations for the "data format/protocol" subproblem include service discovery and content-negotiation. These concepts are normally a part of the protocol used. Additionally there are application-level considerations such sampling rate or image quality. And example of this negotiation between a CAV and an edge node is shown in Figure 3.2.

3.2. Subproblem 1-2: 1st Through 3rd-Party Integration

In order to facilitate all the differences in architectures, languages, and frameworks, a sensible solution from industry is presented: virtualization. Namely containerization through

Listing 3.3. Example MessagePack Go program with RPC client/server

```go
// create and use decoder/encoder
var (
    v interface{} // to decode/encode into
    r io.Reader
    w io.Writer
    b []byte
    mh codec.MsgpackHandle
)

dec = codec.NewDecoder(r, &mh)
dec = codec.NewDecoderBytes(b, &mh)
err = dec.Decode(&v)

enc = codec.NewEncoder(w, &mh)
enc = codec.NewEncoderBytes(&b, &mh)
err = enc.Encode(v)

// RPC Server
go func() {
    for {
        conn, err := listener.Accept()
        rpc.ServeCodec(
            codec.GoRpc.ServerCodec(conn, mh),
        )
    }
}()

//RPC Communication (client side)
conn, err = net.Dial("tcp", "localhost:5555")
client := rpc.NewClientWithCodec(
    codec.GoRpc.ClientCodec(conn, mh),
)
```

the various Linux container runtimes and storage drivers. Containers are a solution to the problem of how to get applications to run reliably when moved from one computing environment to another. This could be from a developer's laptop to a test environment, from a local staging environment into a cloud production platform, and or from an edge node to a CAV.

Containers in Linux refer to lightweight, operating system-level virtualization using the Linux Containers project (LXC). LXC is an extension of *chroot* composed of control groups (cgroups), and kernel namespacing, while Docker containers are composed of LXC (now a pure Go component) and a storage layer (normally a union file system). Although Docker has popularized containers in Linux, operating system level virtualization existed

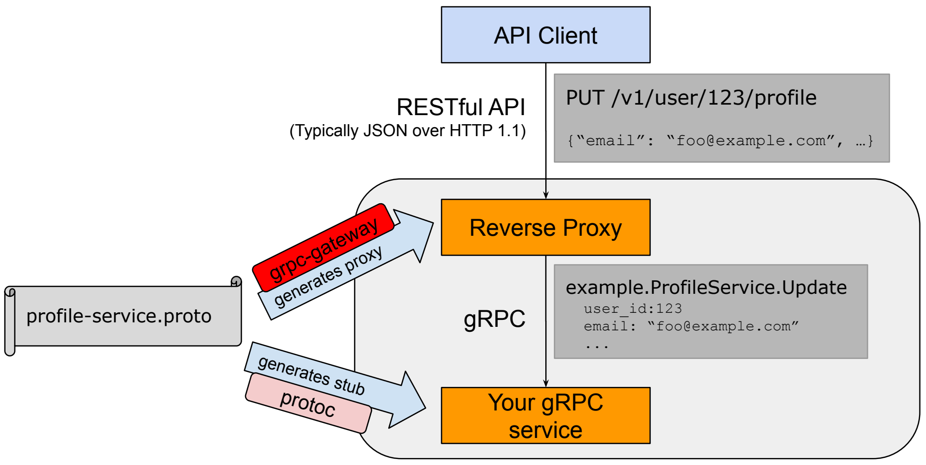


FIGURE 3.1. *gRPC-gateway*: a plugin of the Google protocol buffers compiler *protoc*. It reads protobuf service definitions and generates a reverse-proxy server which translates a RESTful HTTP API into gRPC.

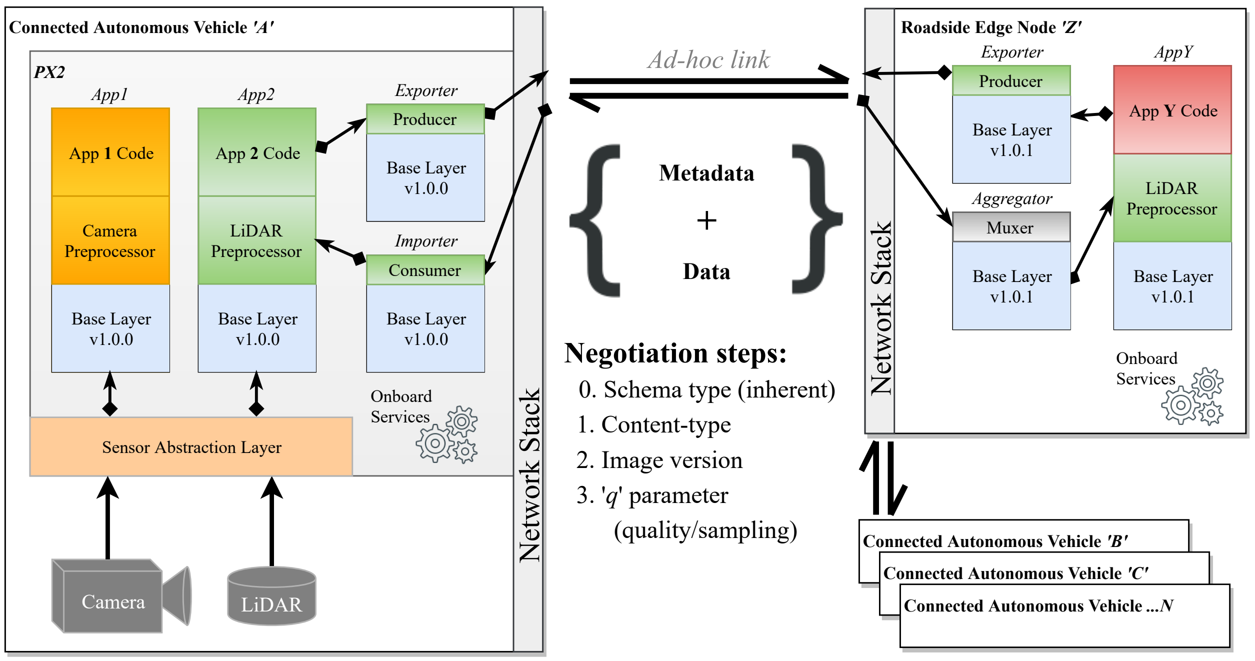


FIGURE 3.2. Communication with content negotiation between a CAV *'A'* and a roadside edge node *'Z'*. Both *App1* and *App2* on the CAV are using outdated container layers (v1.0.0), but the *Importer* and *Exporter* containers on the edge node are semantically within the same major version as the *Aggregator* and *Exporter* on *'Z'* (v1), so backward/forward-compatibility is guaranteed.

before in several forms: direct LXC (Linux), jails (FreeBSD), Workload Partitions (AIX) and Solaris Containers (Sun) [184].

With LXC, processes are isolated, but run straight on the host. This means that CPU performance is almost identical to native performance, while memory performance is a few % slower due to system accounting. Network performance also incurs a small overhead as more encapsulation/deencapsulation has to take place in software. These namespaces have been expanded to provide namespacing for several items:

- Process Identifier (PID) Namespaces: Ensures that processes within one namespace are not aware of process in other namespaces.
- Network Namespaces: Isolation of the network interface controller, iptables, routing tables, and other lower level networking tools.
- Mount Namespaces: Filesystems are mounted, so that the file system scope of a namespace is limited to only the directories mounted.
- User Namespaces: Limits users within a namespace to only that namespace and avoids user ID conflicts across namespaces.

In 2006, engineers at Google invented the Linux "control groups", abbreviated as *cgroups*. This is a feature of the Linux kernel that isolates and controls the resource usage for user processes. These processes can be put into *namespaces*, essentially collections of processes that share the same resource limitations. A computer can have multiple namespaces, each with the resource properties enforced by the kernel. The resource allocation per namespace can be managed in order to limit the amount of the overall CPU, RAM, etc that a set of processes can use.

The third piece to Docker's containerization is its 'union file system'. A container is made up of layers of images, binaries packed together into a single package. The base image contains the operating system of the container, which can be different from the operating system of the host. The operating system of the container is in the form an image. This is not the full operating system as exists on the host, with the difference being that the image is just the file system and binaries for the operating system while the full operating system includes the file system, binaries, and the kernel. This is shown in Figure 3.3. On top of the base image are multiple images that each build a portion of the container.

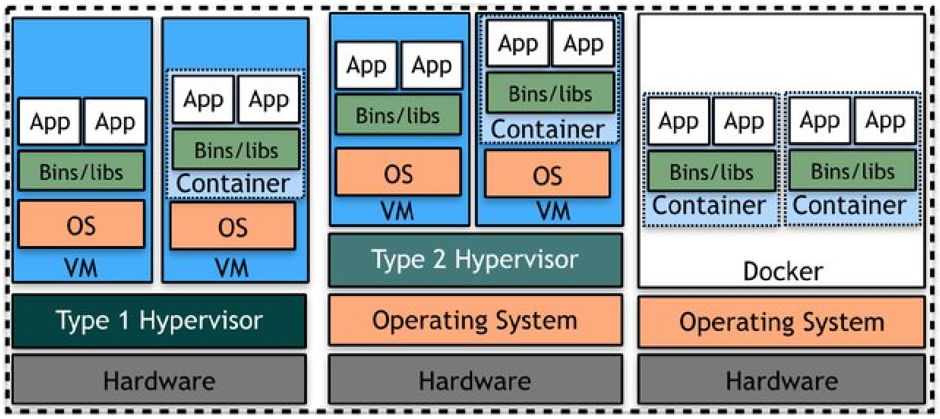


FIGURE 3.3. Operating system, virtualization, and library differences between a Type 1 hypervisor, Type 2 hypervisor, and lightweight containers [5].

While containerization benefits development, testing and deployment, there are several features that contributes to system interoperability. First, each container layer is another image that can both be versioned and de-duplicated on the host system. For example, if two applications (*App1*, *App2*) need to use a Python computer vision library, that library can be pushed into an layer and shared among all containerized applications with a single on-disk image. When *App1* needs to update that library to take advantage of new features, only it's container needs to pull down a new layer, and *App2* can continue to use the 'older' layer. These images can be versioned, cryptographically signed and pushed to a public or private image registry. Secondly, the Open Container Initiative (OCI) is a Linux Foundation project that designs standards governing container-level virtualization. There are three major standards to ensure interoperability of container technologies: the OCI Image, Distribution, and Runtime specifications. Lastly, since there are different container runtimes, each manufacturer can choose to deploy the runtime most suitable for their platform/environment. These could be well-known open-source projects (CoreOS' rkt, Docker, CRI-O, podman/buildah, Google's lmctfy), or closed-source proprietary OCI-conformant runtime. Additionally, by separating every application/service to be containerized, infrastructure orchestration is logically separated from application deployment. This frees each manufacturer from using the same deployment orchestration and enables technologies like Kubernetes (k8s) or RancherOS for deployment of backend services, or even k3s for edge nodes.

3.3. Subproblem 2-1: Privacy (Data Ownership and Retention)

In order to ensure both privacy of vehicle data and legal requirements are satisfied, I propose categorizing the type of privacy being described. For simplicity and clarity I have defined three levels of privacy, following in order of increasing data leakage:

(1) **Private**: restricted to the producer/consumer (1^{st} party)

(2) **Internal**: restricted internal to the CAV (2^{nd} party)

(3) **Shareable**: external to the CAV (3^{rd} party)

By clearly defining categories for privacy, end-users can easily decide their own comfort level with their data privacy. This gives them the ability to "opt-out" of specific data sharing, while retaining sharing needed for common CAV tasks.

To ensure that the end-user's privacy levels are respected, I will experiment with an onboard proxy server that is positioned between data producers, consumers, and the network stack on the CAV. Another onboard server will serve as a privacy registry, tagging data with the three previously defined privacy categories. This privacy registry could even scrubbing user data at the drive resource level (i.e. GPS scrambling data protected structs). An example of this architecture is shown in Figure 3.4. This work will be done with guidance from the GDPR and [219].

3.4. Subproblem 2-2: Trustworthiness of Non-Self Platforms

They are many solutions to trust and reputation, from blanket authoritative hierarchies to learned trust between unique nodes. Authoritative hierarchies include systems such as government or manufacturers *I*dentity *A*ccess *M*anagement (IAM), while learned trust could include a trustbank of compared sensor data. Each system is trying to solve two distinct issues, bad/unreliable sensors (unintentional), and bad/unreliable actors (intentional).

A hybrid approach to this problem seems most likely to be successful. A government agency could act as a root authority, and cryptographically sign identities assigned to each manufacturer. The manufacturers in turn could as an intermediate authority and sign identities to specific platforms. Once the platforms are "in the wild", they can create trust

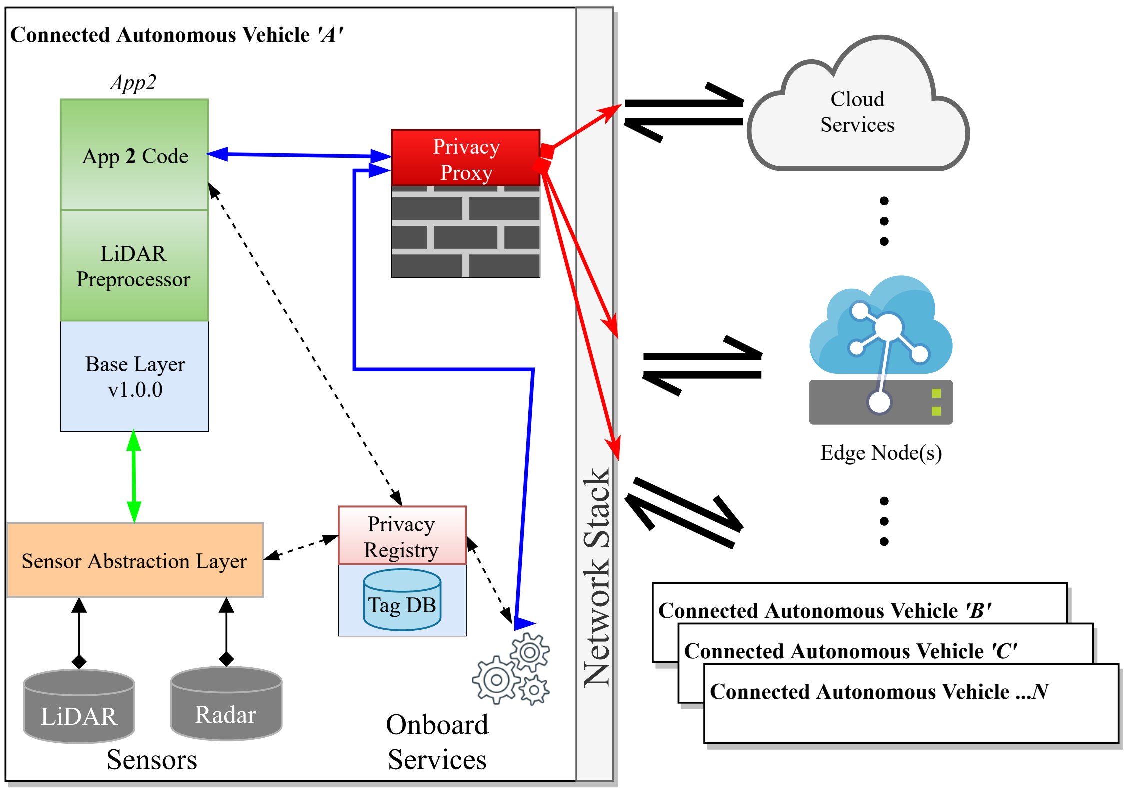


FIGURE 3.4. Architecture and data flow of a privacy proxy implemented in a Connected Autonomous Vehicle. The actual privacy proxy is shown in red, with the privacy registry shown in light red. The data flow shows an example application *App2* using the privacy registry to tag datum, and then the privacy proxy uses the registry to either block or allow the datum to private (solid green), internal (solid blue), or external consumers (solid red).

between each other through a heuristic reputation value based on various attributes, e.g. same manufacturer, same model, same sensor data, etc.

This trust can be implemented in a similar fashion to the 'privacy proxy' above, with the additional component of the TRS engine as a feedback loop for the trust system. Other concerns include 'transitive trust', also known as 'friend of a friend' (FOAF), and implementation of trust revocation.

3.5. Conclusion

My work will be limited to solving the challenge of extensibility and compatibility, by focusing on efficient data communication between CAV services and modern application orchestration. Benchmarking of example protocols with deployment onto embedded hardware will be shown in the following chapters, along with comparisons against competing standards and methodologies. Previous work in benchmarking of neural nets on embedded platforms has been performed on the Raspberry Pi and specialized low-power vision processing units.

CHAPTER 4

DATA FORMAT

The formatting of data so it can be exchanged easily and efficiently is both a technical problem and a cultural problem. Namely, just because a format is the most efficient in a certain metric, e.g. packed bytes, does not mean that developers will readily use it or integrate it within their architecture. This is evident with the proliferation of standards among all aspects of computer science and computer engineering. So my goal is not to create another "universal standard" based on specific technical merits which will change over time, but rather push forward one specific standard based on its immutable cultural merits.

Google's language-neutral data interchange format, Protocol Buffers, meet both the technical merits and the ease of exchange and development. This format standard was designed inside Google around 2001, and had evolved over the years to eventually become open-sourced in 2008. Named `protocol buffers`, the original meaning of a *buffered* C++ class has long since disappeared, while the name stuck internally at Google. Today a 'protocol message' refers to a message in the abstract sense, while 'protocol buffer' refers to the serialized copy of that message, and 'protocol message object' for the in-memory object representation of the parsed message. Culturally, Google employees embraced *protobufs* for green-field (new development) work, and then slowly over the next decade as applications and systems were redesigned and rearchitected, *protobufs* were integrated into "legacy" systems. At this point the decision to open-source *protobufs* was made because many projects Google wanted to open-source already contained *protobufs*. Soon thereafter it was re-written from scratch as version 2 (`proto2`). After some industry exposure and lessons learned, version 3 was released (`proto3`) in 2016.

Currently Google provides code generators for many languages, including targeting embedded/edge systems such as C, C++, Golang, and JavaNano. As the appeal of *protobufs* is gaining traction, third-party code generators have been created that target such diverse languages as Ada, Haskell, R, Rust, and Julia.

4.1. Details

Protocol Buffers are composed of a Interface Definition Language (IDL), a binary scheme to encode/decode formats described in the IDL, combined with a code generator/-compiler (`protoc`). The IDL specification is openly documented [89] and is defined using Extended Backus-Naur Form. Proto messages are composed of fields, which can be one of three types: `normal`, `oneof`, or `map`. Each field consists of a wire `type`, `name` and a field `number`. There may be additional options appended. Wire types can either scalar values (integer, float, string, etc.) or composite types. Arrays are represented by the `repeated` keyword in front of a field. The field number is required and must be unique as it is the tag the field uses for binary encoding, and cannot be reused within the encoding. Numbers 1-15 only require one byte, so they should be used for commonly used or repeated fields, while higher number fields should be used for less-common values. This optimization is useful because if a field isn't set in a message, a default value is used instead and not serialized and set over the network.

In order to use `protoc3`, you must have two things: a well-formatted proto file, and a proto compiler. As stated earlier, the compilers are available for almost every popular language. A toy example of a proto message can be found in Listing 3.1 above. Using the protoc compiler to generate Go code for that proto file is just the command '`protoc --go_out=$APP_DIR person.proto`'. This will generate `person.pb.go` in the given app directory, with a Go struct with defined pointer fields, along with getter functions for each field. These getters will return the previously reference default value if the field is blank. Oddly enough, these getters functions are prefixed with `Get`, in direct opposition to the Golang style guide, "neither idiomatic nor necessary to put `Get` into the getter's name." The reason for this is to wrap `nil` values with defaults while still exposing the original, exported field. What files are generated by the `protoc` compiler depends on the target language, .e.g.:

- **C++:** a header file (`.h`) and an implementation file (`.cc`) from each `.proto`, with a class for each message type described in your file.
- **Java:** a `.java` file with a class for each message type, as well as special `Builder` classes for creating message class instances.
- **Python:** a module with a static descriptor of each message type in the `.proto` file, which is then used with a metaclass to create the necessary Python data access class at runtime.
- **Go:** a `.pb.go` file with a type for each message type in your file.

Once the *.pb..go* file is generated, it can be used in the main application as an object/struct, composed of reading, and writing. Writing a message is as simple as initializing a struct from the generated code, such as `person := &pb.Person{name: "John Doe"}`, and then serialization that person struct with `data := proto.Marshal(person)`. `data` can then be saved to a file or sent across the network to another service. Reading a PB message is almost the same, just in reverse. First data is received from a source, a file for instance: `data := ioutil.ReadFile(filename)`. Then a person struct is initialized, `person := &pb.Person{}`, and the data is deserialized: `proto.Unmarshal(data, person)`. Error checking was eliminated from these reader/writer examples for clarity. The usage of protobufs within applications is similar for all supported languages. At a higher level, the developer workflow for using protobufs in an application is generically shown in Figure 4.1 below.

The encoding of a proto message produces a binary message, minimized for transmission across networks, and suitable for low-power consumers (no automatic compression/decompression). For example, using the populated message from Listing 3.2, the message size would be 31 bytes. The exact byte for byte encoding is shown in Figure 4.2. The person record is just a concatenation of its fields. Within the message, each field starts with a byte indicating it's tag number and what kind of field it is. The numbers 1, 2, and 3 correspond to the same numbers within the proto definition. If the wire type is a string, the next bytes give the length of the string, and then the UTF-8 encoding of the string. If the field is an

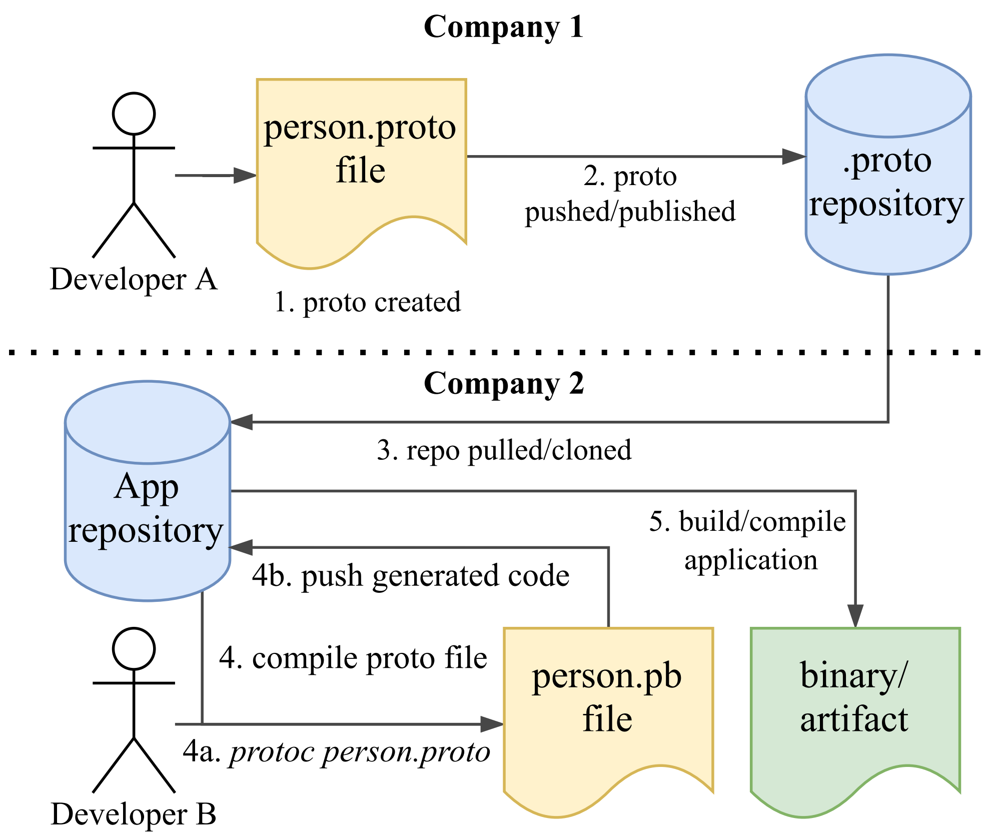


FIGURE 4.1. Developer workflow with protobufs: 1. Proto file composed; 2. Proto file published; 3. repo pulled; 4. PB file(s) generated with `protoc` compiler; 5. build the application and generate the binary/artifacts; optionally repeat step 2-5 for updates to the .proto file.

integer, then a a variable-length encoding of the number is next.

Variable integers, or `varints`, are based on the idea that most numbers are not uniformly distributed. For example, the *id* field within an organization would start with 1, and values such as invoice numbers are assigned sequentially. As smaller numbers are more common, the encoding gives preference to these smaller numbers. Two common ways of encoding `varints` are length prefixed, and continuation bits. Protobuf's use continuation bits using a simple technique: the top bit of each byte is used to determine if the number continues to the next byte, and the least significant group comes first. Using the example value of 1234, converting to binary becomes `10011010010`. Splitting them into 7-bit chunks gives `1001` and `1010010`. Moving them to least significant group first and padding to 7 bits gives `1010010` and `0001001`. Since the first byte continues to the second byte, we add a top

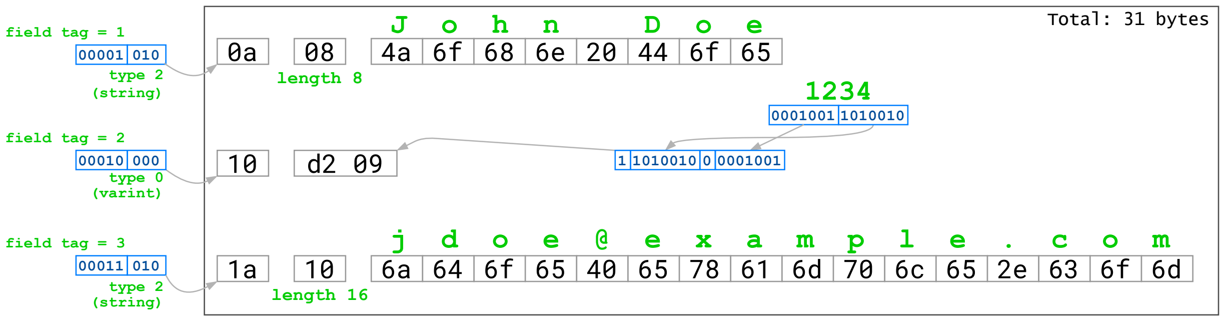


FIGURE 4.2. Illustration of example protobuf message composed of three fields: hexadecimal wire values shown in black, with decoded values and field descriptions in green.

bit to the first byte as `1`, and since there are no more bytes after the second, we add a top bit of `0`. This becomes two full bytes of `11010010 00001001`, encoding into hex as `d2 09`. This encoding is very efficient for small numbers since they will take up less space in the message and consequently less space across the network, but still allows for encoding of any sized integer. Some encoding/decoding computation has to take place for`varints` beyond 127, but the trade off is high efficiency for a majority of message transmissions. In the real world, decodings happen through prefix searches in lookup tables, negating many of the performance drawbacks. There is no need to pad encode to a byte boundary since `varints` are byte aligned and a 64-bit number can be decoded in at most 10 bytes. Additionally, because of the self-delimiting nature of `varints`, they can be concatenated together into arrays within the message without the need for other bytes in between. Non-varint numbers, composed of wire types 1 (`double`/`float`) and 5 (`double`/`fixed32`), are simply a 64/32-bit lump of data, stored in little-endian order.

UTF-8 encoding uses prefixed length encoding, with the length encoded in unary. The most significant group is first, in contrast to `varints`. UTF-8 is compatible with existing ASCII character encodings. Since the leading number of bits indicate how many extra bytes follow, decoding of UTF-8 can be quicker than continuation formatting. UTF-8 values are also more error tolerant due to the amount of "dead" bits within the encoding that can be mangled and the characters can still be decoded. Since the most significant group is first, sorting is also quicker, as values that differ greatly can immediately be shuffled.

4.2. Benefits

The simple workflow inherent in the design of protobufs is the real benefit. This publish/compile/push workflow allows developers from different teams, and even different companies to coordinate on shared definitions of data formats. With automated workflows using modern CI/CD tools, when a change is detected in a published protobuf, webhooks can be used to pull down those changes, compile the new generated code, and immediately run unit and integration tests against it. For many teams, this would be a simple, one-time change to their continuous integration/continuous delivery (or deployment) pipeline. As more abstraction is is used in the team's application architecture, having an "open" data format is an immense cultural benefit. By using a self-describing standard, business value can be created quicker and delivered to the customer, and since this format can be updated at any time, it couples well with the modern agile delivery method of "iteration over milestones".

Technically, protobufs has three main merits: wide language support, actual serialization size on the wire, and forward/backwards compatibility. By having multiple languages supported by the protobuf compiler, development can take place in whatever works best for the application architecture. This is in contrast to some 3rd party libraries that either force developers into a certain stack to interface with their APIs (libraries), or worse, cause the developers to reverse engineer the data structures so they can pack their own bytes.

4.3. Concerns

There are both technical and non-technical concerns with `protobufs`. Much like any other open-source software, there is a concern of ownership over the software and how to maintain the hosting. "Closing" of the software repository is a valid concern that has affected numerous real-world application deployments. This single point of failure also exists for hosted proto files. To mitigate risk, development teams normally clone down the repository from the hosting site and mark them as 'vendored'.

Another issue that arise from openly-hosting proto files is the risk of governance fracturing. If for instance Toyota disagreed with the format of a message that the SAE standardized, they could host their own proto files, forcing integrators to “choose” between competing, incompatible encodings. Legal enforcement, either through contracts or body membership would discourage this behavior, although there is still risk of competing standards within organizations.

4.4. Comparisons

In the following sections I will briefly describe competing encoding standards, and compare them against `protobufs`.

4.4.1. ASN.1

Abstract Syntax Notation One (*ASN.1*), has its roots in the joint 1984 standard (CCITT X.409:1984) developed by the International Telecommunications Union Telecommunication Standardization Sector (*ITU-T*) and International Organization for Standardization (ISO) with International Electrotechnical Commission (IEC). Since 1984, it has been moved to it’s own standard, and been revised several times, with the latest being in 2015, specifically defined in ISO/IEC 8824 and the X.680 series of ITU-T Recommendations.

ASN.1 defines how data structures should be assembled, so they can be serialized and deserialized across various languages and platforms [61]. Much like *protobufs*, data schemas (ASN.1 `modules`) are compiled into code libraries (normally into *codecs*) using a separate *ASN.1* compiler. Large numbers of protocols are backed by *ASN.1*, from a precursor to email, to modern cryptography standards.

ASN.1 uses *types*, *identifiers*, and *constraints* to define interfaces, which are organized in *modules*. Each *type* (such as `People`) in *ASN.1* must begin with an uppercase letter. *Items* that are components of a message (such as `name`, `id`, `email` and `telNumbers`) are called *identifiers* and must begin with a lowercase letter. The limitations that are defined on *items* are called *constraints* and can even be combined using union and intersection logical operators. These *constraints* can either be explicit individual allowed values, specific value ranges, or restrictions on the length of the value. In *ASN.1*, all definitions are placed inside of a module which starts with the `BEGIN` keyword and closed with the `END` keyword. Listing

3.1 below shows an equivalent 'person' message using the *types*, *identifiers* and *constraints* of *ASN.1* organized into a *module* named `People`.

Listing 4.1. Example *ASN.1* message declaration `person.asn`

```
People DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
  Person ::= SEQUENCE {
    name PrintableString,
    id INTEGER (-2147483648..2147483647),
    email UTF8String OPTIONAL,

    telNumbers ListOfTelNumbers
  }

  ListOfTelNumbers ::= SEQUENCE (SIZE (1..4)) OF TelNumber

  TelNumber ::= SEQUENCE {
    telType ENUMERATED {mobile(0),home(1),work(2)} DEFAULT home,
    number NumericString (SIZE (10))
  }
END
```

A key difference with *ASN.1* is that it offers several different human-readable or binary-only encodings (notes in italics):

- Basic Encoding Rules (BER) *binary*[179]
- Distinguished Encoding Rules (DER) *binary*[179]
- Canonical Encoding Rules (CER) *binary*[179]
- Basic Packed Encoding Rules (PER) *aligned/unaligned, binary*[180]
- Canonical Packed Encoding Rules (CPER) *aligned/unaligned, binary*[180]
- Basic XML Encoding Rules (XER) *human*[181]
- Canonical XML Encoding Rules (CXER) *human*[181]
- Extended XML Encoding Rules (EXER) *human*[181]
- Octet Encoding Rules (OER) *binary*[182]
- Canonical Octet Encoding Rules (OER) *binary*[182]
- JSON Encoding Rules (JER) *human*[183]
- Generic String Encoding Rules (GSER) *human*[133]
- BACNet Encoding Rules *binary*[155]

- Signalling Specific Encoding Rules (SER) *binary*[36]

Many more encodings have been proposed but were abandoned due to lack of support or incompleteness. Additionally, if none of the existing encoding rules are suitable for a given application, the Encoding Control Notation (ECN) of *ASN.1* provides a way for a programmer to define their own customized encoding rules. With regards to remote procedure calls, *ASN.1* is like *protobufs* whereas they are not tied to a single RPC language/stack, and can be used by different library implementations.

While *ASN.1* has been used almost exclusively within the telecom industry (e.g. SS7, DSRC, GSM, 3G), there is heavy usage in computer networking (e.g. LDAP, Microsoft RDP, SNMP) and cryptography (e.g. PKCS, Kerberos). Despite being used in almost every computer communication in one form or another, *ASN.1* is surprisingly still obscure after two decades of use. This is due to a number of factors, namely:

(1) Documentation/specifications are not freely available from ITU-T/ISO
(2) The wide-variety of encoding rules makes efficient encoding for composite types non-trivial
(3) The complexity of the specification/kitchen-sink syndrome

These are not new issues, and even a 20-year-old IETF draft references some of these problems [249]. The lack of word alignment means poor performance, and buggy *ASN.1* parsers produced security issues that many have not forgotten. While *ASN.1* has been embraced by many fields within computer science/engineering, simpler is usually better, and the marginal difference in wire size does not make up for its shortcomings, especially for intra-node communication.

4.4.2. XML

The E**x**tensible **M**arkup **L**anguage was created by the World Wide Web Consortium (W3C) in a 1998 Recommendation [246]. Like HTML, XML is based on SGML (Standard Generalized Markup Language), and was really designed to be a simpler markup language, as the perceived complexity of SGML was not driving adoption. SGML itself came from

Scribe, an earlier attempt at a word processing system similar to LaTeX [82]. XML was designed to be semantic-free, as opposed to HTML. XML is a meta-language in that is a language for describing other languages. HTML is not a suitable format for data serialization because it's really just a mechanic to generate a semantic layer for displaying of embedded data (normally in combination with presentation through style sheets). In other words, just because the string 'John Doe' is wrapped in an HTML `<div>` tag, there is no information present that this is a **Person** data structure, even though if this HTML is shown to a human with extra context like a preceding `<b>Name:</b>` fragment, a human will be able to ascertain what information is being conveyed. While HTML tags tell a browser how to display this **Person** information, the tags don't tell the browser what the information is. With XML, meaning can be assigned to the tags in the document, and that 'John Doe' fragment would become `<name>John Doe</name>`. Thousands of different formats have been established based on XML, from instant messaging protocols (XMPP) to desktop application (Microsoft Office documents). Within the Java community, it became the de facto standard for data-exchange, with Sun Microsystems' Jon Bosak, chair of the XML Working Group, commenting "XML gives Java something to do" [21]. An example of a XML message file is given in Listing 4.2. Note that this is exactly what would be transmitted over the wire, excluding any inter-tag whitespace minification or external compression.

Although there exists XML compilers for some languages, most are not used as XML parser libraries are feature-complete and provide everything needed to efficiently serialize and deserialize data structures. For most languages these parsers are either document object model-based (DOM), or stream-based. DOM refers to the in-memory representation of the XML document, and must be completely loaded for transversal and processing. Stream-based loads individual nodes of the XML document and either runs callbacks or fires events (depending on the language involved). Each have advantage, DOM allows for random access to each node in the document, while stream driven algorithms have a smaller memory footprint and are typically much faster due to avoiding tree-traversals. Some languages even have a third parser type that is a middle ground between the two former types, like Java's

StAX.

Additionally, to validate that a given XML document is formatted correctly, a two step process is invoked. First the document is passed through a 'Well-Formedness' checker, to ensure the document conforms to the XML specification. Then an external file called an XML Schema Definition (XSD) is required. XSD provides 19 primitive data types, e.g. `boolean`, `float`, `duration`, `string`, etc. Since XSD is an XML schema language, XSD is used to express a set of rules which an XML document must conform in order to be considered "valid". XSD was also designed with the intent that determination of a document's validity would produce the context of the document's conformity. Such a post-validation infoset is useful in XML document processing and versioning. The XML Schema data model includes the *vocabulary* (element and attribute names), *content model* (relationships and structure) and *data types*. This collection of information is called the Post-Schema-Validation Infoset (*PSVI*).

While the primary purpose of an XML schema is to validate XML documents, XML schemas can also be used for code generation and generation of human-readable documentation for a given XML format. An example of a companion XSD to Listing 4.2 is given below in Listing 4.3. Regarding document validity, an XML document can be one of three kinds:

- **Well-formed**: meet the XML syntax rules but don't have a schema.
- **Valid**: are well-formed and meet the rules defined by their schema.
- **Invalid**: that don't meet the syntax rules defined by the XML specification or their schema.

The major benefit of XML is being human-readable/editable, but since most, if not all, communication in CAV systems is machine-to-machine, the benefits of human readability are eliminated. XML also has extensive tooling and integration with all modern languages. Additionally, it is the basis for several other remote procedure protocols, namely SOAP. XML is schema-based, but it is grossly verbose due to the inherit nested wrapping of the markup language. Almost all XML for transmission is compressed, either during the encoding phase, or at the proxy server. In either case, the extra CPU cycles needed for compression/decom-

Listing 4.2. Example *XML* `person.xml`

```xml
<?xml version="1.0" encoding="UTF-8"?>

<person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="person.xsd">
  <name>John Doe</name>
  <id>1234</id>
  <email>jdoe@example.com</email>
  <telNumbers>
    <telNumber>
      <telType>mobile</telType>
      <number>123-456-7890</number>
    </telNumber>
    <telNumber>
      <number>555-555-5555</number>
    </telNumber>
  </telNumbers>
</person>
```

pression are not suited for edge devices. Even though it is schema-based, an XML document can be parsed without knowledge the schema due to the nature of how the markup tags are formatted. Prior schema knowledge is needed for validation of an XML message though, similar to `protobufs` and `ASN.1`. As stated in 2003 by Siméon and Wadler [200, p. 1], "the essence of XML is this: the problem it solves is not hard, and it does not solve the problem well".

Using the equivalent, minimal comparison in Listing 4.4, the protobuf version on the wire would be approximately be 28 bytes long and take around 100 to 200 nanoseconds to parse. The XML version is at least 69 bytes after whitespace minification, and would take around 5,000 to 10,000 nanoseconds to parse. This is not including the compression/decompression that normally takes place during transmission of an XML document, but since `gzipping` such a small fragment actually increases the wire size (to ~80 bytes), it is outside the scope of this comparison. While this is a toy example, the orders of magnitude difference between the two is an insurmountable barrier to cross and Protocol buffers have other advantages over XML for serializing structured data:

- simpler (specification and documentation)
- 3 to 10 times smaller (both representation and wire-size)

Listing 4.3. Example *XSD* `person.xsd`

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  attributeFormDefault="unqualified"
  elementFormDefault="qualified">
  <xs:element name="person">
    <xs:complexType>
      <xs:sequence>
        <xs:element type="xs:string" name="name" />
        <xs:element type="xs:integer" name="id" />
        <xs:element type="xs:string" name="email" />
        <xs:element name="telNumbers">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="telNumber" maxOccurs="4" minOccurs="1">
                <xs:complexType>
                <xs:sequence>
                    <xs:element
                    name="telType"
                    default="home"
                    minOccurs="0"
                    maxOccurs="1">
                      <xs:simpleType>
                        <xs:restriction base="xs:string">
                          <xs:enumeration value="mobile" />
                          <xs:enumeration value="home" />
                          <xs:enumeration value="work" />
                        </xs:restriction>
                      </xs:simpleType>
                    </xs:element>
                    <xs:element type="xs:string" name="number" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

- 20 to 100 times faster (due to less parsing and validation overhead)
- less ambiguous (explicit rules for types and composition)
- easier to use programmatically (fewer, less verbose data access classes are generated) [shown in Listing 4.5 below]

LISTING 4.4. Comparison of an XML fragment and the equivalent `protobuf` textual format representation.

```
<person>
  <name>John Doe</name>
  <email>jdoe@example.com</email>
</person>
```

```
person {
  name: "John Doe"
  email: "jdoe@example.com"
}
```

LISTING 4.5. Comparison of manipulating an equivalent XML fragment and the corresponding `protobuf` in C++

```
/* Accessing XML elements */
cout << "Name: "
     << person.getElementsByTagName("name")->item(0)->innerText()
     << endl;
cout << "E-mail: "
     << person.getElementsByTagName("email")->item(0)->innerText()
     << endl;

/* Accessing protobuf equivalent fields */
cout << "Name: "   << person.name()  << endl;
cout << "E-mail: " << person.email() << endl;
```

4.4.3. CORBA

The **C**ommon **O**bject **R**equest **B**roker **A**rchitecture (CORBA) is a standard defined by the **O**bject **M**anagement **G**roup (OMG) in 1991 for the exchange of "objects" between disparate computer systems. There has been three major versions of CORBA ratified (the first version was only for C and did not even provide interoperability), with the latest being in 2012. CORBA defines both an Interface Definition Language and communication broker, with the design being a "distributed" objects (in the Object Orientated Programming-sense). This communication broker, an Object Request Broker (ORB), contains an Object Adapter, which maintains internal structures like reference counts, object/reference instantiation policies, and object lifetime policies. The Object Adapter is used to register instances of the generated code classes.

To use a CORBA-based distributed object interface, a developer must write the IDL code that defines the object-oriented interface to the logic the system will use or implement. Most ORB implementations includes a tool called an IDL compiler that translates the IDL interface into the target language for use in that part of the system. A traditional compiler then compiles the generated code to create the linkable-object files for use in the application. CORBA mappings were first developed for C, then C++ and Java, with other languages following. As CORBA is a distributed object system, there are several ways to share objects, namely either by reference, or by value. An example of a CORBA IDL file is given in Listing 4.6, note that the CORBA IDL has no definition for *optional* fields or *default* values for data types.

As given in [81], the typical life-cycle of a CORBA application is as follows:

(1) define the service as interfaces in IDL

(2) compile the IDL to generate client stub and server skeletons

(3) implement the service and associate it with the skeletons via the portable object adapter (POA)

(4) publish the service with a Naming or Trading Service for use by clients.

The CORBA client processing involves the following:

(1) contact the Naming Service for the desired service and retrieve the appropriate object reference,

(2) invoke operations on the object reference using the IDL-compiler generated stubs. Alternatively, clients can infer the operations supported by the service by consulting an interface repository (IR) and dynamically create requests populating them with the appropriate parameters using the dynamic invocation interface (DII)

(3) process incoming reply or exceptions.

An overview of CORBA components is given in Figure 4.3.

CORBA was just not designed for modern distributed systems and is considered both a technological and procedural failure. As stated by Henning [101] in 2006 , "some of the OMG's early object services specifications, such as the life cycle, query, concurrency control, Listing 4.6. Example *CORBA* IDL `people.idl`. Note that the lack of *optional* fields or *default* values for data types.

```
module People {
    enum TelType {MOBILE, HOME, WORK};

    struct TelNumber {
        string number;
        TelType type;
    };

    struct Person {
        string name;
        long id;
        string email;

        TelNumber telNumber[4];
    };
};
```

relationship, and collection services, were not only complex, but also performed no useful function whatsoever. They only added noise to an already complex suite of specifications, confused customers, and reinforced CORBA's reputation of being hard to use." It's shortcomings far outweigh any benefits it once had, and the industry has mostly moved on to better paradigms with CORBA seen as a 'lesson learned'.

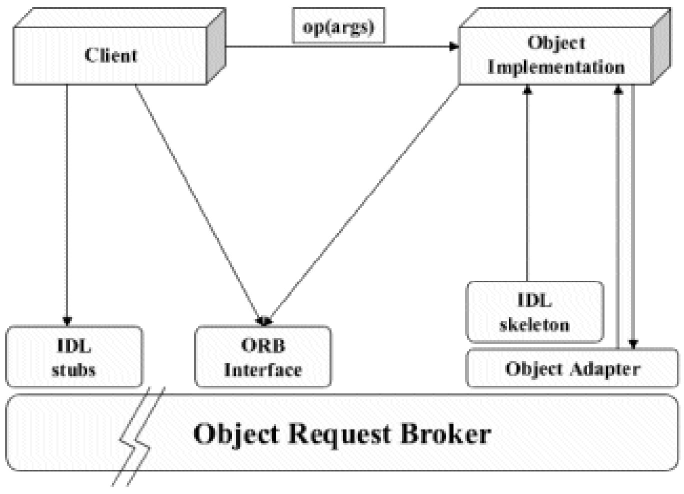


Figure 4.3. Components in the CORBA 2.x Reference Model. [81]

4.4.4. JSON

Around the year 2000, **J**ava**S**cript **O**bject **N**otation (JSON) arose out of a need to transfer data between a web server and a client browser without using plugins like Java applets or Adobe Flash. This was in conjunction with the development of Ajax (Asynchronous JavaScript and XML) technologies in web browsers. Today the *XML* in Ajax has been supplanted by JSON, which lead to the rise of the RESTful paradigm [69] being embraced by everything from startups to industry leaders and government bodies.

JSON has been defined as a IETF standard with several documents evolving the specification over several years: ECMA-262, RFC 4627, RFC 7158, RFC 7159, ECMA-404, and finally RFC 8259/STD 90. It's also an ISO standard under ISO/IEC 21778:2017. These changes in the specification have mostly been driving JSON to be a more generalized data interchange format, as JSON has become less tied to JavaScript over the preceding years with languages other than JavaScript adding 'batteries-included' support for JSON. In most languages, accessing JSON data fields/attributes is similar to XML, using marshaling/unmarshaling techniques or direct pathing. A notable difference is that while streaming XML is an established specification, there is only limited support for the JSON streaming specification formalized in IETF RFC 7464, and many libraries implement their own streaming tokenizer, either line-delimited, length-prefixed, concatenated, or record-separated.

JSON is a human-readable (and editable) format, which maps one-to-one to corresponding JavaScript data structures. An example message of the recurring 'person' model is given in Listing 4.7 below. Like XML, a JSON message relies upon an external file/process for validation (a JSON schema file), but unlike XML's `noNamespaceSchemaLocation` tag attribute, there is no built-in mechanism for a JSON message to reference their corresponding schema. JSON Schema is still an IETF draft, with the latest being the 2019-09 IETF Draft (formerly Draft 8). [236]. JSON Schema is verbose, but includes many features like collection limits, value length limits/ranges, and stranger rules like *multipleOf* for numeric types [59]. A complementary JSON Schema to Listing 4.7 is shown in Listing 4.8 below.

JSON is a context-free grammar and can be parsed without advanced schema knowledge, but unlike XML, does include some basic types. Namely four simple data types: `number`, `string`, `boolean`, and `null`, along with the two composite types: `array`, and `object`. Noticeably absent are types for *date*, or *time*, or *duration* along with *enums*. All the types are straight forward except for `object`, which is JavaScript's loose equivalent to a key-value 'map' (hash, associative array, etc.), and `number`, which is actually defined as a 64-bit IEEE 754 *float*. `Number` makes no requirements regarding implementation details such as overflow, underflow, loss of precision, rounding, or signed zeros, and different implementations may treat *integers*, *floats* or *scientific notation* differently. `Objects` in JSON can only use strings for keys and the values can only be other JSON defined types, i.e. they cannot hold references to other keys, regular expressions or other esoteric data like anonymous functions. Regarding language differences, note that JavaScript `arrays` and `objects` can hold multiple types at once, i.e. they are heterogeneous collections which many languages do not support.

With JSON, the data representational format is the same format for wire transmission, so there is no need for a compiler for code generation or 'helper' classes. Like XML, most of the performance bottlenecks with JSON are found in the parser implementations, and this has created a proliferation of third-party libraries trying to edge out incremental performance. For Java, there are over 24 third-party libraries and 23 for C++. Academic papers are also a hot topic for JSON performance, with 42,000 papers returned for a Google Scholar search for "JSON performance comparision". For CAV/CV systems, the benefit of a human readable/editable format like JSON (and XML before it), doesn't really apply. While less verbose then XML, JSON is not a binary format and consequently suffers from unpacked "byte bloat".

Listing 4.7. Example *JSON* message `person.json`

```json
{
    "person": {
        "name": "John Doe",
        "id": 1234,
        "email": "jdoe@example.com",
        "telNumbers": [
            {
```

```
          "number": "123-456-7890",
          "telType": "mobile"
        },
        {
          "number": "555-555-5555"
        }
      ]
  }
}
```

Listing 4.8. Example *JSON* schema message `person.schema.json`

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "additionalProperties": false,
  "properties": {
    "person": {
      "type": "object",
      "additionalProperties": false,
      "properties": {
        "name": {
          "type": "string"
        },
        "id": {
          "type": "integer"
        },
        "email": {
          "type": "string"
        },
        "telNumbers": {
          "type": "array",
          "items": [
            {
              "type": "object",
              "additionalProperties": false,
              "properties": {
                "number": {
                  "type": "string"
                },
                "telType": {
                  "type": "string",
                  "default": "home",
                  "enum": [
                    "mobile",
                    "home",
                    "work"
                  ]
                }
              },
              "required": [
                "number"
              ]
            }
          ],
          "minItems": 1,
          "maxItems": 4
        }
      },
      "required": [
        "name",
        "id",
        "telNumbers"
      ]
    }
  },
  "required": [
    "person"
  ]
}
```

MessagePack (`msgpack`) is a binary ‘form’ of JSON and has types that loosely correspond to those in the JSON format, with explicit types for *integers*, *floats* and *timestamps* [74]. It was designed for efficient and fast network transmission though, and not necessarily for low-resource computing. Like JSON, it is schema-less, and like protocol buffers, it requires interface libraries to encode/decode messages. MessagePack doesn’t support external validation of messages, as there are enough differences in types between JSON and MessagePack that JSON Schemas cannot be used. MessagePack has gained a significant following, as for many applications it can be a ‘drop-in’ replacement for JSON, while eliminating performance issues with JSON parsing, and reducing bandwidth cost (a common theme with public cloud deployments). Using the JSON example in Listing 4.7, the wire size would be 314 bytes (whitespace minified, uncompressed). Converted to binary MessagePack, the wire size is 121 bytes, a reduction of over 250%. Over 50 languages support MessagePack, but as with all binary formats, there still is the downside of losing human-readability,

YAML is another JSON offshoot, and is a superset of JSON. Widely used for storing data inside configuration files, it is not a popular format for data exchange since it is even more verbose than JSON is. YAML is more human-readable than JSON and even includes the ability to add comments to documents, along with allowing embedding of other formats (such as JSON or XML) within a YAML file. The YAML specification has evolved drastically over the years, with latest revisions having support for references, and relational anchors. These features have penalized the time and resources needed for serialization/deserialzation though, with YAML parsers having to contend with a much larger grammar set than JSON.

JSON is a great format for RESTFful, human-readable data exchange, and the widespread usage within industry reinforces this. Due to the issues stated above, I can not recommend it for machine-to-machine communication. In this context it offers no benefits over other formats, and even has major drawbacks, including the lack of a built-in schema, versioning, or streaming support. The large wire size of a JSON message is the final problem for an environment like CAVs, where bandwidth and latency are paramount concerns.

4.4.5. Thrift

Apache Thrift is both an IDL and an RPC client/server. Designed internally by Facebook and open-sourced in 2007, Apache Thrift was brought into the Apache Software Foundation (ASF) [176]. To address performance and scalability concerns within Facebook services, Facebook forked *Thrift* and rewrote much of the C++ RPC backend for the next seven years. In 2014 they re-open sourced their forked *Thrift* as *fbthrift*, while 'classic' *Thrift* still survives as a "Top Level Project" within ASF. Besides heavy usage at Facebook, Thrift users include Twitter (through `Finagle`), Foursquare, Pinterest, Uber (through `TChannel`), and Evernote. Thrift is a reliable, mature choice for a Service-Oriented Architecture (SOA), and is a favorite for including in data serialization case-studies.

Since the first component of *Thrift* is an IDL, a compiler must be used to generate code and bindings for the target language, similar to `ASN.1` and `CORBA`. The *Thrift* IDL specifies *basic types*, *structs* and *containers*, along with enumerations ((`enum`), type aliasing (`typedef`) and constants (`const`). The type system allows programmers to use native types as much as possible, no matter what programming language Thrift is used with. The *basic types* include boolean (`bool`), byte (`byte`), 16, 32, and 64-bit integers (`i16, i32, i64`), 64-bit floating point numbers (`double`), and strings (`string`). Thrift `structs` are similar to C/C++ structs whereas they are the common objects in Thrift and contain a set of strongly-typed fields. Containers include an ordered array (`list`), an unordered array of unique items (`set`), and an associative array (`map`). Documentation can also be added to `structs`, *fields* and `services`, using the Javadoc comment style (`/** ... */`). HTML documentation can be automatically generated from these comments using the Thrift compiler. An example of the 'person' definition is given below in Listing 4.9. Note that the Thrift IDL does not allow nested structs and line separators are either not present or must be either a comma or semi-colon. Service definitions for the Thrift RPC are added to the same file as the data IDL, and stubs for service functions/methods are generated during the compilation phase. An overview of Apache Thrift components and architecture is given in Figure 4.4. Apache Thrift components are show in *green* and code generated from the Thrift IDL in *white*. There are four protocols supported for encoding data:

- TBinaryProtocol: A binary format that is not optimized for space efficiency. This protocol is faster to process than the plain text protocols but is more difficult to debug than the human-readable formats.
- TCompactProtocol: A more compact binary format which is more efficient to process.
- TJSONProtocol: This protocol uses JSON for serialization.
- TSimpleJSONProtocol: A write-only protocol that cannot be parsed by Thrift because it drops metadata using JSON. Suitable for parsing by scripting languages.

Thrift IDL is similar to `protobufs`, with an emphasis on forwards and backwards-compatibility. Attention must be made to not rename or reorder field numbers, and there are several best practices that must be follow. This includes making every field `optional`, which seems counter-intuitive, but since the IDL and RPC are coupled, message validation is pushed up the stack to the application level. In direct comparison to `protobufs`, Thrift was originally more feature-complete, had faster serialization/deserialization, and supported dozens more languages as the original *v2* release of `protobufs` didn't even include support

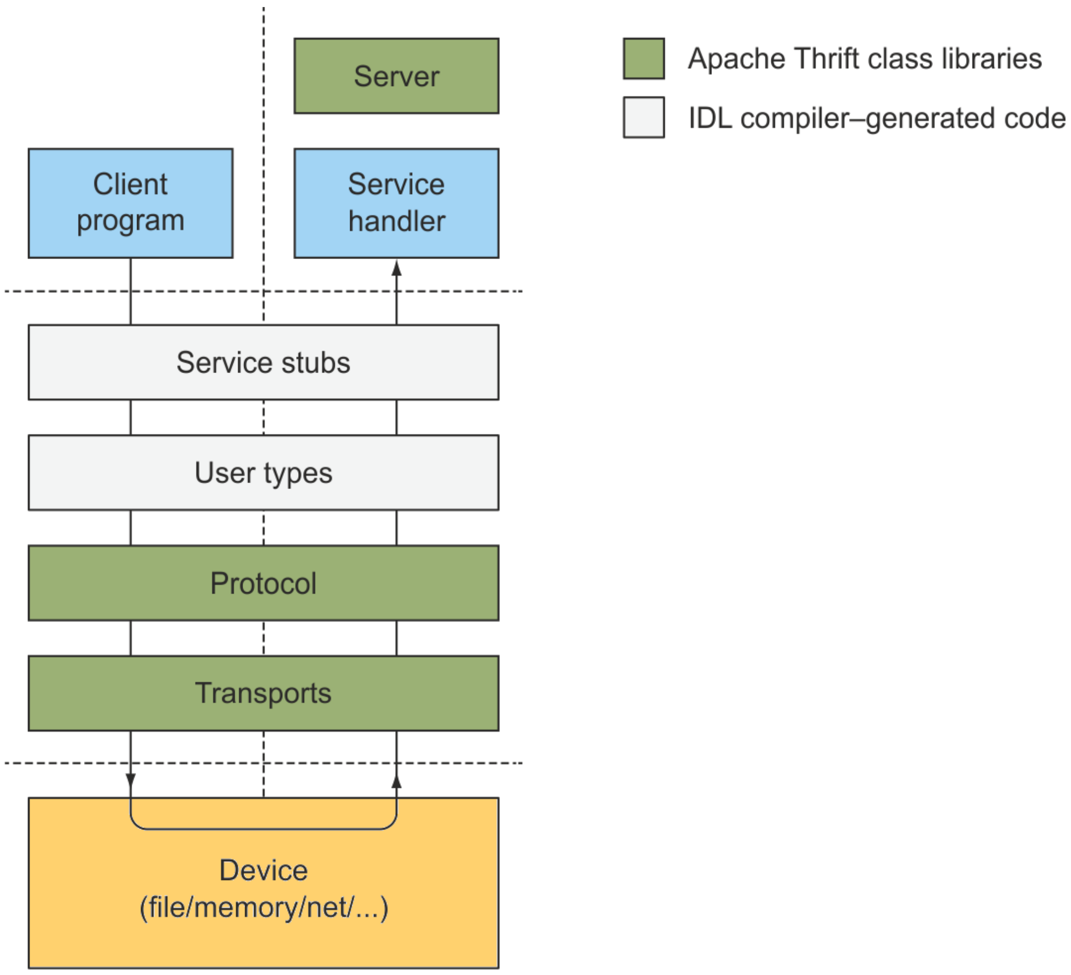

Server
Apache Thrift class libraries
IDL compiler–generated code
Client program
Service handler
Service stubs
User types
Protocol
Transports
Device (file/memory/net/...)

FIGURE 4.4. Components and Architecture Diagram of Apache Thrift data exchange between a client and server [1]. Apache Thrift components are show in *green* and code generated from the Thrift IDL in *white*.

for Google's own language, Golang. Since *v3*, the feature-set has surpassed Thrift and gone beyond, especially with IDL options like class naming and JSON field tagging. Additionally, `protobufs` on the wire are still ~30% smaller than equivalent Thrift messages due to variable length integer coding, both for values and for field identifiers. The only thing that `protobufs` lack are unique ordered lists (sets), which do not have , Thrift also differs in that it is both an IDL and an RPC implementation, but as evidenced by the multiple 'versions' of Thrift, tightly coupling the IDL to RPC interfaces has led to friction within the open-source community. Another thing to consider is the developer community, with `protobufs` having 700 public contributors and 46,000 stars on GitHub, while Thrift and `fbthrift` have ~400 contributors each and 10,000 stars combined.

Listing 4.9. Example *Thrift* IDL file `person.thrift`. Note: lack of nested structs, line separators are defined as `,` (comma), `;` (semi-colon), or blank.

```
enum TelType {
  MOBILE,
  HOME,
  WORK
}

struct TelNumber {
  1: required string number,
  2: optional TelType type = TelType.HOME
}

struct Person {
  1: required string name;
  2: required i32 id;
  3: optional string email;
  4: list<TelNumber> phone;
}
```

4.4.6. Avro

Apache Avro is a data serialization framework and RPC developed through the Apache Hadoop project (distributed map/reduce). For Hadoop, Avro is the main technique for serializing data for persistent storage, and also for distributing messages to other nodes in a binary, wire-compact format. Avro's data format is actually JSON, with a formal schema defining what constitutes an Avro Object Container File (OCF), the serialized representation of Avro data.

The OCF is composed of a header consisting of the Avro version, the schema definition, and a sync marker, and then one or more data blocks holding the actual data. The OCF can be binary-encoded for space, or JSON-encoded for human-readability and debugging. The Avro specification designates two kinds of available types [70]:

'Primitive' types:

- `null`: no value
- `boolean`: a binary value
- `int`: 32-bit signed integer
- `long`: 64-bit signed integer

- `float`: 32-bit floating-point number (IEEE 754)
- `double`: 64-bit floating-point number (IEEE 754)
- `bytes`: sequence of 8-bit unsigned bytes
- `string`: unicode character sequence

'Complex' types:

- `record`: schema's main container, holding an array of `fields` objects composed of four fields:
  - `name`: a JSON string providing the name of the field
  - `type`: one of these 'primitive' or 'complex' types
  - `doc`: a JSON string describing this field for users
  - `default`: default value
- `enum`: an enumerated type composed of two fields:
  - `name`: a JSON string providing the name of the field
  - `symbols`: a JSON array, listing symbols, as JSON strings
- `array`: an ordered list composed of two fields:
  - `name`: a JSON string providing the name of the field
  - `items`: the schema type of the array's items
- `map`: an associative array composed of two fields (the map keys must be `strings`):
  - `name`: a JSON string providing the name of the field
  - `values`: the schema type of the map's values
- `union`: a union datatype is used whenever the field has one or more datatypes represented as a JSON array
  - e.g. if a field that could be either an int or null, then the union is represented as `["int","null"]`
- `fixed`: a fixed number of bytes for storing binary data composed of two fields:
  - `name`: a JSON string providing the name of the field
  - `size`: an integer, specifying the number of bytes per value

Using the Avro schema, the 'person' model is given in Listing 4.10 below. Notice that the Avro schema has no specification for limits on array sizes and all field are mandatory in Avro. For a field to be optional, the first `union` type must be `"null"`, and the field's `default` value must be `null`, e.g. the 'email' field with the 'person' field on line 21.

Listing 4.10. Example *Avro* schema `person.avsc`

```
 1 {
 2   "name": "PersonClass",
 3   "type": "record",
 4   "namespace": "com.hochstetler.avro",
 5   "fields": [
 6     {
 7       "name": "person",
 8       "type": {
 9         "name": "person",
10         "type": "record",
11         "fields": [
12           {
13             "name": "name",
14             "type": "string"
15           },
16           {
17             "name": "id",
18             "type": "int"
19           },
20           {
21             "name": "email",
22             "type": [
23               "null",
24               "string"
25             ],
26             "default": null
27           },
28           {
29             "name": "telNumbers",
30             "type": {
31               "type": "array",
32               "items": {
33                 "name": "telNumbers_record",
34                 "type": "record",
35                 "fields": [
36                   {
37                     "name": "number",
38                     "type": "string"
39                   },
40                   {
41                     "name": "telType",
42                     "type": "enum",
                        "symbols" : ["MOBILE", "HOME", "WORK"],
                        "default": "HOME"
                    }
                ]
            }
        }
    }
  ]
 }
}
]
}
```

With Avro's use of JSON, there is no need for an 'extra' compile phase for code and interface generation when used in application development. This simplifies working with dynamically typed languages like JavaScript, Ruby and Python immensely. For strongly-type languages like C++, Go, Rust, Java, and TypeScript, there is no real benefit gained. Avro's OCF format that joins both the schema and data does eliminate an entire step in the JSON/XML validation process, while also allowing dynamic languages to reflect on the data structures inside. While Avro is used heavily in the Hadoop project, there is little use outside of that, especially with languages other than Java.

4.5. Protobuf Examples

In this section I will discuss several `protobuf` examples and and explain their importance.

4.5.1. Serialization/Deserialization Benchmark Example

Benchmarking is a notoriously thorny endeavour, fraught with many issues arising from seemingly inconsequential techniques. The following benchmarks are not meant to be rigorous, merely to show consistent differences between serialization libraries within a single language. In Figure 4.5, Go's own benchmarking framework is used to serialize and deserialize toy data models, to illustrate the differences in speed, RAM usage and raw allocations. The testing file that generated this benchmark is in Listing 4.11.

Listing 4.11. Benchmark serialization/deserialization testing file.

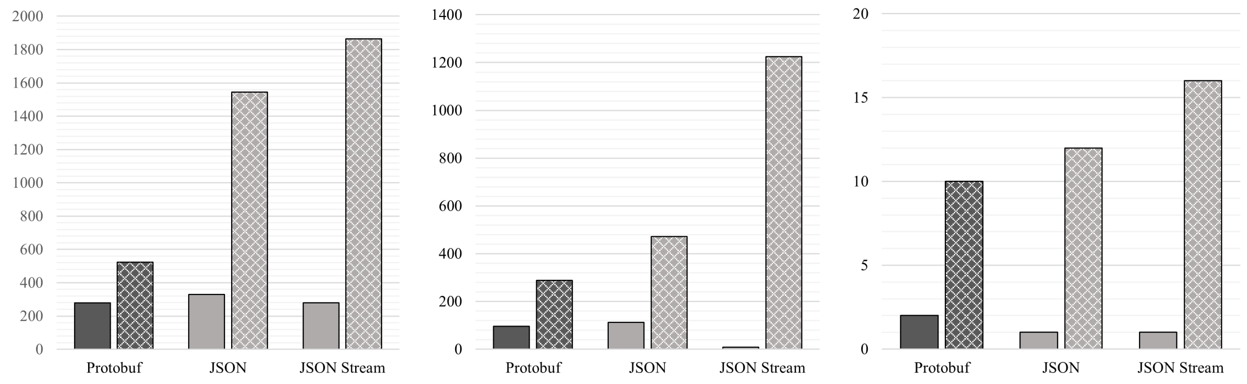
```go
package benchmarks_test

import (
  "encoding/json"
  "io/ioutil"
  "testing"
  "github.com/golang/protobuf/proto"
)

var fixtureData = &TestData{
    Message: "Nullam congue sapien eu nunc",
    Data:    []string{"Aenean", "sit", "amet", "tellus", "vel"},
}
var result TestData

func BenchmarkJSON(b *testing.B) {
    b.Run("Serialize", func(b *testing.B) {
      for n := 0; n < b.N; n++ {
        json.Marshal(fixtureData)
      }
    })
    b.Run("SerializeStream", func(b *testing.B) {
      for n := 0; n < b.N; n++ {
        json.NewEncoder(ioutil.Discard).Encode(fixtureData)
      }
    })
    data, _ := json.Marshal(fixtureData)
    b.Run("Deserialize", func(b *testing.B) {
      for n := 0; n < b.N; n++ {
        json.Unmarshal(data, &result)
      }
    })
    b.Run("DeserializeStream", func(b *testing.B) {
      for n := 0; n < b.N; n++ {
        json.NewDecoder(bytes.NewReader(data)).Decode(&result)
      }
    })
}

func BenchmarkPb(b *testing.B) {
    b.Run("Serialize", func(b *testing.B) {
        for n := 0; n < b.N; n++ {
            proto.Marshal(fixtureData)
      }
    })
    data, _ := proto.Marshal(fixtureData)
    b.Run("Deserialize", func(b *testing.B) {
      for n := 0; n < b.N; n++ {
          proto.Unmarshal(data, &result)
      }
    })
}
```



(A) µs/operation (B) kB RAM allocated/operation (C) allocations/operation

FIGURE 4.5. Plots of serialization/deserialization benchmarks: gRPC is in dark gray and JSON is in light gray, while *serialization* is solid, and *deserialization* is hatched.

4.5.2. Open Source Projects

While there may be many closed-source, commercial products using `protobufs`, surveying and auditing open-source projects provides insight into industry trends and is useful to compare implementation details between organizations. For review I have chosen a third-party `protobuf` compiler *nanopb*, a 'web-scale' SQL ACID database *CockroachDB*, and the open-source base of the most popular web browser in the world, *chromium*.

Nanopb is a Protobuf compiler specifically designed for embedded-C systems. It uses the same proto files as the normal `protoc`, but reading additional model metadata from an *options* file. This allows more granularity in creating interfaces for microcontrollers, where every kB of RAM matters, and statically generating arrays beforehand has measurable performance implications. Since starting in 2011, nanopb has fairly healthy open-source community, with almost 100 contributors and ~140 pull requests. Nanopb has been deployed in Android phones, Garmin watches, TomTom GPS devices, and Cisco Telepresence Servers.

CockroachDB (crdb) is an open-sourced, distributed SQL database built on a transactional and strongly-consistent key-value store. It is wire-compatible with PostgreSQL, and supports strongly-consistent ACID transactions. Scaling horizontally, it can survive disk, machine, rack, and even data center failures with minimal latency disruption and no manual intervention. For cluster communication, it uses proto files for everything between nodes, from simple datum timestamps, to encapsulation of actual SQL requests, response rows and associated telemetry. Actual communication is handled through gRPC, and is muxed with an HTTP server to provide the Web dashboard on the same port that PostgreSQL drivers communicate with. This lowers the requirements for firewalls and corresponding networking. CockroachDB has over 400 contributors on Github and over 20k stars.

The Chromium web-browser is an open-source browser that several browsers are built on: Google's Chrome, Opera, Amazon's Silk and Microsoft's Edge. While many end-users may not be aware of open-sourced Chromium, it powers every modern browser besides Firefox and Safari, amounting to ~38 market share worldwide in 2020. It also is the main interface in Chromium OS, a Linux derivative upstream of Chrome OS. Protobufs are used in Chromium to maintain state in the browser, and hold configuration data for policies. This includes everything from how bookmark history status is polled, to syncing of user profiles.

4.5.3. Edge Node

An example of a service needed on an edge node that `protobufs` would be useful for is a cluster heartbeat. In simple, tightly clusters, heartbeats are normally just a 'check-in' from established members of the cluster. In more loosely defined clusters, where an unreliable network, or constrained resources force members to join/leave ad-hoc, heartbeats contain more information about the cluster, for example in Redis each ping/pong packet carries important pieces of information about the state of the cluster from the point of view of the sender node [129]:

- hash slots distribution
- configuration information
- additional data about other trusted nodes

From the opposite perspective, if there is a central dispatcher in the cluster, e.g. a job queue, then heartbeat information will need to include more endogenous node information for the dispatcher to make decisions. This information could include the current processing load, state of resources and other telemetry data.

4.5.4. Resource Modeling/Service Discovery

As CAVs have many different sensors with a wide array of data characteristics, there should be a catalog of available resources that the CAV can provide. This catalog should include all the pertinent data about the sensor/resource, along with what service on the CAV is advertising, and the format of the data. This can all be accomplished with a CAV service announcing resources through standard `protobufs` definitions. Through automating the process by which resources are discovered, a larger scale of services can be accessed and managed.

4.6. CAV-Specific Implementation

Since the proto files would be used by many teams, the independent ownership of proto files should be established. These repositories should be owned and maintained by a standards body like ISO or SAE for example, preferably by the organization holding the standard. These proto files should be hosted openly, i.e. not behind paywalls or subscriber licensing agreements. Manufacturer-specific protobufs should be hosted in their respective open-source repositories for consumption by 2nd and 3rd parties.

CHAPTER 5

DATA PROTOCOL

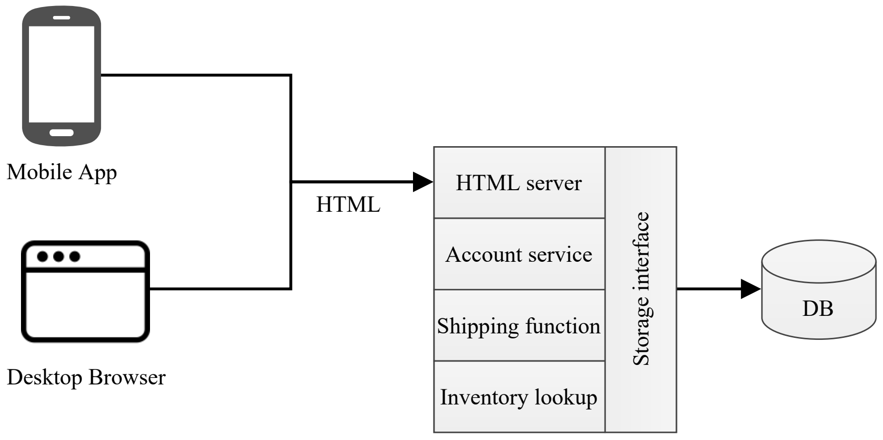
Gone are the days of shared memory programming, single-server services and localized resources. Some of today's modern application stacks involve hundreds of services powered by thousands of servers, spread not only across data centers, but across continents. A large, complex system cannot change direction quickly though, and therefore cannot adapt to market pressures in a timely manner. There are several solutions to this problem, and while many companies still run a complex, monolithic, "do all" architecture, over the past decade the most innovative companies have transitioned to microservices. A simple visual comparison of a monolithic stack versus a microservice-based application is given in Figure 5.1.

In contrast to a monolithic service, a microservice contains just enough functionality to do one thing (or service). Some defining characteristics are:

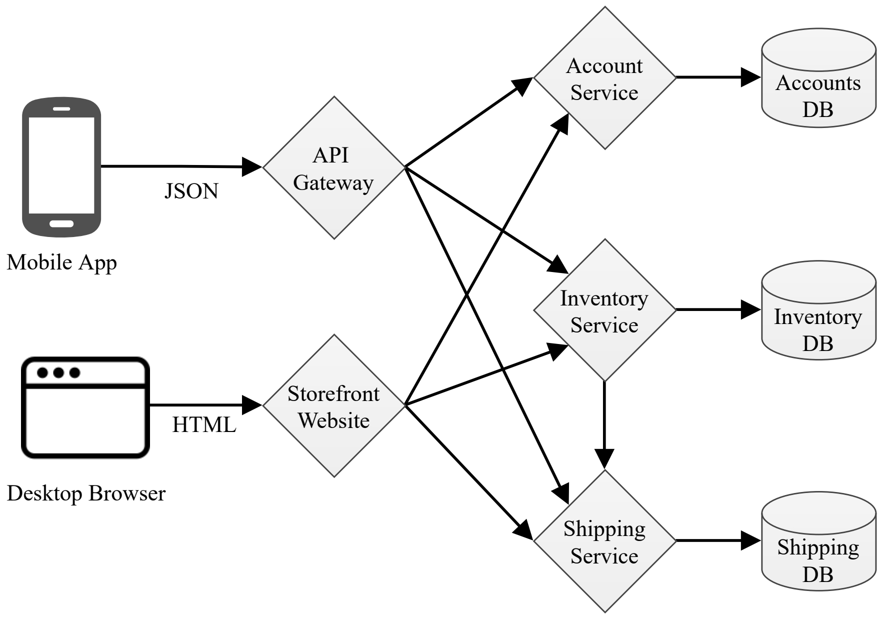
- Highly maintainable and testable
- Loosely coupled
- Independently deployable
- Organized around capabilities
- Owned by a small team

Some definitions go beyond these attributes and add 'idempotence' as a microservice trait. Idempotence can be thought of as an extension of deterministic behavior (same input results in same output each time), i.e., clients calling the same service repeatedly will produce the same result. In other words, making multiple identical requests has the same effect as making a single request. Note that while idempotent operations produce the same result on the server (no side effects), the response itself may not be the same, e.g. a resource's state may change between requests.

Architecturally, microservices can also reduce code silos and reinforce DRY (Don't Repeat Yourself) development. This can be accomplished in monolithic applications, but it is a reinforced pattern in microservices. Additionally microservices reinforce network



(A) Monolithic style.

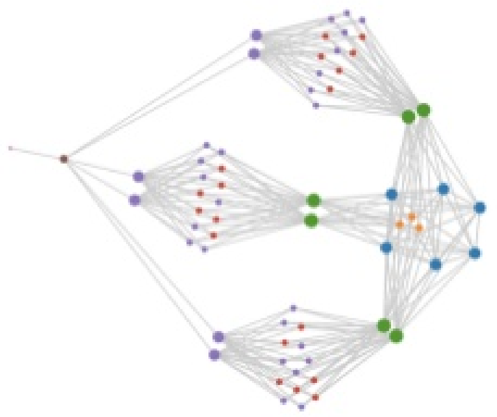


(B) Microservice-based architecture.

FIGURE 5.1. Architectural illustrating comparing a monolithic application stack versus a microservice-based application stack of an example E-Commerce website with two kinds of consumers and persistent backing data storage.

abstraction, since a microservice doesn't need to know where (local, remote, etc.) it's running as all communication happens over its API. Microservices offer technical benefits in addition to architectural ones. Services that are 'hot', i.e. are resource-intensive or time-intensive, can be scaled individually. Similarly, services in a 'critical' path can be scaled for resiliency and redundancy. Since microservices can be deployed independently from one another, they can be scaled without affecting other services. This independent deployment capability also leads to more advance deployments techniques like Blue/Green [72], which will be covered in more detail in Chapter 6, and independent version evolution. This independent version evolution allows microservices to move away from the “lock step” feature deployment of an entire application.

A microservice API can be considered a contract between the service and its clients. The only way the service can communicate is through this API, and therefore the only way to change or maintain state with each other is through this contract. This emphasis on communication is demonstrated in another way: by the difference in connections between Figure 5.1a and 5.1b. For non-toy examples, these connections can explode in number as services interconnect during their lifetime. For instance, notice the differences between the simplified Netfix architecture circa 2012 in Figure 5.2a and the actual arch in Figure 5.2b. This is colloquially referred to as a ‘Death Star’ diagram, and has presented new problems, both for the industry [98] (interdependency anti-patterns) and for academia [75] (QoS/utilization).


(A) Simplified architecture.

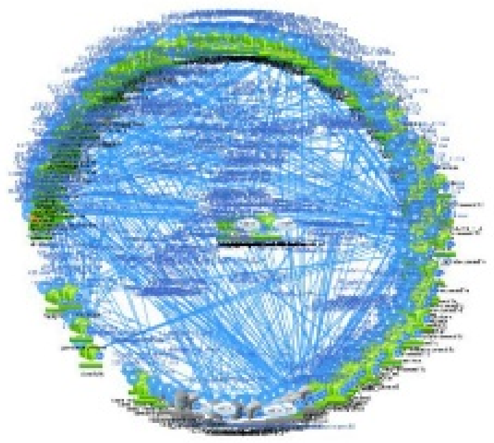

(B) Actual architecture.

FIGURE 5.2. Netflix microservice architecture generated by Adrian Cockcroft using SPIGO [50] [49].

The future of CAV is microservices. Not only do all the ‘normal’ benefits of microservices apply, but the layering of CAV communication (between the cloud, edge nods, and inter-CAV), is especially well suited for services that aren’t tightly coupled within the same network. In other words, since there is already a hard domain separation between CAV, edge and cloud, services already must enforce communication contracts to produce and consume data. For microservices to effectively communicate across these barriers they must agree on a standard protocol for transmission. Within distributed systems this is communication is generically a remote procedure call (RPC), and refers to literally calling a procedure in a remote address space on a different computer. Researching all the modern RPC systems in an exhaustive manner is counter-productive, as many of the same companies that have migrated to microservices independently chose a definitive RPC system, namely `gRPC`.

5.1. Details

Like, protocol buffers, `gRPC` was developed internally at Google and released in 2015 [142]. Originally named *Stubby*, `gRPC` was designed to power "massively distributed systems that span data centers, as well as power mobile apps, real-time communications, IoT devices and APIs." While released at the same time as version 3 of protocol buffers, `gRPC` is not tied specifically to protocol buffers, and different data formats can be used (JSON, Protobuf, Thrift, XML) at varying levels of maturity.

`gRPC` evolved out of lessons learned running web-scale services, resulting in the principles defined in Table 5.1 [selected]. The 'fallacies of ignoring the network' referred to in the principle "Services not Objects, Messages not References" deserve more explicit enumeration, which arose from seminal distributed computing work at Sun Microsystems in the 1990's:

(1) The network is reliable.
(2) Latency is zero.
(3) Bandwidth is infinite.
(4) The network is secure.
(5) Topology doesn't change.
(6) There is one administrator.
(7) Transport cost is zero.
(8) The network is homogeneous.

Table 5.1. gRPC Principles [selected] [191].

Principles & Requirements	Description/Explanation
Services not Objects, Messages not References	Promote the microservices design philosophy of coarse-grained message exchange between systems while avoiding the pitfalls of distributed objects [73] and the fallacies of ignoring the network [189].
Coverage & Simplicity	The stack should be available on every popular development platform and easy for someone to build for their platform of choice. It should be viable on CPU and memory-limited devices.
Free & Open	Make the fundamental features free for all to use. Release all artifacts as open-source efforts with licensing that should facilitate and not impede adoption.
Interoperability & Reach	The wire protocol must be capable of surviving traversal over common internet infrastructure.
General Purpose & Performant	The stack should be applicable to a broad class of use-cases while sacrificing little in performance when compared to a use-case specific stack.
Layered	Key facets of the stack must be able to evolve independently. A revision to the wire-format should not disrupt application layer bindings.
Payload Agnostic	Different services need to use different message types and encodings.
Streaming	Storage systems rely on streaming and flow-control to express large data-sets. Other services, like voice-to-text or stock-tickers, rely on streaming to represent temporally related message sequences.
Pluggable	A wire protocol is only part of a functioning API infrastructure. Large distributed systems need security, health-checking, load-balancing and failover, monitoring, tracing, logging, etc.
Extensions as APIs	Extensions that require collaboration among services should favor using APIs rather than protocol extensions where possible.
Metadata Exchange	Common cross-cutting concerns like authentication or tracing rely on the exchange of data that is not part of the declared interface of a service. Deployments rely on their ability to evolve these features at a different rate to the individual APIs exposed by services.
Standardized Status Codes	Clients typically respond to errors returned by API calls in a limited number of ways. The status code namespace should be constrained to make these error handling decisions clearer.

gRPC services are defined in a service files, by default using the same Interface Definition Language (IDL) as protocol buffers. This service definition specifies the methods that can be called remotely, along with their parameters and return types. Continuing the 'Person' example, an example 'Contacts' service is given in Listing 5.1. There are four kinds of services that can be defined:

Unary: a client sends a single request to the server and gets a single response back.

Server streaming: a client sends a request to the server and gets a stream to read a sequence of messages back. The client reads from the returned stream until there are no more messages.

Client streaming: the client writes a sequence of messages and sends them to the server, again using a provided stream. Once the client has finished writing the messages, it waits for the server to read them and return its response.

Bidirectional streaming: both client and server send a sequence of messages using a read-write stream. The two streams operate independently, so clients and servers can read and write in whatever order they prefer.

Message ordering is guaranteed within an individual RPC call, and in bidirectional streaming the order of messages in each stream is also preserved. Additionally, these services can be defined as either asynchronous, or synchronous, target language permitting.

Using the `gRPC` plugin for `protoc` generates both client-side and server-side code from the service definition. Clients the can call these APIs on the client side while developers can implement the corresponding API on the server side. For the server side, the developer implements the methods declared by the service and runs a `gRPC` server to handle client calls. The infrastructure stack decodes incoming requests, executes service methods, and encodes service responses. On the client side, the client has a local object known as stub (for some languages, the preferred term is client) that implements the same methods as the service. The client can then just call those methods on the local object, wrapping the parameters for the call in the appropriate message type.

Each of the ten default supported languages has multiple layers, allowing users to customize what pieces they want in your application. There are three main stacks in `gRPC`: C-core, Go, and Java. Most of the languages supported are just thin wrappers on top of the C-based gRPC core library, show in Figure 5.3. For example, a Python application calls into the generated Python stubs. These calls pass through interceptors, and into the wrapping library where the calls are translated into C calls. The gRPC C-core will encode the RPC as HTTP/2, optionally encrypt the data with TLS, and then write it to the network. By layering the framework, different pieces can be swapped out depending on application requirements. For example, if C++ is needed for performance, it can use an In-Process transport, saving calls from having to go all the way down to the OS network layer. Another example is enabling experimental projects, like QUIC protocol (see 5.2.1.4). For each of the wrapped languages, the default HTTP/2 implementation is built into the C-core library, but it is possible to exchange it, for instance replacing HTTP/2 with Cronet (the Chrome networking library). The structure of the Go stack is much simpler since it only supports one language, as shown in Figure 5.4. The flow from the top of the stack to the bottom is more linear, and unlike wrapped languages, gRPC Go can use either its own HTTP/2 implementation, or the standard Go library `net/http` package. While the Java stack shown in Figure 5.5 also only supports one language, the Java stack supports multiple configurations for underlying layers. Java supports HTTP/2, QUIC, and In Process like the C-core. Unlike the C-Core though, applications commonly can bypass the generated stubs and interceptors, and speak directly to the Java Core library. Each structure is slightly different based on the needs of each language implementation of gRPC. Also unlike wrapped languages, gRPC Java separates the HTTP/2 implementation into pluggable libraries, such as Java's Netty, OkHttp, or Cronet.

5.2. Benefits

The benefits of `gRPC` broadly fall into three points:

(1) Leverage server-side streaming from underlying HTTP/2.
(2) Efficiency gains during serialization and deserialization by using compact encodings.
(3) Ability to auto-generate and publish services SDKs and documentation.

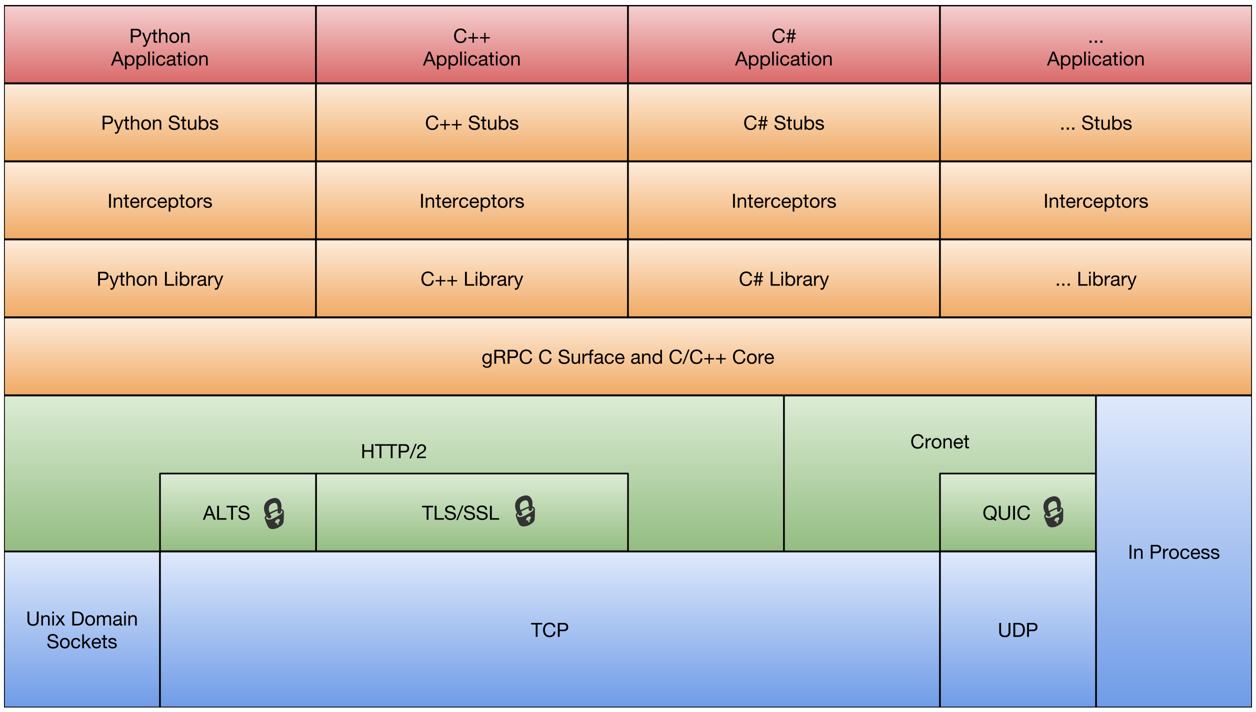


FIGURE 5.3. gRPC Wrapped Languages stack (C-core) [144].

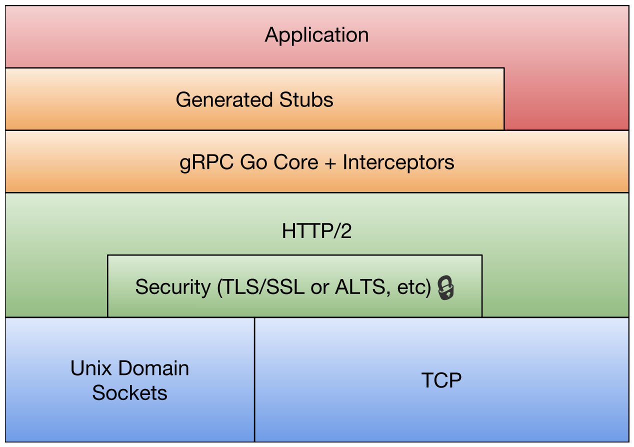


FIGURE 5.4. gRPC Go stack [144].

5.2.1. HTTP/2

gRPC largely follows HTTP semantics over HTTP/2 but explicitly allows for full-duplex streaming. Typical REST conventions are avoided, with static paths used instead for performance reasons during call dispatch. This for performance reasons due to parsing call parameters from paths, query parameters and payload body adds latency and complexity. Also included is a formalized set of errors that are more directly applicable to API use cases

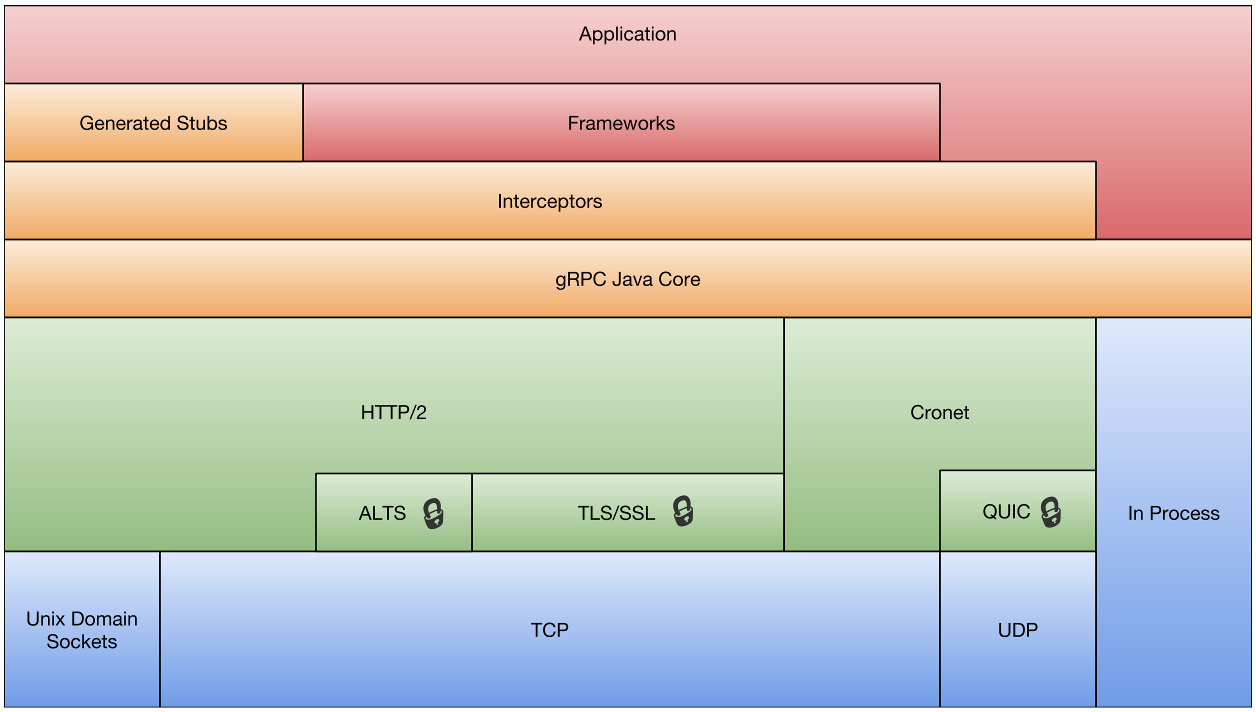


FIGURE 5.5. gRPC Java stack [144].

than the HTTP status codes.

The benefits of using HTTP as an underlying RPC layer:

Software:

- Libraries already designed to handle HTTP (standard codes, error handling, etc.)
- Extensive server-side optimization (caching, resource tagging, etc.)
- Wide client support
- Replaceable with newer standards (see Section 5.2.1.4)

Hardware:

- Specialized hardware availability to offload HTTP cryptography (TLS accelerators)
- Network equipment is already designed/optimized for HTTP traffic, e.g. firewalls/Intrusion Detection Systems/Layer 7 load balancers

HTTP/2 is an IETF standard published as RFC 7540 in May 2015 [14]. It supersececed HTTP/1.1, which had been the HTTP standard since 1997. HTTP/2 is based almost entirely on an early Google experimental protocol called SPDY, with the only real difference being replacment of the dynamic header compression with a fixed scheme. Compared to

Listing 5.1. Example IDL file `contacts.proto` for a 'Contacts' `gRPC` service.

```
service Contacts {
  // Adds a Person to the Contacts
  rpc CreateContact(CreateContactRequest) returns (Empty) {
    options (google.api.http) = {
      // Create maps to HTTP POST.
      post: "/v1/contacts",
      body: "person",
    }
  }
  // Get returns person for ID
  rpc GetContact(Person) returns (Person) {
    // Get maps to HTTP GET. No body.
    options (google.api.http) = {
      // 'id' field mapped from Person definition
      get: "/v1/contacts/{id}"
    }
  }
  // Searches Contacts for People matching 'name'
  rpc ListContacts(Name) returns (People) {
    // Get maps to HTTP GET. No body.
    options (google.api.http) = {
      // URL query params are automatically mapped, e.g. ?name=$name
      get: "/v1/contacts"
    }
  }
}

message CreateContactRequest {
  string person_id = 3;
  // The contact resource to create.
  // The field name should match the Noun in the method name.
  Person person = 2;
}

message People { // an array of 0 or more Persons
  repeated Person people = 1;
}

message Name {
  string name = 1;
}

message Empty {} // empty message to represent an empty response
```

HTTP/1.x, HTTP/2 provides three major enhancements: encoding, multiplexing, and flow control [91].

5.2.1.1. Encoding

Compared to the newline-delimited, plain text HTTP/1.x protocol, HTTP/2 uses binary encoding for frames. The binary encoding is much more compact, efficient for processing, and easier to implement correctly. In HTTP/2, headers are compressed using fixed Huffman compression scheme to reduce the size of HTTP headers. When HTTP/1.x was developed, headers between clients and servers was minuscule, but with the advent of cookies and session tokens, header sizes has been pushed up the limit of web servers, up to 16 KB. Headers are often repeated across requests and responses, and HTTP/2 leverages static codes to compress literals. In addition to the compression, the client and server also maintain a list of frequently seen fields and their compressed values. So when these fields are repeated they simply include the reference to the compression values, instead of passing the actual value across the wire.

5.2.1.2. Multiplexing

HTTP/1 was initially a single request and response flow. Client had to wait for the response before issuing the next request. HTTP/1.1 introduced pipelining, whereas a client could send multiple requests without waiting for the response. However, the server is still required to send the responses in the order of incoming requests (FIFO queue). So HTTP/1.1 suffered from requests getting blocked on high latency requests in the front (referred to as Head-of-line blocking). HTTP/2 introduces fully asynchronous multiplexing of requests by introducing the concept of streams. A single underlying TCP connection can contain streams initiated by both clients and servers. Compared to HTTP/1.x, in HTTP/2 even the server can initiate a stream for transferring data which it anticipates will be required by the client. For example when client request a web page from an HTTP/2 web server, in addition to sending the main HTML content, the server can initiate a separate stream to transfer images or other resources that it knows the client will require to render the full page (referred to as HTTP Push). Figure 5.6 illustrates a single multiplexed TCP connection between a client and server, over four streams and three TCP packets. While HTTP/2 streams are mostly independent, there are provisions to establish priority and dependencies across streams as

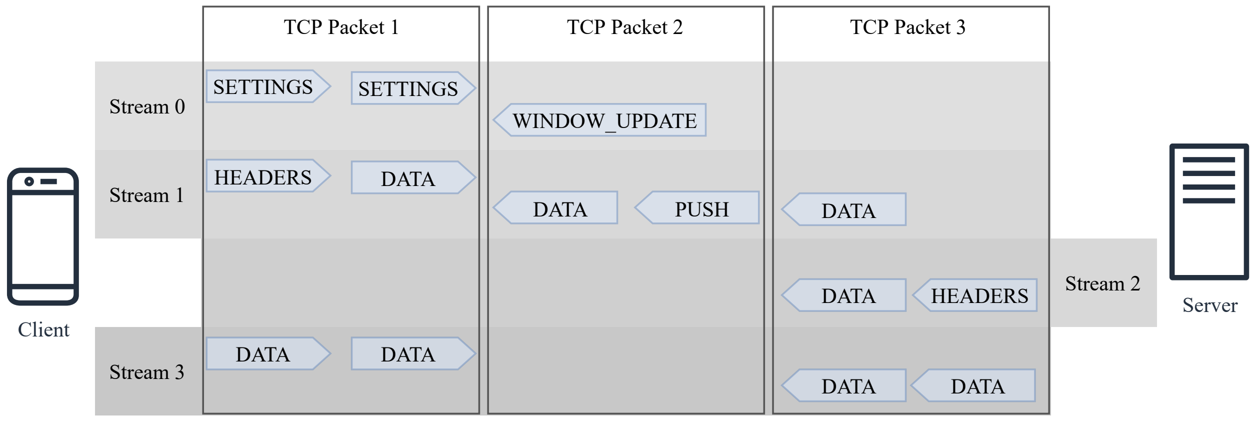


FIGURE 5.6. HTTP/2 multiplexing four streams and using 'server push' between a client and server over one TCP connection containing three packets. *Stream 0* is reserved for control messages/frames, while *Stream 1* & *Stream 3* are initiated by the client. *Stream 2* is initiated by the server, pushing headers and data in *Packet 3*.

well.

5.2.1.3. Flow Control

Successful multiplexing requires flow control in place to avoid contention for underlying TCP resources and avoid destructive behavior across streams. Rather than enforce a particular control flow algorithm, HTTP/2 provides the building blocks for client and servers to implement flow control suitable for their specific situation and context. Application-layer (OSI model layer 7) flow control allows the browser to fetch only a part of a particular resource, put the fetch on hold by reducing the stream flow control window down to zero, and then resume it later, e.g., fetch a preview image, display it and then allow other high priority fetches to proceed, then resume the fetch once more critical resources have finished loading. Note than during the RFC process, allowing a layer 7 protocol to duplicate flow control that should be handled by TCP (layer 4) was a contentious topic, and was one of the main objections to the HTTP/2 standard as is.

5.2.1.4. HTTP/3

HTTP/2 will soon be superseceded by HTTP/3, addressing some flaws that were found in HTTP/2 during the RFC process [17]. Work on switching out HTTP/2 for HTTP/3 in `gRPC` has already been started, with no impact to users. HTTP/3 uses QUIC, a transport layer network protocol also developed initially by Google. Compared to the version jump from HTTP/1 to HTTP/2, replacing TCP with QUIC was relatively trivial. QUIC differs from TCP in where user space congestion control is used over the User Datagram Protocol (UDP). The switch to QUIC aims to fix a major problem of HTTP/2 called "head-of-line blocking": because the parallel nature of HTTP/2's multiplexing is not visible to TCP's loss recovery mechanisms, a lost or reordered packet causes all active transactions to experience a stall regardless of whether that transaction was impacted by the lost packet. Because QUIC provides native multiplexing, lost packets only impact the streams where data has been lost. QUIC also introduces a low-latency initial connection establishment, by reducing session startup time. Zero roundtrip time connection resumption (0-RTT) is also supported, which means that subsequent connections can start up much faster by eliminating the TLS acknowledgement from the server when setting up the connection. Since there is no need to renegotiate a connection with 0-RTT, connections can migrate and are resilient to NAT rebinding, i.e. clients on HTTP/3 can switch between WiFi and mobile networks without penalty. All these improvements will have meaningful impacts in imperfect, high-latency network environments like WiFI and mobile.

5.2.2. Compact Encodings

While different encoding formats are supported, as stated previously the default encoding for `gRPC` is protocol buffers. For reference, the benefits of protocol buffers are covered in Chapter 4. Besides the technical reasons for using protocol buffers with `gRPC`, keeping object interface definitions adjacent to the services that access/mutate them lessens developer cognitive load and simplifies CI/CD pipelines.

5.2.3. SDK Auto-Generation

Committing to `gRPC` as the PRC framework means moving from documenting APIs for developer consumption to providing SDKs to customer integration into their own products/services/adapter By releasing SDKs instead of APIs, developers can copy-paste example code written in their language into their adapters. Since these SDKs are auto-generated from the service definition files, this means the service owner does not need to manually update API contracts, and because they are abstracted, there is no need to maintain SDKs for multiple languages. Additionally, these changes are in locked between all SDKs, and avoids the issue of documentation/APIs drifting out of spec with the actual service definitions. This is especially important with Agile delivery, where deployment of new services versions could be overlap customer's development windows.

Along with the SDKs, OpenAPI (formerly Swagger) schemas can be automatically generated from service files using `protoc` during the compilation phase. OpenAPI has become the standard specification for machine-readable interface files for describing, producing, consuming, and visualizing RESTful web services, with several innovative tools created for API exploration [126], [250]. Building beyond OpenAPI, API management products like Apigee integrate directly to enforce throttling and security policies, collect telemetry and analytics data, and even monetization of commercial endpoints.

5.3. Concerns

The same concerns that I raised with `protobufs` in Section 4.3 also apply to `gRPC`. There are also concerns with legacy applications that cannot take advantage of `gRPC`, or with a specific languages that cannot bind to `gRPC`. This concern is lessened with the deployment of a JSON to `gRPC` gateway. The *gRPC-Gateway* is part of the gRPC ecosystem and is a plugin for `protoc`. By adding an *option* tag with an endpoint URL to an RPC method within the definition file, the *gRPC-Gateway* provides the API in both a `gRPC` and RESTful style at the same time. An overview of this gateway is shown in Figure 3.1.

Debugging, along with API exploration, is also a concern. Whereas a developer's normal tools used for APIs, like command-line-based *curl* or GUI-based *Postman*, cannot call `gRPC` endpoints. For this, several new tools like *gRPC-swagger* can take advantage of service reflection, and can be used to list and call gRPC methods using *swagger-ui* conveniently. Since it's based on reflection, the reflection feature only needs to be enabled when starting the service and there is no need to modify encoding files or related code implementations.

`gRPC` is a Cloud Native Computing Foundation (CNCF) project, which means that no single company can steer the direction of development. Although in practice, like most open-source projects, development is driven by the company devoting the most developers to producing/merging features.

5.4. Comparisons

In the following sections I will briefly describe competing protocol standards, and compare them against `gRPC`.

5.4.1. HTTP/JSON

The history of HTTP, being closely tied to the internet, is long, and while worthy of an entire chapter or book, instead of detailing the early years, I will jump ahead to the advent of REST and the rise of JSON-based APIs.

5.4.1.1. REST

Roy Fielding's seminal 2000 doctoral dissertation on "Representational state transfer" [69] shaped the next two decades of World Wide Web development, from influencing the HTTP/1.1 and URI standards to framing the transition to API-driven web development. The REST architectural style today is regarded as essentially for modern web development, but REST has also "become an industry buzzword: frequently abused to suit a particular argument, confused with the general notion of using HTTP, and denigrated for not being more like a programming methodology or implementation framework" [68]. Unlike SOAP, REST isn't an actual defined standard, but rather a set of restrictions on how the HTTP protocol is used and how object (resource) state is represented. If an API follows REST guidelines, it's termed 'RESTful'. The core concepts of REST include stateless resource transfer between client and server, and uniform interfaces. REST and RESTful principles are a deep subject, with over thirty books with "REST" in the title published by O'Reilly & Associates alone, and according to Google Scholar Fielding's dissertation itself has been cited over 6,000 times. Although the REST architectural style doesn't wholly pertain to

RPC, as it was designed to solve issues arising from HTTP client-server interactions, almost all of it's principles apply to `gRPC`:

Uniform interface: An API must expose specific application resources to API consumers.

Client-server: The client and the server function independently. The client will only know the URIs that point to the application's resources, i.e. the server is a blackbox to the client (besides knowledge of the endpoints).

Stateless: The server does not save any data pertaining to the client request. Each HTTP connection should be a 'clean slate' connection, unaware of previous transactions.

Cacheable: Resources exposed by the API need to be cacheable by the client or intermediary servers. This is normally accomplished with the HTTP headers *Expires/Cache-Control* [1] and *Last-Modified/ETag* [2].

Layered: The architecture is layered, which allows different components to be maintained on different servers. A client should not ordinarily be aware of whether it is connected directly to the end server or to an intermediary along the way. This allows proxies, traffic management (load balancers/global server load balancers), and authZ/authN layers to function seamlessly in a client-server session.

Code-on-Demand (optional): This allows the client to receive executable code as part of a response from the server. Originally this was intended to spotlight Java applets executing in the browser, but is largely ignored today.

To solve some of the limitations in RESTful APIs, Facebook created GraphQL, which can be thought of as a query language (like SQL) for APIs. Its query language is based on JSON, and includes the ability to join several 'resources' together into a single response. Individual attributes can also be selected for a response, and like SQL, constraints can be placed on data returned. In other words, clients calling GraphQL endpoints can specify exactly the data they needs in a query and the structure of the response follows precisely the nested structure defined in the query. This eliminates the REST problems of 'overfetching' (return-

[1] "strong caching headers"

[2] "weak caching headers"

ing extraneous data you don't need), and 'underfetching' (requiring additional requests for nested data, i.e. the *n+1* request problem. In order for `gRPC` to support GraphQL clients, the `protoc` plugin *grpc-graphql-gateway* can produce GraphQL schema for consumption by clients.

5.4.1.2. AJAX and JSON

The history of JSON actually starts just a year later, in April of 2001, when Douglas Crockford and Chip Morningstar sent the first HTML-wrapped JSON message [54]. Continuing the long history of other Silicon Valley inventors, this message was sent while they were hacking in their garage. What Crockford and Morningstar were trying to accomplish was passing data to a web page after the initial page load. Browser support was not good for what they were attempting, and they had not found a way to that would properly work across all the browsers they were targeting. This was at a time when Microsoft's Internet Explorer browser provided primordial support through the now defunct ActiveX framework. This did not work on the competing Netscape Navigator browser, so they decided to abuse an HTML tag element to try and support both platforms. The first JSON message ever sent is shown in Listing 5.2. Crockford himself admits that he did not "invent" JSON, instead that he "just discovered it, and gave it a specification and a little website". Oddly enough, they first tried to name it "JSML", for **J**ava**S**cript **M**arkup **L**anguage, but quickly found that it conflicted with **J**ava **S**peech **M**arkup **L**anguage. In the end they decided to go with **J**ava**S**cript **O**bject **N**otation.

Listing 5.2. The "first" JSON message ever sent. This was to solve the problem of dynamic loading of data after initial browser page (HTML) loading [54]. Note there is a syntax error on line 4, as `do` is a reserved JavaScript keyword.

```
1 <html><head><script>
2     document.domain = 'fudco';
3     parent.session.receive(
4         { to: "session", do: "test", text: "Hello world" }
5     )
6 </script></head></html>
```

How JSON became the standard for web APIs is an equally *grassroots* Silicon Valley story. In 2005, Jesse Garrett coined the term “Ajax” in a blog post about asynchronous data exchange. He stressed that AJAX wasn’t any one new technology he invented, but instead “several technologies … coming together in powerful new ways” [76]. AJAX stands for **A**synchronous **J**avaScript and **X**ML, but in a followup *Q&A* post he explained that JSON was an entirely acceptable alternative to XML. Writing that “XML is the most fully-developed means of getting data in and out of an AJAX client, but there’s no reason you couldn’t accomplish the same effects using a technology like JavaScript Object Notation or any similar means of structuring data.” [76]. Over time the ‘X’ in AJAX languished as developers found JSON easier to work with, both natively through JavaScript, and in stateful changes of the HTML document object model. In the resulting years, the paradigm shift to frontend/backend style development pushed API development to programmers who only had experience with JavaScript applications and corresponding web technologies. This made JSON an obvious choice for backend APIs that needed to send data to a frontend UI, or even other backend APIs. This was exacerbated by the need for “Web 2.0” websites to become as performant as possible due to industry pressures. With the need to eek out performance that could exist anywhere between the browser and server (JavaScript engine, compression libraries, language constructs, etc.) and XML’s parsing speed comparatively lagging [158], JSON also beat XML on a performance level.

Although JSON has become widespread in APIs due to it’s roots in web applications and developers familiarity with such, it’s not the best fit for machine-to-machine communication. While I have already stated my issues with JSON for this purpose in Section 4.4.4, I additionally think that the *gRPC-JSON gateway* resolves many of the issues in abandoning JSON as a ‘first-class’ format. Namely the secondary features that have organically grown around JSON like API exploration through Swagger, API management with Apigee, or 3rd-party tooling like Postman/Insomnia.

5.4.2. CoAP

A recently standardized protocol that has been gaining support among the industry is Constrained Application Protocol, or CoAP [20]. Standardized in RFC 7252 [197], CoAP is designed for communication of internet-of-things (IoT) devices in a 'lossy'/'noisy' network environment. Published by the Constrained RESTful Environments (CoRE) Parameters Group, CoAP is based on REST principles and is designed to easily translate to HTTP for integration with web services. This means that CoAP is a replacement for HTTP at the application layer by extending HTTP semantics and even uses a superset of HTTP verbs and response codes. The protocol uses Datagram Transport Layer Security (DTLS) for encryption of network traffic. DTLS is based on TLS, and operates at the transport layer with UDP, preventing eavesdropping, tampering, or message forgery. CoAP defines four security models based around DTLS :

NoSec: DTLS is disabled, for testing and debugging.

PreSharedKey: DTLS is enabled and there is a list of pre-shared keys.

RawPublicKey: DTLS is enabled and the device uses an asymmetric key pair without a certificate, which is validated out of band.

Certificate: DTLS is enabled and the device uses X.509 certificates for validation.

CoAp has recently come into prominence with the introduction of the Thread protocol [216]. Thread is an IPv6-based, low-power mesh networking technology for IoT products. Using 6LoWPAN (IEEE 802.15.4), Thread is focused on home automation and generic IoT communication in a mesh environment. It is promoted through the "Connected Home over IP" a working group within the Zigbee Alliance, formed by Amazon, Apple, and Google to create a "new royalty-free connectivity standard to increase compatibility among smart home products, with security as a fundamental design tenet" [255].

Unfortunately CoAP and Thread are too focused on home automation to be useful for CAV purposes, where low-latency and streaming communication is a critical requirement. This niche in home automation may expand in the future, and replacing HTTP in the process. Of note, many production implementations of CoAP use `protobufs` as the data format for exchange between services, using `nanopb` [4], an ANSI C implementation of protocol buffers targeted at 32 bit microcontrollers and other memory restricted systems (<10 kB ROM/<1 kB RAM).

5.4.3. Thrift RPC

Thrift was discussed in Section 4.4.5, and Thrift itself is more of an RPC framework than a data serialization one. As stated earlier, there are two versions of Thrift, Apache Thrift and Facebook's own open-sourced fork `fbthrift`. The 'Contacts' service is shown in Listing 5.3. As shown in Figure 4.4, Apache Thrift provides a 'server', 'protocol', and 'transport'. The five transports available are:

- TSimpleFileTransport: This transport writes to a file.
- TFramedTransport: This transport is required when using a non-blocking server (TNonblockingServer). It sends data in frames, where each frame is preceded by length information.
- TMemoryTransport: Uses memory for I/O. The Java implementation uses a simple ByteArrayOutputStream internally.
- TSocket] Uses blocking socket I/O for transport.
- TZlibTransport: Used in conjunction with another transport to provide compression using `zlib`.

Thrift also supports four different servers:

- TSimpleServer: A single-threaded server using standard blocking I/O. Useful for testing.
- TNonblockingServer: A multi-threaded server using non-blocking I/O (Java implementation uses NIO channels). TFramedTransport referenced above must be used with this server.
- TThreadedServer: A multi-threaded server using a thread per connection model and standard blocking I/O.
- TThreadPoolServer: TThreadedServer using a thread pool.

LISTING 5.3. Example IDL file `contacts.thrift` for a 'Contacts' Thrift service.

```
// Namespaces in Thrift are similar to C++ namespaces or Java packages.
// They are a way to organize or isolate code.
// Thrift allows namespaces on a per-language basis:
namespace cpp com.example.contacts
namespace java com.example.contacts
namespace py com.example.contacts
namespace go com.example.contacts
namespace js com.example.contacts

service Contacts {
  /**
   * A method definition looks like C code. It has a return type,
   * arguments, and optionally a list of exceptions that it may throw.
   * Note that argument lists and exception lists are specified using
   * the exact same syntax as field lists in Thrift structs or
   * exception definitions.
   */

    // Structs can also be exceptions
    exception InvalidOperation {
      1: i32 code,
      2: string message
    }

    // save creates or updates a Person record
    // returns an error if person cannot be saved
    void save(1:Person person) throws (1:InvalidOperation error),

    // get returns a single Person given the ID parameter
    // returns an error if person not found
    Person get(1:i32 id) throws (1:InvalidOperation error),

    // search returns a set of Persons matching the given 'name' param
    set<Person> search(1:string name)
}
```

One of the biggest concerns with using Thrift as the RPC framework for CAV applications is the lack of support for streaming data. Also while multiple configurations and options may be viewed as beneficial, this can lead to much confusion over what is 'best' for specific applications. In this regard 'opinionated' frameworks like `gRPC` that default to 'reasonable' choices will perform better for almost all use cases. As for service definitions, `gRPC` provides *interceptors*, which can be used to add common functionalities to multiple endpoints. This makes it trivial to implement health checks, telemetry or authentication shared by master and worker interfaces. Thrift has no equivalent API. Lastly, by abstracting on top of HTTP/2, `gRPC` can use full-duplex bi-directional streams for it's communication channels. This advantage over Thrift also means it can benefit from future advancements in flow control and underlying network protocols to alleviate congestion and QoS pressures.

5.5. gRPC Examples

In this section I will benchmark a few protocol examples and explain the results. All benchmarks were run on a 64-bit workstation with a 16-core Intel®Core™i9-9900K 3.60 GHz CPU and 32 GB RAM. This workstation is meant to represent a high-end Edge server communicating with client nodes over a 'non-lossy' network connection. All testing was done on Linux with a 4.4 kernel.

5.5.1. Heartbeat

Heartbeats are a mainstay of cluster communications. Besides informing other nodes of each other's status, heartbeats are used to form subclusters, failover operations and maintaining quorum. A heartbeat client and server was created to benchmark examples of both gRPC and JSON over HTTP. Three run averages were collected, and each benchmark ran for 3 s. The benchmark results are shown in Figure 5.7 while the heartbeat message itself is detailed in Listing 5.4 and the actual test/benchmark file in Listing 5.5. The benchmark, while isolated from network interference, shows that gRPC running in parallel is almost an order of magnitude faster than JSON over HTTP in parallel. In single request mode, gRPC is almost twice as slow as the equivalent JSON heartbeat. This is likely due to three things:

(1) JSON serialization/deserialization has been heavily optimized in the standard library.
(2) The small wire size of the "heartbeat message" is not worth the marshalling and compression overhead of `protobufs`. This is reinforced in Figure 5.7c where the single request allocations are similar for both gRPC and JSON.
(3) The impact of reflecting on the heartbeat `protobuf` data structure slows down single request performance as that reflected meta-information cannot be used

Listing 5.4. Protocol buffer IDL file `heartbeat.proto` used for benchmarking gRPC and JSON/HTTP remote procedure call.

```
syntax = "proto3";

package messages;

message PingRequest {
  string ping = 1;
  Point location = 2;
  string origin_addr = 3;
  string origin_cluster_id = 4;
  string server_version = 5;
  int32 target_node_id = 6;
  int32 origin_node_id = 7;
}

message PingResponse {
  string pong = 1;
  int32 client_id = 2;
  int64 server_time = 3;
  string server_version = 4;
  int32 server_id = 5;
  string cluster_name = 6;
}

message Point {
  double latitude = 1;
  double longitude = 2;
}

service Heartbeat {
  rpc Ping (PingRequest) returns (PingResponse) {}
}
```

again for the next request. This is shown in Figure 5.7b where the allocation sizes are higher for the gRPC service.

5.5.2. Streaming Resource Transfer

A comparison of a gRPC service with a comparable HTTP service for different payload sizes and connection types is given in Table 5.2. This benchmark uses four different built-in servers, and includes both unary responses and streaming messages.

5.6. CAV-Specific Implementation

Since the technology lifecycle management for `gRPC` services is the same as for `protobufs`, the same development pipelines and repositories can be used. This includes the same gover-

Listing 5.5. Benchmark RPC testing file.

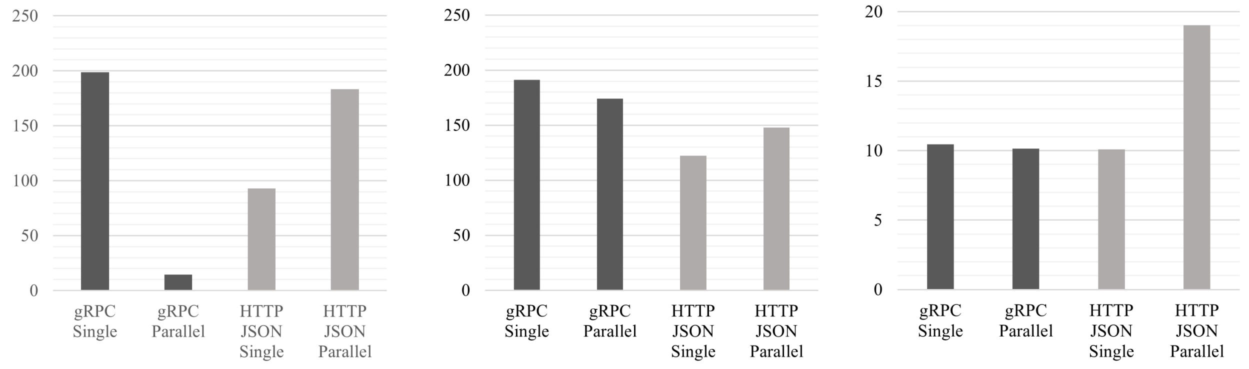
```go
package benchmarks_test

import (
  "testing"
  "github.com/jh125486/benchmarks"
)

func setup(b *testing.B) (*benchmarks.Client, func()) {
  rpcAddr, rpcStop := benchmarks.InitRPCServer("Testing-GRPC")
  httpAddr, httpStop := benchmarks.InitHTTPServer("Testing-HTTP")
  client, err := benchmarks.NewClient(rpcAddr, httpAddr)
  if err != nil {
    b.Fatal(err)
  }
  return client, func() {
    b.StopTimer() // stop benchmark timer before tear down happens
    client.Shutdown() // tear down client TCP connections
    rpcStop() // stop gRPC server (graceful shutdown)
    httpStop() // stop HTTP server
  }
}

func checkPing(b *testing.B, fn benchmarks.Pinger) {
  if _, err := fn(); err != nil {
    b.Fatal(err)
  }
}

func BenchmarkSendHeartBeat(b *testing.B) {
  client, stop := setup(b)
  defer stop()
  for name, fn := range map[string]benchmarks.Pinger{
    "JSON": client.PingJSON,
    "gRPC": client.PingRPC,
  } {
    b.Run(name, func(b *testing.B) {
      for i := 0; i < b.N; i++ {
        checkPing(b, fn)
      }
    })
    b.Run(name+"Parallel", func(b *testing.B) {
      b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
          checkPing(b, fn)
        }
      })
    })
  }
}
```



(A) µs/operation (B) kB RAM allocated/operation (C) allocations/operation

Figure 5.7. Plots of remote client/server heartbeat benchmarks: gRPC is in dark gray and JSON over HTTP is in light gray.

Table 5.2. RPC Benchmarking with gRPC and HTTP.

Benchmark Name	µs/**op**		**speed** MB/s		kB **alloc/op**		**allocs/op**	
Request Count	1K	64K	1K	64K	1K	64K	1K	64K
gRPC `Serve()`^{u}	28	201	74	653	16	459	155	172
gRPC `Serve()`^{s}	16	215	125	610	8	457	27	50
gRPC `ServeHTTP()`^{u}	93	395	22	332	35	565	211	310
gRPC `ServeHTTP()`^{s}	28	346	72	379	9	545	35	160
ProtoBuf RPC^u	20	218	101	601	5	433	16	24
ProtoBuf HTTP/1.1^u	194	368	11	357	60	960	457	612
ProtoBuf HTTP/2^u	108	485	19	271	33	1100	112	212

^{u} Unary message.

^{s} Streaming message.

nance policies dictating ownership and maintenance, as in Section 4.6.

CHAPTER 6

APPLICATION ORCHESTRATION

While solving the issue of data interoperability and service communication is a paramount issue for all levels within the CAV ecosystem, how applications are built and deployed is just as important for longevity and 'Day 2' operations. Day 2 refers to the time between an application being fully deployed and that application being terminated or deprecated. This includes everything from deploying new versions of the application, or patching vulnerabilities and even things like rotating security certificates as shown in Figure 6.1. Obviously by making sensible and forward-thinking choices in 'Day 0' and 'Day 1', 'Day 2' operations will result in measurable impacts to KPIs like application uptime or vulnerability scorecards.

The move from physical, bare-metal servers to hypervisor-based virtualization has shortened 'Day 1' operations, and the arrival of containerization almost removed them entirely, moving those tasks back into 'Day 0'. This movement is closely tied to infrastructure-as-code, and refers to the way of treating infrastructure as any other programmatic interface, e.g. provisioning VMs through domain languages and automatic configuration of network load balancers through service discovery. Several tools are built for infrastructure-as-code, like *Chef*, *Ansible*, *Terraform*, and a favorite among HPC distributions, *CFEngine* [202]. Infrastructure-as-code is also tangentially related to Immutable Server Architecture, the act of infrastructure provisioning and configuration being 100% automated, so that instead of upgrading or patching a live, production server during 'Day 2', the automation should be able to spin up an exact server replica that contains the upgrades and security patches needed. Since it's an independent, exact copy, the patches/updates can be tested out of band to ensure correctness. Once the new immutable infra is live, network load balancers can switch traffic to the new server address and the termination process can begin for the obsolete infrastructure. If two separate sets of infrastructure are maintained, they can be alternately used for new application versions or patches, while automated processes are used to route between them. This is called "Blue-Green" deployments [72] and is an integral part of “Continuous Deployment”.

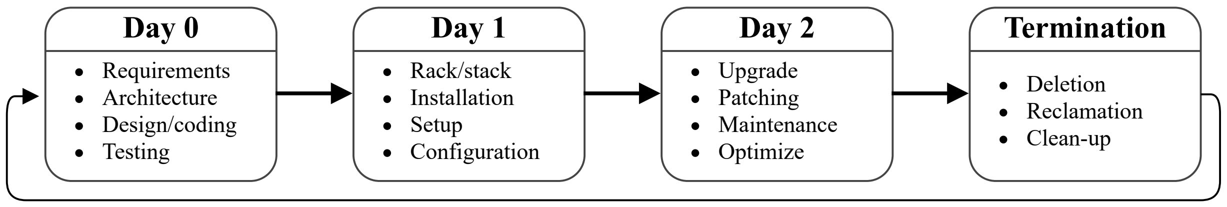


Figure 6.1. Application lifecycle.

6.1. Details

The arrival of containerization has brought more advantages, by abstracting further away from the underlying operating system and libraries. Each container that is repeatable and through the standardization from having dependencies included leads to deterministic behavior during deployment. By decoupling applications from underlying host infrastructure deployment becomes simple in a different operating system or even a different cloud environment. With containers ‘Day 1’ operations can be reduced to a few seconds while the container is starting, to milliseconds with advanced frameworks like AWS’ *Firecracker* [3]. As containers are immutably configured by default, ‘Day 2’ operations are reduced to Blue-Green deployments, further tightening the software development lifecycle loop. For CAV applications, containers provide benefits conferred beyond what applies in data centers and developer laptops, namely the integration of 1st through 3rd party applications. If deployed through containers, these non-OEM applications would be tenant-isolated, and still reap performance benefits without the resource overhead of an entire VM separation.

An industry example of this was the recent U.S. Air Force U-2 spyplane flight in October 2020, which flew four computers on-board as part of a Kubernetes cluster [217]. This airframe houses many sensor packages, from electronic signals, imagery intelligence, including radar and electro-optical, and air samples, and this Kubernetes cluster was used to perform advanced machine learning algorithms on those four individual, flight-certified computers. The next flight included two updates: a logging container that wrote some text along with a timestamp in a file and deployment of “improved automatic target recognition algorithms" in an unspecified test application/sensor. This work was not unique either, as early in 2019 the U.S Air Force deployed three Kubernetes clusters to legacy hardware in an F-16 fighter, detailed in talk at KubeCon 2019 [35]. Of note, NASA flies two planes as the ER-2, with missions ranging from soil nitrate observation, celestial observations and wildfire mapping.

Released in 2014, Kubernetes was designed to deploy and automate microservice-based applications running in containers. It's analogous to an operating system scheduler that decides which processes to run on specific CPUs. In Kubernetes, the 'processes' are containers and the 'CPUs' are host machines. In other words, Kubernetes abstracts application orchestration from single machines to a data center, or even across multiple data centers and regions. Like `gRPC`, Kubernetes (abbreviated and stylized as *k8s*), was originally designed by Google and is now maintained by the Cloud Native Computing Foundation. A deployed Kubernetes cluster is composed of a Control Plane, along with at least one worker Node. Important Kubernetes deployment concepts include:

Control Plane: makes global decisions about the cluster, e.g. scheduling, as well as detecting and responding to cluster events, e.g. starting up a new **Pod** when a deployment's replicas count is unsatisfied.

Node: a worker machine that runs containerized applications and every cluster has at least one worker.

Namespace: a virtual cluster backed by the same physical cluster.

Pod: the smallest deployable units of computing in Kubernetes, a group of one or more containers, with shared storage and network resources. Always co-located, co-scheduled onto the same **Node** and run in the same shared context.

Service: an abstract way to expose an application running on a set of **Pods** as a network service. Kubernetes gives **Pods** their own IP addresses and a single DNS name for a set of **Pods**, and can load-balance across them.

Ingress: manages external access to the **Services** in a cluster and may provide load balancing, SSL termination and name-based virtual hosting.

Resource Quotas: provides constraints that limit aggregate resource (milli-CPUs and GB RAM) consumption per **Namespace**. It can limit the quantity of objects that can be created in a **Namespace** by type, as well as the total amount of compute resources that may be consumed by resources in that **Namespace**.

The Control Plane is composed of the following pieces:

***kube-apiserver*:** the front end for the Kubernetes control plane and designed to scale horizontally, balancing traffic between them.

***etcd*:** a highly-available component of the control plane which contains the overall state of the cluster at any given point of time. Uses *raft* for distributed quorum consensus.

***kube-scheduler*:** watches for newly created Pods with no assigned node, and selects a node for them to run on, based on: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

***kube-controller-manager*:** manages the *controllers*, control loops that watch the state of the cluster, then make or request changes where needed.

***cloud-controller-manager*:** links the cluster into a cloud provider's API, and separates out the components that interact with that cloud platform from components that just interact with the local cluster.

Components of the Control Plane and how they interact with Workers is shown in Figure 6.2. A worker Node is composed of:

***kubelet*:** an agent that runs on each node in the cluster. It makes sure that containers are running in a Pod.

***kube-proxy*:** a network proxy that runs on each node in the cluster, implementing part of the Kubernetes Service concept. *kube-proxy* maintains network rules on nodes and allows network communication to the Pods from network sessions inside or outside of the cluster.

Container Runtime: daemon responsible for running containers. Kubernetes supports several container runtimes: Docker, containerd, CRI-O or anything meeting the

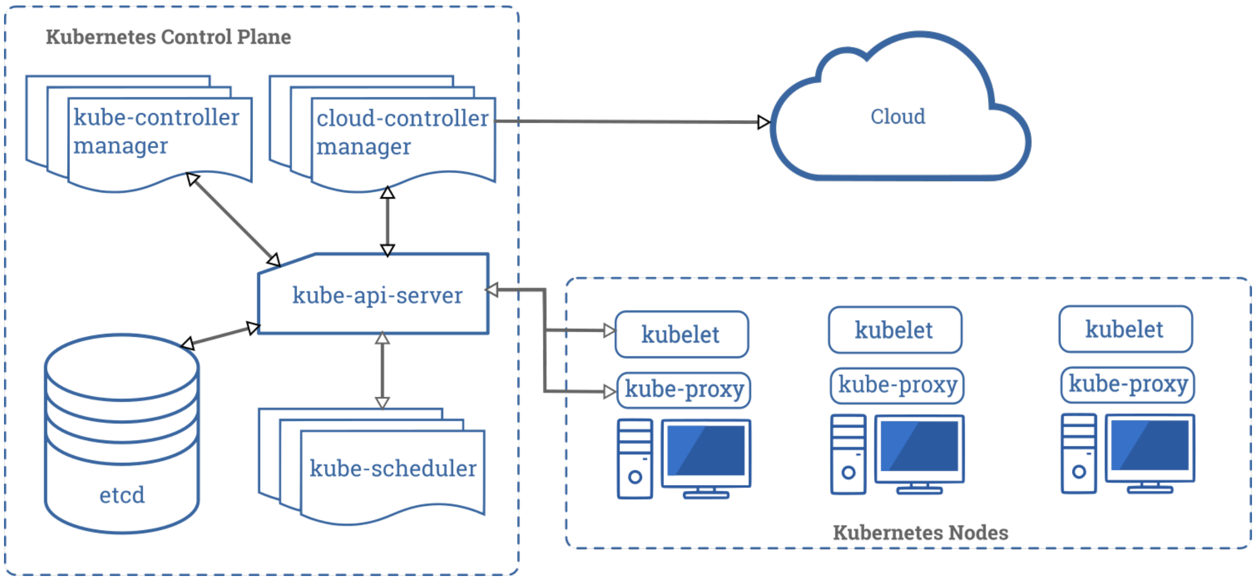


FIGURE 6.2. Components of a Kubernetes cluster.

Kubernetes CRI (Container Runtime Interface).

6.2. Benefits

The pod isolation, network segregation and integrated service discovery make Kubernetes ideal for CAVs that need to deploy applications from multiple-companies or manufacturers. For manufacturers that want to include and deploy 2nd or 3rd party applications, Kubernetes' Namespaces along with Network Policies provide multi-tenancy isolation to guarantee data is not leaked across manufacturers. Resource Quotas ensure that applications that are 'noisy' or experiencing transient load will not affect other applications in the same Node. Even the operator model can be tailored to only give access appropriate to role, so other developers cannot access applications outside their own Namespace. The downside of Kubernetes is the administration overhead and complicated deployment of cluster components. With highly-available services deployed to clouds, Kubernetes clusters may include three-times over replicas for essential clusters services, which is not feasible for single-board computers powering CAVs.

6.2.1. K3s: Kubernetes at the Edge

In 2019, Rancher Labs released *k3s* [11], a fully certified Kubernetes distribution (distro) designed to address the issues that prevent *k8s* from running on resource-constrained or embedded hardware. While still including all the benefits of Kubernetes, *k3s* is packaged as a single, lightweight <40 MB binary, and removes the need for more complex pieces of Kubernetes. This reduces the controller plane memory size to ~300 MB and ~100 MB for the agent. Just like the benefits of moving to container-based virtualization, deploying *k3s* in production allows developers to use the same Kubernetes commands locally to write applications, and testing pipelines can use the same Kubernetes configurations to run hundreds of tests in cloud infrastructure. It also designed to run on both ARM64 and ARMv7 architectures, bridging the gap between developers' hardware, targeted edge devices and cloud infrastructure. Compared to Figure 6.2, an overview of a *k3s* server and agent node is shown in 6.3. In other to reduce the complexity and administrative overhead of a full Kubernetes installation, several components were replaced with lighter-weight equivalents:

Flannel: a very simple L2 overlay network that satisfies the Kubernetes requirements.
CoreDNS: a flexible, extensible DNS server that can serve as the Kubernetes cluster DNS.
Traefik: is a modern HTTP reverse proxy and load balancer.
Klipper Load Balancer: a Service load balancer that uses available host ports.
SQLite3: The storage backend used by default (also support *MySQL*, *Postgres*, and *etcd3*).
Containerd: is a stripped down runtime container daemon.

6.2.2. K3OS: The Kubernetes Operating System

As compact and edge-friendly that *k3s* is, it still needs a full Linux environment underneath it to provide all the operating system services. This underlying operating system must be patched and maintained after initial deployment, just like all other operating systems. Timing operating system updates with Kubernetes updates is especially important since many Kubernetes features are tied to developments in the Linux kernel and related drivers. Shortly after releasing *k3s*, Rancher Labs announced another open-sourced project,

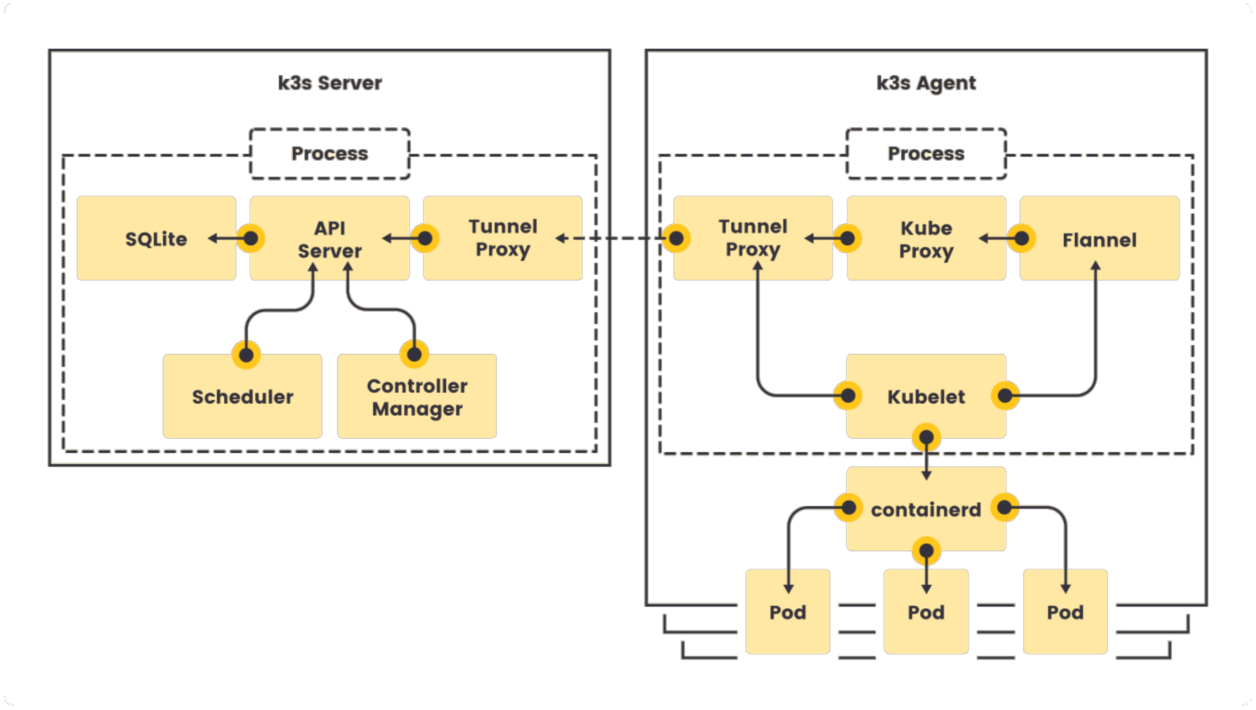


FIGURE 6.3. Components of *k3s* from K3s 'How it Works' [11].

k3OS [10], or "the Kubernetes operating system". *K3OS* is a Linux distro built for the sole purpose of running Kubernetes clusters. It differs from a normal Linux distro in that all services not required to run Kubernetes have been removed, and every system service is run within *k3s* Pods. A diagram of *k3OS* is shown in Figure 6.4. This enables administrators to roll out updates to Edge devices without bringing them offline or disturbing the separate Kubernetes cluster running end-user applications. By not patching the underlying operating system, significant security risks are introduced to the Kubernetes cluster. These unpatched CVEs in the underlying operating system threaten the security of the entire cluster. In the other direction, by patching Linux, and not coordinating with with the Kubernetes installation, operating system upgrades can cause multiple nodes to become unavailable at the same time. Even though Kubernetes is designed to withstand individual node reboots, this can cause the Kubernetes master to lose quorum or disrupt the application workload. Both Linux and Kubernetes are part of the foundational computing platform. Combining a Linux distro with a Kubernetes distro into a Kubernetes operating system simplifies Kubernetes cluster operations and improves system security and reliability, fully eliminating infrastruc-

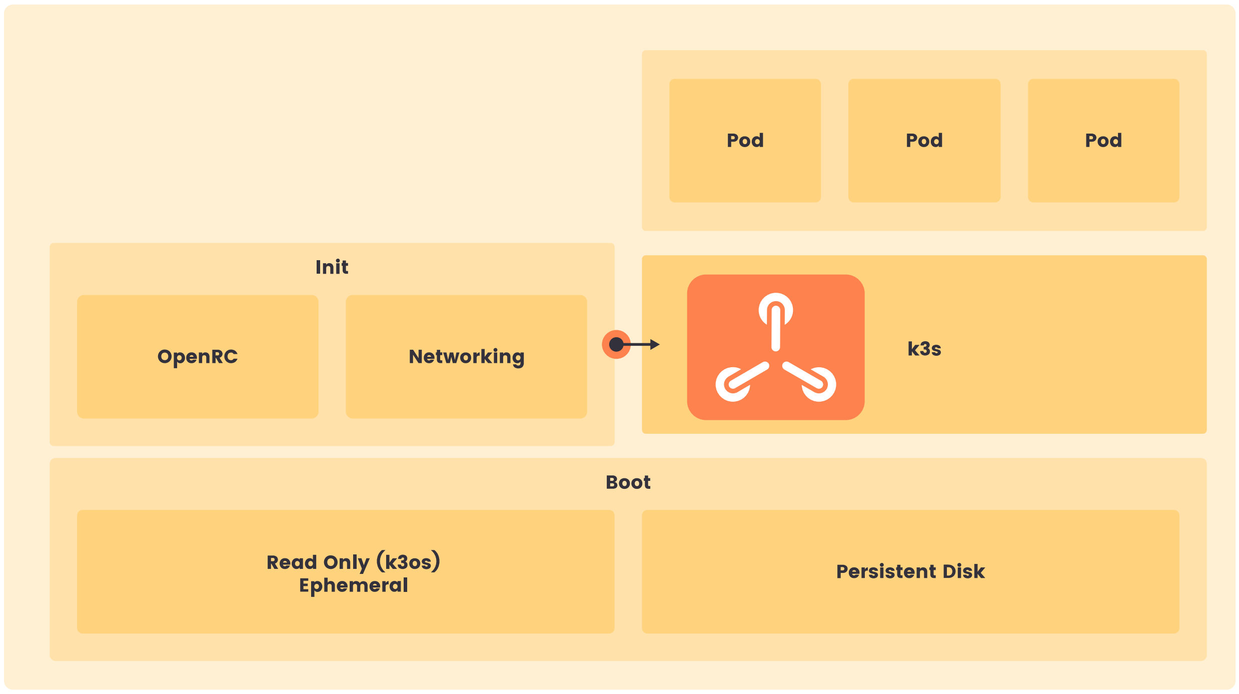


Figure 6.4. Components of *k3OS* from K3OS 'How it Works' [10].

ture 'Day 1' operations and merging 'Day 2' infrastructure operations with the application cluster operations. This allows Edge administrators and operators to focus on other 'Day 2' operations:

- Application tracing/observability
- Application log aggregation
- Monitoring/alerting
- Storage persistence
- Hierarchical storage management
- Metering/billing chargeback
- Tenant networking policy
- Resource segmentation
- Multi-environment rollout
- Cluster/application elasticity

6.3. K3OS Example

Installed on a Raspberry Pi 3 B, a bare *k3OS* Server and Agent, showed the configuration in Listing 6.1 with *kubectl*. This output shows both the ‘system’ **Namespace** `kube-system` and the user **Namespace** `default`. I installed an example application, MediaWiki, consisting of a webserver and database.

6.4. Concerns

The concerns with moving to a Kubernetes-backed deployment are issues with scale. Kubernetes, and by inference a *k3s* fleet, only makes sense when working at web-scale, with thousands of cluster nodes and hundreds of applications. Additionally, the foundational components to be successful at web-scale applications must be in place: infrastructure-as-code, strongly opinionated configurations and mature CI/CD pipelines. If any of those pieces are lacking, companies will not be able to deploy Kubernetes effectively, or worse, once deployed they will suffer high-availability outages. For many companies that are just now making the leap to containers, or even virtualization, the bar for Kubernetes is just too high to surpass.

Listing 6.1. Example *k3OS* installation showing Pods and Services. Ephemeral postfixes and labels removed for clarity.

```
rancher [~]kubectl get all --all-namespaces
NAMESPACE   NAME                          READY   STATUS      RESTARTS   AGE
kube-system pod/helm-install-traefik      0/1     Completed   1          25h
k3os-system pod/system-upgrade-controller 1/1     Running     1          25h
kube-system pod/local-path-provisioner    1/1     Running     1          25h
kube-system pod/metrics-server            1/1     Running     1          25h
kube-system pod/svclb-traefik             2/2     Running     2          25h
kube-system pod/coredns                   1/1     Running     1          25h
kube-system pod/traefik                   1/1     Running     1          25h
default     pod/svclb-mediawiki5          0/1     Pending     0          20h
default     pod/mediawiki-mariadb-0       1/1     Running     0          20h

NAMESPACE   NAME                      TYPE CLUSTER-IP    EXTERNAL-IP PORT(S)                AGE
default     svc/kubernetes            IP   10.43.0.1     <none>      443/TCP                25h
kube-system svc/kube-dns              IP   10.43.0.10    <none>      53/UDP,53/TCP,9153/TCP 25h
kube-system svc/metrics-server        IP   10.43.241.154 <none>      443/TCP                25h
kube-system svc/traefik-prometheus    IP   10.43.199.18  <none>      9100/TCP               25h
kube-system svc/traefik               LB   10.43.83.123  192.168.1.9 80/TCP,443/TCP         25h
default     svc/mediawiki-mariadb     IP   10.43.240.144 <none>      3306/TCP               20h
default     svc/mediawiki             LB   10.43.100.237 <pending>   80:30490/TCP           20h

NAMESPACE   NAME                          DESIRED CURRENT READY UP-TO-DATE AVAILABLE AGE
kube-system daemonset.apps/svclb-traefik   1       1       1     1          1         25h
default     daemonset.apps/svclb-mediawiki 1       1       0     1          0         20h

NAMESPACE   NAME                                        READY   UP-TO-DATE   AVAILABLE AGE
k3os-system deployment.apps/system-upgrade-controller   1/1     1            1         25h
kube-system deployment.apps/local-path-provisioner      1/1     1            1         25h
kube-system deployment.apps/metrics-server              1/1     1            1         25h
kube-system deployment.apps/coredns                     1/1     1            1         25h
kube-system deployment.apps/traefik                     1/1     1            1         25h

NAMESPACE   NAME                                        DESIRED   CURRENT   READY   AGE
k3os-system replicaset.apps/system-upgrade-controller   1         1         1       25h
kube-system replicaset.apps/local-path-provisioner      1         1         1       25h
kube-system replicaset.apps/metrics-server              1         1         1       25h
kube-system replicaset.apps/coredns                     1         1         1       25h
kube-system replicaset.apps/traefik                     1         1         1       25h

NAMESPACE NAME                                READY   AGE
default   statefulset.apps/mediawiki-mariadb  1/1     20h

NAMESPACE   NAME                          COMPLETIONS   DURATION   AGE
kube-system job.batch/helm-install-traefik   1/1           68s        25h
```

CHAPTER 7

PREVIOUS RESEARCH

Applications of connected vehicles vary widely, but one of the easiest transitions to make is automated Police Patrol placement. Police patrol placement is not currently highly automated, but can take advantage of simple GPS sensors on each patrol car, along with cloud resources data mining relevant crime datasets. In the future, unmanned drones could replaced the patrolling aspect of policing, leading to better utilization of non-automated (human) resources. To enable the automation of patrol selection to scale, and not need to rely upon a centralized decision system, each vehicle should have an application that determines it's own placement, communicating to other police vehicles to determine patrol routes for each time of day.

7.1. Optimal Police Patrol Planning Strategy for Smart City Safety [104][1]

The safety of citizens is an integral part of any smart city project. Police patrol provides an effective way to detect suspects and possible crimes. However, policing is a limited resource just like any other service that a Smart City provides. In order to efficiently consume this resource, the city has several aspects that can be controlled to make efficient use of Police patrolling: where (area), what (number), when (hour). In this paper, we utilize the LA County Sheriff's open crime dataset to study the police patrol planning problem. We propose a novel approach to build a network of clusters to efficiently assign patrols based on informational entropy. This minimizes Police time-to-arrival and lowers the overall numbers of police on patrol. Our algorithm relies upon the categories of crimes, and the locations of crimes. Since we use real-time traffic analysis to join crime clusters, our solution is extensible enough to be applied to any metropolitan area.

[1] Section 7.1 is reproduced in its entirety from Jacob Hochstetler, Lauren Hochstetler, and Song Fu, An optimal police patrol planning strategy for smart city safety, 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPC-C/SmartCity/DSS), pp. 1256-1263, with permission from IEEE

7.1.1. Introduction

The concept of Smart City as a means to enhance the life quality of citizens has been gaining increasing importance in the agenda of policy makers [154]. Most recently, 10 cities in the United States were selected to participate in the Smart Cities Initiative [220], including Los Angeles, Dallas, and eight other cities. The involved projects deal with smart transit/parking, Internet of Things, micro-grids, streetlights and smart poles, integrated smart city systems and more [64].

A smart city must be a secure city first. While many smart-city projects will increase well-being or quality of life, the safety of the citizens should be an integral part of any city project. With advancement of civilization, day-to-day human life is safer today than ever before. However, in view of new threats of terrorism, organized crimes, gang violence and gun crimes, securing cities remains an equally important and a big challenge that smart city initiatives could provide unique solutions to.

Routine activities theory posits that crimes, both individual and serial, occur when a motivated offender encounters a suitable target in a time and place where there is an absence of capable guardianship [24]. Police patrol provides an effective way to detect suspects and possible crimes, thereby deterring offenders from committing crimes. For example, *suspect-oriented patrol* occurs when a suspect matches the description of an offender in a series, and *directed patrol* involves instructing officers to visit certain locations at certain times. The current practice of police patrol is either based on officers' experience or neighborhood crime statistics, that is, if an area has more crimes in the near past, more officers will patrol in that area.

Motorized and foot police patrols have been utilized as a crime deterrent method for many years. Studies have been done on this method to discover whether or not it is effective in eliminating crime, or whether crime just moves away from the areas patrols have increased. An experiment [178] conducted in Philadelphia of more than 200 foot patrol officers in the summer of 2009 found that police foot patrols raised the public's perception of the police in the communities, reducing their fear of crime.They studied police effectiveness over 60 violent crime problem areas. In 12 weeks,there was a significant reduction in the amount of crimes in the targeted areas,exceeding the control site numbers by 23% with 53 violent crimes prevented. These findings show that police patrols in targeted areas where violent crime occurs can have a significant impact on violent crime rates at a microspatial level.

Patrol placement is a critical process used to maximize the effectiveness of the police department in its activities. However, police are limited in supply. To be effective at policing, officers must be qualified, formally trained, and developed. For example, officers in the City of Los Angeles Police Department (LAPD) must be 21 years of age, and have completed a six-month training course before beginning their year-long probationary period [48]. These constraints mean that the number of officers, like many other city resources, is inelastic. Further, the limited availability of resources in comparison to the city's large population and size makes it more difficult to patrol the entire city of Los Angeles. Limited budget implies limited number of patrols cars which can pose a problem to availability of back up officers present at crime scenes or in the pursuit of criminals. The limited police personnel and budget makes patrol planning a challenging task. A smart placement strategy is needed to assure the safety of smart cities.

In this paper, we propose a data-driven, smart decision making approach to place police officers for the optimal patrol. Our goal is to maximize the responsiveness of police departments in their activities while having limited police officers and resources. Our proposed approach leverages an entropic metric in selecting patrol locations and in searching for the cluster locations that maximize the total entropy in the Patrol Police Network (PPN), relating maximal entropy with maximal clusters coverage. We use Los Angeles Open Data (LAOD) [138], especially the crime related datasets, and evaluate the performance of our proposed patrol planning strategy. The experimental results show that response-time can be reduced and police can cover more crimes through our entropy-based approach. We want to emphasize that this is not a crime-prevention strategy, as crime factors are complex and numerous. Instead, we will treat crime like a resource that should be monitored, with patrols as the sensors that do the monitoring.

Table 7.1. Information About Los Angeles County and Budget/Equipment of LAPD

Policing coverage & equipment	**Quantity**
LA area	~12,300 km^2
LA population	3.8 million residents
Police budget	$1.4 billion
Number of police stations	21
Number of police officers	~10,000
Officers per 10k residents	~25.6
Number of patrol cars	Unknown
Number of police boats	2
Number of police helicopters	26
Number of police fixed-wing	3
Number of mounted (horses)	21
Number of K-9 units (canine)	22

Table 7.2. Selected Attributes of LA County Crime Data

Attribute	**Explanation**
Incident Date	Date a crime incident occurred.
Category	Incident crime category.
Statistical Code	A three digit number to identify the primary crime category for an incident.
Statistical Code Description	The definition of the statistical code number.
Full Address	The street number, street name, state and zip where the incident occurred.
Street Address	The street number and street name where the incident occurred.
City	The city where the incident occurred.
X / Y Geocode	Used to map the general location of the incident.

The rest of the paper is organized as follows. We describe the crime datasets from LAOD and a classification of crime types in Section 7.1.2 and Section 7.1.3. The methodology and patrol planning design are presented in Section 7.1.4. The implementation and detailed

Table 7.3. Sample of Crime Categories

Category	**Count**	**Weight**	**Reasoning**
Arson	4337	0	Investigation occurs after the fact in coordination with the Fire Services Investigator. Officer must be present to cordon off the scene.
Commitments	14	1	Drunk/Drug tank; officer must be present to take suspect into custody.
Criminal Homicide	1689	5	Most heinous crime which requires immediate Police response.
Forgery	16531	0	Not time-sensitive.
Fraud/NSF Checks	44681	0	Not time-sensitive or critical.
Traffic Accidents	14402	4	First responders are needed to direct traffic and resource other assets.
Warrants	2326	0	Served on an as-needed basis.

evaluation results are reported in Section 7.1.5 and Section 7.1.6. We conclude this paper and discuss possible future work in Section 7.1.7.

7.1.2. City Safety Dataset

We explore the Los Angeles County Sheriff's Department Jurisdiction Data available from Los Angeles County GIS Data Portal [47] in this study. Los Angeles County was chosen because of three main reasons:

- Weather stability
- Land size
- Dataset size & crime variation

The historical average temperature in LA County varies from a December low of 8.6 °C, to an August high of 23.5 °C [157]. Month by month, this results in a mean daily temperature variation of only 10.8 °C, with a standard deviation of 0.54. For comparison, Chicago, which is similar in population size, has a daily average low of -10.8 °C in January to an average high of 28.6 °C in July. This results in a mean of 11.6 °C with a standard deviation of 1.63. With regards to rain and snow, LA receives an average 379.2 mm of rainfall each year, while Chicago receives 936.2 mm of precipitation. LA County has no average snow or ice. Compared to Chicago, LA has more stable weather and this stability should produce more consistent crime data that isn't affected by seasonal patterns.

Since we utilize the drive-time between clusters for our calculations, it makes sense to have a dataset that is geographically distributed. Once again, compared to Cook County (Chicago), LA county is much larger at 12,300 km^2 versus 4,235 km^2. Ideally this would produce geographically distributed clusters of crime which would show the effectiveness of the strategy.

The dataset consists of 2,130,504 crimes from 2005 to 2015 (11 years). Each year is formatted as a CSV file, and there are 18 data fields for each record. Selected attributes are listed in Table 7.2. The variations in crime is discussed in Section 7.1.3 below.

In addition to the crime data, we also collect the information of equipment and resources of Los Angeles Police Department (LAPD). Our goal is to effectively allocate these resources for optimal police patrolling to respond to and suppress crimes. For now, we will focus on just allocating patrol cars, as specialized assets require specific use cases for deployment. Information and selected resources of the LAPD are shown in Table 7.1.

7.1.3. Crime Prioritization and Statistical Analysis

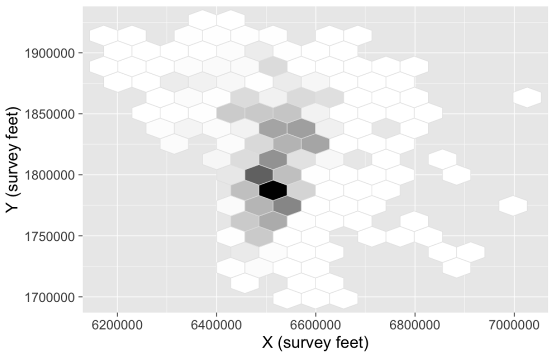
In the LA county crime dataset, a total of 42 major crime categories are specified. Due to the limited police personnel and resources, we prioritize these crime types by assigning higher priorities to those types that need more immediate police response. To this end, we have consulted with a domain expert in Criminal Justice to help us better understand the datasets we use. Since some crimes are more critical than other crimes, i.e. rape vs. larceny, we weight each crime on a scale of zero to five. A weighting of five is the most important, whereas a 1 is of the lowest importance. A weighting of zero means we do not include this crime in our algorithm. Each crime belongs to a “Category” and we can exploit this field to base our weighting. There are a total of 42 major crime categories and 369 sub-categories. The domain expert finds many of these categories are not time-sensitive. A sample of some of the categories, the count in our dataset, the weighting and our reasoning is presented in Table 7.3. Due to space limitations, we have only listed some of the crime categories. Figure 7.2a shows the number of crime records at each of the five priority levels.

Each crime record contains an X and a Y coordinate that locate the crime within LA County. Although each complete crime also includes the full address, we would have to geocode each address through an API, severely limiting our throughput. The X and Y coordinates are in the State Plane Coordinate System (SPS) format, specifically California Zone V (5). We remove those records that do not contain X or Y coordinates. A histogram plot of these records results in a spike from coordinates X[6200000:7000000] and Y[1700000:1900000]. Some of the outliers are as far away as Colorado, and include bad negative coordinates that are not even possible with the SPCS. The LA crime dataset of 2015 does not include the time for each crime. This skews our hourly results towards midnight, since every crime occurs at 00:00 in that dataset.

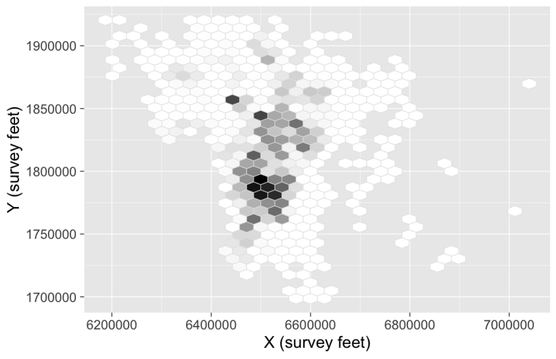
After preprocessing we end up with 1,486,678 crimes, i.e., about 70% of the original data. We use *hexagonal binning* to check if the crime distribution is uniform across coordinates. Sample sizes for 15, 30 and 50 bins are show in Figure 7.1. The clusters are not uniform, and the figures show the centroids begin to dissipate at bin size 30.

We further analyze the distribution of crimes by hours of the day and days of the week using both the overall layout of the crimes and the density of the crimes. These are shown in Figure 7.2b and Figure 7.2c. From the figures, we can see that the distributions for hourly and daily crimes are not uniform. More crimes happen during 16:00-01:00 and on Fridays than other times and days. Meanwhile, 03:00-06:00 and Sundays have the least number of crimes. These uneven crime distributions suggest that we should differentiate police allocation to achieve better cost-effective patrolling. To minimize the affect of time, we will group the crime data into hours for this paper. More specialized implementation can be performed on a per-day basis. Holidays and other city events were not selected as attributes, but could be used for further study since they also affect crime patterns.

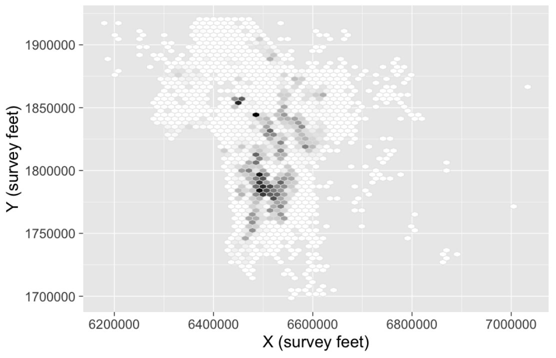
Since we use data clustering methods for our patrol planning algorithm, we need for further preprocessing of the data. We filter the crime data set, first by removing all zero-



(A) Bin size 15



(B) Bin size 30



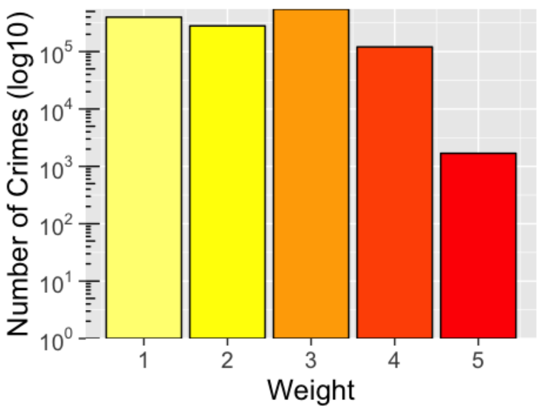
(C) Bin size 50

FIGURE 7.1. Hexbin clustering of crimes in LA County based on X and Y coordinates in survey feet.

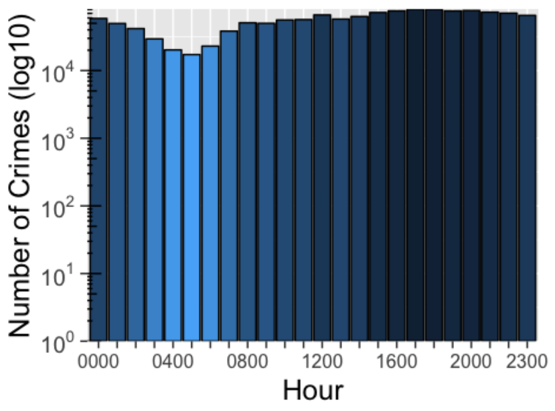
weight crimes, and further restricting the X geo-coordinate to between 6270000 and 6680000 survey feet. This leaves us with 1,351,527 crime records. Since data clustering methods do not use weighting, we replicate each of the weight lines the number of its respective weight. This results in a total count of 3,101,851 crime records.

7.1.4. Optimal Police Patrolling Strategy

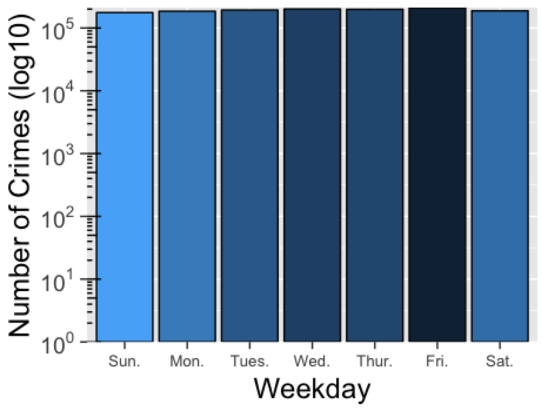
The patrol planning problem can be informally described as follows. For an area in question (such as a city or a county), given its historical crime data and available police resources, find the best officer placement for patrol that can maximally suppress possible crimes. This is a challenging problem as it involves geolocations of many streets in the area and unevenly distributed crime occurrences at those locations under the constraint of limited police resources.



(A) By weight/priority.



(B) By hour.



(C) By day of week.

FIGURE 7.2. Distribution of crimes in LA County based on data attributes ($\log_{10}$).

7.1.4.1. Design of Police Patrolling Strategy

To address these challenges and make the best patrol planning, we formalize the preceding problem as an *entropy maximization problem in a police placement network* and derive the optimal solution. Before presenting the details of our police patrolling strategy, we briefly describe entropy and equivocation. Entropy [90] quantifies the smoothness with which a transformation occurs and of the disorder and the amount of wasted energy during the transformation from one state to another. Mathematically, entropy can be expressed as $H_a = \sum p_a \ln(1/p_a)$, where p_a is the probability mass function of variable a. The entropy of two variables can be calculated as

$$H(A,B) = H(A|B) + H(B|A) + I(A,B)\,, \tag{7.1}$$

where $I(A,B)$ is the mutual information between A and B, which is determined by

$$I(A,B) = \sum_{a\in A, b\in B} p(a,b)\ln\left(\frac{p(a,b)}{p(a)p(b)}\right). \tag{7.2}$$

$H(A|B)$ in Equation 7.1 is the entropy of A conditional on B, which is called equivocation as computed with

$$H(A|B) = \sum_{a\in A, b\in B} p(a,b)\ln\left(\frac{p(b)}{p(a,b)}\right). \tag{7.3}$$

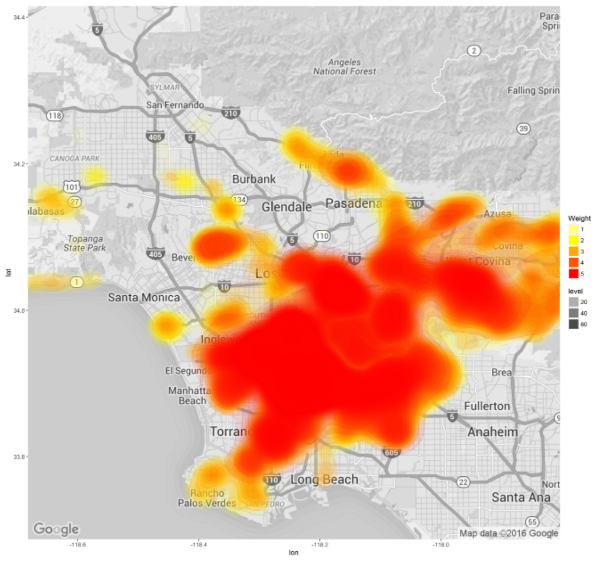
For a system with n variables, entropy $H(p1,\ldots,pn)$ is a measure of a system's order and stability. Its value is maximized when the system is at an equiprobability state. A greater value of entropy indicates a more balanced system in terms of the measured information.

The police patrol planning problem can be restated as one in which patrol locations are sought so that the system entropy is maximized. We define the probability term p_a in terms of a statistical measure of the ratio of an officer's patrolling radius (r) with regard to the quickest path to other locations over the length of the network. It also includes the effect of historical crime data in terms of the crime weight (w, defined in Section 7.1.3). That is

$$H_{c1} = -p(c_1)\ln p(c_1)\,, \quad p(c_1) = \left(\frac{r_{c1}}{r_{sys}} + \frac{w_{c1}}{w_{sys}}\right)\Big/ 2\,. \tag{7.4}$$

The entropy is a combination of the weight of a crime centroid (w_{c1}), generated from crime clustering, over the total system weight (w_{sys}), added to the quickest path from the centroid to any other centroid (r_{c1}), over the quickest path in the entire system (r_{sys}). This balances the need for multiple patrols to cover a larger area that has few short paths between centroids. The value of r_{sys} can be determined by the length of an area's all paths being patrolled.

The total system entropy can be computed by summing up the entropy values for each path, as $H_{System} = \sum_{ci} H_{ci}$. The goal, therefore, is to maximize the system entropy subject to the allowable maximum number of officers and patrolling resources, or equivalently to maximize the entropy while minimizing the number of officers and resources used.


(A) *stat_density2d* plot with size=1, bins=128.

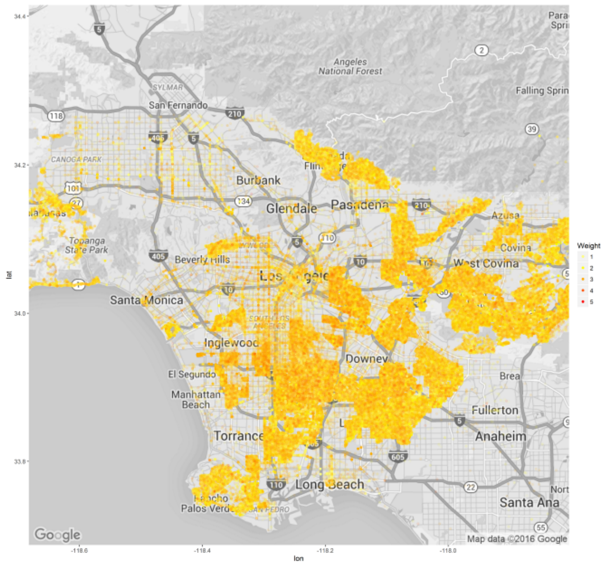

(B) *geom_point* plot with alpha=1/10.

FIGURE 7.3. *ggplot2* plots of all crimes by weight.

The entropy maximization approach works as follows. It starts with the nodal entropy values from the all officer configuration as the calculation base, and assumes that the entropy contributions to the total system entropy from the patrolling officers are not subject to the equivocation property. After ranking the nodal entropies in descending order, the method selects the node that contributes the maximum to the system entropy and places an officer at that node. After assigning an officer to a node, the entropy values of the connected nodes are adjusted, considering equivocation and entropy maximization approach. The nodal entropies are re-calculated and the node with the highest entropy is selected for patrol placement. The process is repeated until the entire area is covered or the number of available officers is reached.

7.1.4.2. Implementation of Police Patrolling Strategy

To ensure that the geospatial distribution is fairly uniform, we plot the crimes against a map of LA County using the library *ggmap* [119]. A 2d density plot of the crimes is shown in Figure 7.3a and a plot of the all the crimes shaded by crime weight is shown in Figure 7.3b. Both figures were generated using the *R* [210] library *ggplot2* [226].

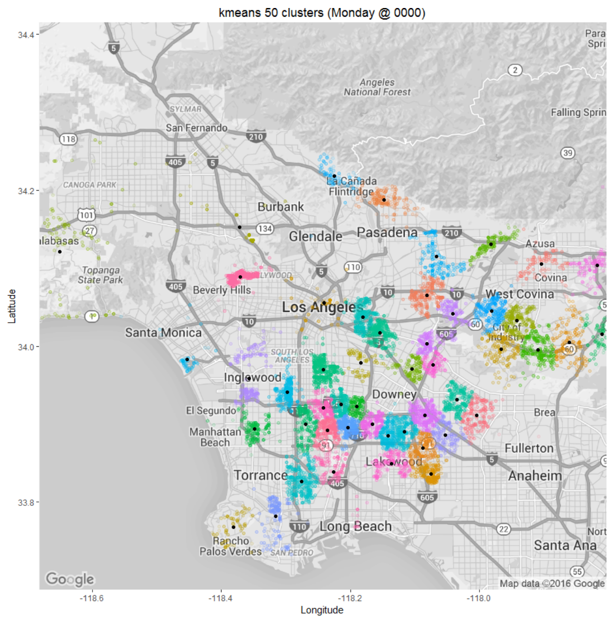
We use Google Maps as the base layer for *ggmap* to present crime clusters and police patrol placement. We convert each crime record from the State Plane Coordinate System (SPS) format to the actual latitude/longitude since that is the only format Google Maps API will allow. The correct State Plane projection for California V is European Petroleum Survey Group (EPSG) 2229 [96]. To get the correct latitude/longitude we use the *rgdal* [123] library in R to convert each X/Y pair to EPSG 4326, which is the EPSG identifier for World Geodetic System (WGS) 84.

After the coordinates are converted, we cluster consecutive sets of data for different hours of a day and different days of a week. We use a simple data clustering method, i.e., K-Means clustering, with a size of 50 based on the hexbin plots shown in Figure 7.1c. Since K-Means clustering does not implement a weight feature, we replicate each row times its weight for clustering. For the optimum clustering, Mclust or DBSCAN should be used and data should not have to be replicated. The resulting cluster centroids are aggregated with their sum weights and saved.

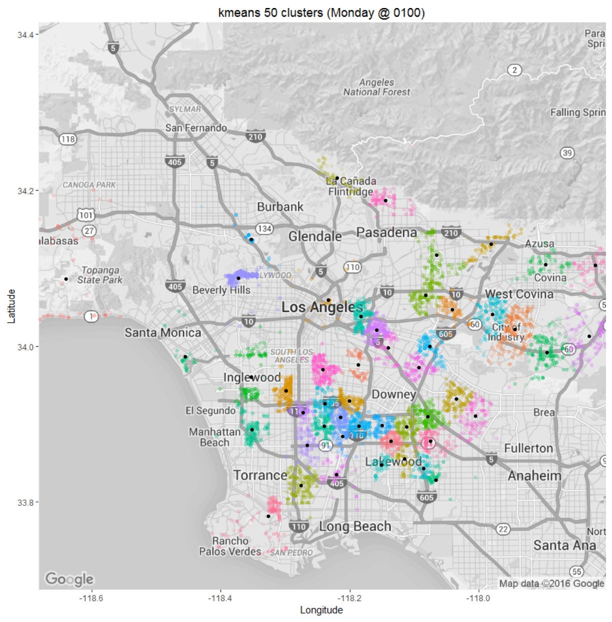
We use real-time traffic analysis as the basis for our "pipes" between clusters. The Google Maps Distance Matrix API [87] provides traffic-time between points, and can be used to predict traffic at future dates/times. Using the Google Maps Distance Matrix API, the distance between each pair of the centroids is the drive-time calculated to each other. For cluster i and cluster j we use the first drive-time calculation, i.e. from i to j only. A more robust solution would be to calculate bi-directional travel-times for each centroid in the fully-connected graph. Since the free-limit for the API is 2500/per day, we find that the bi-directional aspect would not affect our experimental data.

As we collect all the traffic data, our algorithm connects the cluster centroids using this traffic-time. Using this network, we then place patrols within the centroids. Patrols are placed according to the entropy calculation using Equation (4) for each edge in the connected centroid graph.

We place a fraction (denoted by f) of the number of centroids as patrols (denoted by n), such as 50%. This is a fair number of patrols to emphasize our approach, although this number would obviously be adjusted for real cities. This is a small number for such a



(A) Monday at 0000 hours.



(B) Monday at 0100 hours.

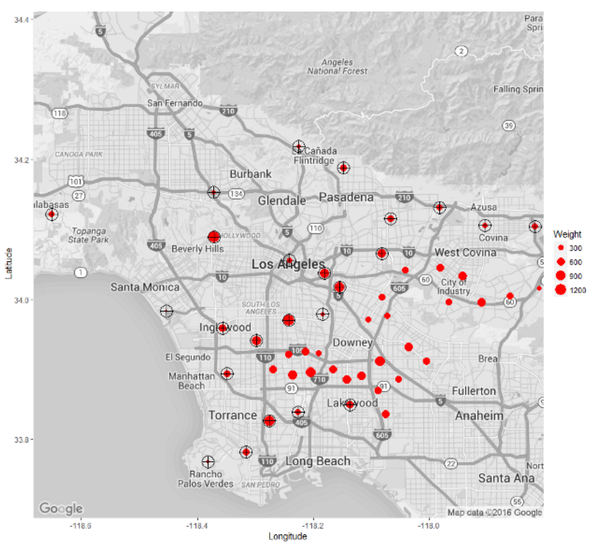
FIGURE 7.4. Crime clusters generated by K-means clustering. Black dots represent each cluster's center (centroid).

large area, but our algorithm is generalized enough that any area and cluster combination is calculable. The only restriction is the drive-time between clusters using the Google API, since for a robust algorithm, the number of edges is an ${}_nP_2$ permutation of the number of centroids.

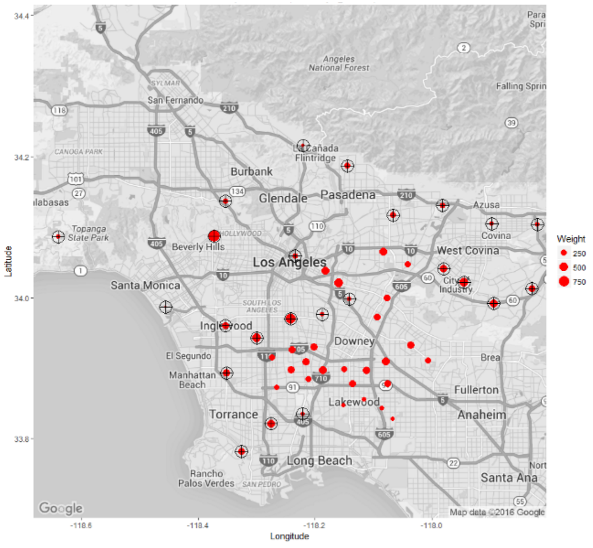
After each edge entropy is calculated, the $f \times n$ patrols are placed according to the most information gained from the entropy calculation. All preprocessing, filtering, and feature selection is done using the Go programming language for speed and concurrency, and then the data are exported to CSV for visualization in *R*.

7.1.5. Experimental Results

Figure 7.4a and 7.4b show the results of crime groups by using K-Means clustering approach. One is for Monday at 0000 hours and another is for Monday at 0100 hours. Using our entropy calculation to place patrols, we emphasize the long drive time needed between crime clusters and the actual amount and severity of crime in a given area. There are several outliers located in the south-east corner of LA County, which means that the entropy formula



(A) Monday at 0000 hours.



(B) Monday at 0100 hours.

FIGURE 7.5. 50 crime clusters with 25 patrols placed using entropy algorithm. Black cross-hairs show placed patrols.

must be balanced between the aggregate weighting and drive-time. These plots are available in Figure 7.5a and 7.5b. Larger area cities might need to assign a higher weight to drive-time during entropy calculation, whereas smaller cities could emphasize crime aggregates and disregard drive-time.

The choice of 25 patrols is used as the number of cluster nodes. For real-world use, the entire entropy of the system could be calculated over hundreds of nodes. Then patrols would be placed on the highest-entropy nodes, and the entropy of the system minus that node would be recalculated. Patrols would be placed until there is no change in entropy, or there is no more "information" gained by covering any other nodes. This is much easier if hundreds of clusters have been generated since patrols will be able to overlap closely (drive-time) related nodes.

Another solution to the closely clustered centroids is described below, which is to collapse clusters that are "clustered" together inside some predetermined constant. Neither solution addresses the aggregate weighting of crimes at nodes, merely the presence of police at a given node. In higher weighted nodes, more physical police presence should be emplaced, while in lower weighted nodes lower-profile solutions could be used instead of police officers. These lower-profile solutions could include community-organized watch or at-risk youth programs.

While the results follow the information theory of entropy, there is unfortunately no way to determine if the calculated patrols can reduce crime. It would be up to the LAPD to change patrol areas and implement new strategies. Therefore, the impact of this strategy in real-world applications is uncertain.

7.1.6. Discussion

In our experiments, we need to remove about 37% of the original dataset because of incorrect X/Y coordinates or incomplete data. There is a possibility that the crime densities would not be uniform across weightings or hours due to these records. Since our algorithm can work at a more granular level, i.e., by day and hour, this should not affect our performance.

There are many external threats to validity. While police patrols can be varied and changed, police officers are not a resource that can be turned on or off each shift. If officers are not out on patrol, they will be at their station on administrative duty. In addition, while our algorithm balances car patrols among the higher crime areas, criminologists agree that Police presence in communities is one of the main deterrents to crimes being committed. Adding random car or foot patrols to lightly surveilled areas could counteract those criminals.

We do not partition our data into hourly chunks, but for actual implementation, each hour could be treated as its own maximization problem, with carry-over between centroids that cross hourly barriers. In addition, we do not take into consideration the aggregate size of each centroid. Patrol sizes should be adjusted to account for both the aggregate weighting of the centroid and the numbers of crimes within a predefined area of time around the centroid. Statistically we can determine how many crimes will occur in this centroid per hour and adjust the patrol count to adequately cover most crimes within the entire cluster, but this would be up to police management and budget constraints.

There are numerous problems with the crime dataset, from missing time, to blank coordinates, and locations that are not even in the state of California. For future study, completing the dataset and mapping crimes with correct coordinates will improve the results for effective patrol planning. Regarding the weighting system, since each police force categorizes and prioritizes crime differently, this would be adjusted by city-policy. For Los Angeles, we felt five levels were sufficient, but for other cities that number would be adjusted up or down. In addition, the crime weighting system could be improved by adding a sliding time scale so that more recent crimes are favored over older crimes. The window used in the experiment by Ratcliffe et al. [178] weighted crimes committed each year half as much as the next year, e.g. 2008 = 1.0, 2007 = 0.5, 2006 = 0.25. Since crime hotspots usually have a long-term crime trajectory, weighing each crime's date with an exponential decay may be better suited to robust crime data that spans many years.

7.1.7. Conclusion

In this paper, we study the police patrol planning problem for the safety of citizens in cities. We formalize the problem as an entropy maximization process in a police placement network and derive the optimal solution. Our algorithm shows that using entropy composed of both the drive-time between cluster centroids and the actual severity and aggregate of the centroid is effective in placing police patrols. These patrols are placed according to the resource maximization of available patrols, and police time is not spent driving between hot-spots of crime within the city. Since our approach is generalizable, any city utilizing smart crime collection can benefit from our work, provided they defined their crime weight and reporting system.

In our study, the main categories are a good estimate for our algorithm, but for future work we will focus on broader datasets and be more pragmatic in filtering records that do not meet a timeliness threshold, i.e., "Exploitation of Child via internet" should not necessitate an immediate police response. For cities with multiple types of police response, e.g., car, foot, and bicycle, the Google API can provide a drive-time for each type of patrol, further benefiting the overall patrol coverage. Lastly, we plan to define a response time and collapse all the clusters that are within that time window into one super-cluster with the aggregate weighting of all sub-clusters before the entropy of the system is calculated.

7.2. TuranGo: Mutation Testing a Language

The modern CAV landscape includes almost every programming language know. Starting at the data center, Golang is used for a host of backend services, like schedulers in Cloud Foundary, or networking ingress controllers in Kubernetes. Every cloud provider has extensive Golang codebases, with most composing the majority of their open-source contributions. For developing machine learning models, Python has been dominating machine learning with popular libraries like *TensorFlow*, *PyTorch*, *Theano*, *Keras* or of course *scikit-learn*. The other most popular ML language is C++, also used with *TensorFlow*. These languages normally use the NVidia CUDA API, written for C, C++, and Fortran, to run large training sets in parallel on the GPUS in graphics cards. For running inference models, Python, C++, Java and C#are popular. Low-level, microcontroller programming is handled by C, with more and more work being done in C++. Even Golang is being deployed to microcontrollers with projects like TinyGo [65]. For general purpose applications, Java is still popular for many applications, as the JVM is relied upon to provide platform abstraction between the developers workstation and the end-user's device. All these languages have extensive testing frameworks, with Java's JUnit pushing forward the concept of test-driven development (TDD) with its release in 1997.

Passing tests gives your project confidence, and with enough coverage, this confidence is justified. But what about the libraries your application relies upon? A language's standard libraries are taken for granted, and universally used. Normally these libraries are trusted to be quality implementations and tested to be free from bugs. One of the most effective ways to verify your test suite is complete is with mutation testing. To test if mutation testing is suitable for modern Continuous Integration (CI) environments, in this paper we present a mutation tool for Go packages called TuranGo and evaluate it on open-source application and packages. In addition, since Go is relatively new, the standard libraries are treated as a reference for the “Go way”. Thus we develop an experiment to investigate whether the test suites become better over time. To do this, we take a sample of the standard libraries and mutation and test it throughout each Go version.

7.2.1. Introduction

The main idea of mutation testing [160] is to create a different version of a codebase (called a mutant), and then run the original test suite on this mutant. There are three possible outcomes for the test suite:

(1) failed because the mutant is killed,

(2) passed because the mutant lives, or

(3) passed due to an equivalent mutant that cannot be killed.

According to the Reachability, Infection, and Propagation (RIP) model [7], a mutant can only be killed if the following three conditions are met.

(1) *Reached* —- The test must actually execute the statement.

(2) *Infected* —- The statement must actually be mutated.

(3) *Propagated* -— The result of the test must be visible.

While being an effective way to judge the quality of a test suite and thus the codebase, mutation testing is expensive [159] in terms of both computational power and time spent on testing. Many modern mutation tools have complex algorithms to pre-select tests and pair those tests with the "correct" mutants, thereby reducing the overall testing time.

Go offers an interesting way to address that expense, as by design it is a modern language built for fast compilation with a built-in testing framework and baked-in concurrency.

To evaluate if mutation testing is an appropriate fit for the Go language, in this paper we develop a Go mutation tool, called *TuranGo*. We evaluate it using experiments. In the first set of experiments, we sample open-source programs along with some packages to study if the benefits outweigh the disadvantages in a continuous integration (CI) environment. For the second set of experiments, we write another tool to test a sample of Go's standard packages to analyze if the test suites developed better quality tests through each version of Go released. Our experimental results show that there are good benefits for Go CI mutation testing, even though it extends testing time significantly. Additionally, our second experiment shows that the quality of tests in the actual Go language has decreased slightly over time, with some specific packages having unexpected and irregular quality trends.

The rest of this paper is organized as follows. Section 7.2.2 introduces the Go language and Section 8.2.1 discusses the related work. The design of TuranGo is described in Section 7.2.3. Sections 7.2.4 and 7.2.5 present and analyze the experimental results. Section 7.2.6 concludes the paper with remarks on future research.

7.2.2. Background

Go is a "new" language invented in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson. It was started as a pure research experiment while all three were working at Google. Robert Griesemer previously worked for Sun Microsystems focusing on the Java Hotspot VM, and later wrote the V8 JavaScript engine for Google. Rob Pike previously worked for Bell Labs, where he wrote the first windowing system for UNIX. With Ken Thompson, he developed UTF-8. Ken Thompson is best known for inventing UNIX while working at Bell Labs.

7.2.2.1. Go

Go, also called golang, is a derivative in the C-family of computer languages. Designed to be concise, the Go language consists of only 25 keywords and 47 operators/delimiters. This compares to 112 keywords for C++ (120 including identifiers) and 44 for C, according to the C++20 and C++17 standards [115]. In addition, Go was designed for the "modern age", with a number of design goals [84]:

- Computers are enormously quicker but software development is not faster. It is possible to compile a large Go program in a few seconds on a single computer.
- Dependency management is a big part of software development today but the "header files" of languages in the C tradition are antithetical to clean dependency analysis and fast compilation. Go provides a model for software construction that makes dependency analysis easy and avoids much of the overhead of C-style include files and libraries.

- There is a growing rebellion against cumbersome type systems like those of Java and C++, pushing users towards dynamically typed languages such as Python and JavaScript. Go's type system has no hierarchy, so no time is spent defining the relationships between types. Although Go has static types, the language attempts to make types feel lighter weight than in typical object oriented languages.
- Some fundamental concepts such as garbage collection and parallel computation are not well supported by popular systems languages. Go is fully garbage-collected and provides fundamental support for concurrent execution and communication.
- The emergence of multicore computers has generated worry and confusion. By its design, Go proposes an approach for the construction of system software on multicore machines.

Go also has a built-in testing framework. Any `*_test.go` file in the package directory will be loaded by the `go test` command and any functions prefixed with "Test" will be executed. Test coverage is also built-in, and can be generated by appending the `-cover` flag to the `go test` command.

Go is particularly suitable for mutation testing because of its ability to quickly compile. Some of the main reasons for this efficiency include:

(1) A clearly defined syntax that is mathematically sound, for efficient scanning and parsing.

(2) A type-safe and statically-compiled language that uses separate compilation with dependency and type checking across module boundaries, to avoid unnecessary re-reading of header files and re-compiling of other modules - as opposed to independent compilation like in C/C++ where no such cross-module checks are performed by the compiler. Compared to C++ that needs to re-read all those header files over and over again, even for a simple one-line "hello world" program.

(3) An efficient compiler implementation (e.g. single-pass, recursive-descent top-down parsing).

Go uses semantic versioning [171]. There have been 66 *patch* releases of Go since 1.0

was released March 28th, 2012. There have been 11 *minor* releases with an average of three *patch* releases for each *minor* release. There has only been one *major* release of Go.

7.2.3. TuranGo: A Mutation Tool for Go Packages

In order to examine mutation testing in the Go language, we first survey the open-source community for current mutation testing packages. Two packages have been well developed, but have not fully met the needs for our experiments. One does not include enough mutators and the other tool includes platform-specific UNIX code that makes it unsuitable for cross-platform development.

We develop a mutation tool for Go packages, named *TuranGo*. TuranGo is composed of three main components: a Parser, a pluggable Mutator framework, and a Tester. An architecture overview is shown in Fig 7.6. The *Parser* loads the Go package and loops through each non-test file in the package, then parses that source code into an abstract syntax tree (AST). We use an AST because Go has very good support for transferring/modifying an AST tree through packages '*go/ast*', '*go/parser*' and '*go/token*'. Using ASTs provides many benefits since the Mutators do not have to get bogged down the details of syntax, correctness, or commented-code. The Parser also does a benchmark test of the package to both ensure that all tests pass in the test suite and generate a baseline timeout number for the test mutants.

The *Mutator framework* contains a number of mutators that are registered and enabled during the mutation phase. These mutators are separate packages and are listed in Table 7.4. Packaging each mutator separately ensures that mutators can be enabled or disabled by the developer during testing, and facilitates a concurrent/parallel solution for larger target packages. Each mutator walks the AST, returning both the count of available mutations and a channel for signaling mutant creation. Some Mutators remove statements entirely and other Mutators swap operators to their logical opposite. For unary tokens, their mutation is the removal of the token entirely. Table 7.5 lists all the tokens *TuranGo* currently recognizes during mutation. These tokens are taken from the Go language specification [86].

Once a mutant has been created, the source code is saved to a temporary directory within the `GOROOT` environment variable. This is because starting with Go version 1.5, "internal" packages may only be used within `GOROOT`, and compilation/testing fails otherwise.

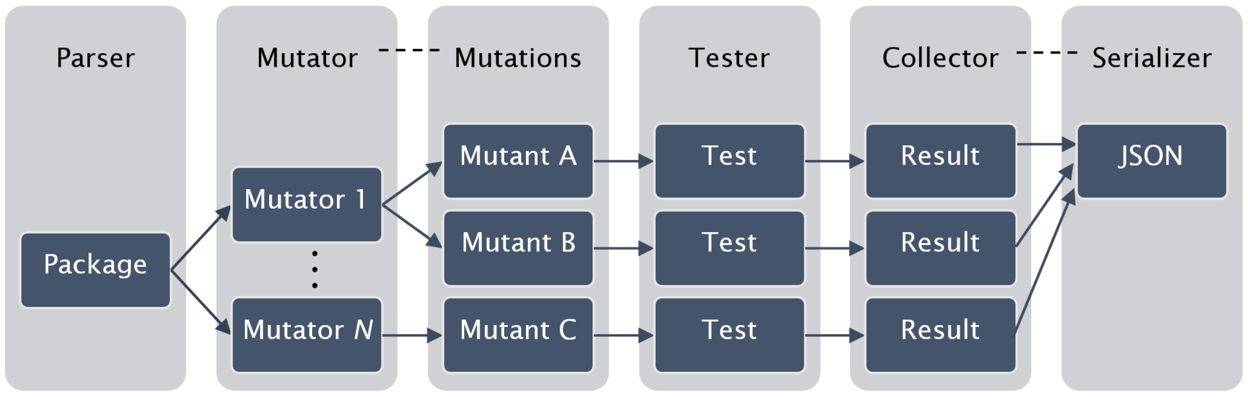


FIGURE 7.6. *TuranGo* architecture

The *Tester* functionally tests the mutant code using Go's `go test` command, and records the results as either **Alive**, **Killed**, **TimedOut**, or **ErrorCompiling**. If the mutate code is compilable, the Tester also records a difference of the source file and the mutant file. Currently this functionality relies upon the command line tool `diff` being installed. After the mutant has been tested, the Tester sends a message back down the Mutator channel to start the next mutant generation.

After all the mutators have been iterated through, the results from all the mutations are collected into a struct. The struct is then serialized to JSON in the specified output directory.

While Go does include some object-oriented logic, it is not a conventional object-oriented programming (OOP) language like C++ or Java. The only objects are structs, and nested structs, and public or private is determined by either a CamelCased name for an exported function/variable, or an all lowercase name for unexported function/variable. We initially choose to mutate the exported/unexported functions in Go. However, this results in a huge number of mutants which almost all fail to compile. With the large number of failed mutants, we deem mutating unexported/exported unnecessary.

Table 7.4. *TuranGo* Mutators

Package/Mutator	Description/Capabilities
`control/case`	Empties a case body in a switch statement
`control/else`	Empties the body of an else statement in a conditional
`control/if`	Empties the body of an if statement in a conditional
`expression/remove`	Removes terms from a binary expression
`expression/remove/LAND`	Short-circuits a `&&` expression by setting the first term to `TRUE`
`expression/remove/LOR`	Short-circuits a || expression by setting the first term to `FALSE`
`operator/assignment`	Swaps assignment operators to their logical opposite
`operator/binary`	Swaps binary operators to their logical opposite
`operator/inc_dec`	Swaps increment/decrement operators to their logical opposite
`operator/unary`	Removes unary operators from expressions
`statement/remove`	Removes statements, both from blocks and case clauses

7.2.4. Empirical Study and Experimental Results

We conduct experiments on a quad-core Intel Core i5-3570K with 16GB memory. Testing needs to be completed in several phases because of the total number of mutations involved. Since *TuranGo* writes each mutation to disk, the execution time is limited by the slower magnetic disk. An SSD or a memory-based file system would improve overall speed. Although Go is cross-platform and can cross-compile, Windows 10 is selected as the host OS for convenience since mutation testing 26 Go versions across 28 packages results in 728 separate *TuranGo* runs and 272,574 mutations.

7.2.4.1. Would Mutation Testing Be a Good Fit For a CI Pipeline?

In order to test if *TuranGo* would be a good fit for CI, we choose to evaluate an open-source application through the CI cycle combined with *TuranGo* mutation testing. Go applications can range from large (many thousands of lines of code in hundreds of packages) to small (hundreds of lines of code in only one package ‘main’). Software application [32] for instance is composed of 229 packages and over 190,000 lines of code (KLOC), with 261

Table 7.5. List of Tokens and Their Logical Opposites

Token name	**Operator**	**Go token**	**Opposite**
Addition	`+`, `+=`	ADD, ADD_ASSIGN	SUB, SUB_ASSIGN
Subtraction	`-`, `-=`	SUB, SUB_ASSIGN	ADD, ADD_ASSIGN
Multiplication	`*`, `*=`	MUL, MUL_ASSIGN	QUO, QUO_ASSIGN
Quotient	`/`, `/=`	QUO, QUO_ASSIGN	MUL, MUL_ASSIGN
Remainder	`%`, `%=`	REM, REM_ASSIGN	QUO, QUO_ASSIGN
Unary Plus	`+`	ADD	*None*[a]
Negation	`-`	SUB	*None*[a]
Bitwise AND	`&`, `&=`	AND, AND_ASSIGN	OR, OR_ASSIGN
Bitwise OR	|, |=	OR, OR_ASSIGN	AND, AND_ASSIGN
Bitwise XOR	`^`, `^=`	XOR, XOR_ASSIGN	AND, AND_ASSIGN
Bitwise Shift Left	<<, <<=	SHL, SHL_ASSIGN	SHR, SHR_ASSIGN
Bitwise Shift Right	>>, >>=	SHR, SHR_ASSIGN	SHL, SHL_ASSIGN
Bitwise AND NOT	`&^`, `&^=`	AND_NOT, AND_NOT_ASSIGN	XOR, XOR_ASSIGN
Bitwise Complement	`^`	XOR	*None*[a]
Logical AND	`&&`	LAND	LOR
Logical OR	||	LOR	LAND
Logical NOT	`!`	NOT	*None*[a]
Increment	`++`	INC	DEC
Decrement	`--`	DEC	INC
Equal to	`==`	EQL	NEQ
Less than	<	LSS	GEQ
Greater than	>	GTR	LEQ
Not equal to	`!=`	NEQ	EQL
Less than or equal to	<=	LEQ	GTR
Greater than or equal to	>=	GEQ	LSS

[a] Unary tokens have no opposite, so they are removed in a mutant

KLOC for testing. Just compiling the application using Go 1.6.2 takes approximately 180 seconds.

In order to determine if *TuranGo* would be beneficial during CI, we select an application that is on the smaller side, but still has a good ratio of code to tests. For this set of experiments, we choose *plot* [127], a Go plotting application composed of 15 KLOC in 14 packages with 3 test KLOC. Running a baseline test in all packages takes 1.1 seconds in total.

After baseline testing (with coverage) of each package within *plot*, we run *TuranGo* on each package that contains tests and record the results which are presented in Table 7.6.

The overall testing time increases drastically from 1.097s to 9,593s. Of the two main subpackages, *palette* and *plotter*, test coverage is already reasonably high at ≥80%. This makes mutation testing effective, as the high coverage and large codebase produced many mutations. While the total mutation testing time of 2.5 hours is well outside of the normal time cycle of ∼30 minutes for CI, the majority of the mutation testing time takes place in the *plot/plotter* package with over 1200 mutations. A higher individual test baseline also contributes to this mutation testing time since the time-out baseline is a CPU multiple of that initial baseline. That high time-out combined with the large number of mutations results in the over 2 hours and 39 minutes in testing.

While testing an entire application is one use case for CI, many developers work on single packages that do not belong to any one application in particular. To ensure that we see the larger perspective of CI during development, we sample selected open-source Go packages, along with some Go standard packages for CI simulation with *TuranGo*. Results of those mutation tests are in shown Table 7.7.

Testing the sampled open-source packages resulted in much lower numbers. While *jessevdk/go-flags* takes only 0.27s to baseline test, but a staggering 5467s to mutation test, that package includes 3 KLOC of testing code and run through 1645 mutations. The rest of the experiment shows that testing time increases, but is still manageable compared to test frameworks from other languages.

TABLE 7.6. *gonum/plot* Packages and the Test Results of *TuranGo*

Packages	**Baseline**[a]	**KLOC**	**Coverage**	**Mutations**	**Score**	**Time**[a]
gonum/plot	0.030s	1.0	15%	464	100%	609s
../gob	0.035s	0.14	100%	20	0%	54s
../palette	0.127s	5.4	90%	115	84%	540s
../plotter	0.536s	4.3	80%	1219	66%	6180s
../plotutil	0.016s	0.58	1%	12	83%	59s
../vg	0.258s	1.8	79%	92	0%	392s
../vg/draw	0.016s	0.53	12%	303	7%	950s
../vg/recorder	0.032s	0.38	87%	48	100%	112s
../vg/vgimg	0.025s	0.37	48%	125	0%	391s
../vg/vgtex	0.022s	0.28	71%	101	64%	306s
Total	1.097s	14.78	58%	2499	51%	9593s

[a] Three-run average time in seconds

To get real-world experience with Go CI, we enlist the help of a developer at USAA, a major American insurance and financial company. This developer is in charge of a medium size team that is converting all of their internal Node.js and Python backend APIs to Go. On their team they use JIRA for issue and project tracking, and Jenkins for their CI environment. Developers in the team run a mixture of operating systems, from Windows 7 to Ubuntu, and Mac OSX. In the interest of time, they ran *TuranGo* only on packages that had changed between commits. When a developer on the team merged a branch, a Jenkins *git* hook initiated the *TuranGo* testing job on the changed package after the normal package test suite passed. Unfortunately, because of confidentiality on internal non-open source projects, this developer could not provide any empirical numbers during the developer cycle. Instead, he completed a survey after using *TuranGo* for two weeks constituting one Agile sprint.

Evaluation was done on a dual-core Intel Core i7-620M with 8GB memory. An SSD was used for testing, which reduced the I/O wait-time. The host OS was Mac v10.11.4. This

TABLE 7.7. Packages with Baseline *TuranGo* Test Times

Packages	**Baseline**[a]	**KLOC**[b]		**Mutants**	**Score**	**Time**[a]	**Time Q**
			Go standard library packages				
image/color	0.095s	0.3	(0.1)	311	53%	353s	3.7$k\times$
hash/crc64	0.011s	0.05	(0.07)	28	71%	89s	8.1$k\times$
hash/crc32	0.011s	0.1	(0.1)	77	27%	141s	12.8$k\times$
html	0.013s	2.4	(0.16)	185	74%	271s	20.8$k\times$
log	0.012s	0.2	(0.2)	131	73%	151s	12.6$k\times$
			Open-source packages				
gorilla/websocket	0.287s	2.0	(1.0)	678	62%	2310s	8.0$k\times$
jessevdk/go-flags	0.027s	3.0	(3.0)	1645	77%	5467s	202.5$k\times$
gosuri/uitable	0.012s	0.2	(0.1)	47	83%	72s	6.0$k\times$

[a] Three-run average time.

[b] Test KLOC in parenthesis.

is representative of a normal Go developer's day-to-day environment.

7.2.4.2. Has Go Package Testing Improved Over Time?

To determine if the Go standard packages have increased the completeness of their test suites, we develop an experiment to mutation test the standard packages over releases. we use the latest Go binaries as of writing, 1.6.2, and use the official github.com Go repo [85] for the source code.

The program we write first downloads all the selected Go versions through 'git', and then mutation tests each selected package in that version. The results are saved to JSON for easy parsing into charts or other data manipulation. Since the user must define what packages and versions to test in a JSON configuration file, the program we write is extensible enough to test any standard packages, and any repository tags.

To first see how many versions that we need to test, we use 'git' to list all the tags for

Table 7.8. Sample of Go Packages Selected for Version Testing

Package	Description
compress/gzip	Implements reading and writing of gzip format compressed files.
container/heap	Provides heap operations for any type that implements the interface.
container/list	Implements a doubly linked list.
crypto/aes	Implements AES encryption (formerly Rijndael).
encoding/csv	Reads and writes comma-separated values (CSV) files.
hash/crc64	Implements the 64-bit cyclic redundancy check (CRC-64) checksum.
html	Provides functions for escaping and unescaping HTML text.
image/color	Implements a basic color library.
math/cmplx	Provides constants and mathematical functions for complex numbers.
text/scanner	Provides a scanner and tokenizer for UTF-8-encoded text.

the repository. Releases are tagged “go$VER”, with $VER representing the short form of the Go semantic version. We save all the tags that start with “go” and remove all the ‘beta’ or ‘release candidate’ tags since those are not under consideration in semantic versioning. Previous to Go 1.4, the standard packages were located in ‘*src/pkg*’, and starting with 1.4 they moved to ‘*src*’.

We then run a full test on all the Go standard packages to see if any problems are encountered during the testing phase. There are 139 total packages, of which there are eight packages with no tests all. Testing “*math/big*” takes 446 seconds. It is composed of 310 tests in 14 files. The “*errors*” package is tested in 8 ms, and there are only two tests in one file. Using these testing times as a guide, we decide to mutation test a sample of packages instead of the full standard library. Because of time limitations, we randomly select 28 packages, first removing any packages that takes over 10s to test. A sample of the Go standard packages that we choose to mutate is listed below in Table 7.8.

Once the packages are chosen, we complete the JSON configuration and start the testing phase. Since we are using a larger testing environment, we write this experiment as a producer/consumer pipeline to take advantage of the greater number of CPUs and main memory. We limit the number of workers (consumers) to the number of CPUs minus one to avoid false negatives from mutants erroneously timing out during testing.

The producer loops through each enabled Go version, and `git` clones down the source repository tagged with that Go version. It then loops through each enabled package and passes the version and package name to a worker through a channel.

The workers loop through the job queue channel, pulling messages off, then running *TuranGo* on that Go version and package. Since the workers are independent, they can reside on different CPUs to enable parallel testing and increase the overall throughput. After a worker has tested a unique combination of Go version and package, the JSON results are saved to the output directory under that Go version and package name. The results of the sampled testing are shown in Figure 7.7. A black dotted line shows the average score for each version. Figure7.8 shows the average score and trend over time of the selected packages.

7.2.5. Discussion

7.2.5.1. Would Mutation Testing Be a Good Fit For a CI Pipeline?

As Table 7.6 shows, the overall testing time increased significantly, moving this from a CI tool to something more akin to a integration test or special testing phase, rather than something that is hooked up to run every commit.

TuranGo is a good tool to use during the CI process even though it lengthens the testing process. The immediate feedback with diffs of codes will greatly benefit finding faults in tests, and the test suite in general. While testing time was substantially longer using mutation testing, several factors can mitigate this lengthy testing time.

First, while mutation testing a package may take over 20,000 times slower than a baseline test (see *jessevdk/go-flags*), the mutation testing phase will still be within the acceptable limit for the test phase in CI to provide constructive feedback [71]. In addition, individual

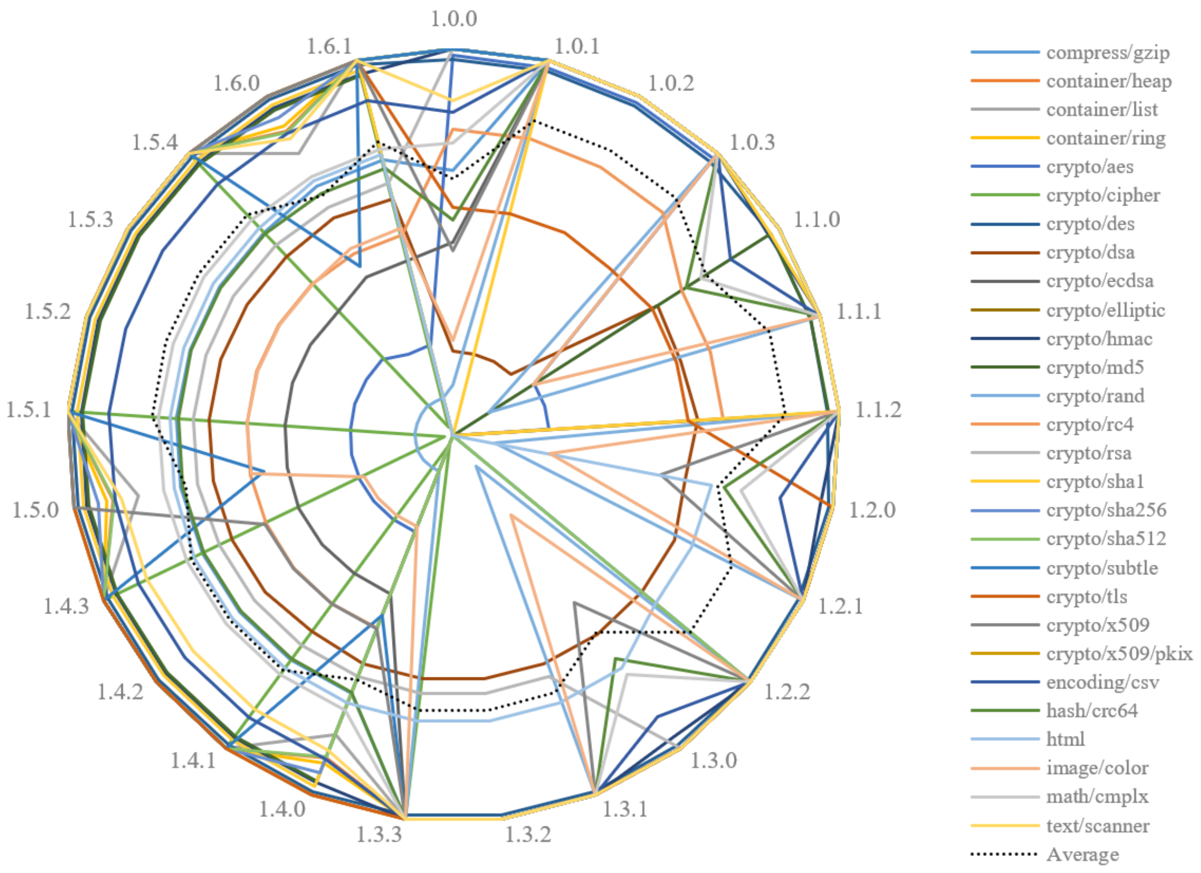


Figure 7.7. Mutation scores for each Go version of the sampled packages

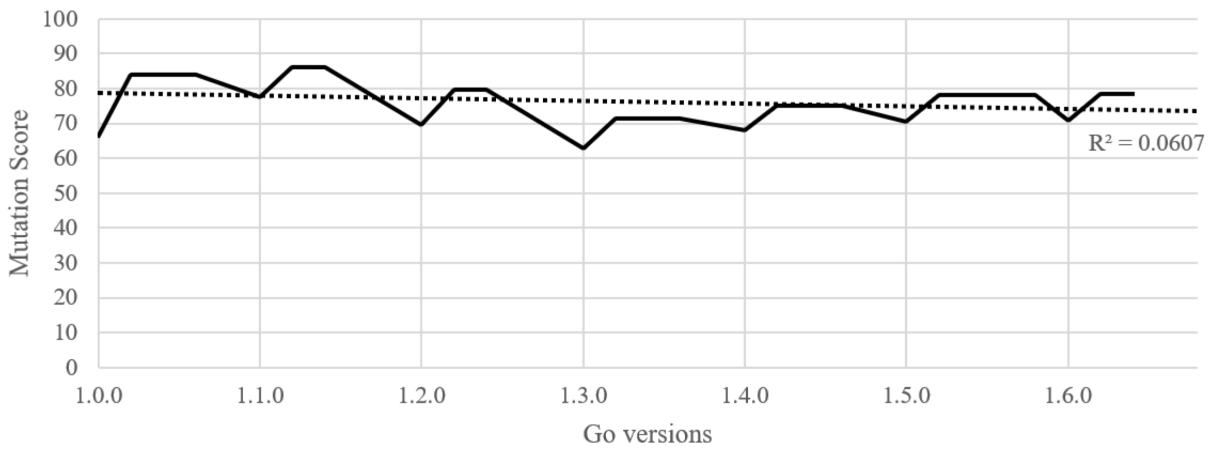


Figure 7.8. Average mutation scores over versions with linear trend line (dashed $R^2 = 0.0607$) of the sampled packages

packages can be mutation tested in parallel, greatly reducing the overall test phase. Go specifically gains from this because the design of Go, encourages small, reusable packages which can be tested independently.

Also, if package tests are one-degree away from the code base, mutation tests are two-degrees away. Since there is no mutation score without testing coverage, the mutation score will always be a lower priority than the code coverage. Then project management would have to dictate what mutation score is acceptable for the QA or production environment. In this regard, package selection could be used to only mutation test packages that have changed, or even further to only mutation test packages with a lower mutation score history or history of breaking builds.

Finally, mutation testing can be viewed as a kind of regression test, where the regressions are actually the coverage of the package's test suite. A lower mutation score does not mean that a build is broken, or even that the test suite did not pass. Actually the opposite is true. So in a CI environment, mutation tests should only be run after the baseline tests have run and no errors have been found. In a real-world example, a developer would only need to run mutation tests before a pulled merge, or an environment promotion.

For Table 7.6, the implications are that the *plot/plotter* package should be broken up into smaller, more mutation-friendly packages. Each source code file within the package is responsible for a different kind of chart or graph. It would be trivial to separate those into individual packages, and retain shared structs/methods/interfaces under the parent *plot/plotter* package. That is only single packages that would need to be tested when a change is made.

The open-source *jessevdk/go-flags* package suffers from the same problem that *plot/-plotter*, namely too much code under one namespace. With 3 KLOC of code under just two packages, main and flags, *jessevdk/go-flags* could be broken up into smaller, more testable packages.

While the developer at USAA could not provide empirical numbers on defects or bugs found, he felt that *TuranGo* enabled the team to get a better perspective on how their testing suites were being developed. Using *TuranGo* he was able to show to his management team their critical backend services were woefully under-tested, and many of the tests they had previously written were of poor quality. Secondly, he used *TuranGo* as a benchmark when choosing between third-party Go libraries that provided the same or similar functionality.

7.2.5.2. Has Go Package Testing Improved Over Time?

As shown by Figure 7.8, the quality of Go standard tests has remained between 60 and 85 since 2012. Even though the results of the experiment show an average downward trend, this only represents a small sample of all the available standard packages. And while there are a few trends among subsets of packages, it seems that the overall trend can be decided by chance since the sample set was small relative to the overall number of standard packages. Note that the average mutation score is calculated by an average of the packages scores, with no weight given to number of mutations or package size.

Among the sampled packages, *crypto/aes* has a strange trajectory. In Figure 7.7, with Go version 1.0 it scores a 98.3%, but then falls to 24.8% for version 1.1. With version 1.2, it falls further to 0%. Starting with version 1.4 it climbs up to 26.8%, which it has remained about till version 1.6, dropping again to 24.1%. This should raise some concern among the Go language contributors, as the *crypto/aes* package is used widely within the security community for systems programming and infrastructure services.

Among *crypto* packages, *crypto/x509/pkix* is a significant outlier with a score of 0% across all Go versions. With 195 lines of code among five functions, that package provides "shared, low level structures used for ASN.1 parsing and serialization of X.509 certificates, CRL and OCSP". Examining the package shows that there are actually no tests written for*pkix*, which is why every mutant lived. Since there are 18 total *crypto* packages, writing simple tests for *pkix* would raise the overall mutation score of the *crypto* group almost 5%.

It is evident that every minor Go release has a severe mutation regression, driving down the average score. The mutation score rises on the next immediate patch release, so it is prudent to wait for a "dot one" release before updating Go binaries/packages in a production environment. Also, something in the test suites of several packages starting at

Go version 1.4 results in a dismal overall mutation score that only recently starts to increase. Two design decisions at 1.4 could account for those lower scores:

(1) Moving away from the C compiler entirely, forcing optimization in pure Go
(2) Private unexported packages could be relocated into an *internal* package namespace, separate from the main package tree

7.2.6. Conclusions

Although not by design, Go is a great language for employing mutation testing. While mutation testing is not widely used due to expense of both computation and time, Go's quick compilation, built-in testing framework, and smaller, modular packaging, mitigates the main reason mutation testing is not widely used, namely the expense of time. For CI, TuranGo provides many benefits without slowly down individual developer feedback, since it should only be run during branch merges.

It is disappointing to observe that Go standard package tests have not improved over time. However, these results are not statistically significant due to the small sample size. The scores should be viewed as interesting assessment on what has changed in Go over the minor revisions since the 1.0.0 release.

As future work, we plan to rewrite *TuranGo* in a producer/consumer architecture to reduce the disk I/O required during mutation testing. One of the major bottlenecks during our performance testing is having to waiting on I/O while iterating writing mutants to disk. Pipelining these operations will enable concurrency during the testing phase and parallelism on systems with more than one CPU. On UNIX systems there is also a possibility to use a memory-based filesystem to speed up large tests. Additionally, there are many planned features, for example, completely in memory difference engine for source file comparisons, adding coverage information through cover out and checking line differences, enabling/disabling mutators on the command line or configuration file, blacklisting code sections that should not be mutation tested or is known equivalents, Jenkins integration utilizing Pitest's CI plugin, and smarter test picking ala Mothra [161].

7.3. Embedded Deep Learning for Vehicular Edge Computing [105][2]

The rapid development of deep learning has greatly increased the accuracy of object recognition, but this deep learning generally requires a massive amount of training data and the training process is very slow and complex. The complex models generated from this training can also require more CPU and RAM during inference than resource-constrained embedded systems are capable of. The following work performs benchmarks on such a system, a Raspberry Pi® 3 Model B, diagrammed in Figure 7.9 below.

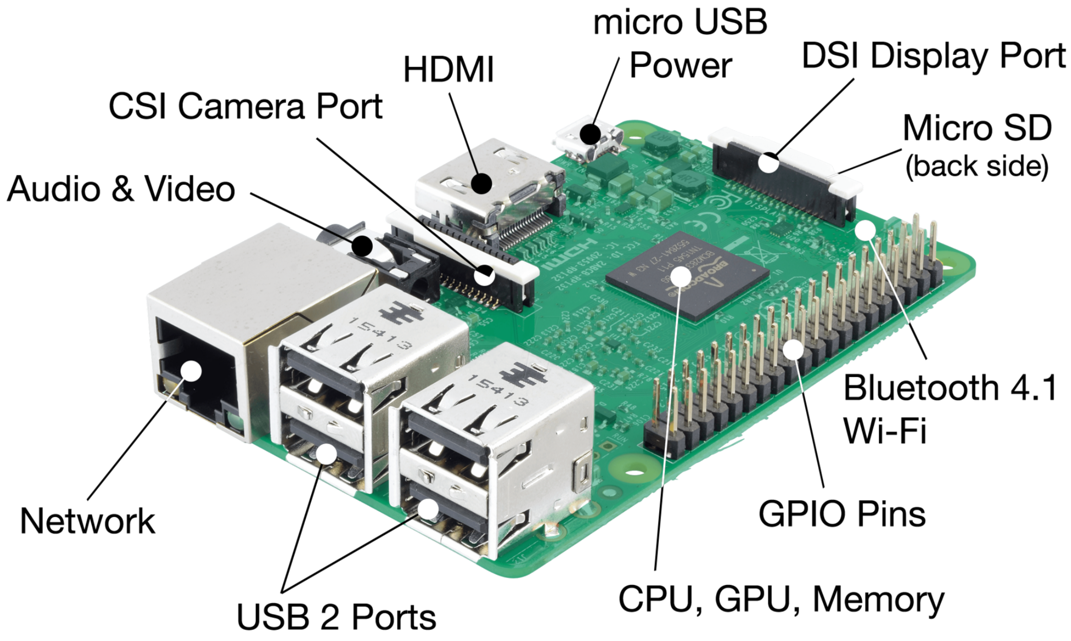


FIGURE 7.9. The Raspberry Pi® 3 Model B used in my experiments contains a 1.2 GHz 64-bit quad-core Cortex-A53 (ARMv8) CPU, 1 GB of low-power DDR2 SDRAM and four USB 2.0 ports via on-board 5-port USB hub. It draws a maximum of 6.7 W at peak load.[177]

This work includes benchmarks with a novel class of inference hardware, an Intel Movidius™ Neural Compute Stick, shown in Figure 7.10. This specialized Vision Processing Unit is used to analyze the objects in the real time images and videos for vehicular edge

[2] Section 7.3 is reproduced in its entirety from Jacob Hochstetler, Rahul Padidela, Qi Chen, Qing Yang, and Song Fu, Embedded deep learning for vehicular edge computing, 2018 IEEE/ACM Symposium on Edge Computing (SEC), pp. 341-343, with permission from IEEE.

computing. Distinct from a more generalized Graphics Processing Unit, this VPU is designed for low-power applications such as 32-bit embedded edge nodes and even drones with USB capability. The results shown in this study explains how the stick performs in conjunction with different operating systems and processing power.

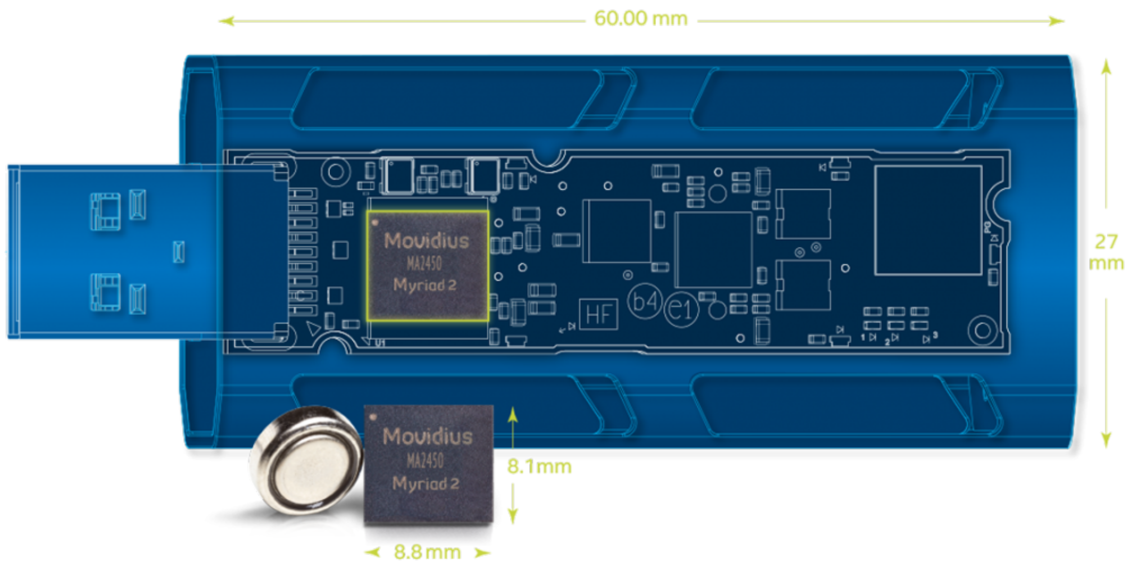


FIGURE 7.10. Intel® Movidius™ Neural Compute Stick (NCS)is a tiny fanless, USB 3.0 Type-A deep learning device containing a Myriad 2 Vision Processing Unit producing almost 100 GFLOPS while only using 1 W of power.[114]

7.3.1. Introduction

The need for low-power but capable inference processors has produced a new class of computing called edge, or sometimes "fog" computing. These stand-alone, specialized edge-computing devices has become more and more popular for three main reasons: lower network delay, energy efficiency, and better privacy protection. The promise of edge computing is that by processing data at the network edge would result in shorter response time, more efficient processing, and less congestion on the network [198].

Connected and Autonomous Vehicles (CAV) are a new class of vehicles that can both communicate with each other or infrastructure (CV) and employ a number of systems that enable automated driving functions (AV). The Society of Automotive Engineers (SAE) has created a six-level hierarchy to categorize AV's capabilities [8], ranging from no capability at all to full automation with no driver engagement.

The CV's communication capability is also categorized as Vehicle to Infrastructure (V2I), Vehicle to Vehicle (V2V), Vehicle to Cloud (V2C), Vehicle to Pedestrian (V2P) or the all-encompassing Vehicle to Everything (V2X).

Artificial neural networks (ANNs), or connectionist systems, are computing systems vaguely inspired by the biological neural networks (NN) that constitute animal brains. Such systems "learn" (i.e. progressively improve performance on) tasks by considering examples, generally without task-specific programming. For example, in image recognition, they might learn to identify images that contain cars by analyzing training images that have been manually labeled as "car" or "not car" and use those results to identify cars in future, unseen images. This was demonstrated in the TV show *Silicon Valley* by a character's SeeFood mobile app, which determined through an ANN if photos contained "hotdog or not hotdog" [169].

This work presents benchmarks from deep learning networks running on an edge computing node assisted by a Vision Processing Unit (VPU). Our results show that the a mobile edge device assisted by a VPU is able to process video using a popular NN in real-time. This is important for a CAV, since it leaves the main CPU and memory free for V2X communication or other tasks.

7.3.2. Mobile Edge Computing Setup

7.3.2.1. Mobile Edge Device

The Raspberry Pi (RPi) is a small, single-board computer (SBC) [177]. Each RPi is based around a Broadcom system on a chip (SoC), which contains the ARM CPU, RAM, GPU and general purpose input/output controllers (GPIO). Originally intended to just be used in teaching computer science and basic embedded computing, but through three different models and several iterations, has become widely successful in home automation, industrial (edge) computing and packaged commercial products.

The RPi3B used in our experiments contains a 1.2 GHz 64-bit quad-core Cortex-A53 (ARMv8) CPU, 1 GB of low-power DDR2 SDRAM and four USB 2.0 ports via on-board 5-port USB hub. It draws a maximum of 6.7 W at peak load.

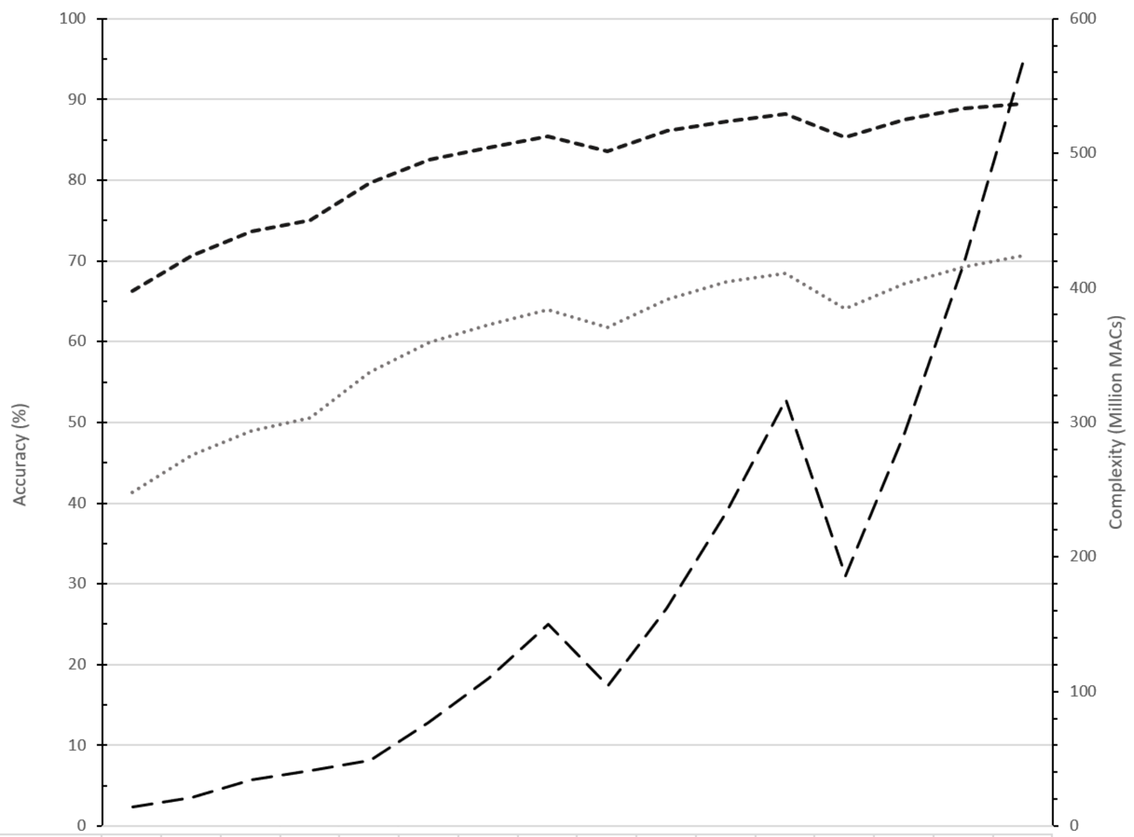
7.3.2.2. Embedded Deep Learning Device

The Intel® Movidius™ [114] Neural Compute Stick (NCS) is a tiny fanless, USB 3.0 Type-A deep learning device that can be used to learn AI programming at the edge. NCS is powered by the same low-power, high-performance Myriad 2 VPU) that can be found in smart security cameras, gesture-controlled drones, industrial machine vision equipment, and other embedded systems. Ubuntu 16.04 is supported installed on a physical x86_64 system, or Debian Stretch running on a Raspberry Pi 3 Model B. The Neural Compute SDK comes with a C++ and Python (2.7/3.5) API [113].

The Myriad 2 VPU within the NCS produces almost 100 GFLOPS using only 1 W of power and generates between 10 to 15 inferences per second. The VPU includes 4 GB of low-power DDR3 DRAM, imaging and vision accelerators, and an array of 12 very long instruction word (VLIW) vector *SHAVE* processors. The entire NCS draws 2.5 W of peak power through its USB 3.0 port. A trained Caffe-based or TensorFlow™ CNN is compiled into an embedded neural network that is optimized to run on the VPU inside the NCS.

7.3.3. Experimental Results

Google's MobileNets [109] is a Convolutional Neural Network (CNN) that uses depthwise separable convolutions to build light weight deep neural networks. MobileNets is tunable by two hyper-parameters (e.g., size and depth), so that a less powerful embedded system can trade accuracy for model latency (i.e., time to result). The Top-1 and Top-5 accuracy in Figure 7.11 is measured against the Large Scale Visual Recognition Challenge [190]. The



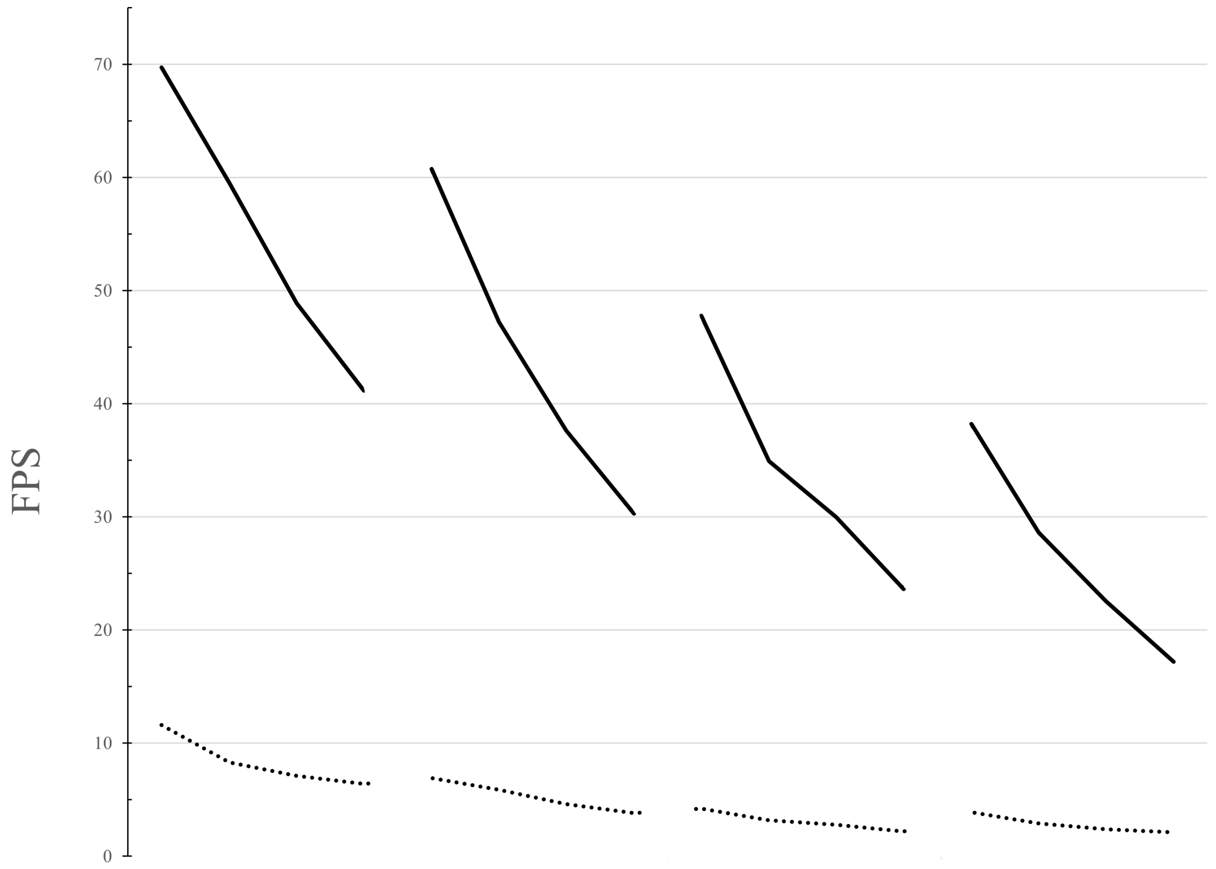
Size	128	160	192	224	128	160	192	224	128	160	192	224	128	160	192	224
Depth	0.25	0.25	0.25	0.25	0.50	0.50	0.50	0.50	0.75	0.75	0.75	0.75	1.00	1.00	1.00	1.00
······ Top-1 Accuracy	41.3	46	49	50.6	56.2	59.9	62.1	64	61.8	65.2	67.4	68.4	64.1	67.2	69.3	70.7
- - - Top-5 Accuracy	66.2	70.7	73.6	75	79.6	82.5	84	85.4	83.6	86.1	87.3	88.2	85.3	87.5	88.9	89.5
— — Million MACs	14	21	34	41	49	77	110	150	104	162	233	317	186	291	418	569

Figure 7.11. Google's MobileNets Accuracy and Complexity

results from benchmarking MobileNets on a bare RPi3B and assisted with an NCS are shown in Figure 7.12 below.

The next two experiments, depicted in Figure 7.13, compare a RPi3B to a standalone Ubuntu Desktop. The Ubuntu 16.04 LTS system used for comparison is powered by a 3.06 GHz dual-core Intel® E7600 CPU, 4 GB of DDR3/1066 MHz RAM. The Linux kernel used during testing was 4.15.0-23.

The first experiment used `video_objects.py` from the Movidius™ `ncappzoo/apps`, along with the six example videos used for testing and training. The desktop achieved 9.3 frames-per-second (FPS) with a single NCS attached while the RPi3B produced 5.7 FPS. The



Size	128	160	192	224	128	160	192	224	128	160	192	224	128	160	192	224
Depth	0.25	0.25	0.25	0.25	0.50	0.50	0.50	0.50	0.75	0.75	0.75	0.75	1.00	1.00	1.00	1.00
⋯ Pi3	11.60	8.30	7.10	6.40	6.90	5.90	4.60	3.80	4.20	3.20	2.80	2.20	3.90	2.90	2.40	2.10
— Pi3 w/NCS	69.71	59.52	48.89	41.15	60.77	47.22	37.61	30.27	47.77	34.94	29.95	23.61	38.20	28.62	22.53	17.21
NCS Speedup	601%	717%	689%	643%	881%	800%	818%	797%	1137%	1092%	1070%	1073%	979%	987%	939%	820%

FIGURE 7.12. MobileNets RPi3B (FPS): CPU only vs. 1 x NCS

second experiment used the GoogLeNet [206] CNN classifier `street_cam.py` with example videos. With two sticks the desktop produced 6.6 FPS, while the RPi3B produced 3.5 FPS.

7.3.4. Discussion

In addition to a lower clock speed, the RPi3B only has a 60 MBps USB 2.0 bus, which is ten times slower than the theoretical maximum of a USB 3.0 bus. This greatly limits the speed that data can flow into the NCS from the host RPi3B.

KITTI is one of the main open resources for training CAV models and all the examples are synchronized at 10 Hz (cameras, lidar, and GPS) [77]. The bare RPi3B is not able to keep up with real-time processing when the depth and size increases in MobileNets. In

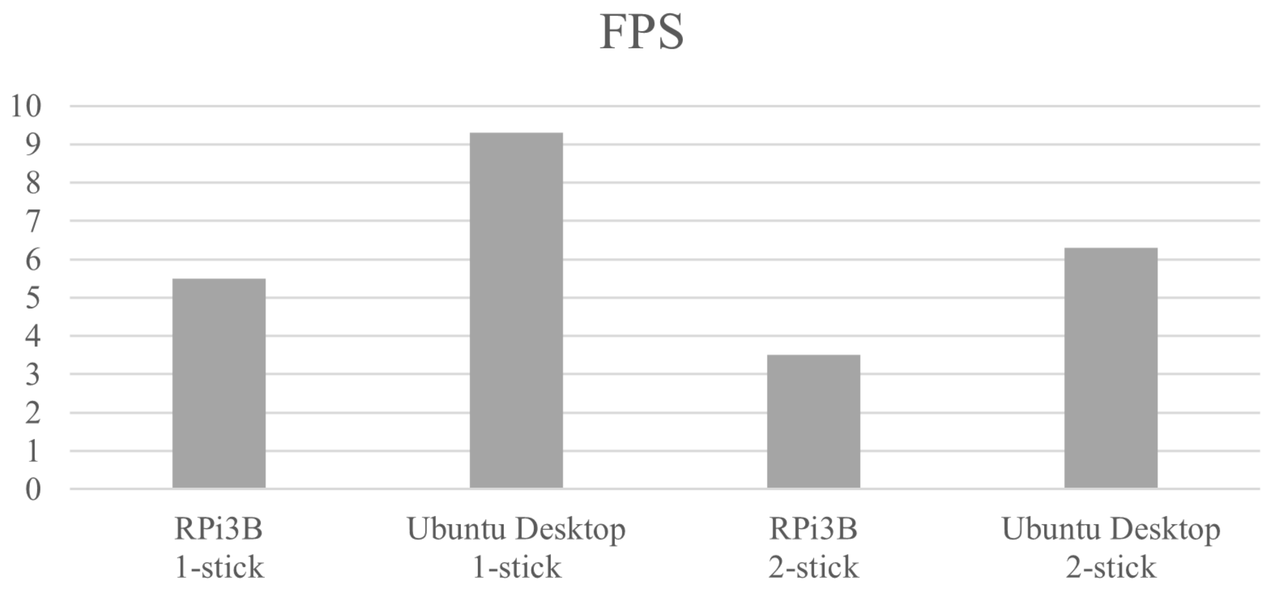


Figure 7.13. Frames per second (FPS) platform comparison running 1 or 2-sticks.

contrast, a single NCS can process a single 10 Hz feed in real-time at the largest size and depth, producing the most accurate results. Since each NCS is uniquely indexed, different feeds can be sent to different sticks for independent processing, or independent networks can be loaded on each stick for processing. Lab tests at Movidius™ show linear performance increases up to 4 sticks, with validation pending for 6 to 8-stick configurations.

Neither the desktop, nor the RPi3B were able to process `video_objects.py` or `street_cam.py` in real-time, which means that the models used could be further optimized to produce better results on either platform.

7.3.5. Conclusions

The results show that the RPi3B is capable of processing real time video and recognizing objects using the embedded deep learning device. Multiple sensor feeds, which are prevalent in CAVs, can be processed independently by different sticks. This frees the edge computing device's CPU and memory for other tasks, like real-time diagnostics, V2X communication or Advanced Driver-Assistant Systems.

7.4. Low-Latency High-Level Data Sharing for Connected and Autonomous Vehicular Networks [42][3]

Autonomous vehicles can combine their own data with that of other vehicles to enhance their perceptive ability, and thus improve detection accuracy and driving safety. Data sharing among autonomous vehicles, however, is a challenging problem due to the sheer volume of data generated by various types of sensors on the vehicles. In this paper, we propose a low-latency, high-level (L3) data sharing protocol for connected and autonomous vehicular networks. Based on the L3 protocol, sensing results generated by individual vehicles will be broadcasted simultaneously within a limited sensing zone. The L3 protocol reduces the networking latency by taking advantage of the capture effect of the wireless transmissions occurred among vehicles. With the proposed design principles, we implement and test the L3 protocol in a simulated environment. Simulation results demonstrate that the L3 protocol is able to achieve reliable and fast data sharing among autonomous vehicles.

7.4.1. Introduction

For years, the development of connected and autonomous vehicles (CAV) technology has garnered significant interest from both research institutes and industry alike. CAV incorporate a variety of different technologies, ranging from computer vision [78] to wireless networking [243], to facilitate a safe and efficient movement of people and goods, revolutionizing the current transportation system. It brings a host of benefits such as improved safety, convenient mobility for the elderly and disabled, and a better public transportation system [95]. Ideally, CAV could help drive fatalities to near zero, given the technologies continue to improve.

Autonomous vehicles are typically equipped with high-precision sensing systems, producing a healthy amount of sensor data that need to be processed in real time. For example, the autonomous vehicles developed by companies such as Google, Tesla, Mobileye autopilot

[3] Section 7.4 is reproduced in its entirety from Qi Chen, Shihai Tang, Jacob Hochstetler, Jingda Guo, Yuan Li, Jinbo Xiong, Qing Yang, and Song Fu, Low-latency high-level data sharing for connected and autonomous vehicular networks, 2019 IEEE International Conference on Industrial Internet (ICII), pp. 287-296, with permission from IEEE.

systems, are mainly equipped with LiDAR (light detection and ranging), camera, Radar, ultrasound, thermal camera, GPS, and IMU (inertial measurement unit), etc. Currently, the data generated by these sensors are processed and stored locally on individual vehicles, and rarely shared among autonomous vehicles. The current solutions, however, come with several limitations. When driving in the evening, rain, snow, fog or other bad weather conditions, cameras may not work properly. Similarly with LiDAR and Radar sensors, intersections, turning corners and other scenarios may witness the sensing systems not functioning properly. For example, Tesla autonomous car once had a fatal accident on a freeway. The vehicle failed to identify the white body of a truck under an intense sunshine condition, and therefore did not activate the brake system in time. While developing more powerful sensors may solve these issues, the associated cost will rise to the point where consumers cannot afford the product, i.e., individual customers aren't likely to see such vehicles in much volume.

A possible solution to the above mentioned issues is to allow autonomous vehicles to exchange real-time sensing data to each other, realizing a connected and autonomous vehicular system. Although data sharing among vehicles is promising, it faces several challenges that must be adequately addressed before the technology is deployed in real world. These challenges are related to both data processing and data sharing, e.g., it is difficult to synchronize the sensing data among vehicles, the networking bandwidth of existing wireless technologies is too limited to transmit the data, the large networking delay may be prohibitive for autonomous driving applications. In summary, without efficient data processing and transmission mechanisms, the sheer amount of resources that will be consumed by autonomous vehicles can dramatically slow the deployment of CAV technologies.

7.4.1.1. Motivations

To achieve data sharing among autonomous vehicles, an effective way is to adopt the V2X communications [245]. Intuitively, V2X communication connects cars to other cars or the Internet to form a vehicular networks, including V2V (vehicle to vehicle), V2I (vehicle to infrastructure), V2N (vehicle to Internet) and V2P (vehicle to people) communications. V2X communication can be viewed as a means that allows the sensors on vehicles to extend their sensing range well beyond what they are physically capable of. By sharing the sensing results with nearby vehicles and roadside infrastructures, vehicles can greatly enhance the perception of the surrounding environment and thus enhancing their decision making. Even though the self-driving function can be partially achieved by the vehicle itself, using V2X can further improve safety and driving performance by reducing the cost of deploying high-precision sensors. In addition to improving its own perception and decision making, the enabled autonomous vehicle can also improve the driving reliability for the normal human operated vehicles, making it more encouraging for the adoption of more vehicles equipping V2X devices.

7.4.1.2. Challenges

The challenge of a CAV system comes from the massive deployment of sensors on the autonomous vehicles and the huge amount of data that they can pick up from the environment [247]. First of all, it is challenging to synchronize and fuse data generated from different vehicles which may use different sensors (and algorithms) to perceive the surrounding environment. Data fusion is the process of combining multiple vehicles' data to produce a more consistent, accurate, and reliable perception than what is offered by an individual vehicle [97]. The data fusion process in CAV is usually classified into three categories: low-level, intermediate-level, and high-level fusions, depending on the processing stage at which the fusion takes place [125]. As their names imply, low-level fusion refers to raw data fusion, which requires the highest network bandwidth to transmit the data. Intermediate-level fusion, such as feature-based fusion, takes the features extracted from the raw data before fusion. Finally, high-level fusion takes the processing results, e.g., the objects detected from cameras, to conduct the fusion. For wireless vehicular networks, regardless which type of fusion is adopted, the large amount of data generated and shared among vehicles will pose significant research challenges to existing wireless technologies, e.g., dedicated shorte range communication (DSRC) [124] and 5G networks.

7.4.1.3. Proposed Solution

To facilitate data sharing among autonomous vehicles, high-level fusion is often opted over the other two levels of fusion, due to the small amount of data exchanged between vehicles. In this way, each vehicle processes its sensing data locally and exchange its sensing results with nearby vehicles. As long as the format of the sensing results are standardized, it does not matter what sensing technologies individual vehicles adopt. In this paper, we focus on the object detection results generated by the perception system on autonomous vehicles. The object detection function is itself one of key components for autonomous vehicles, as it allows a vehicle to account for obstacles when considering possible moving trajectories. The object detection results of a vehicle are represented in a sensing matrix, which provides an overview of objects existing in the vehicle's surrounding environment. Each vehicle will broadcast its sensing matrix to nearby vehicles to achieve a high-level data sharing in CAV systems. To mitigate the potential collision of packets simultaneously transmitted by multiple vehicles, a low-latency data sharing mechanism is designed, leveraging the capture effect that is widely observed in various wireless communication techniques. With the above design principles, we propose a Low-Latency and high-Level (L3) data sharing protocol for connected and autonomous vehicles.

7.4.1.4. Contributions

Inaccurate object detection and recognition are major impediments in achieving a powerful and effective perception system on autonomous vehicles. To address these issues, we propose the L3 protocol in which an autonomous vehicle combines its own sensing data with that of other vehicles to help enhance its own perception. We believe that data redundancy, as mentioned, is the solution to this problem and we can achieve it through data sharing and combination between autonomous vehicles. The L3 protocol is effective and efficient, which can improve the detection performance and driving experience thus providing better safety. Specifically, we make the following contributions. We propose the mechanism to divide a digital map into sensing zone and vehicles will only exchange sensing data about one zone in which it current resides, leading to scalable data sharing. The object detection results on each vehicle are represented in a sensing matrix, which facilitates a quick information sharing among vehicle, leveraging the capture effect. The proposed L3 protocol is evaluated in simulations and it significantly outperforms existing solutions, i.e., offering a lower network latency in sharing data among vehicles.

7.4.2. Data Sharing for Connected and Autonomous Vehicles

To design an efficient data sharing protocol for connected autonomous vehicle, it is necessary to first understand the characteristics of data produced by various sensors on autonomous vehicles. In this section, we investigate the mismatch between the huge amount of data generated from autonomous vehicles and the limited network bandwidth available for vehicular communications.

7.4.2.1. Characteristics of Data on Autonomous Vehicles

Autonomous vehicle is typically equipped with various types of sensors to obtain fine-grained, real-time and accurate information about its surrounding driving environments. The perception system on an autonomous vehicle usually consists of several LiDAR, Radar, camera sensors, ultrasonic sensors, GPS and IMU (inertial measurement unit) sensors. Through these sensors, sheer amount of data will be generated and these data must be processed in real time. Each autonomous vehicle will collect almost 4,000 Gigabytes of data per day, according to [194]. A LiDAR sensor, e.g., Velodyne LiDAR HDL-64, will generate 9.75 Mbps of data when it scans at 5Hz, and up to 39 Mbps at 20Hz.

LiDAR is an essential component for autonomous vehicles as as it is used to detect dynamic and static objects including other cars or pedestrians in order to navigate around them. LiDAR is also applied to create high-definition maps and achieve high-accuracy localization of autonomous vehicles. Due to the popularity of installing LiDAR sensors on autonomous vehicles, in this paper, we use the point cloud data generated by LiDAR as a case study to illustrate how the proposed L3 protocol works for connected and autonomous vehicles.

To process the data generated by LiDAR sensors, several methods are proposed, e.g., MV3D [44] and VoxelNet [254], to detect the objects existing in point cloud data. Due to the sparsity of LiDAR data, it is quite challenging to accurately detect all objects in point cloud data. Recently, VoxelNet [254] has announced its experiments on the KITTI dataset, i.e., it offers an acceptable object detection performance on LiDAR data. However, its detection accuracy is far from the performance of camera-based solutions. The average car detection precision of VoxelNet is only 81.97%. For smaller objects, e.g., pedestrian and cyclist, its average precision drops to 57.86% and 67.17%, respectively. While in a hard condition, VoxelNet's detection accuracy of car, pedestrian and cyclist further drop to 62.85%, 48.87%, and 45.11%, respectively.

7.4.2.2. Detection Failures on Autonomous Vehicles

Detection failures occur on autonomous vehicles for various reasons, e.g., objects are too far away, low-quality sensing data, errors in object detection algorithms. Therefore, it is critical to share data among autonomous vehicles to achieve cooperative perception wherein vehicles help each other to gain a better perception of their surrounding environments. Leveraging the sensing data provided from other vehicles, an autonomous vehicle can essentially extent its sensing range and enhance its sensing capability, e.g, accurately detect more objects on roads.

Using LiDAR sensor as an example, we illustrate several cases where detection failures could happen on individual vehicles that rely only on their own sensors. As shown in Figure 7.14, we collect LiDAR data on an autonomous vehicle, referred as the sensing vehicle. It stops in front of an intersection and move towards the north direction. In the figure, we identify four major areas (marked with numbers) that are blocked by obstacles around the intersection. The area #1 is totally non-observable as it is completely blocked by the vehicles moving along the west-east direction. These vehicles are indicated by green boxes, and we can see there is a big truck blocking the sight of our autonomous vehicle. Similar situations can be found in area #2, in which a car (marked as yellow) is located in front of the sensing vehicle. The relatively-large objects (e.g., buildings and trees) are the root cause of the blocked areas #3 and #4. To improve the perception capability of the sensing vehicle, it would be beneficial to let other vehicles share what they sense. For example, the green-boxed vehicle(s) can help provide information within areas #1, #3 and #4. The objects in area #2 could be detected by the yellow-boxed car, and the information can be shared with the sensing vehicle.

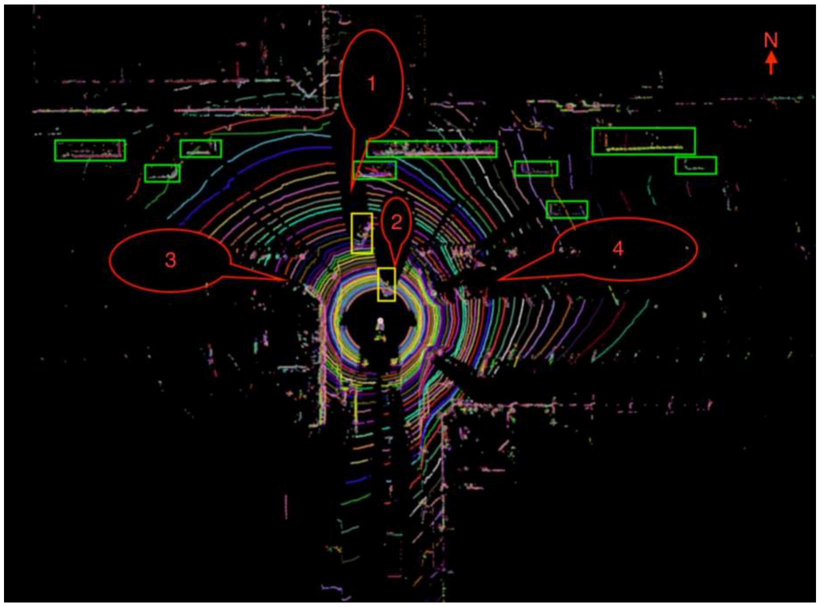
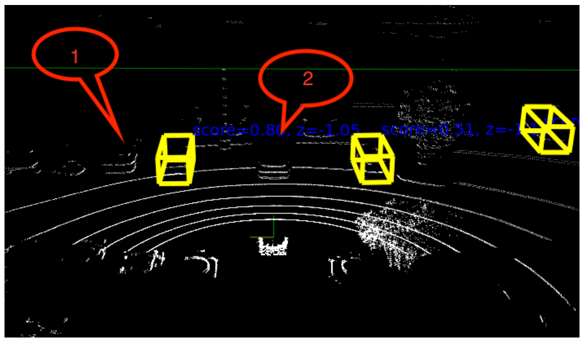


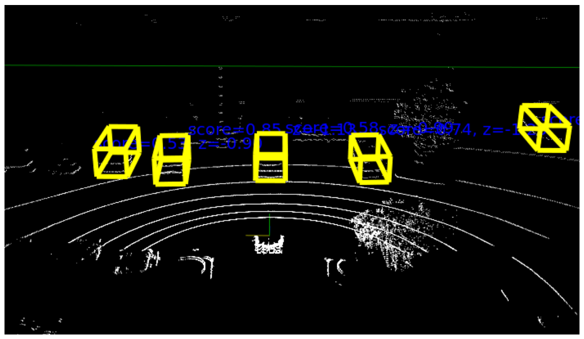
FIGURE 7.14. Detection failures occur on individual vehicles because objects are blocked or exist in the blind zones of sensors.

Detection failures on autonomous vehicles could also happen due to bad recognition, e.g., sensing data is too weak or is missing. As shown in Figure 7.15, we provide several driving scenarios that are recorded in the T&J dataset [43]. These are the LiDAR data of an autonomous vehicle, with each key frame being a time step forward from the previous scanning position as the vehicles is moving along a straight path. Detected objects/vehicles are marked by yellow boxes, including driving and parking vehicles. As we can see in Figure 7.15, in the top two frames, we have vehicle #1 detecting only three objects. When the vehicle moves forward, however, it is able to detect previous undetected objects. The same happens with vehicle #2, except in this case, the objects are hidden from the LiDAR sensor in the previous time step by other objects in the way.

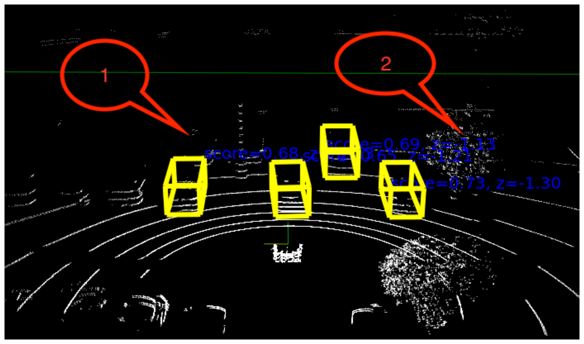
We can imagine how dangerous this situation is should the vehicle still be blind to those regions in its path. Should the vehicle not detect moving objects on a collusion course with itself, then an accident is bound to happen. Similarly, detection failures can also contribute to the same situation should the camera sensors being obstructed in bad weather conditions. However, this problem could be fixed if nearby vehicles exchanged sensing information.



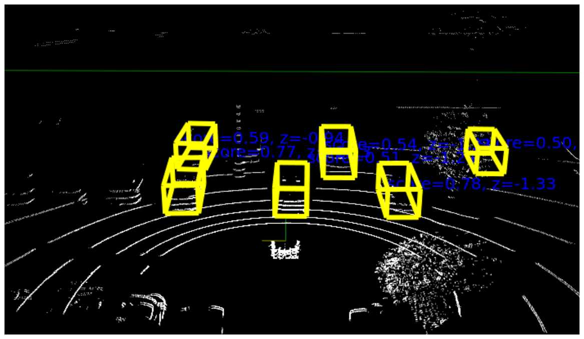
(A) Detected by vehicle #1 at time t.



(B) Detected by vehicle #1 at $t + 5\,\text{s}$.



(C) Detected by vehicle #2 at t.



(D) Detected by vehicle #2 at $t + 5\,\text{s}$.

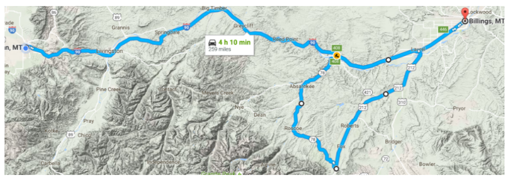
FIGURE 7.15. Objects (outlined as yellow boxes) detected by different vehicles at different time instances.

7.4.2.3. Challenges in Vehicular Networks

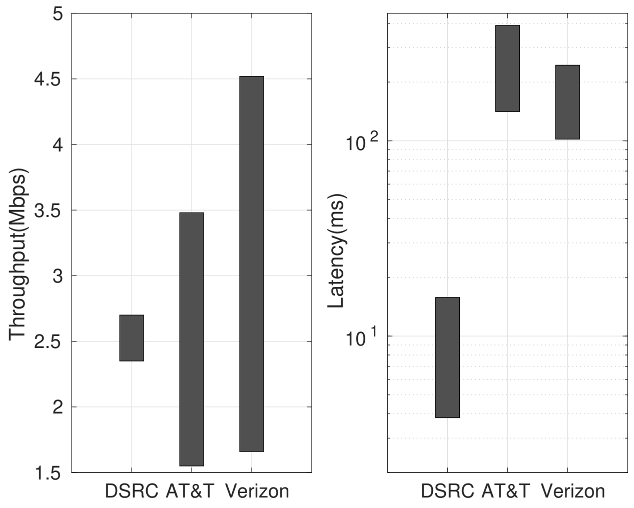
It is impractical to directly transmit the raw sensing data through current known wireless networks available to autonomous vehicles. Optimally, the detected objects are labeled with detailed sensing information before transmitting out. The information of detected objects should include when, where and which kind of object is detected by what sensor on what vehicle, along with what size of this object and its movement conditions. We can image it will be still challenging even if flooding the high-level object detection results in a high frequency among vehicles.

According to the research work [41], tested vehicles are communicating using DSRC (Dedicated Short Range Communications) and Cellular networks along the Interstate highway I-90 in the Montana state, USA. It is found that the DSRC throughput between two moving vehicles is less than 3 Mbps when the BPSK modulation technique is applied. As for the cellular network performance, using Verizon and AT&T carriers, it is shown that the LTE network can support up to 4.5 Mbps throughput, and 3G network only offer < 2 Mbps throughput. In summary, none of existing wireless network technology would support the high-level data sharing among autonomous vehicles.

Another technical challenge lie in vehicular networks is the large network latency introduced by transmitting the sensing data among vehicles. For V2X communications, especially in V2V communications, low latency is required because of the high mobility of autonomous vehicles. As shown in Figure 7.16, hundreds of millisecond delay is observed for cellular networks, while a substantially lower delay for DSRC.



(A) Measurement routes along the I-90 freeway in USA.



(B) Throughput and latency of DSRC and cellular networks.

FIGURE 7.16. Field tests of V2X networking technologies.

7.4.3. L3 Data Sharing Protocol for Connected and Autonomous Vehicles

The huge amount of data can become impractical to transmit over any existing wireless networks due to unacceptable network latency and limited, especially in a mobile environment with a large number of vehicles. In order to develop the low-latency, high-level (L3) data sharing protocol, the high-volume of data must be reduced appropriately. While the amount of to-be-shared data is reduced, we must guarantee the useful information obtained from the raw sensing date is still kept. Another challenge in a CAV system is that what vehicles share may not be trustworthy [244, 45], which is an important issue but out of the scope of this paper. Therefore, we assume here that all data exchanged between vehicles are trustworthy, although detection errors may exist in the data.

7.4.3.1. High-Level Data Sharing

Based on the data shared from others, a vehicle must be able to extend its sensing range or increase its sensing capability; otherwise, the data transmission would be useless and should be omitted. For example, blocked areas behind obstacles on the road could not be sensed, while this can be filled in by collecting "unseen" information from others. Meanwhile, vehicles in adjacent districts or crowded areas can keep their connections for a longer duration, hence data sharing can greatly help them get more useful information. In summary, complementary data are always the most valuable information to share among vehicles.

7.4.3.2. Sensing Zones on Digital Map

Generally, letting every vehicle report all observations they make would provide enough information for object detection. However, this is not the case as doing so would transmit an enormous amount of redundant data. For example, in crowded areas, many vehicles may transmit a slight variant of the same information. The effectiveness drops rapidly as redundancy increases. To address this issue, we compress sensing results into small data packets to reduce the network traffic.

We introduce an approach to position every vehicle into a zone pre-indexed on a digital map. As shown in Figure 7.17, a digital map is divided into equal-sized zones, depicted as groups of red and blue blocks. Depending on the sizing choices, the sensing area of a particular sensor could be a few zones or dozens of blocks. For a particular vehicle, it will be located in only one zone. Should it occupied two adjacent zones, the one it most recently touched is considered the zone where it resides in. For this vehicle, the information of objects within its zone becomes more important than those from other zones.

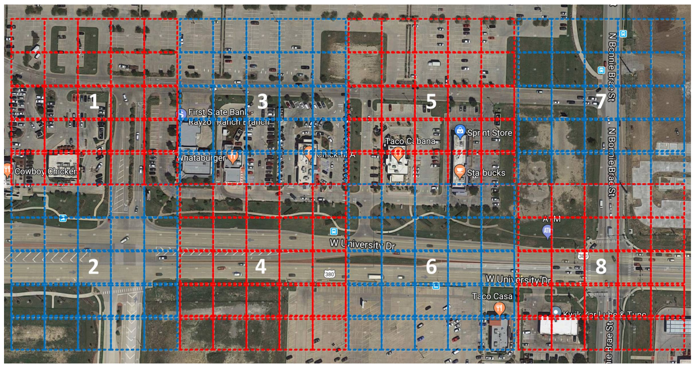


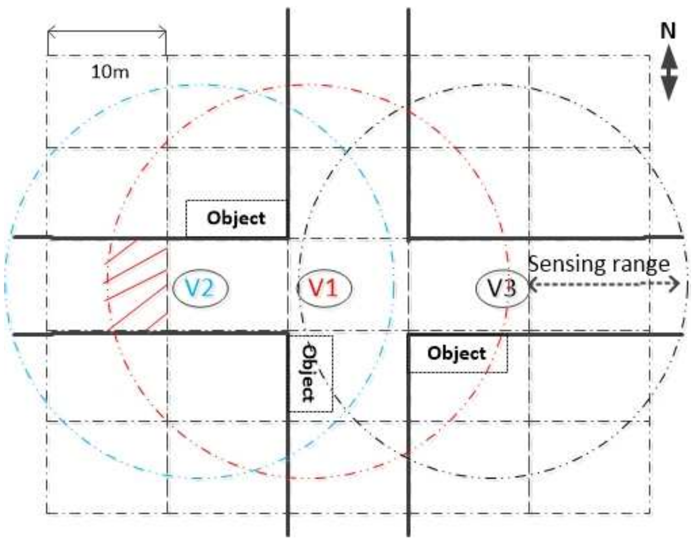
FIGURE 7.17. A digital map is divided into equal-sized zones and blocks so that each vehicle is positioned into one zone and detected objects are placed into blocks.

7.4.3.3. Sensing Blocks Within a Zone

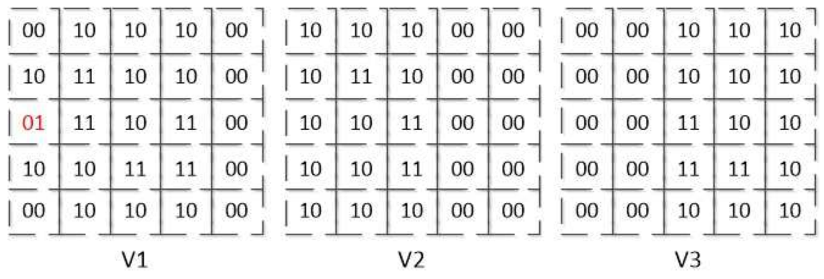
While vehicles move on roads, each one of them will locates itself (e.g., 10Hz) into one zone based on its current location informed by its GPS sensor. As each zone is assigned an index in the pre-installed digital map that is available to all vehicles, vehicles in the same zone would share information to each other. When a vehicle is moving on the road, from its sensing data, it can detect various objects, including pedestrians, cars, motorcycles, and bicycles. These objects are then labeled with their locations and size information. The smaller the blocks, the more details about the objects, and thus the larger communication overhead on the vehicular network.

Once a vehicle enters an indexed zone, it maps its sensing information (i.e., the object detection results) to corresponding blocks. If a block is occupied by a detected object, the location of this blocked will be marked as object detected. Otherwise, there is either no object in the block or the vehicle is uncertain about whether an object exists in the block. It worth noting that in some case a block may be out of the sensing range of a vehicle, and this case must be considered when we encode the information of each block. To illustrate the concept of high-level sensing data sharing among autonomous vehicles, we only consider one type of objects, e.g., cars, detected by a vehicle. For the blocks within the zone of this particular vehicle, four possible values will be assigned: *No object*, *Objected detected*, *Out of sensing*, and *Uncertain*.

In this way, the value of each block could be represented by two bits, denoted as $b_1 b_0$. Here, b_1 indicates whether the block is sensed (1) or not (0). If $b_1 = 1$, b_0 indicates whether objects exist (1) or not (0). When $b_1 = 0$, b_0 presents whether this block is blocked (1) or out of sensing range (0). In summary, we can assign 10 (*No object*), 11 (*Objects detected*), 00 (*Out of sensing*) and 01 (*Uncertain*) four possible values to each block within the sensing area of a vehicle.



(A) Three vehicles are located within an intersection where the front view of vehicle #1 is blocked by vehicle #2. Dashed circle indicates the sensing range and the shaded area depicts the blind zone of vehicle #1.



V1

00	10	10	10	00
10	11	10	10	00
01	11	10	11	00
10	10	11	11	00
00	10	10	10	00

V2

10	10	10	00	00
10	11	10	00	00
10	10	11	00	00
10	10	11	00	00
10	10	10	00	00

V3

00	00	10	10	10
00	00	10	10	10
00	00	11	10	10
00	00	11	11	10
00	00	10	10	10

(B) Sensing matrices generated by three vehicles. The element with value of (01) in vehicle #1's sensing matrix indicates that vehicle #1 has no idea if there is any object in that block. The element with a value of (00), (11), or (10) indicates the corresponding block is out of sensing range, contains objects and no objects, respectively.

FIGURE 7.18. Sensing zones of three vehicles.

As Figure 7.18 shows, three vehicles are located within one zone, consisted of 5×5 blocks. Each vehicle maps its detected objects into a block in its zone. As such, each vehicle can prepare a 5×5 matrix, with each element representing the value of a block in vehicle's zone.

In this way, instead of sharing raw data, a high-level object detection results captured in a matrix could be shared among vehicles. Different from the messages defined in the SAE J2735 standard [124], smaller packets are adopted in L3, and thus smaller network bandwidth consumption is expected. The SAE J2735 standard defined a DSRC message set dictionary to support interoperability among DSRC applications through the use of standardized message sets. However, the SAE J2735 packet size is usually on the level dozens of bytes; therefore, it is not considered a light-weighted solution to data sharing among autonomous vehicles.

7.4.3.4. Low-Latency Data Sharing

As many vehicles may co-locate within one zone, the information shared among nearby vehicles will become a huge load of network traffic. In addition, the frequency of data produced by sensors is usually very high, in order to meet the real-time requirements for autonomous vehicle's applications. Given high-frequency and huge-volume of data exchange among vehicles, network collisions are inevitable and must be carefully addressed. In this section, we propose a low-latency data sharing protocol for V2V communications, leveraging the capture effect that widely exists in wireless communications.

After the sensing data is processed, a vehicle will create a matrix to record all objects it detects and use this matrix to determine whether it needs help from others, or it is the best vehicle to provide information for others. For example, when a vehicle can clearly sense its surrounding environment, i.e., the value of b_1 of all elements in its sensing matrix is 1, it is unnecessary for this vehicle to receive or process any information shared from others. On the other hand, if a vehicle's sensing matrix contains many elements with $b_1 = 0$, it must request helps from nearby vehicles to convert these b_1's from 0 to 1. Based on this simple principle, we design a distributed data sharing protocol for connected and autonomous vehicles. In the L3 protocol, sensing matrices are shared among vehicles in a synchronous manner where vehicles can only send data within pre-defined slotted time intervals. Because a vehicle's local clock is continuously synchronized with the atomic clocks on the satellites, here we assume all vehicles within a zone is well synchronized.

7.4.3.5. Synchronous Data Communication Based on Capture Effect

As DSRC was standardized as the V2V communication protocol in USA [124], in this paper, we focus on designing a synchronous data transmission mechanism for DSRC. Based on the distributed coordination function (DCF) defined in the IEEE 802.11p protocol, multiple access control is implemented based on the well-known carrier sense multiple access with collision avoidance (CSMA/CA) mechanism [16]. The DCF approach is proved to be efficient for relatively-low network traffics, however, its performance degrades significantly in the cases where large amount of devices transmitting data simultaneously. In these cases, as packet collisions occurs frequently, an larger contention window on each vehicle is expected, which will not only increase the network delay but also reduce the overall network throughput.

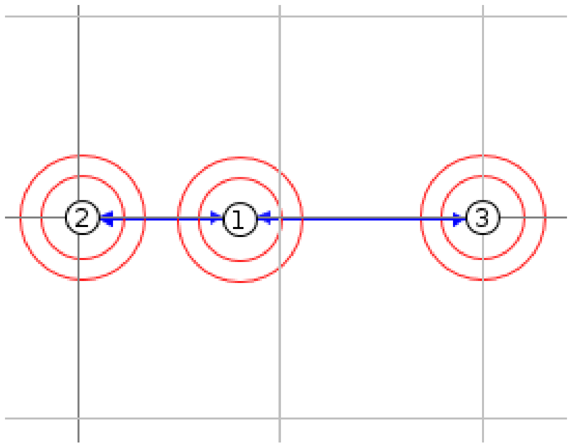
To address the above-mentioned issues, we propose to leverage the capture effect that was widely studied in IEEE 802.11 protocols [130]. Capture effect enables a receiver to correctly decode a packet when the received signal is about 3 dB stronger than the sum of the received signals from all others [9, 62]. As such, given multiple simultaneous wireless transmissions, only the one with the strongest received signal can be received and decoded. To ensure capture effect, the strongest signal must arrive no later than the air time of synchronization header, after the first weaker signal [130]. If these conditions are satisfied, collided packets (from the strongest signal) can be successfully decoded on the receiver. Due to the capture effect, vehicles can receive packets despite interference from other vehicles that are transmitting packets at the same time. As such, the network throughput is improved and the network latency is reduced.

The synchronous data communication protocol works as follows. Vehicles owning uncertain blocks initiate the data communication process by sending their sensing matrices to nearby vehicles. The nearby vehicles overhearing the data will receive these packets with a high probability, due to the capture effect. On reception of these packets, the vehicles combine their own sensing data with the received ones and update their sensing matrices accordingly. The updated sensing matrices will be again shared with other vehicles. This data aggregation process continues in a fully distributed manner until all vehicles in the same zone have the same sensing matrix. When the protocol is executed multiple times, several vehicles may have the same sensing matrix. In this case, when these vehicles simultaneously send the same sensing matrix to others, a constructive inference could be observed. Constructive inference occurs only when packets are identical and overlap with each other within 500 ns. Apparently, constructive inference would speed up the data sharing process among vehicles.

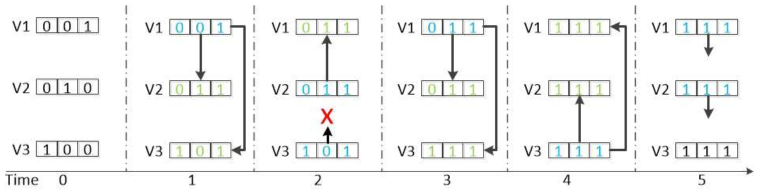
We use an example shown in Figure 7.19a to illustrate how capture effect would facilitate faster data sharing among three vehicles. Here, we assume vehicles `#1`, `#2`, and `#3` are within the communication range of each other. The three vehicles are assumed to reside in a zone containing 25 10 m $\times$ 10 m blocks. Based on its sensors, each vehicle could prepare a sensing matrix. In the example, as vehicle `#2` blocks the front view of vehicle `#1`, there is an uncertain block in vehicle `#1's` sensing matrix. According to the L3 protocol, vehicle `#1` will initiate the data sharing process via sending its sensing matrix in time slot `#1`. When vehicles `#2` and `#3` receive the message from vehicle `#1`, they will aggregate the received data with their own data and update their sensing matrices. The updated sensing matrices are then transmitted from vehicles `#2` and `#3` simultaneously in time slot `#2`. Vehicle `#1` will receive the update sensing matrix from vehicle `#2`, due to capture effect. With the new information contained in the receive message, vehicle `#1` will update its sensing matrix and send it in time slot `#3`. As now new information is received, vehicle `#2` does not send anything in time slot `#4`. At time slot `#5`, because all vehicles have the same sensing matrix, the data sharing process stops.

7.4.3.6. Data Aggregation Process

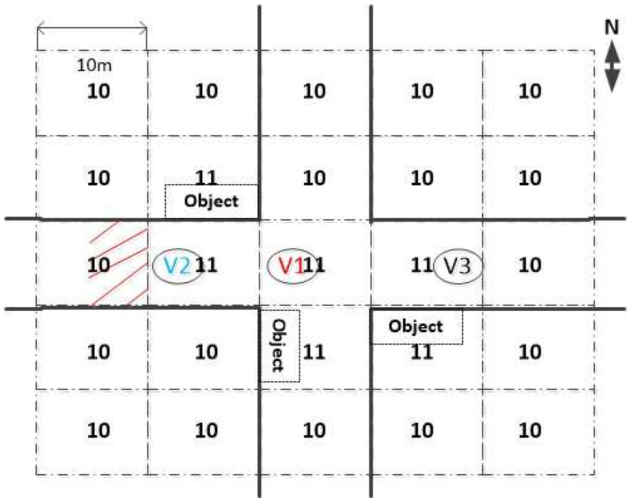
When sensing matrices are received from other vehicles, it is necessary to design a data aggregation process to combine the received data with the current one. As we are focusing



(A) Three vehicles are located along a line with vehicle #2 being closer to vehicle #1 than vehicle #3.



(B) The data communication process among three vehicles. Arrowed line indicates a packet is successfully delivered, due to capture effect. Red cross means a packet is dropped.



(C) The final sensing matrix on vehicle #1. It starts the data sharing process as it has a block with a value of (01). After five rounds of data exchange with others, the value of this block is updated to (10).

FIGURE 7.19. Illustration of how sensing matrices are exchanged among three vehicles.

on enhancing the object detection capability of autonomous vehicles, the data aggregation must produce a sensing matrix that contains all the objects detected by the sharing vehicles.

The data aggregation process on a vehicle starts from identifying whether the received data is generated from another vehicle within the same zone. This can be done by comparing the index of the zones where the vehicles reside. If the received sensing data, denoted as $R_{m\times n}$, are for the same sensing zone, the aggregation will be carried out as follows. Here, we assume there are $m \times n$ blocks within the current zone. Similarly, we use $C_{m\times n}$ to denote the sensing matrix on the current vehicle which takes $R_{m\times n}$ to update its own sensing matrix.

To aggregate two matrices $R_{m\times n}$ and $C_{m\times n}$, we will compare all elements from these two matrices one by one. For a particular pair of elements, we use $b_1^r b_0^r$ and $b_1^c b_0^c$ to represent the sensing data in the received and current sensing matrices, respectively. If $b_1^r = 0$, it implies the received data do not contain any useful information for the corresponding block. Therefore, the value of $b_1^c b_0^c$ will be kept unchanged. On the other hand, if $b_1^r = 1$ and $b_1^c = 0$, the the value of $b_1^c b_0^c$ will be replaced by $b_1^r b_0^r$. If $b_1^r = b_1^c = 1$ but $b_0^c \neq b_0^r$, it means there is inconsistency on the object detection from the two vehicles. In this case, as it is difficult to determine which one offers the best detection result, we consider uncertain observations were made. As a result, the value of $b_1^c b_0^c$ becomes 01, which will again trigger the data sharing process. We believe this case is very rare and only occurs occasionally. The aggregation process will be applied to all pairs of elements from two sensing matrices. The entire data aggregation process is summarized in Algorithm 1.

Algorithm 1: Data Aggregation

```
Result: Sensing matrix $C_{m\times n}$ is updated.
for $\forall b_1^r b_0^r \in R_{m\times n}$ do
    $b_1^c b_0^c \leftarrow$ Corresponding element in $C_{m\times n}$;
    if $b_1^r = 1 \wedge b_1^c = 0$ then
        $b_1^c b_0^c \leftarrow b_1^r b_0^r$;
        else if $b_1^r = b_1^c = 1 \wedge b_0^c \neq b_0^r$ then
            $b_1^c b_0^c \leftarrow 01$;
        end
    end
end
```

Figure 7.19b illustrates how messages are exchanged among the three vehicles. We can see that the blocked area from vehicle `#1's` perspective is updated based on the sensing data provided by vehicle `#2`. Such information is then transmitted to vehicle `#3`. After five rounds of communications, the sensing matrix converges to the one shown in Figure 7.19c. As such, vehicle `#1` is able to extend its sensing capability by detecting there is no object existing in the area blocked by vehicle `#2`. With the proposed L3 protocol, consistent sensing results could be derived on individual vehicles which shared their own sensing data to others.

7.4.4. Evaluation and Result Analysis

In this section, we evaluate the performance of L3 protocol in simulations. Although the capture effect is widely observed on IEEE 802.11 devices, it is not yet implemented in IEEE 802.11p. As most DSRC devices are not open-source platforms, it is prohibitively expensive to conduct reverse engineering on these devices to implement capture effect. As such, we adopt the COOJA [163] simulator to evaluate L3 protocol. Although COOJA is a Contiki IEEE 802.15.4 network simulator, it can adequately approximate the data communication process among vehicles using DSRC. With the COOJA simulator, we simulate scenarios where several vehicles communicate with each other to share the object detection results via the L3 protocol. Particularly, we are interested in how many rounds of data communications are needed to realize a consistent sensing matrix on all participating vehicles. Next, we use the NS-3 simulator [100] to simulate and measure the network delay and scalability of the L3 protocol.

7.4.4.1. Simulation Setup

According to the DSRC protocol, we note that the reliable communication range of DSRC is about hundreds of meters. On the other hand, the effective sensing ranges of regular sensors, e.g., LiDAR, Radar and camera, are far less than the communication range. In the simulations, we set a zone as a $100\,\mathrm{m}^2$ square, and we assume all vehicles within a zone can communicate to each other. Meanwhile, we set each vehicle's sensing range as $25\,\mathrm{m}$. We also set the block size as $5\,\mathrm{m}$. As such, there will be 400 blocks in one sensing zone, i.e., 800 bits are needed to record the sensing results of each block in a zone. As there are only 800 bits in each packet, the payload of the L3 protocol is 100 bytes. Overall, the simulation setup parameters are shown in Table 7.9.

Table 7.9. Simulation Setup Parameters

Parameter	**Value**
Communication Range	100 m
Sensing Range	25 m
Payload Size	100 bytes
Zone Size	100 m^2
Block Size	5 m^2

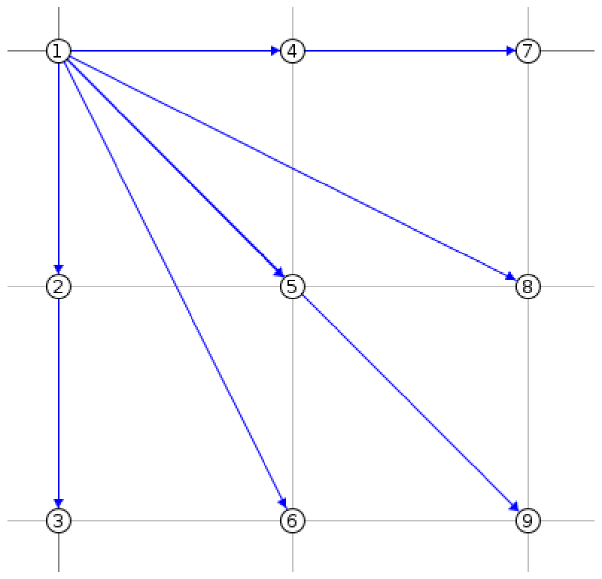
7.4.4.2. Convergence Time

L3 is designed to realize low-latency data sharing among autonomous vehicles, therefore, it is important to evaluate how long it takes to ensure all participating vehicles have the consistent sensing matrix. The latency can be measured in two dimensions: (1) number of time slots taken and (2) the actual time taken to reach a consistent sensing matrix on vehicles. In this section, we evaluate the how many time slots are needed to complete the data sharing process among vehicles.

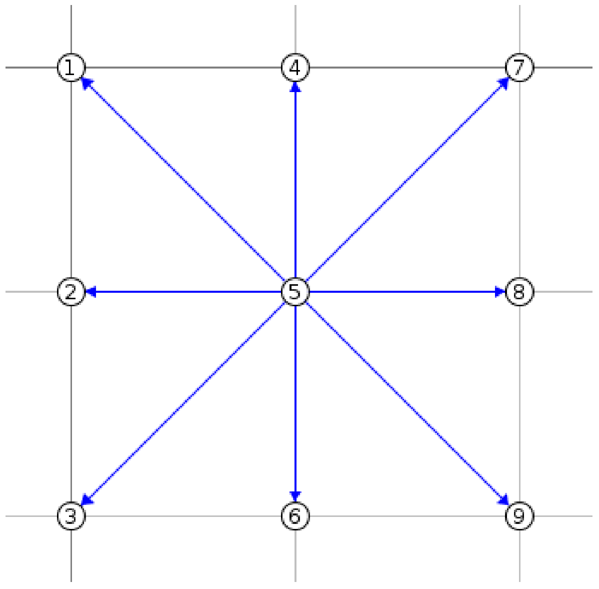
In the simulation, we place nine vehicles in a grid using the COOJA simulator, as shown in Figure 7.20a. The horizontal/vertical distance between two adjacent vehicles is set to be 10 m. We first let vehicle #1 to initiate the data sharing process, which represents the cases where vehicles located around the corners of the grid to start communications. We then record how many time slots a vehicle is in its transmission or reception modes, until all nine vehicles have the same sensing matrix. As shown in Figure 7.20c, after a total of 15 rounds of data exchange, all vehicles reach the same sensing matrix, i.e., the sensing results converge. For vehicle #1, it transmits its original (or updated) sensing matrix for seven time slots and receive data from others in eight time slots. For vehicles #2 and #3, as they are in the perfect location of receiving sensing data, their sensing matrix converges after the 12th round of data exchange. After that, they simply broadcast the converged sensing matrix one more time. Due to constructive interference, their transmissions will not collide even though two packets are transmitted during the same time slot. After the 13th time slot, vehicles #2 and #3 finish the data sharing process and they do not send or receive any new data. At the 14th time slot, vehicle #6 receives the message from vehicle #3, due to capture effect as vehicle #6 is closer to vehicle #3. As such, vehicle #3 received the converged sensing matrix and concludes its data sharing process as well. The remaining vehicles will finish the sensing matrix updating process in a similar manner. After 15 rounds of data exchange, all vehicles obtain the same sensing matrix for the targeted sensing zone.

Next, we make vehicle #5 serve as the initiator, which represents the cases where vehicles in the middle of a sensing zone starts the data sharing process. As shown in Figure 7.20b, vehicle #5 starts sending and collecting sensing data from others. Different from our expectation, in this case, it takes a total of 17 time slots to finish the data sharing process. This is mainly because it takes a longer time for data from vehicles at one side to propagate to those at the other side. The distribution of transmission and reception activities of each vehicle is plotted in Figure 7.20d. As we can see, vehicle #3, #5, #6 and #9 receive the converged sensing matrix after the 14th time slot. They broadcast the sensing result one more time and then keep silent. Vehicles #1, #2, and #4, on the other hand, receive vehicle #5's message at the 15th time slot, due to capture effect. As the message contains the final sensing matrix, they all halt the sharing process after one more round of broadcasting. The last two vehicles (#7 and #8) complete their updating process at the 17th time slot, and the entire sharing process is finished.

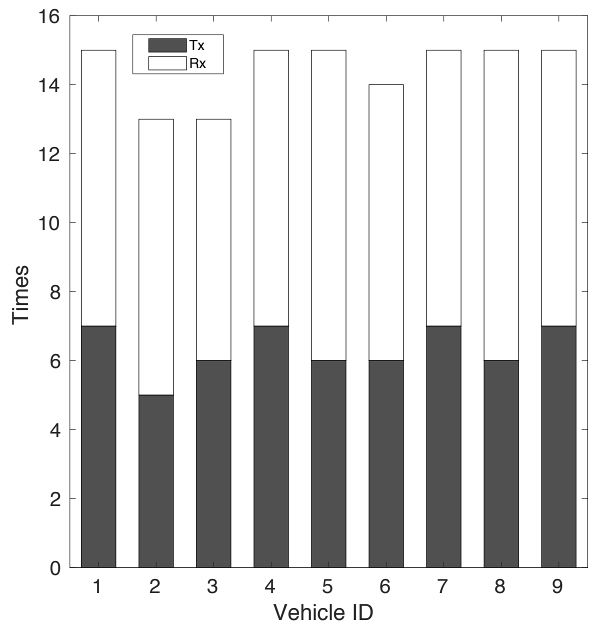
Besides the above simulations, we also conduct experiments with different number of vehicles that are randomly deployed in a certain area. The convergence times of different scenarios, i.e., deploying three to 15 vehicles randomly in a zone, are summarized and plotted in Figure 7.21. We first deploy three vehicles in a zone and it takes four time slots to finish the data sharing among the three vehicles. The convergence time grows as more and more vehicles join in the data sharing process. The total number of rounds increases up to 26 time slots when there are 15 vehicles in the network.



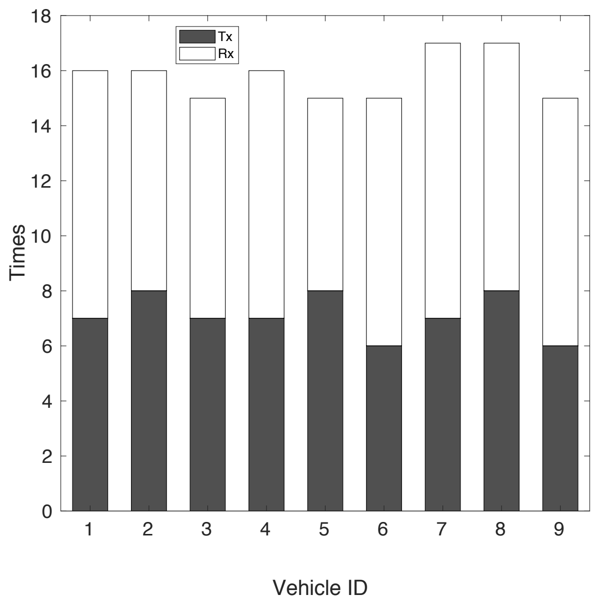
(A) Vehicle #1 located at top-left corner initiates the data sharing process.



(B) Vehicle #5 located in the middle initiates the data sharing process.



(C) The distribution of time slots in which vehicles are in transmission and reception modes, respectively.



(D) The distribution of time slots in which vehicles are in transmission and reception modes when vehicle #5 is the initiator.

FIGURE 7.20. Convergence time of nine vehicles exchanging sensing matrices between each other.

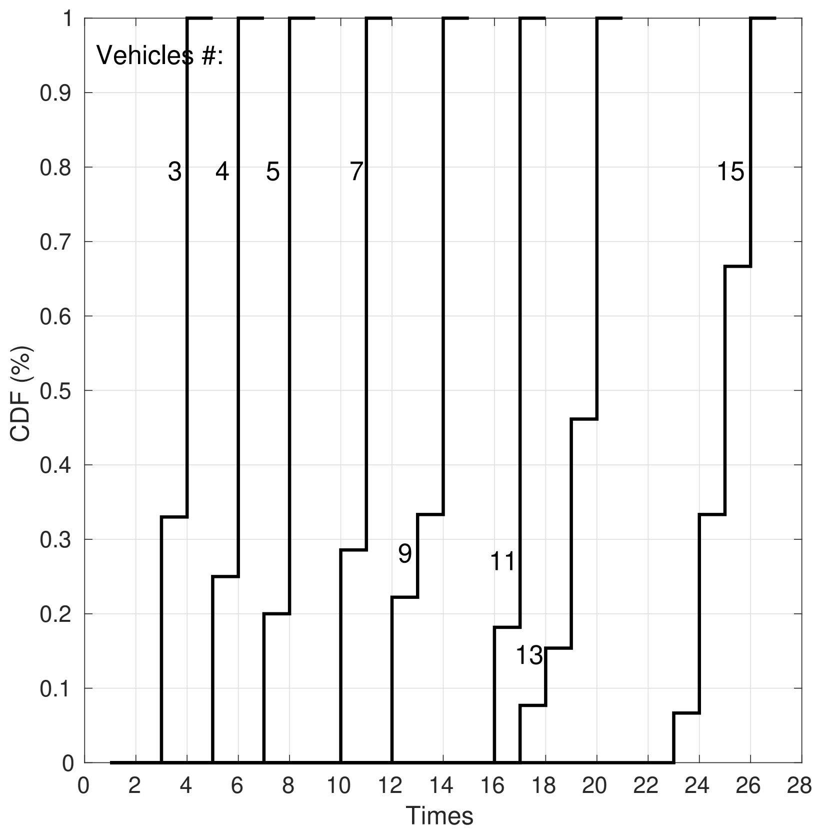


FIGURE 7.21. Completion times of data sharing with different number of vehicles in the network.

7.4.4.3. Network Latency

Networking latency of the L3 protocol highly depends on the setting of the time slot, i.e., longer the time slot, larger the networking delay. To achieve a low-latency protocol, it is critical to set the time slot as small as possible. To identify the best setting of time slot, we need to understand how long it takes to transmit, receive and process 100 bytes of data in a vehicular network. We adopt NS-3 simulator [100] to find the minimal required time to finish each round of data transmission between vehicles. To obtain an accurate measurement of the time, we simulate two vehicles (100m away from each other) communicating to each other in NS-3. In the simulation, one vehicle transmits a 100 B message to another vehicle, using the IEEE 802.11p protocol with CSMA/CA disabled. In this case, the time needed to transmit and receive a 100 B message is similar to that obtained from capture effect.

Here, the time is what is needed for the receiving vehicle to successfully receive the message from the transmitting vehicle. In our simulation, it requires less than 2ms to share a 100 B message between two vehicles. When the vehicles are closer to each other, the time will be a bit smaller, due to a shorter propagation delay that is neglected in this paper. As such, we configure the time slot to be 2 ms in our simulations. Figure 7.22 shows the actual delay of the data sharing process, with different numbers of vehicles in the simulations. In the figure, there is a notable improvement on latency in L3 over the IEEE 802.11p. This is because the IEEE 802.11p protocol requires vehicles to compete to access the wireless channels, which may cause a significant networking delay.

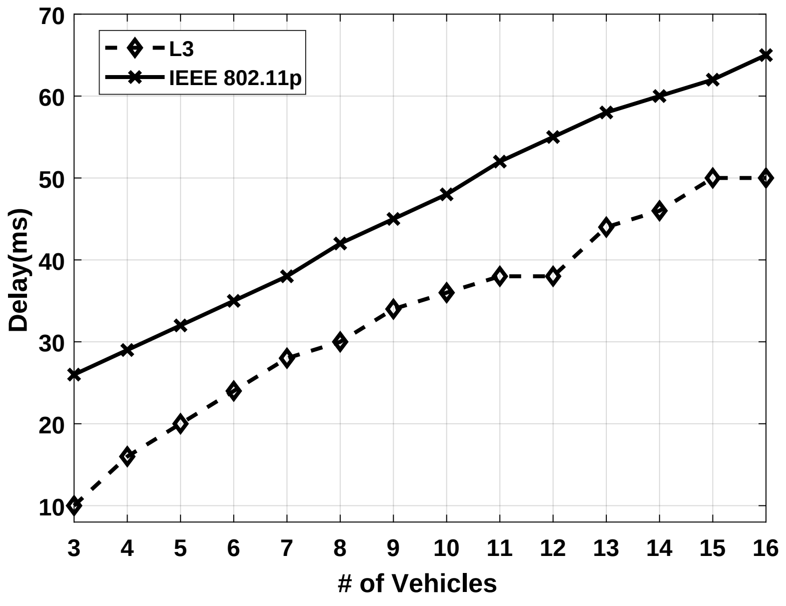


FIGURE 7.22. Network delay with different numbers of vehicles.

7.4.4.4. Scalability

With more vehicles, the data sharing among vehicles may take a longer time to complete. In this section, we conduct experiments to evaluate L3's scalability, i.e., understanding how L3 performs when the number of vehicles increases in the network. As seen in Figure 7.23, we witness that the network delay of L3 is increases slightly as the number of participating vehicles increases. On the other hand, the latency of IEEE 802.11p tends to perform poorly when there are large number of vehicles transmitting packets simultaneously.

As vehicles benefiting from the shared data and not suffering from the consequences of large latency, L3 is proved to be effective for up to as many as 225 vehicles in a sensing zone. With the current traffic infrastructure, there is usually less than 225 vehicle within any reasonable intersection in any city. In some extremely crowded areas, the number of vehicles could be larger than 255, which may cause a longer network latency. To address this issue, we could reduce the size of each sensing zone to include no more than 225 vehicles. The parameter setting of L3 protocol is not static and needs to be changed based on real-world applications.

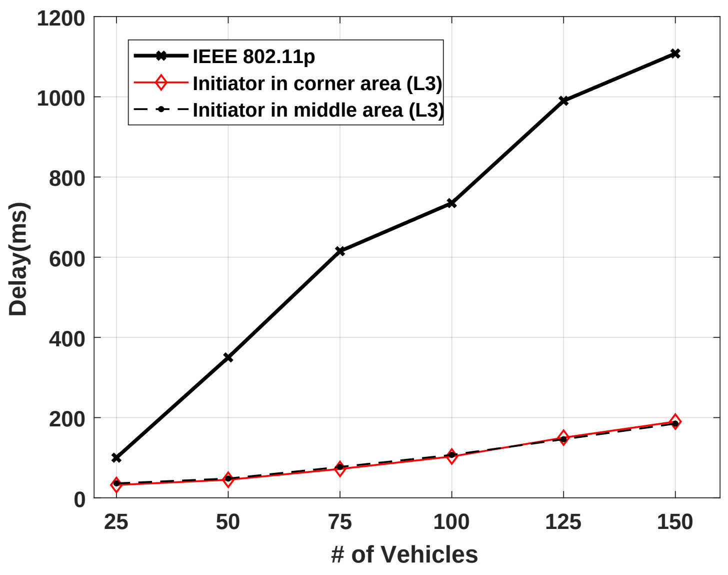


FIGURE 7.23. Network delay in a large scale vehicular network.

7.4.5. Conclusions

We propose the L3 protocol to support low-latency data sharing among autonomous vehicles towards the goal of a better detection of objects around autonomous driving cars. Due to the capture effect, all vehicles transmit their sensing matrices simultaneously and thus a low-latency data sharing among vehicles is achieved. Although the design of L3 protocol is verified and evaluated in simulations, it is worth noting that the implementation of L3 on real-world hardware is still a challenging problem. In the future, we will explore the possibility of integrating L3 into existing DSRC devices and demonstrate how L3 works in real-world experiments.

7.5. Reliability Characterization of Solid State Drives in a Scalable Production Data Center [134][4]

In recent years, NAND flash-based solid state drives (SSD) have been widely used in data centers due to their better performance compared with the traditional hard disk drives. However, little is known about the reliability characteristics of SSDs in production systems. Existing works study the statistical distributions of SSD failures in the field. However, they do not go deep into SSD drives and investigate the unique error types and health dynamics that distinguish SSDs from hard disk drives. In this paper, we explore the SSD-specific SMART (Self-Monitoring, Analysis, and Reporting Technology) attributes to conduct an in-depth analysis of SSD reliability in a production environment. Data is collected from a scalable production system having several physical locations. Our dataset contains over a million records with more than twenty attributes. We leverage machine learning technologies, specifically data clustering and correlation analysis methods, to discover groups of SSDs which have different health status and relations among SSD-specific SMART attributes. Our results show that 1) Media wear affects the reliability of SSDs more than any other factors, and 2) SSDs transit from one health group to another which infers the reliability degradation of those drives. To the best of our knowledge, this is the first study that investigates SSD-specific SMART data to characterize SSD reliability in a production environment.

7.5.1. Introduction

Solid state drive (SSD) based storage systems are receiving wide attention for high performance computing (HPC) and their deployment is steadily increasing due to their higher performance and lower power consumption compared with hard disk drive (HDD) based storage systems. The memory-storage hierarchy has been shift from using HDD as permanent storage to using SSDs or a fusion of Flash cache and HDD storage. While their deployment is increasing, the write endurance of SSDs still remains as one of the main

[4] Section 7.5 is reproduced in its entirety from Shuwen Liang, Zhi Qiao, Jacob Hochstetler, Song Huang, Song Fu, Weisong Shi, Deesh Tiwari, Hsing-Bung Chen, Bradley Settlemyer, and David Montoya, Reliability characterization of solid state drives in a scalable production datacenter, 2018 IEEE International Conference on Big Data (Big Data), pp. 3341-3349, with permission from IEEE.

concerns. As write and erase operations on an SSD wear it out gradually, after a certain number of operations, the SSD could fail and its data could be lost.

Many studies have investigated the bit error failure behavior of multi-level cells (MLC) and single-level cells (SLC). They find that the bit error rate of the flash memory increases with an increased number of Program/Erase (P/E) cycles. These studies model the bit error rate as an exponential function of the number of P/E cycles that a cell has gone through. There are also a number of recent studies analyzing the statistical distributions of SSD failures in the field. They also find that although flash drives offer a lower field replacement rate than HDDs, they have a significantly higher rate of uncorrectable errors that can impact the stored data.

However, the existing studies of SSD reliability are either at the circuit level (i.e., MLC and SLC) or for the entire storage system level using field data. They do not explore the rich set of performance and reliability related attributes provided by the SSD Self-Monitoring, Analysis, and Reporting Technology (SMART) at the drive level. Compared with the SSD failure field data which do not provide insight into how SSD deteriorates and what factors dominate the process or workload and environment data which complicate SSDs' failure analysis, SSD-specific SMART data provide a direct and insightful way to characterize SSD failures with a generic method.

In this paper, we perform an in-depth analysis of SSD reliability using SSDs' SMART data collected from an active, production environment. The SSDs are used as caching devices in storage nodes spread across several data centers. The dataset contains six-months of SSD SMART data. We use machine learning models and statistical analysis to investigate more than 20 SSD-specific performance and error related SMART attributes from over a million complete records. Results from our comprehensive correlation analysis reveal that 1) In a well maintained data center, environmental factors rarely affect the reliability of SSDs. 2) Certain attributes, e.g., Media Wear which reports the number of cycles that the NAND media has undergone, influence the reliability of SSDs more significantly than other attributes. 3) The wear-out process is very slow and the manufacturer-provided flash cell endurance rating is conservative. 4) We can identify imbalanced I/O workload by cross-comparison of certain attributes, such as power-on hours (POH), power cycle count (PCC), and Average Write/Erase Count (AWEC). By using unsupervised machine learning techniques, we discover latent relationship for the first time, that is we find groups of SMART records corresponding to different health status of SSDs. We observe the transition of several SSDs from one group to another, which represents the degradation of their reliability.

To the best of our knowledge, this is the first study of SSD SMART data from a production data center to characterize SSD reliability. Our analytic results provide a deeper understanding of SSD reliability and the dominating factors in production environments. Our findings will help system operators develop countermeasures to extend SSD's lifetime and protect the stored data, for example by balancing I/O workload, predicting SSD failures, and proactive migration of data.

The reminder of the paper is organized as follows. Subsection 7.5.2 presents the background and an overview of our method. Subsection 7.5.3 describes the SSD-specific SMART attributes and results from the correlation analysis. Subsection 7.5.4 characterizes SSD reliability. The related research is discussed in Subsection 8.1.1. Subsection 7.5.5 concludes the paper with remarks on future research.

7.5.2. Fault Modes and SMART Data of SSDs

7.5.2.1. Architecture of SSD

SSD consists of many tightly-coupled flash chips and memory controllers. A *NAND* type floating gate forms a *Non-Volatile Memories* (NVM) cell, whose content can be electrically altered and can be preserved without power. Based on number of bits that each flash cell stores, flash memory has Single Level Cell (SLC), Multi-Level Cell (MLC), and Triple-Level Cell (TLC) that store 1, 2, and 3 bits per cell, respectively. Each flash chip contains arrays of such flash cells which are organized into pages, blocks, planes, dies, and banks in the corresponding constructive order. An SSD device contains multiple flash chips.

For example, a typical 32 Gbit flash chip is logically organized into 8192 blocks, and each block consists of 64 pages with 4K bytes per page in addition to the 128-byte spare space. In NAND flash memories, *a logical page is the smallest addressable unit for read and programming, while block is the smallest erasable unit.* The spare space is used for storing ECC parity data and reallocated sectors for bad pages.

Over the past few decades, HDD was used as permanent storage media in the memory-storage hierarchy. HDDs are organized into sectors, and use Logical Block Addressing (LBA) to specify the locations of blocks of data. In contrast, SSDs are organized into planes, blocks, and pages. The Flash Translation Layer (FTL) translates a sector access into a page read or block write. The FTL manages the mapping between LBAs and physical block addresses. In addition, FTL 1) employs an aggressive *Error Correction Code* (ECC) to ensure data reliability; 2) adopts *wear-leveling* to distribute erasures and re-writes evenly across cells since each flash cell can only sustain a certain number of program-erase cycles; 3) performs *garbage collection* periodically to reclaim blocks previously deleted by the host system; 4) enables *bad block management* to handle invalid blocks that have more uncorrectable errors than what ECC can fix, and map data to spare blocks.

7.5.2.2. Fault Modes of SSD

SSD manufacturers have their proprietary FTL policies to manage ECC, wear-leveling, and garbage collection. Thus there exists a variety of reliability characteristics of different SSDs. For example, reading and writing data causes wear of flash cells, which degrades SSD reliability gradually. A number of prior works studied the correlation between wear and the increase of error rate [30] [38] [143] [147] [149] [195]. Wear-leveling is designed to distribute data across SSD to address this issue.

Retention errors that are caused by the leakage current increase with usage [13] [31] [27] [56] [131] [241]. If not confined or corrected in time, retention errors quickly propagate. As read and programming operations can affect the threshold voltage of the neighboring blocks [23] [26] [117] [122] [209], causing untouched cell susceptible to *read disturb errors* and *program disturb errors*. Besides wear-leveling, FTL uses ECC as a countermeasure to prevent the above-mentioned error propagation to the upper data hierarchy, that is checksum in each page of the spare space is used by ECC to protect against these errors. If the number of bit errors exceeds the capability of page-level ECC, SSD controller performs error correction at the host driver level by using more complex error correction algorithms. From the SSD's perspective, page-level ECC are correctable errors, while host driver level ECC are uncorrectable errors. When the number of uncorrectable errors in a block exceeds a preset threshold, FTL marks it as a bad block and remaps the data to a reallocated block in the spare space. SSD manufacturers are conservative about the *endurance rating* and reserve a considerable amount of space for remapping data from bad blocks.

7.5.2.3. SSD Caching and Flash Storage

Server-side flash or *SSD caching* refers to the deployment of SSDs as flash memory for caching and tiering data between the main memory and the storage system. As a cost-effective alternative to flash storage, it is often coupled with slower HDDs to improve the read and write throughput. When using SSDs as read cache, compute nodes retrieve data from permanent storage (HDDs) or via a storage area network (SAN), and store temporary copies of active data on NAND flash memory. Thus, data can be accessed quickly when needed. When used as write cache, SSDs buffer data until the slower and persistent storage has space and bandwidth to complete write operations. SSD caching is managed by system's storage controllers and is secondary to the main memory. Because the footprint of active data is relatively small, the capacity requirement of SSDs is lower than that of a full flash storage. *Flash storage* is getting popular in high performance storage appliances that use flash memory based technologies as the permanent storage media. It has higher capacity and reliability requirements, while the access frequency is usually less than that of SSD caching. In this paper, we study the reliability of SSDs as caching/buffering devices, which is similar to the use of SSDs as burst buffers in HPC systems.

7.5.2.4. SSD SMART Data

Our SSD SMART dataset was collected from several active production data centers owned by a major financial service provider. Each server in the data center has 2 sockets, 10-18 cores per socket, 1.5 TB of DRAM, 1-2 SSDs, and 8-45 HDDs. SSDs are used as a data buffer between the main memory and the storage subsystem. They accelerate data accesses, improve I/O throughput, and reduce access latency from main disks in a Ceph storage node. The workloads run on the servers include investment/portfolio management, brokerage order management systems, and financial planning management. This represents a wide range of operating system main disks, persistent database storage, and sequential message queues.

The SMART monitoring system is employed for both HDDs and SSDs to collect access information and fault/error indicators. SSD SMART data are collected hourly at runtime. By the time of this study, we have six months of SMART records from both HDDs and SSDs. We pre-process the raw SMART data by extracting SMART attributes and their values and removing incomplete records. Over half of a million SSD records are kept with over 20 attributes from two SSD brands and models. Then, we analyze the correlation among these attributes and further explore machine learning technologies to characterize and study SSD reliability. The results are presented in the following subsections.

7.5.3. Correlations Analysis of SSD SMART Data

The SMART technology monitors drives' accesses and errors, and provides various attributes, many of which are particularly designed for SSDs. An SSD manufacturer adopts a subset of all SMART attributes which may be different from that of another manufacturer. Even among drives produced by the same manufacturer, those of different models may have different sets of SMART attributes. In general, we group the SMART attributes into three categories, i.e., environmental factors (e.g., temperature and power-on hours), workload related statistics (e.g., the amount of data read from or written to flash chips), and error attributes (e.g., the number of seek errors and the number of uncorrectable errors). In the data center that we study, two SSD models from different vendors are found. Table 7.10 lists the SMART attributes provided by the SSDs.

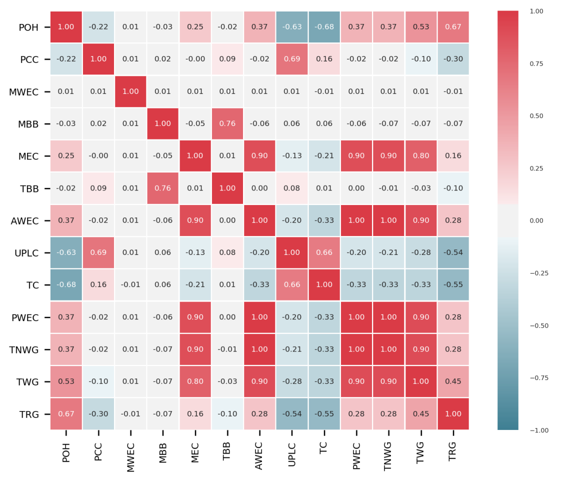
Table 7.10. SSD SMART Attributes and Description

SMART ID	Attribute	Description
Environmental Attributes		
9 [c]	POH	Power On Hour
12 [c]	PCC	Power Cycle Count
174 [c]	UPLC	Unexpected Power Lost
194 [c]	TC	Temperature Celsius
Workload Related Attributes		
166 [a]	MWEC	Min Write/Erase Count
168 [a]	MEC	Max Erase Count
173 [c]	AWEC	Average Write/Erase Count
180 [b]	URNB	Unused Reserve NAND Blocks
202 [b]	PLR	Percent Lifetime Remaining
230 [a]	PWEC	% of Write/Erase Count
232 [a]	PARS	% Avaliable Reserved Space
233 [a]	TNWG	Total NAND Write(GB)
241 [a]	TWG	Total Write(GB)
242 [a]	TRG	Total Read(GB)
246 [b]	CHSW	Cumulative Host Sectors Written
247 [b]	NONPWYTH	Number of NAND page written by Host
248 [b]	NONPWNTF	Number of NAND page written by FTL
Error Related Attributes		
4 [b]	RRER	Raw Read Error Rate
5 [c]	RSC	Reallocated Sector Count
167 [a]	MBB	Min Bad Block/Die
169 [a]	TBB	Total Bad Block
171 [c]	PFC	Program Fail Count
172 [c]	EFC	Erase Fail Count
183 [b]	SID	SATA Interface Downshift
184 [b]	ECC	Error Correction Count
187 [c]	RU	Reported Uncorrected
196 [b]	REC	Reallocation Event Count
197 [b]	CPEC	Current Pending ECC Count
206 [b]	WER	Write Error Rate
212 [a]	SPE	SATA Physical Error

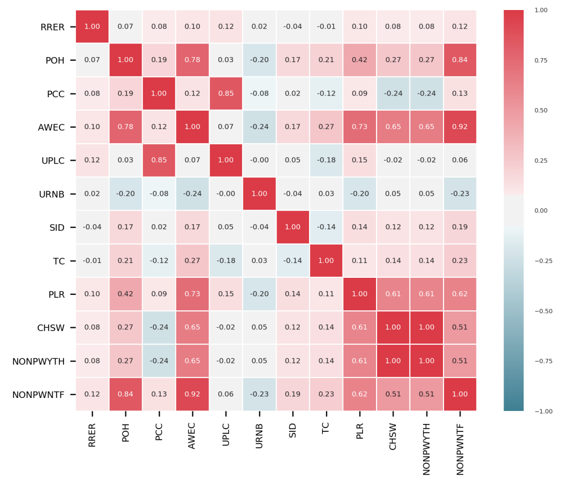
[a] Attributes exclusive to Model A SSDs [colored in *red*]
[b] Attributes exclusive to Model B SSDs [colored in *blue*]
[c] Attributes common to Model A and Model B SSDs models.

Compared with several decades of deployment of HDDs in the field, SSDs are still at the early stage of usage as the mainstream storage media in production systems. Little is known about SSDs' reliability characteristics in real-world settings. Most recently, large-scale field studies, such as [195], identify substantial differences from those SSD fault data collected in controlled environments. Production systems involve a wide range of conditions, e.g., real-world applications display a variety of access patterns and frequencies compared with synthetic benchmarks. Additionally, the entire software stack on top of a flash storage also affects data accesses, SSD performance degradation, and lifespan.

In this subsection, we investigate the relationship among SMART attributes by quantifying pair-wise correlation coefficients. We also use *boxplot* to visualize the distributions of SSD SMART data. By analyzing the correlation among SSD SMART attributes, we obtain a better understanding of the influence among various factors and their criticalness for characterizing SSD reliability. We compare several correlation coefficients, i.e., Pearson, Spearman, and Kendall. We select the Spearman rank correlation coefficient, because it provides the best modeling of monotonic linear/non-linear relations which are common for SMART attributes. To prevent invalid correlation, we remove SMART attributes whose values remain constant over time. Figure 7.24 shows the pair-wise correlation calculated by using Spearman coefficients for the two SSD models in our dataset. Values of the Spearman correlation coefficients range from -1 (i.e., strong negative monotone correlation) to $+1$ (i.e., strong positive monotone correlation), while 0 indicates no correlation. We compare the correlations of environmental attributes with workload-related attributes, and also with error related attributes. We show the results with over 95% of confidence. In the correlation heat-map (Figure 7.24), redder colors indicate stronger correlations. Note that the solid red blocks along the diagonal show the correlation of an attributes with itself, which is not considered in our analysis. Other blocks with a correlation coefficient greater than a threshold, say 0.9, infer that the corresponding attribute pairs are significantly correlated. The major findings on the correlations of SSD SMART attributes are as follows.



(A) Attributes of Model-A SSDs.



(B) Attributes of Model-B SSDs.

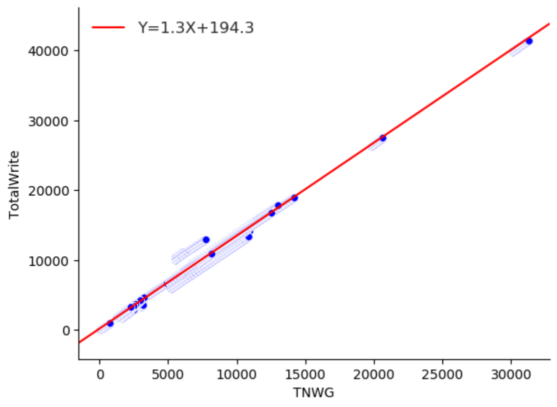
FIGURE 7.24. Spearman correlation among SSD SMART attributes.

7.5.3.1. Finding 1: Environmental Attributes Barely Affect SSD Reliability

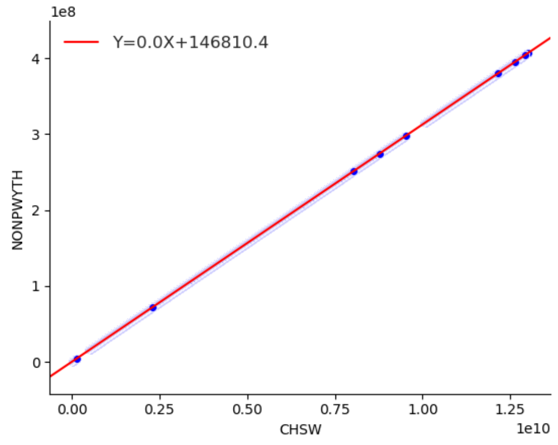
After removing constant-valued attributes, we use 13 SMART attributes to calculate the correlation coefficients for Model-A SSDs, and 12 attributes for the Model B. Among these attributes, environmental attributes, i.e., Temperature in Celsius (TC), Power On Hours (POH), Power Cycle Count (PCC), and Unexpected Power Lost Count (UPLC), do not possess strong correlations with other attributes. If the threshold for the correlation coefficient is set to 0.6, the correlation between the following attributes needs analysis. Power On Hours (POH) and the Number of NAND Page Write by FTL (NONPWNTF) for Model-B SSDs have a correlation of 0.84, 0.67 between POH and Total Read in GB (TRG) for Model A, and 0.53 between POH and Total Write in GB (TWG). This is because older SSDs (i.e., higher POH values) are more likely to experience more read/write/erase operations. Thus, *environment related SMART attributes do not significantly influence SSD reliability*, which confirms with prior studies [147, 195].

7.5.3.2. Finding 2: Workload Related Attributes Do Not Directly Indicate Occurrences of SSD Failures

Flash cells can endure a limited number of program and erase (PE) cycles. I/O workloads could provide useful information about the wear level of flash cells. Early research reported an exponential growth of the Raw Bit Error Rate (RBER) with the increase of PE cycles [149][30][31]. However, recent field studies show the contradictory results, that is the increase of RBER is linear[195].



(A) Write Amplification



(B) Host writes vs. NAND writes

FIGURE 7.25. Relationship of write operations.

In Figure 7.24a, we observe that those attributes that are related to write and erase operations of SSDs, such as Max Erase Count (MEC), Percentage of Write Erase Count (PWEC), Average Write Erase Count (AWEC), Total NAND Write in GB (TNWG), and Total Write in GB (TWG), have significant correlations which are higher than 0.9 between each other. However, the dataset does not show any correlation between these wear related attributes with failure symptoms such as the Raw Bit Error Rate and Bad Blocks. Since the dataset contains six months of SSD SMART records, it is possible that flash chips have not experienced failures during that period of time.

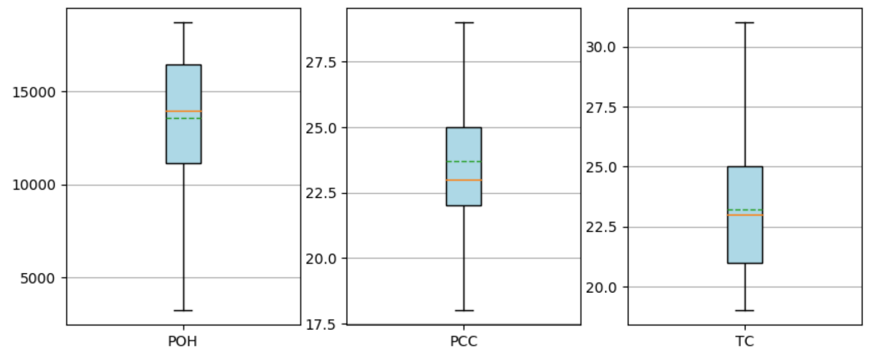
The raw values of TNWG and TWG of Model A, as well as the Number of NAND Page Written by Host (NONPWYTH) and Number of NAND Page Written by FTL (NON-PWNTF) of Model B can be used to calculate flash memory's write amplification ratio. Figure 7.25a shows the linear regression that fits the correlation data between TNWG and TWG. We observe a 1.3X write amplification from host initiated writes to the actual NAND page writes by FTL. Note the ideal write amplification is 1X. *The workload related attributes do not directly indicate the occurrences of SSD failures.*

A similar pattern is observed for Model-B SSDs. As illustrated in Figure 7.24b, Average Write Erase Count (AWEC) has a strong correlation with Percent Lifetime Remaining (PLR), Cumulative Host Sectors Written (CHSW), Number of NAND Page Written by Host (NONPWYTH), and Number of NAND Page Written by FTL (NONPWNTF). Among these attributes, PLR indicates the estimated percentage of lifetime remaining based upon the average number of block erase operations and the number of rated block erase operations. The average number of block erase operations has a positive correlation with AWEC. Henceforth, PLR becomes positively correlated with AWEC. We also find that 90.8% of SSDs in the data center have PLR equal to zero, which means those SSDs reach the end of their lifetime according to the specification of PLR. However, the error related SMART attribute for those SSDs show no significant difference from other SSDs whose PLRs are greater than zero. In addition, those SSDs run smoothly for a long period of time with PLR remaining as 0%. This finding also confirms that manufacturers' rated block erase count is conservative, and SSDs' actual lifetime in field is longer than that provided by the manufacturers.

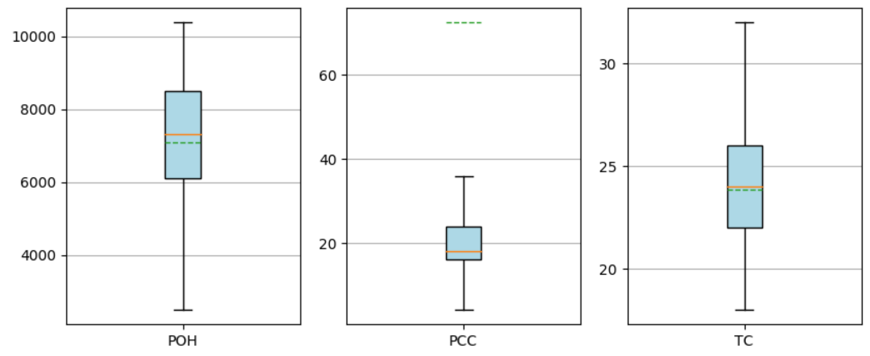
Our results also show that Cumulative Host Sectors Written (CHSW) and NAND Page Written by Host (NONPWYTH) have a correlation coefficient as 1.0. CHSW indicates the amount of data that the host writes to the LBA device. FTL then translates and maps the LBA sector requests to physical pages on an SSD. The number of pages written is recorded by NONPWYTH. The number of bytes written to the SSD recorded by the two attributes should be the same. A typical sector size is 512 bytes, and the page size of Model-B flash memory is 16 KB. This corresponds to our observation that the mapping from sectors to pages has a ratio around 30:1 as shown in Figure 7.25b. The correlation results between PLR and error related attributes from Model-B SSDs also confirm that *not every workload related SMART attribute, such as PLR, directly indicates that SSDs fail.*

7.5.3.3. Finding 3: I/O Workloads Are Not Evenly Distributed in the Data Center

We also investigate the distributions of environmental attributes and their strongly correlated attributes. The cross-comparison identifies two models for workload, wear level, and cumulative failure symptoms. In this study, we use boxplot to illustrate the attributes'



(A) Model A



(B) Model B

FIGURE 7.26. Distributions of Environmental Attributes from SSDs.

distributions for ease of visualization and analysis. Figure 7.26 shows the environmental attributes from both SSD models. After removing negligible outliers, the box region shows the 95th percentile of raw values for environmental attributes, that is Power On Hours (POH), Power Cycle Count (PCC), and Temperature Celsius (TC). In these figures, solid and dash lines represent the median value and arithmetic mean of each attribute respectively. We analyze the median values in the following discussion as medians represent the values of those attributes from the majority of SSDs in the data center.

The median TC value for both SSD models is relatively close, i.e., around 23 Celsius with a variance of ±2. This is because 1) the cooling system in the data center functions well, and 2) the internal thermal management of SSDs keeps drives under a stable temperature. Meanwhile, we observe that about Model-A SSDs have two times longer operation time than Model-B drives based on the POH. The variance of POH is high, ranging from 2,000 to 19,000 for Model A, and from 2,000 to 11,000 for Model B. Considering the median PCC for both models is close (i.e., around 20-22), we consider that the value of POH is not solely determined by the operation time. After checking manufacturers' documents, we find that the raw value of POH reflects a device's online hour (i.e., under power), and excludes the increment in offline states such as *SATA Partial*, *SATA Slumber*, and *SATA Device Sleep*. We also consulted with system administrators working at the data center who confirmed that both SSD models in the system were deployed in the same time frame. Despite less than 5% of the SSDs experienced infant mortality and were replaced, the majority of SSDs operates in an active system since their deployment. Since enterprise data centers employ more aggressive power saving policies, we believe that Model-B SSDs have experienced less workload as the time spent in offline states is about two times more than Model-A SSDs. We also notice that this workload imbalance not only happens between different models of SSDs, but also among drives of the same model. The difference between PCC median and PCC mean is caused by a few drives ($< 10\%$ of SSD population) have very high PCC values. All these observations show that the *I/O workloads are not uniformly distributed among SSDs*.

In summary, we find that 1) write and erase operations have a strong correlation between each other; 2) environmental attributes do not directly affect SSDs' health; 3) workload related attributes do not directly indicate the occurrences of SSD failures; 4) I/O operations are not evenly distributed in the system.

7.5.4. Characteristics of SSD Reliability

In order to understand SSD's reliability and discover the relations between SMART attributes and SSD's health status at the drive level, we analyze the SSD SMART dataset by using machine learning methods. Specifically, we explore K-means clustering on SMART records. K-means is an unsupervised machine learning method, which is used to discover groups of data items with similar feature patterns. It can help us establish a dynamic view of SSD's health status with possible transitions over time.

Base on the material composition and architecture of SSDs, I/O operations, including read, write and erase, can influence the health status of SSDs. We analyze the categories of

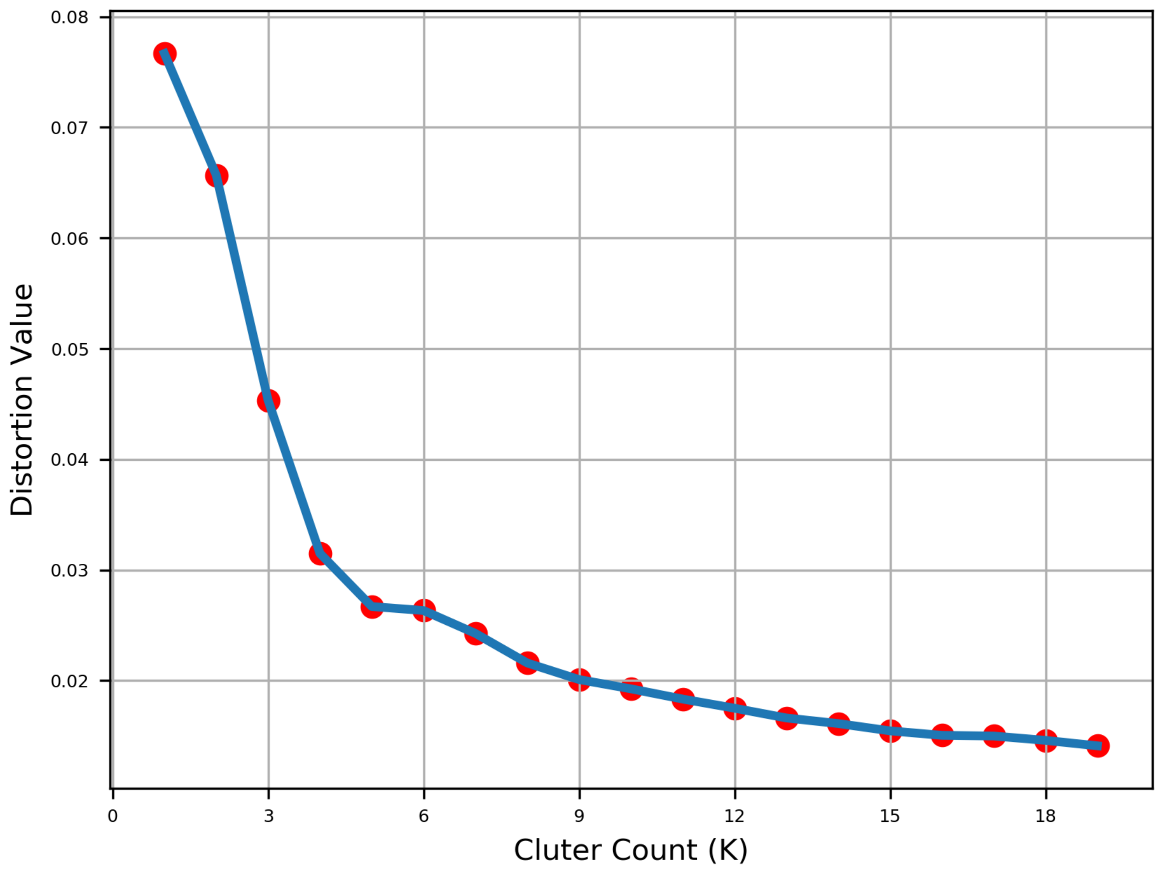


FIGURE 7.27. Clustering method produces five groups of SSD SMART records (Model A). SMART attributes MEC, WEC, PWEC, TNWG and TWG are used in clustering.

SSD health states and their possible transitions over time. Experimental results show that the SSD SMART records can be grouped into five clusters for both Model A and Model B, which is shown in Figure 7.27 and Figure7.28. For Model-A SSDs, SMART attributes MEC, WEC, PWEC, TNWG and TWG etc. are selected for clustering, while AWEC and NONPWNTF, CHSW and NONPWYTH etc. are used to cluster SMART records for Model B. Due to the high dimensionality, we do not plot the five clusters in this paper. With the euclidean distance cost function, Model-A SSDs have a distortion value lower than 0.03, while Model-B drives have a lower than 0.01 distortion value.

An important finding from our experimental results is that, for both models of SSDs, the health states of SSDs may change from one group to another as the wear level changes. In several cases, such transitions happen more than once. In our study, a maximum of three

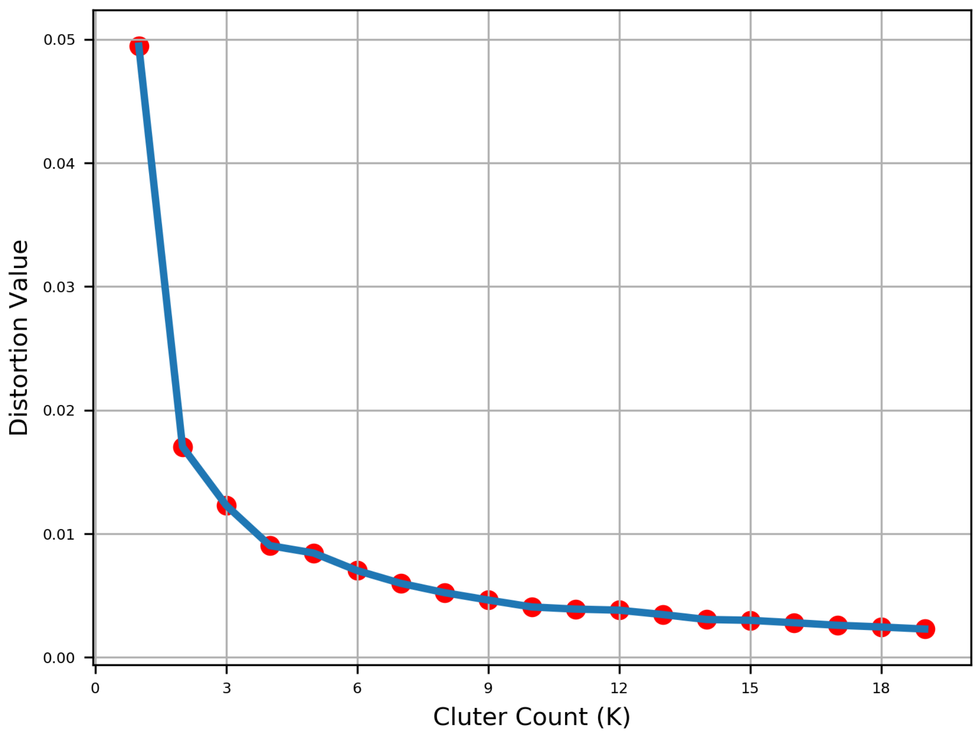


FIGURE 7.28. Clustering method produces five groups of SSD SMART records (Model B). SMART attributes AWEC and NONPWNTF, CHSW and NONPWYTH are used in clustering.

transitions is observed for Model-A SSDs and a maximum of nine transitions is observed for Model-B SSDs. For Model A, over 84% of SSDs experience health state transitions, which we call *reliability degradation*. Model-B SSDs also have over 36% drives experience health state transitions. Table 7.11 and Table 7.12 present the relative size of each SMART record cluster and the frequency of reliability degradation. From the tables we can see the patterns of reliability degradation between the two models of SSDs are different. Specifically, the majority of Model-A SSDs have reliability degradation, while Model-B SSDs have more stable health states. As we discuss before that I/O workload is unbalanced between Model-A and Model-B SSDs, those drives of Model B with less workload are more likely to stay in one health state.

To analyze the wear levels and relationship between SMART records and SSD health states, we investigate each cluster produced by K-Means. We find each SSD model has its own reliability characteristics and has some properties in common. The distributions of

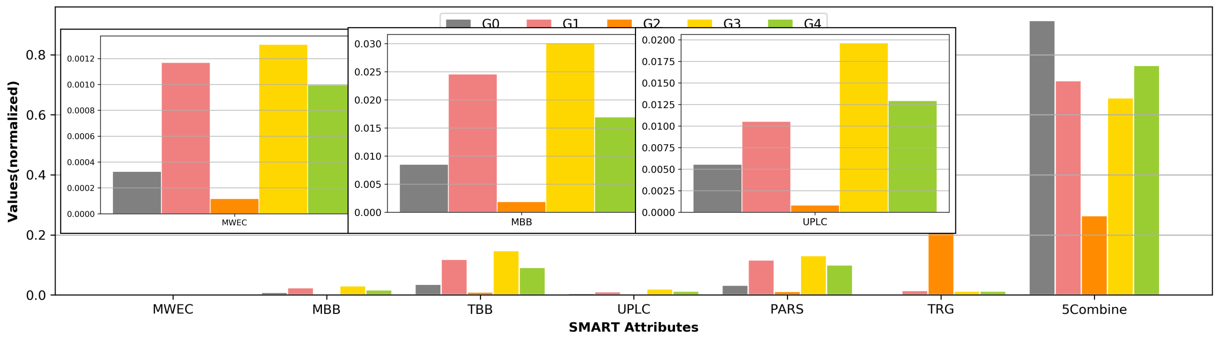
Table 7.11. Groups of SSD SMART Records and Transitions of SSD Health States: Model-A Drives.

Categories	Clusters	Percentage of SSDs in Category
Cluster	Cluster 0	10.7%
	Cluster 1	0.0%
	Cluster 2	1.3%
	Cluster 3	2.0%
	Cluster 4	2.0%
Cluster Transition	Cluster 1 $\rightarrow$ 4	27.3%
	Cluster 3 $\rightarrow$ 1	18.0%
	Cluster 3 $\rightarrow$ 1 $\rightarrow$ 4	38.0%
	Cluster 3 $\rightarrow$ 1 $\rightarrow$ 4 $\rightarrow$ 0	0.7%

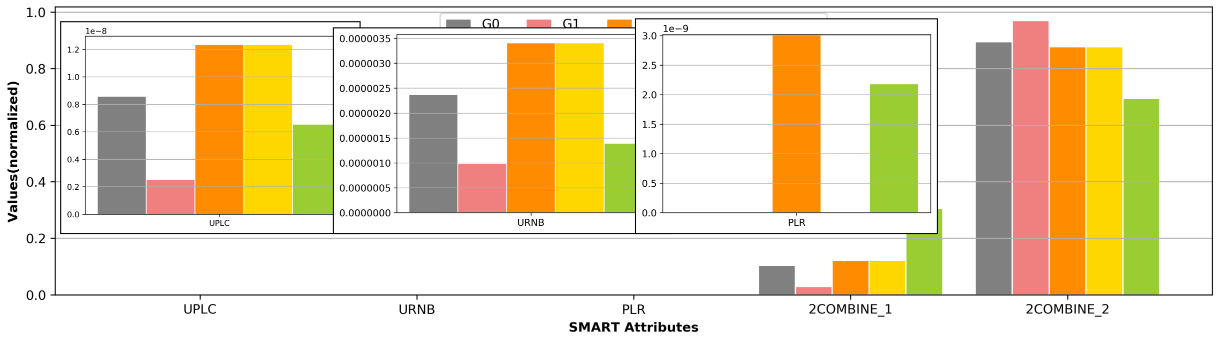
Table 7.12. Groups of SSD SMART Records and Transitions of SSD Health States: Model-B Drives.

Categories	Clusters	Percentage of SSDs in Category
Cluster	Cluster 0	19.8%
	Cluster 1	27.0%
	Cluster 2	0.7%
	Cluster 3	15.6%
	Cluster 4	0.7%
Cluster Transition	Cluster 3 $\rightarrow$ 0	29.8%
	Cluster 0 $\rightarrow$ 3	0.7%
	Cluster 3 $\longleftrightarrow$ 0	4.3%
	Cluster 3 $\rightarrow$ 0 $\rightarrow$ 1	0.7%
	Cluster 1 $\rightarrow$ 0 $\rightarrow$ 3	0.7%

the selected attributes among different SSD groups are shown in Figure 7.29. For Model-A SSDs, we find that 1) drives in Cluster 2 experience I/O intensive operations. The number of read operations is the highest, and the number of write and erase operations (as shown by 5Combine in Figure 7.29) is lower than those in other clusters. 2) Drives in Cluster 0 experience the highest number of write and erase operations, while the number of read



(A) Model A



(B) Model B

FIGURE 7.29. Attributes' Values of Cluster Centroids.

operations is the average. (3) Clusters 1, 3 and 4 include the majority of drives which experience the average number of I/O operations. However, the reliability degradation of SSDs in the three clusters follows similar transition patterns, i.e., Cluster 3 → Cluster 1 → Cluster 4. Along this transition, the value of Total Bad Block (TBB) decreases while the number of write and erase operations increases. Based on the preceding findings, we infer that the health states of SSDs in Clusters 2 and 0 is worse than other drives which are still in good shape. Those good SSDs will experience reliability degradation, i.e., transition to Clusters 1 and 4, as more I/O operations and P/E cycles cause wear and errors.

For Model-B SSDs, we find that 1) drives in Cluster 0, 2 and 3 experience similar write and erase operations for both FTL and host. 2) Drives in Cluster 4 experience the highest number of FTL write operations (as shown by 2COMBINE_1 in Figure 7.29), while drives in Cluster 1 experience the lowest number of FTL writes. 3) For Host write operations (i.e., 2COMBINE_2 in Figure 7.29), drives in Cluster 1 experience the highest number while those in Cluster 4 have the lowest Host writes. 4) Counter-intuitively, PLR cannot indicate the remaining lifetime of an SSD. Only SSDs in Clusters 2 and 4 have PLR > 0, while PLR of SSDs in other clusters remains 0. 5) Clusters 3 and 0 have the majority of drives (that is 70.2% as see in Table III). A half of the drives experiences reliability degradation. Among them, 85.8% of SSDs follow a similar degradation pattern, that is Cluster 3 $\rightarrow$ Cluster 0. Drives in Cluster 3 have more unused NAND blocks than those in Cluster 0. Based on the preceding findings, we believe that SSDs in Clusters 0, 2 and 3 experience the similar write workloads. SSDs in Cluster 3 are in a better health state than those in Cluster 0.

For both models, I/O operations affect SSDs' health status, and even cause reliability degradation of the drives. Workload related attributes play an important role for SSD reliability analysis. As a characteristic of SSD reliability, we discover that the reliability degradation of SSDs follows certain patterns which depend on the model of a drive and I/O workload that the drive performs.

7.5.5. Conclusions

We study SSD-specific SMART data collected from an active production data center. We find that SSDs have many unique attributes compared with HDDs. By analyzing these SSD-specific attributes, we find that they are very useful for characterizing and modeling the health status SSDs at the device level. Our analytic results show that the volume of I/O operations and P/E cycles has a significant impact on the wear level of SSDs. Write and erase related attributes display strong correlations. In a well maintained data center, environmental attributes do not have directly influence SSD reliability. We also observe health state transitions which correspond to SSD reliability degradation. As a future work we plan to further study the reliability degradation process of SSDs and accurately model this process for a deeper understanding and better characterization of SSD reliability in production environments.

7.6. Incorporate Proactive Data Protection into ZFS for Reliable Storage Systems [175][5]

Disk drive failure related data lost remains a major threat to storage system reliability. Current counter-measurement all focused on reactive data protection, such as using RAID or erasure coding to reconstruct data after failure occurs. We argue that post-failure recovery will not scalable when the storage demand keep increase. In this paper, we present a proactive data protection (PDP) framework within the ZFS file system to rescue data from disks before actual failure onset. By reducing the risk of data loss and mitigating the prolonged disk rebuilds caused by disk failures, PDP is designed to enhance the overall storage reliability. We extensively evaluate the recovery performance of ZFS with diverse configurations, and further explore disk failure prediction techniques to develop a proactive data protection mechanism in ZFS. We further compare the performance of different data protection strategies, including post-failure disk recovery, proactive disk cloning, and proactive data recovery. We propose an analytic model that uses storage utilization and contextual system information to select the best data protection strategy that can save up to 70% of data rescue time.

7.6.1. Introduction

The big data and machine learning applications have driven the storage demand to exabyte of capacity, which supplied by millions of disk drives. At such a scale, disk failures become the norm. Redundant array of inexpensive disks (RAID) is widely used to enhance the performance and reliability of storage system. However, RAID recovery is a time-consuming process which demands considerable computing resources and stalls user applications due by bringing the RAID system offline for repair or consuming a large portion of the I/O bandwidth. In recent years, the dramatic growth of disk capacity far outpaces the improvement of disk speed, resulting in an even longer RAID recovery time. It takes days and even weeks, to recover an enterprise-grade RAID group composed of helium-filled

[5] Section 7.6 is reproduced in its entirety from Zhi Qiao, Jacob Hochstetler, Shuwen Liang, Song Fu, Hsing-bung Chen, and Bradley Settlemyer, Incorporate proactive data protection in ZFS towards reliable storage systems, 2018 IEEE 16th International Conference on Dependable, Autonomic and Secure Computing, 16th International Conference on Pervasive Intelligence and Computing, 4th International Conference on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), pp. 904-911, with permission from IEEE.

large-capacity hard drives. Furthermore, the RAID recovery process places additional stress on the remaining disk drives due to the intensive read and write activities. During the day or week-long RAID recovery, it becomes increasingly likely that other disk drives(s) in the same RAID group may become failed. Such multiple disk failures can cause data loss and high monetary cost. The prolonged disk rebuilds also compromise the overall system performance, as data accesses to the affected RAID system are delayed for a long period of time.

The existing methods and products are mostly reactive, providing disk or storage remediation after failures occur. Reactive data protection schemes suffer from high recovery overhead which affects storage availability and system performance. Proactive data protection (PDP), on the other hand, can explore the lead time prior to disk failures to overlap data rescue with regular storage operations. From the system's perspective, the disk drives and storage system continuously service I/O requests without obvious disruptions. Failure prediction is an enabling technology for proactive data protection, as it allows data rescue mechanisms to be performed before failures truly happen. Disk manufacturers embedded SMART (Self-Monitoring Analysis and Reporting Technology) monitoring in their products, which reports the health status of a disk. Using such data, studies leveraging advanced statistic models and machine learning technology show promising results of predicting disk failures ahead of their actual occurrences. Recently, many researchers used advanced machine learning algorithms and obtained accurate prediction results. Mahdisoltani et. al. [141] evaluated the effectiveness of a set of machine learning techniques and discussed the proactive error prediction method that aims to improve storage reliability. Their work inspires us to incorporate failure prediction in ZFS file systems to develop a cost-effective data protection solution with important contextual information.

In this paper, we propose a proof-of-concept, proactive data protection scheme that explores failure prediction in ZFS file systems to enhance the storage reliability and reduce the risk of data loss caused by disk failures. By performing counter measures prior to disk failures, PDP mitigates the performance impact caused by lengthy disk rebuilds. Due to the popularity in both the research and industrial communities, we choose the ZFS file system to implement a prototype proactive data protection system. This combination of file-system-level control and data protection with disk-drive-level reliability monitoring and failure prediction provides a pragmatic and holistic approach to build highly reliable storage systems. The disk-drive failure prediction technologies enable ZFS to proactively move or re-compute data from a failing disk to an available and healthy drive. Since there are many factors that affect ZFS' recovery performance, we first evaluating the I/O and recovery performance of ZFS (in Section 7.6.3) using different system configurations. We then propose three PDP strategies (in Section 7.6.4) designed for ZFS. Our performance evaluation of these strategies shows that ZFS can make an optimal decision with the help of contextual file-system information. That is, ZFS can determine the best PDP strategy based on the current system state.

7.6.2. Data Protection in ZFS

ZFS is a state-of-the art open sourced file system that provides unparalleled storage capability, efficiency, and reliability. It is a 128 bit addressed filesystem that can store upto 16 exbibyte (2^{64} bytes) files. ZFS has been widely adopted in HPC storages, data centers, and Network Attached Storages(NAS). It features copy-on-write transactional model, pooled storage, deduplication, filsystem snapshot and clone, etc. In addition, it also provide many reliability centric design such as data block hierarchical checksums, automatic data recovery, and software RAID implementation. We only interested at ZFS's software RAID in this discussion. In terms of ZFS, *Stripe* is equivalent to RAID 0, whereas *Mirror* corresponds to RAID 1 and *RAID-Z* or *RAIDZ-1* and *RAIDZ-2* are equivalent to the standard RAID 5 and RAID 6, respectively.

In ZFS the RAID rebuild process is called *resilvering*. This comes from the word used for the actual repairing of physical mirrors. Resilvering occurs when a disk needs to be replaced due to either failure or data corruption. If a disk fails unexpectedly, the ZFS rebuild algorithm reassembles data on a new disk using either mirrored or parity data. The resilvering process can take an extended period of time, depending on the size of the drive and the amount of data that needs to be recovered.

During resilvering, the performance of a RAID array is degraded because 1) the system resources are usually prioritized for data recovery, and 2) the rest of the disks in the group cannot provide optimal data redundancy for the configured RAID level. The resilvering process causes extra wear and accelerates the failure of the remaining healthy disks. Two or more disk failures can occur when I/O workloads stress individual disks too much. Therefore, the total number of disk failures might exceed the maximum fault tolerance for the RAID level, which causes resilvering to fail. The RAID array then becomes unavailable once such nested failures occur. If no additional backup exists, data on that array will become lost.

7.6.3. How ZFS Performs: Expected and Unexpected

ZFS has been widely used in production storage systems, due to its support for high storage capacity, efficient data compression, integrated volume management and reliability management features. Data resilvering in ZFS is reactive, that is, data and parity blocks are read, regenerated, and stored after failures are detected. Although there are a few works that evaluate the performance of ZFS [99] [166] [150], they focus on certain features such as data compression or the read/write speed as a file system. Little work has been conducted to understand the performance of the new fault management techniques in ZFS. In this section, we evaluate the performance of ZFS' software RAID and the cost of the resilvering process. These results help us obtain a deeper understanding of ZFS' fault management mechanisms influencing the design of our proactive data protection scheme.

7.6.3.1. Test Platform Configuration

Table 7.13 shows the parameters and their values used in our experiments. The storage servers we used in this test were equipped with eight Intel Xeon cores (3 GHz), 32 GB DRAM, Ubuntu 16.04 LTS, and ZFS version 0.6.5.11. The disk drives model we used in the tests are Seagate BarraCuda ST2000DMs (magnetic HDD) and Intel DC3520s (data-center class SSD). The HDDs and SSDs are installed on separate storage server. We use the Bonnie++ benchmark suite to test ZFS' I/O performance. Specifically, we use

Table 7.13. Test Platform Configuration and Experiment Setting

System Configuration	**Setting**
Disk Media	HDD (magnetic spinning), SSD (solid-state flash)
RAID (ZFS) Level	0 (*stripe*), 1 (*mirror*), 5 (*raidz*), 6 (*raidz2*)
RAID stripe size	2 - 6
DRAM Size for ZFS	8, 16, 32 GB
Storage Utilization	0% - 90%

the three subtests 'sequential output', 'sequential input', and 'rewrite' to evaluate the file system functions *write*, *read*, and *modify*. Note that we use the sequential I/O to simulate the workload of typical Big Data application, where terabytes of data were reads and writes intensively. Since ZFS is memory intensive, we vary the system DRAM size to characterize its performance.

To evaluate the resilvering performance of ZFS, we first developed a tool that systematically recreate the disk failure without physical contact with hardware. By default, the first and last 1MB of each disk is reserved by system. In addition, ZFS formats the disk appropriately by assigning its own partition table and meta-data blocks. This reserved area is usually the last 2^{14} byte, or 8MB, next to system reserved blocks in each disk. We inject a stream of zeros into target disk, until the entire ZFS reserved area was flushed. As illustrated in Figure 7.30, the compromised *ZFS reserve* area disconnected the IO channel of ZFS from host system, thus mimic a disk logical failure signal and OS will consider it as a failure disk. Then we immediately issue a system scrub to activate the default ZFS resilvering process. We measure the turnaround time of resilvering and the amount of data that ZFS recovers. During two month of our experiment period, we've simulated hundreds of disk drive failures event, which would take a real world data center few years to generate the equivalent amount of failures with normal operation.

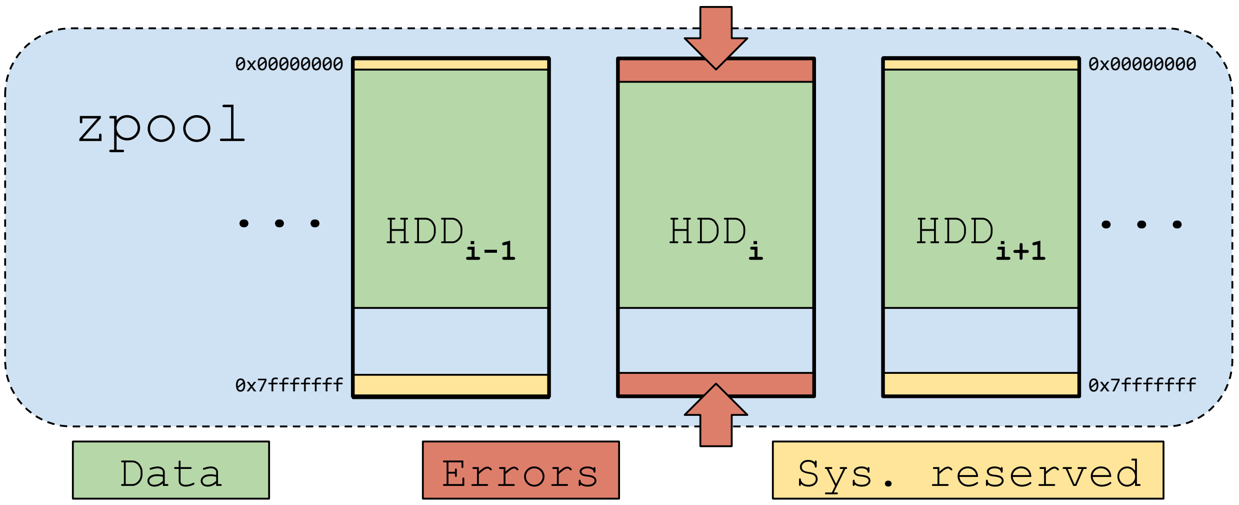
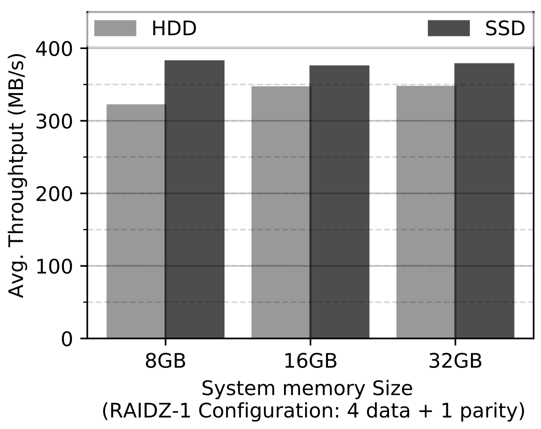


FIGURE 7.30. Injecting errors to disk drive creates a false alarm of disk failure.

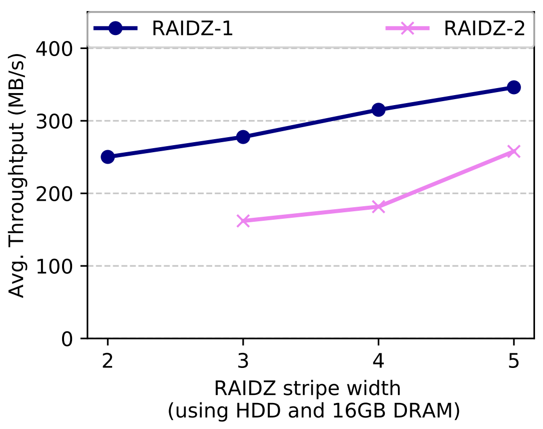
7.6.3.2. Performance Modeling of I/O and Resilvering in ZFS

We run each benchmark five times and compute the average of results. Each of the five runs is within 5% of the average. Thus we believe the experimental results are relatively stable. We present the average values of the results in the following figures for a clear presentation and interpretation.

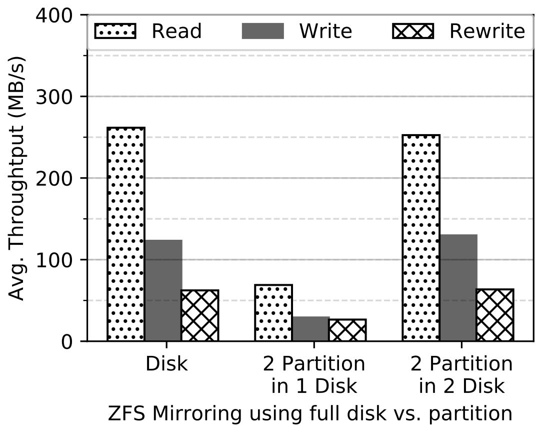
Figure 7.31 shows a list of finding from our I/O performance experiments. We start our discussion with a counter-intuitive result. Figure 7.31a compares the I/O throughput using different memory size. The experiment is conducted on two storage server that equipped with either HDD or SSD. ZFS requires a large amount of DRAM for file caching and metadata management. But during our sequential I/O benchmarking, larger DRAM size leads to about 10% of improvement in HDD-based RAID array, where SSD-based array does not show significant benefit from using more DRAM. However, we cannot simply assume DRAM size is not important for ZFS, since our synthetic benchmark only consider sequential I/O performance. Real-world workload consist of both random and sequential I/Os. The random data access pattern in ZFS might gain huge benefit from larger DRAM size. Similarly, Figure 7.31a also reveals that for sequential I/O, SSD only outperform HDD slightly. If workload consist of mostly large I/O, such as full system check-pointing or write-out petabyte of data to archive storage, using SSD as storage media may not cost-effective. Instead, current



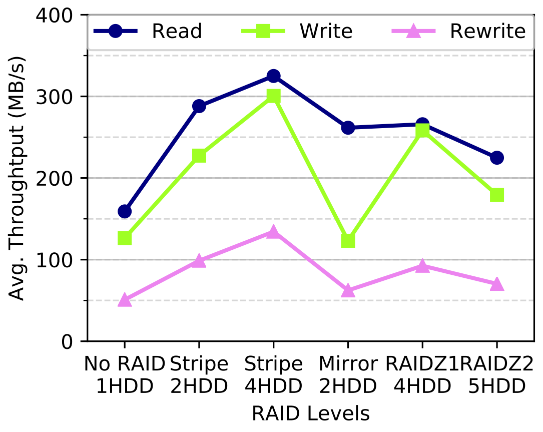
(A) Use SSD and larger DRAM



(B) Enlarge RAID stripe width



(C) Partitions in RAID setup



(D) Various RAID levels

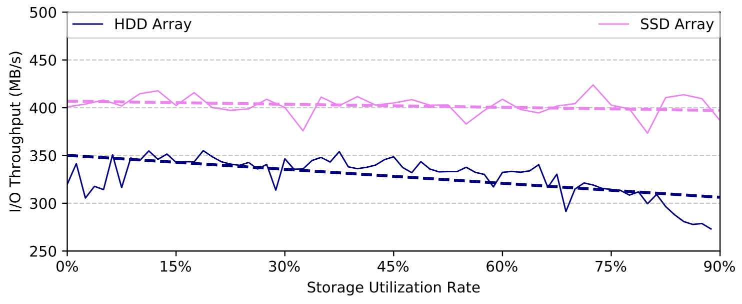
FIGURE 7.31. I/O throughput comparison between each RAID configurations

best practices, which using SSD as Separate Log Devices (SLOG) and Level 2 Adaptive Replacement Cache (L2ARC) to improve read/write performance, can leverages the SSD's benefit of high random I/O throughput.

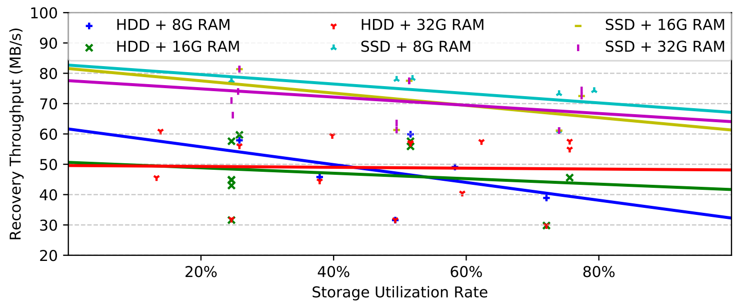
The rest of the findings are much straightforward and similar to what we can observed from system with hardware RAID controller. For example, our experimental results as presented in Figure 7.31b show that for the software RAID in ZFS, **increased stripe width improves I/O throughput and the improvement is super-linear**. The striping process distributes file chunks across multiple disks, which means a file access request is served by all disks within the stripe. The x-axis in the figure is the stripe width of RAID array. When the stripe width is increased from 2 to 5 in a RAIDZ1 array, or from 3 to 5 in a RAIDZ2 array, the overall I/O throughput increases linearly. Figure 7.31c compares the ZFS mirroring performance of using full hard drives versus using partitions. When mirroring two partition on one disk drives, both I/O bandwidth and data management processor are shared by each partitions. As a result, the overall throughput degraded significantly. For ZFS, if one have to use partition instead of full disk drives, placing partitions on different drive can avoid compromising the performance. Figure 7.31d shows the average throughput of ZFS using different RAID configurations. To choose the appropriate RAID level, we often need to trade off the performance and available capacity for added reliability. For example, this figure shows that RAIDZ1 and RAIDZ2 configuration have the same amount of available capacity (3 HDD), but the performance of RAIDZ2 is lower than RAIDZ1 for double failure tolerance.

Our finding reveals that the software RAID in ZFS generally follows the similar performance pattern as we can expected from system with hardware RAID controller. As a result, administrator's domain knowledge still plays an important role in managing ZFS storage. And this similar performance pattern also allows minimum learning curve and ease the transition to software based RAID.

Figure 7.32 illustrate the performance decrease of ZFS I/O and recovery shows positive correlation with the increased storage utilization rate. ZFS employs *copy-on-write* (COW), that is a new copy is created only when the data is modified. It efficiently shares duplicate data to avoid unnecessary resource consumption. However, COW causes more fragmentations as the utilization of the storage increases. Moreover, when the storage usage increases to approximately 80%, ZFS switches to a space-conserving (rather than speed-oriented) mode to preserve working space on the volume. In our experiment, we profile ZFS and measure its throughput as we ramp up zpool's utilization from 0% to 90%. Figure 7.32a presents the experimental results. The solid curve shows the average throughput of ZFS under the RAIDZ1 configuration, and the dashed curve is the corresponding trend-line using a linear regression. From the figure, we can see that **as the utilization of zpool increases, the throughput of ZFS decreases significantly in the RAID array using HDDs**.



(A) I/O throughput

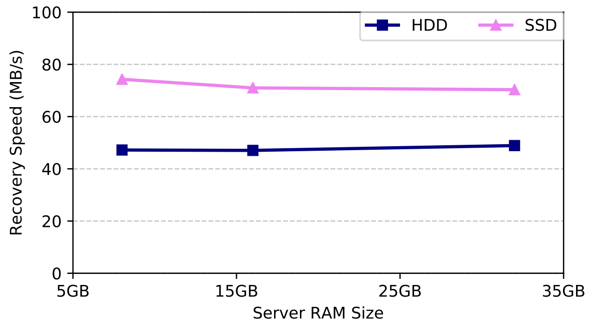


(B) Recovery throughtput

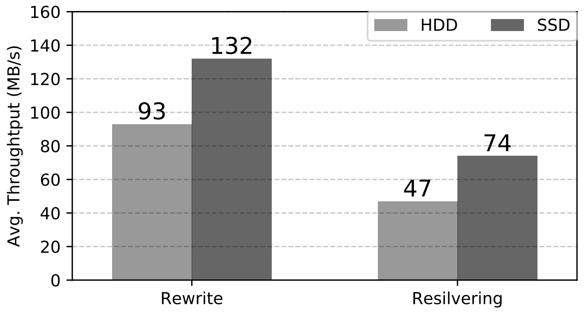
FIGURE 7.32. Storage utilization influences ZFS' I/O and Recovery performance.

As a zpool becomes fully used, ZFS' throughput drops by up to 25%. Similar to the preceding results, **the recovery throughput degrades as the zpool utilization increases**. Figure 7.32b shows the recovery throughput under different zpool utilization for both HDDs and SSDs based array. The least-squares linear regression model fits the results the best. We observe that the recovery performance decreases as the zpool utilization increases. When the storage array is close to a full utilization, the recovery speed is about 37% slower than the average throughput, and 47% slower than the peak speed. The first and last steps of the resilvering process consist of reads and writes. Therefore the performance degradation due to increased zpool utilization plays a major role for the reduced recovery speed.

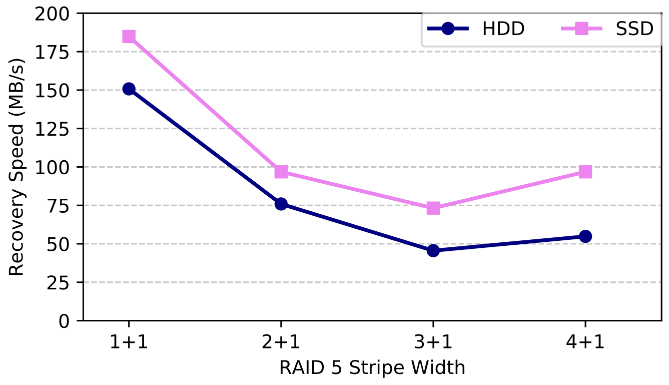
The results from the RAID recovery experiments show that **adding more DRAM does not always improve ZFS' recovery performance**. Figure 7.33a shows that using more DRAM for an HDD-based array improves the recovery speed by 3.8% on average, while the performance on an SSD-based array drops by 5.6%. In order to explain this phenomenon, we decompose the resilvering process of ZFS into three steps: 1) reading data and parity



(A) With the RAM size increases, rebuild performance only shows negligible differences.



(B) Rewrite performance compare with recovery performance.



(C) Recovery performance is not monotonically decreasing when stripe width are increasing.

FIGURE 7.33. Recovery Performance Comparison

blocks, 2) computing the data that is lost, and 3) writing the recovered data to a spare. The performance of the second stage (data recovery computation) significantly influences the overall resilvering speed. We can compare the resilvering process with the *rewrite* operations that Bonnie++ performs, which also consists of reading, modifying, and writing blocks of data. From Figure 7.33b, we can see that the **recovery is a time-consuming process and the CPU performance is a dominant factor**. Although larger DRAM accommodates more metadata and file caching for regular I/O operations, the resilvering process rarely uses cached files. Hence, the DRAM size does not significantly affect the recovery performance of ZFS. Unlike I/O workload which can be serviced in parallel, resilvering is more computation intensive that reconstructs data sequentially. Although employing more disks in the rebuild process can improve the aggregated I/O throughput, the wider stripe increases the amount of data and parity used during the resilvering calculation which offsets the throughput gain. Figure 7.33c shows the **recovery speed does not monotonically decrease as the stripe expands**.

7.6.4. Proactive Data Protection

Disk access speed has been outpaced by the increasing capacity. The stripe width of RAID arrays have grown to fill the gap between disk speed and capacity. Unfortunately, the probability of having double and even triple failures also increases as the disk recovery time is significantly prolonged. Although ZFS supports RAID 6 and RAID 7 (triple parity) to handle disk failures, the longer performance degradation and even unavailability of disk arrays compromise the overall system performance and users' satisfaction. Complementary to post-failure disk rebuilds, data can be rescued proactively prior to disk failures. This is enabled by disk failure prediction techniques [83] [111] [22] which forecast when failures will happen on which drives with promising accuracy. We aim to incorporate disk failure prediction methods in ZFS so that ZFS becomes capable of replacing a failing drive before the failure actually happens. Data on the failing drive (in contrast to failed drive) can be moved to a spare drive without the expensive disk rebuild process, thereby avoiding service disruption of the storage system. In addition, proactive data protection can be scheduled to perform during off-peak hours, which can further improve storage availability and achieve service-level objectives (SLO). Disk failure prediction remains an active research topic. In this work, we leverage the existing techniques of prediction disk failures and explore them in ZFS to develop proactive data protection. Research on the prediction methods is not the focus of this paper. Table 7.14 lists the variables that we use in the following discussion and analysis.

7.6.4.1. Proactive Strategies for Handling Disk Failures

A disk failure prediction model (FPM) uses monitored, real-time status data of disk drives to compute the probability at which the drives will fail in the future. At time t_i, if

Table 7.14. Variables Used in the Analysis of Proactive Data Protection and Strategy Selection

Variables	**Description**
t_l	Lead time of a failure prediction
t_i	Time when a failure prediction is performed
t_f	Time when a disk failure is predicted to happen
t_v	Validation period of a prediction
T	Duration of the data rescue process
S	Data rescue speed
A	Amount of data to be rescued
W	Wasted time due to failure misprediction
p	Precision of failure prediction

FPM predicts that a disk is going to fail, it reports the predicted failure occurrence which will happen at time t_f. We use lead time t_l, i.e., the length of time between the point when FPM makes prediction and the predicted failure occurrence time, to represent the *urgency* of a failure. We can calculate the lead time using $t_l = t_f - t_i$. Proactive data rescue from the failing disk to a spare or available disk also takes time. We use T to denote the time that a proactive strategies uses to rescue data. If $t_l > T$, then proactive data rescue can ensure the safety of the data on the failing drive(s). In this paper, we discuss several proactive strategies to handle disk failures. To proactively rescue data on a failing disk, we need to calculate the estimated recovery time T for each proactive strategy and compare it with the lead time t_l.

(1) **P**roactive **Dis**k **Clo**ning (P-DISCO)

(2) **P**roactive **A**ctive-**da**ta **R**ecovery (P-ADAR)

(3) **P**roactive Active-**da**ta **Clo**ning (P-DACO)

The *Proactive Disk Cloning*, or P-DISCO, migrates data on a predicted, failing drive to a hot spare using disk cloning. In the conventional RAID rebuilds, data reconstruction involves massive data movement and intensive parity computation using data and parity blocks from all of the other disks in an array. CPU and interconnect become the performance bottleneck as they determine the recovery throughput. Disk cloning, on the other hand, moves the data from the failing disk only to a spare one without involving other disks or calculating parities. Theoretically, proactive disk cloning can achieve a much higher throughput. Moreover, the recovery speed is independent of the disk space utilization. In contrast, the performance of ZFS' resilvering process is affected by disk utilization as shown in Section 7.6.3.2. Figure 7.34 compares the two recovery strategies, i.e., ZFS' resilvering and proactive disk cloning, for a fully used drive. From this figure, we can see P-DISCO completed the data rescue in 17.59 hours for an 8TB HDD, and in 4.17 hours for a 3.2TB SSD. More importantly, the data rescue happened before the actual disk failure, thereby avoiding operating in a degraded state. On the other hand, after a disk failure, the default ZFS data recovery took over two times longer to rebuild the same disk. As a result, the data reconstruction process had to compete with other workloads for limited system resources, which prolong the recovery time and the dangerous time spent in a degraded state. However, the volume manager masks the data location to make it transparent to the system. Without knowing which disk sectors contain useful data, disk cloning has to copy everything from each sector to the spare drive. For a nearly empty disk, P-DISCO may not be time or resource optimal.

$$T_{P-DISCO} = \frac{A_{disk\ capacity}}{S_{cloning}} \tag{7.5}$$

ZFS, combining both a file system and volume manager, has the advantage of keeping track of "active data" and the ability to only recovery those tagged data areas when a disk fails. This results in an efficient data rescue in a controlled manner. *Proactive Active-data Recovery*, or P-ADAR, enables ZFS to proactively rescue only active-data to the spares. Although resilvering is a computationally intensive process, recovering only the minimum amount of necessary data can save time. Because the amount of active data in a disk is always less than its capacity, $A_{active\ data} < A_{disk\ capacity}$. Additionally, ZFS resilvering is safe to interrupt. If power loss or reboot occurs during data rescue, the resilvering process resumes at the exact location where it left off without manual intervention. However, the

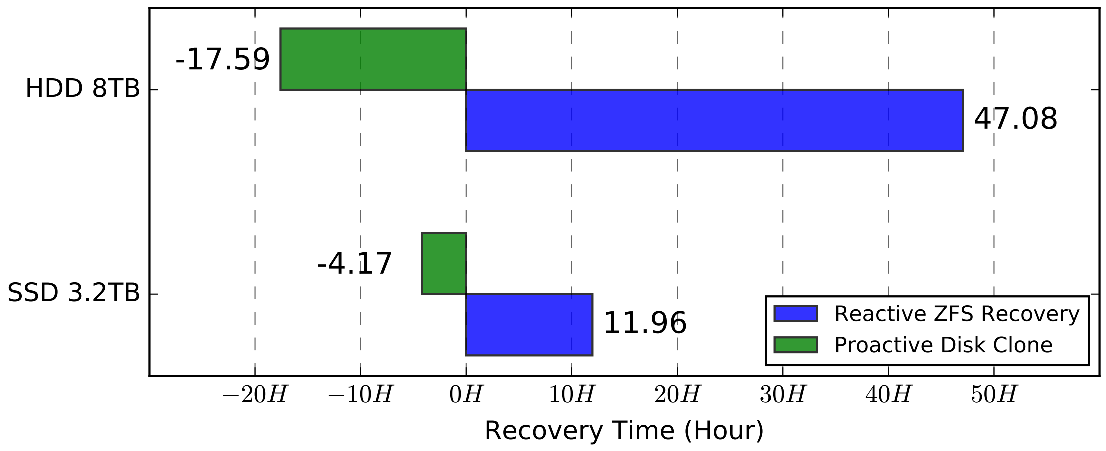


FIGURE 7.34. Time used for proactive disk cloning and post-failure disk recovery for a fully-utilized zpool. ***0H*** denotes the disk failure occurrence.

resilvering process is much more complex than cloning, which involves a full disk scan to gather active data locations, followed by CPU-intensive data reconstruction. Therefore, $S_{resilvering} < S_{cloning}$. The actual recovery time T equals

$$T_{P-ADAR} = \frac{A_{active\ data}}{S_{resilvering}} \tag{7.6}$$

Further improving the speed of recovery, we leveraged the computationally-light cloning process to handle the minimum necessitated data marked with the ZFS active-data tag. The proposed *Proactive Active-data Cloning*, or P-DACO, only clones the active data during rescue, and eliminates the need to compute parity. It combines the benefits from the previous two strategies, so the recovery times equals to

$$T_{P-DACO} = \frac{A_{active\ data}}{S_{cloning}} \tag{7.7}$$

To measure the value of S and A, a set of daemons will frequently check all disk's SMART data and the zpool utilization percentage. A daemon also runs a series of micro-benchmarks to determine the runtime I/O performance and data recovery speed S. The amount of data to rescue A will be determined when the proactive strategies starts to rescue data. Meanwhile, the SMART data of each disk in the system will be streamed to FPM for failure prediction and health status analysis. Since FPM uses a pre-built disk failure model,

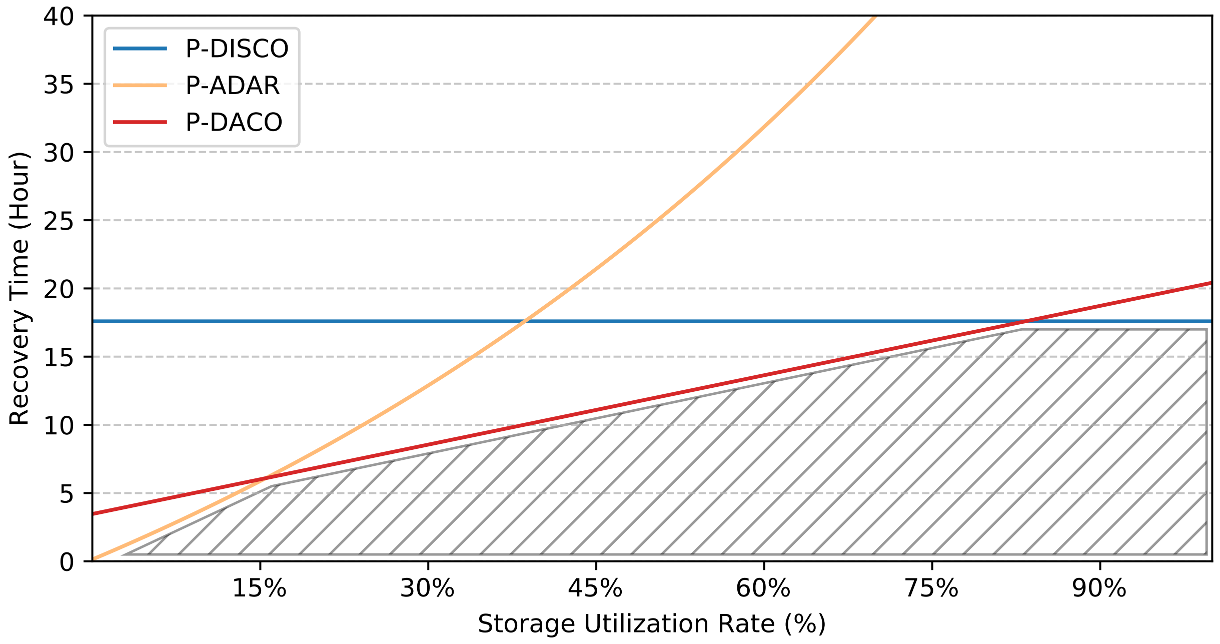


Figure 7.35. Time used by each proactive strategies for data rescue.

runtime SMART data constantly improves the performance of the prediction.

7.6.4.2. Selecting the Optimal Strategies

From the previous discussion, we can see that each proactive strategy is only suitable for a specific system condition. How to choose the appropriate strategy to handle disk failure becomes critical, and can yield up to 70% faster recovery time. Figure 7.35 illustrates the performance of each proactive strategy for recovering data on an 8TB enterprise-grade HDD with different storage utilizations. We observed that P-ADAR is more efficient when the zpool is lightly used (below 15%). As more active data is stored, P-DISCO becomes a better choice. P-DACO seems like a promising proactive strategy that yields better result than P-ADAR and P-DISCO most of the time, but P-DACO cannot tolerate interrupts during data rescue, so it trade-off the dependability for the performance gain.

To systematically pick the optimal strategies to handle disk failure, we set two constraints for the selection process: the *urgency*, and the *dependability*. The prior indicates how fast each proactive strategy completes data rescue, where the latter measures if proactive strategy is safe to interrupt. For example, to minimize the system downtime caused by disk failure, we can prioritize *urgency* over *dependability*. In this case, if FPM provides enough lead time, where $t_l > min(T_{P-DISCO}, T_{P-ADAR}, T_{P-DACO})$, then we can select the strategy with shortest data rescue time T, as illustrated by shaded area. For a system that prioritizes *dependability* over *urgency*, we adopt a simpler approach to narrow down the strategy selection set, since *active-data resilvering* is the only strategy that is safe for interrupt. Therefore, if FPM provides enough lead time for P-ADAR, we choose it as the proactive strategy to handle disk failure. Otherwise, the default reactive data rescue, or R-ADAR, will take over the data rescue.

7.6.4.3. Cost Analysis

In practical use, each FPM only performs well for certain drives. Others may generate excessive false alarms that introduce considerable amount of overhead to the system. We dedicate this subsection to discuss the effectiveness and cost of proactive data protection.

When FPM predicts a failure at time t_f, it also provides a valid period t_v, which indicates the failure occurrence is likely within $t_f \pm \frac{1}{2}t_v$. We can calculate the adjusted lead time as $t'_l = t_f - \frac{1}{2}t_v - t_i$. If the disk is still healthy beyond $t_f + \frac{1}{2}t_v$, we say this is a false positive. For proactive data protection, the cost of a false positive is a wasted healthy disk, in addition to the wasted system resources used during data rescue. If the actual failure occurs before $t_f - \frac{1}{2}t_v$, or FPM didn't report such a failure, we say this is a false negative. In this discussion, we assume the worst case of false negative, i.e., there is no proactive strategy to rescue data at all. Since proactive data protection should always use reactive data rescue (R-ADAR) as backup, the cost of false negative becomes the cost of R-ADAR, which is identical to the system that did not use proactive data protection. The default R-ADAR and proactive strategy P-ADAR takes the same time to complete data rescue ($T_{P-ADAR} = T_{R-ADAR}$), the only difference is their starting time. At time t_i, if FPM predicts a failure will occur by time t_f, P-ADAR will start data rescue immediately, and complete by $t'_l - T_{P-ADAR}$. This is equivalent to $t_f - \frac{1}{2}t_v - T_{P-ADAR}$. For R-ADAR, the rescue process starts only after the actual disk failure occurrence t'_f, and completes by $t'_f + T_{R-ADAR}$. From the definition of the *precision*, we know there are p possibility that

$t_f = t'_f$. Once the FPM provides enough lead time for proactive strategies, or $t'_l > T$, we can conclude that proactive strategy can complete data rescue even before reactive data rescue starts. For the $(1-p)$ possibility that FPM made a wrong prediction, the cost is not worse than the default reactive strategy. Therefore, proactive data rescue improves the overall system reliability by minimizing the data lost risk caused by disk failure.

As Eckart et al. discussed in [63], proactive data rescue can improve overall system reliability even if the failure prediction rate is as low as 40-50%. Moreover, proactive data rescue is not a replacement of the default ZFS resilvering process, but rather an approach to enhance overall system reliability. It can resolve the bandwidth competition between ZFS resilvering processes and the regular I/O workload. Proactive data rescue is complementary to reactive action for handling disk failures and as reactive action are always applicable, proactive data rescue should be combined with reactive action to address false negative prediction.

7.6.5. Conclusions

Ensuring the reliability and availability of storage systems is crucial. Currently fault management such as hardware or software RAID, are reactive, meaning the data recovery process onset after the failure. In this paper, we proposed a proactive data protection scheme that combines machine learning based disk failure prediction techniques in to ZFS filesystem. We use our finding from performance benchmarks to develop an analytic model that derives the optimal data rescue strategy. Our analytic model uses contextual information of the storage system to find the best proactive data recovery strategy that suits the storage array in a cost-effective way. In this proof-of-concept work, we demonstrate that proactive data protection can effectively rescue data ahead of disk failure. It also reduce the chance of data lost during failure recovery. And system level scheduling for data protection also helps to avoid I/O conflict between workload and recovery process. Integrate proactive data protection into ZFS provides additional reliability measurement beyond reactive failure recovery.

7.7. Developing Cost-Effective Data Rescue Schemes to Tackle Disk Failures in Data Centers [174][6]

Ensuring the reliability of large-scale storage systems remains a challenge, especially when there are millions of disk drives deployed. Post-failure disk rebuild takes much longer time nowadays due to the ever-increasing disk capacity, which also increases the risk of service unavailability and even data loss. In this paper, we present a proactive data protection (PDP) framework in the ZFS file system to rescue data from disks before actual failure onset. By reducing the risk of data loss and mitigating the prolonged disk rebuilds caused by disk failures, PDP is designed to enhance the overall storage reliability. We extensively evaluate the recovery performance of ZFS with diverse configurations, and further explore disk failure prediction techniques to develop a proactive data protection mechanism in ZFS. We further compare the performance of different data protection strategies, including post-failure disk recovery, proactive disk cloning, and proactive data recovery. We propose an analytic model that uses storage utilization and contextual system information to select the best data protection strategy to achieve cost-effective and enhanced storage reliability.

7.7.1. Introduction

Nowadays, big data applications require storage systems to possess exabytes of capacity, provided by millions of hard disk drives. At such a scale, disk failures become the norm. Redundant array of inexpensive disks (RAID) is a common practice in most large-scale storage systems for redundancy and performance[79]. In the face of disk failures, a RAID system uses parity data on the remaining, working disk drives to compute and recover the lost data on a spare drive. However, RAID recovery is a time-consuming process which demands considerable computing resources and stalls user applications due by bringing the RAID system offline for repair or consuming a large portion of the I/O bandwidth. In recent

[6] Section 7.7 is reproduced in its entirety from Zhi Qiao, Jacob Hochstetler, Shuwen Liang, Song Fu, Hsing-bung Chen, and Bradley Settlemyer, Developing cost-effective data rescue schemes to tackle disk failures in data centers, International Conference on Big Data, (Francis Y. L. Chin, C. L. Philip Chen, Latifur Khan, Kisung Lee, and Liang-Jie Zhang, eds.), 2018, pp. 194-208, with permission from Springer International Publishing.

years, the dramatic growth of disk capacity far outpaces the improvement of disk speed, resulting in an even longer RAID recovery time. It takes days and even weeks, to recover an enterprise-grade RAID group composed of helium-filled large-capacity hard drives. Furthermore, the RAID recovery process places additional stress on the remaining disk drives due to the intensive read and write activities. During the day or week-long RAID recovery, it becomes increasingly likely that other disk drives(s) in the same RAID group may become failed. Such multiple disk failures can cause data loss and high monetary cost. The prolonged disk rebuilds also compromise the overall system performance, as data accesses to the affected RAID system are delayed for a long period of time.

The existing methods and products are mostly reactive, providing disk or storage remediation after failures occur. Reactive data protection schemes suffer from high recovery overhead which affects storage availability and system performance. Proactive data protection (PDP), on the other hand, can explore the lead time prior to disk failures to overlap data rescue with regular storage operations. From the system's perspective, the disk drives and storage system continuously service I/O requests without obvious disruptions. Failure prediction is an enabling technology for proactive data protection, as it allows data rescue mechanisms to be performed before failures truly happen. Disk manufacturers embedded SMART (Self-Monitoring Analysis and Reporting Technology) monitoring in their products, which reports the health status of a disk. But they set the thresholds of SMART attributes for disk failure detection as low as possible in order to minimize the false alarm rate. As a result, disk drives fail way before these thresholds are reached in the field, which makes the proactive RAID controllers inefficient to protect data.

Studies leveraging advanced statistic models and machine learning technology show promising results of predicting disk failures ahead of their actual occurrences. For early attempts such as [151] [152], the effectiveness of failure prediction was questioned, for example in [167]. Recently, many researchers used advanced machine learning algorithms and obtained accurate prediction results. Mahdisoltani et. al. [141] evaluated the effectiveness of a set of machine learning techniques and discussed the proactive error prediction method that aims to improve storage reliability. Their work inspires us to incorporate failure prediction in file systems to develop a cost-effective data protection solution with important contextual information.

As an open-source high performance file system, ZFS has been used by data centers in their production storage systems, as well as by industry vendors in their software-defined storage products. ZFS implements software RAID and efficient data compression. It was designed from scratch to address fault management issues in storage systems. ZFS can detect and repair silent disk corruptions[19], and rebuild software RAID from disk failures. The performance of RAID recovery in ZFS is affected by many factors, including:

- storage pool utilization,
- the number of virtual disks in a RAID group,
- software RAID configuration,
- the amount of corrupted data that needs to be recovered.

Hardware configuration of the storage system also affects the performance of ZFS recovery through

- the use of flash drives,
- the amount of DRAM available,
- CPU cycles available.

In this paper, we propose a proof-of-concept, proactive data protection scheme that explores failure prediction in file systems to enhance the storage reliability and reduce the risk of data loss caused by disk failures. By performing counter measures prior to disk failures, PDP mitigates the performance impact caused by lengthy disk rebuilds. Due to the popularity in both the research and industrial communities, we choose the ZFS file system to implement a prototype proactive data protection system. This combination of file-system-level control and data protection with disk-drive-level reliability monitoring and failure prediction provides a pragmatic and holistic approach to build highly reliable storage systems. The disk-drive failure prediction technologies enable ZFS to proactively move or re-compute data from a failing disk to an available and healthy drive. Since there are many factors that affect ZFS' recovery performance, we first present an analytic model (in Section 7.7.3) for evaluating the I/O and recovery performance of ZFS using different system configurations. Due to the increasing adoption of flash drives in storage systems, we conduct our experiments in a heterogeneous storage environment that consists of both magnetic hard disk drives (HDD) and solid-state drives (SSD). We then propose three PDP strategies (in Section 7.7.4) designed for ZFS. Our performance evaluation of these strategies shows that ZFS can make an optimal decision with the help of contextual file-system information. That is, ZFS can determine the best PDP strategy based on the current system state. The major contributions of our paper are as follows.

- We extensively evaluate the performance of software RAID and observe that the performance of I/O and data recovery in ZFS degrades as the storage utilization increases, which influences the selection of data protection strategies.
- As far as we know, we are the first to integrate proactive data protection with file systems and leverage the strengths of both to enhance storage reliability. The contextual file-system information enables the maintenance (such as disk drives early retirement and proactive data migration) to be scheduled at off-peak time, thus preserves the valuable I/O and network bandwidth for user's request.
- We provide a quantitative analysis of each proactive data protection strategy and design a method to select the optimal strategy by exploring the run-time system status and cost functions to best protect storage data.

7.7.2. Background

7.7.2.1. RAID, Striping, and Levels

RAID (Redundant Array of Independent Disks, originally Redundant Array of Inexpensive Disks) is a virtualized storage subsystem that combines multiple physical disk drives into one or more logical units to improve the performance and fault tolerance of storage systems. Over the years of development, several *RAID levels* have evolved. These include the standard schemes such as mirroring data and striping data, as well as other variations such as nested levels and proprietary schemes. Depending on the specific RAID level, an array of multiple disks can be configured to achieve a balance between performance, reliability, and capacity. Many RAID levels employ an error protection scheme called *parity*, which uses a relatively small amount of data recover more user data. *Striping* is the underlying concept of all RAID levels other than RAID 1 (mirroring). Striping is a mechanism to split up disk partitions into stripes of contiguous sequences of disk blocks. A RAID Stripe usually consists of multiple data blocks and one or more parity blocks. Disk striping without any data redundancy (or parity) is RAID 0 and is used for increasing the I/O performance. Different RAID levels organize the stripes and parity data differently. RAID 5 uses a distributed parity block with a disk stripe, while RAID 6 employs two parity blocks distributed across all disks in the stripe. In terms of ZFS, *Stripe* is equivalent to RAID 0, whereas *Mirror* corresponds to RAID 1 and *RAID-Z* and *RAIDZ-2* are equivalent to the standard RAID 5 and RAID 6, respectively.

7.7.2.2. RAID Rebuild in ZFS

In ZFS the RAID rebuild process is called resilvering. This comes from the word used for the actual repairing of physical mirrors. Resilvering occurs when a disk needs to be replaced due to either failure or data corruption. If a disk fails unexpectedly, the ZFS rebuild algorithm reassembles data on a new disk using either mirrored or parity data. The resilvering process can take an extended period of time, depending on the size of the drive and the amount of data that needs to be recovered. During resilvering, the performance of a RAID array is degraded because 1) the system resources are usually prioritized for data recovery, and 2) the rest of the disks in the group cannot provide optimal data redundancy for the configured RAID level. The resilvering process causes extra wear and accelerates the failure of the remaining healthy disks. Two or more disk failures can occur when I/O workloads stress individual disks too much. Therefore, the total number of disk failures might exceed the maximum fault tolerance for the RAID level, which causes resilvering to fail. The RAID array then becomes unavailable once such nested failures occur. If no additional backup exists, data on that array will become lost.

ZFS provides two different redundancy strategies, i.e., data mirroring and parity. Data mirroring used in *mirror* is a relatively simply method for rebuilding a RAID group. Since data is simultaneously stored on each component disk in the array, the content of any disk is identical to the rest. In the case of a disk failure, ZFS copies blocks of data from the remaining healthy disks to a spare disk. Although this type of redundancy provides fast RAID rebuild, it suffers from a low space utilization. The cost of scaling such an array increases linearly as a full drive is needed for every copy of the data. On the other hand, parity is more computationally expensive, but has a lower disk cost since it does not need a full set of duplicated disks to operate. *Raidz* uses simple logical operations, such as *exclusive or* (XOR), across the stripe to compute parity P

$$P = \bigoplus_i D_i = D_0 \oplus D_1 \oplus D_2 \oplus \cdots \oplus D_{n-1} \tag{7.8}$$

where D_i is a data block within a stripe.

If one disk fails, ZFS performs XOR operations on the data and parity blocks from the remaining healthy stripes to recover the original data. However, XOR operations do not specify the exact location of each data block in the stripe. If two disks fail in a *raidz2* array, applying XOR twice would be useless, since ZFS cannot determine which data block belongs to which failed disk. To handle this problem, *raidz2* utilizes the Reed-Solomon coding method and *Galois field* $GF(m)$ to generate the second parity Q and embeds the ownership information in the block.

$$Q = \bigoplus_i g^i D_i = g^0 D_0 \oplus g^1 D_1 \oplus \cdots \oplus g^{n-1} D_{n-1} \tag{7.9}$$

Calculating parities and data recovery (i.e., resilvering) based on *Galois field* $GF(m)$ are more compute-intensive than performing XOR in *raidz*. On the other hand, *raidz2* achieves a better redundancy.

7.7.3. Performance Modeling of I/O and Resilvering in ZFS

ZFS has been widely used in production storage systems, due to its support for high storage capacity, efficient data compression, integrated volume management and reliability management features. Data resilvering in ZFS is reactive, that is, data and parity blocks are read, regenerated, and stored after failures are detected. Although there are a few works that evaluate the performance of ZFS [99] [166] [150], they focus on certain features such as data compression or the read/write speed as a file system. Little work has been conducted to understand the performance of the new fault management techniques in ZFS. In this section, we evaluate the performance of ZFS' software RAID and the cost of the resilvering process. These results help us obtain a deeper understanding of ZFS' fault management mechanisms influencing the design of our proactive data protection scheme.

7.7.3.1. Test Platform Configuration

Table 7.15 shows the parameters and their values used in our experiments. The servers in the test platform were equipped with eight Intel Xeon cores (3 GHz), 32 GB DRAM, Ubuntu 16.04 LTS, and ZFS version 0.6.5.11. The disk drives model we used in the tests are Seagate BarraCuda ST2000DMs (magnetic HDD) and Intel DC3520s (data-center class SSD). We use the Bonnie++ benchmark suite to test ZFS' I/O performance. Specifically, we use the three subtests 'sequential output', 'sequential input', and 'rewrite' to evaluate the file system functions *write*, *read*, and *modify*. As ZFS is memory intensive, we vary the DRAM size allocated to ZFS to characterize its performance. To evaluate the resilvering performance of ZFS, we create a disk logical error to activate the resilvering process, then we measure the turnaround time of resilvering and the amount of data that ZFS recovers. The disk logical failure is created by injecting an error into ZFS' reserved area on a target disk to disable the communication channel between ZFS and the target disk.

7.7.3.2. ZFS Performance Characterization

We run each benchmark five times and compute the average of results. Each of the five runs is within 5% of the average. Thus we believe the experimental results are relatively

TABLE 7.15. Test Platform Configuration and Experiment Setting

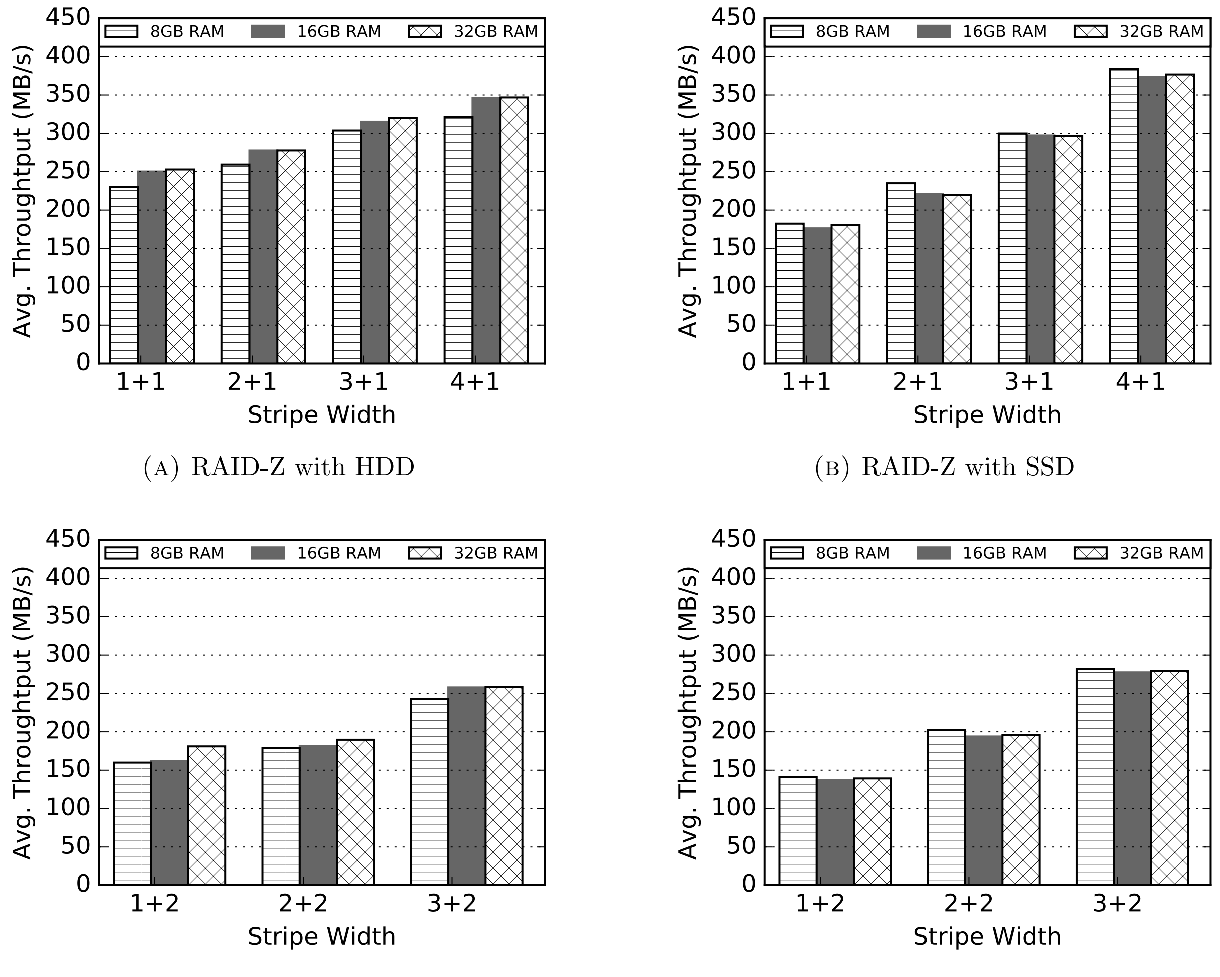
System Configuration	**Setting**
Disk Media	HDD (magnetic spinning), SSD (solid-state flash)
RAID (ZFS) Level	0 (*stripe*), 1 (*mirror*), 5 (*raidz*), 6 (*raidz2*)
RAID stripe size	2 - 6
DRAM Size for ZFS	8, 16, 32 GB
Storage Utilization	0% - 90%

stable. We present the average values of the results in the following figures for a clear presentation and interpretation. We expect to address the following important questions in our experiments.

7.7.3.3. How Does Strip Width Affect ZFS' Performance?

As discussed in Section 7.7.2.1, the striping process distributes file chunks across multiple disks, which means a file access request is served by all disks within the stripe. Our experimental results as presented in Figure 7.36 show that for the software RAID in ZFS, **increased stripe width improves I/O throughput and the improvement is super-linear**. The x-axis in the figure is the stripe width of RAID array. *4+1* denotes that a stripe consists of four data blocks and one parity block. When the stripe width is increased from 2 to 5 in a *raidz* array, or from 3 to 5 in a *raidz2* array, the overall I/O throughput increases linearly. Note that a RAID array using SSDs does not always outperform an HDD-based RAID array. For sequential I/O, SSDs arrays only outperforms the HDD-based array in the largest stripe size (i.e., *4+1* and *3+2* on *raidz* and *raidz2*, respectively).

Unlike I/O workload which can be serviced in parallel, resilvering is more computation intensive that reconstructs data sequentially. Although employing more disks in the rebuild process can improve the aggregated I/O throughput, the wider stripe increases the amount of data and parity used during the resilvering calculation which offsets the throughput gain. Fig. 7.37a shows the **recovery speed does not monotonically decrease as the stripe expands**. This finding inspires us to explore the search space of different hardware and environment settings to achieve the best recovery performance.



(A) RAID-Z with HDD

(B) RAID-Z with SSD

(C) RAIDZ-2 with HDD

(D) RAIDZ-2 with SSD

FIGURE 7.36. I/O throughput under different RAID level, Stripe width, and DRAM size.

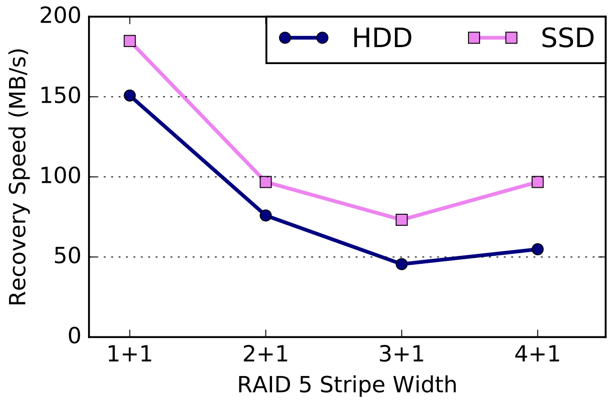
7.7.3.4. How Does the Size of DRAM Affect ZFS' Performance?

ZFS requires a large amount of DRAM for file caching and metadata management. A common practice for system configuration is to provide 5 GB of DRAM for each terabyte of storage. As shown in Figure 7.36, we test three DRAM sizes: 8, 16, and 32 GB for each stripe width and RAID level. From the figure we can see that larger DRAM size leads to about 10% of improvement of sequential I/O performance in HDD-based RAID array. But SSD-based array does not show significant benefit from using more DRAM. However, we cannot simply assume DRAM size is not important for ZFS, since our synthetic benchmark only consider sequential I/O performance. Real-world workload consist of both random and sequential I/Os. The random data access pattern in ZFS will gain huge benefit from large DRAM size.

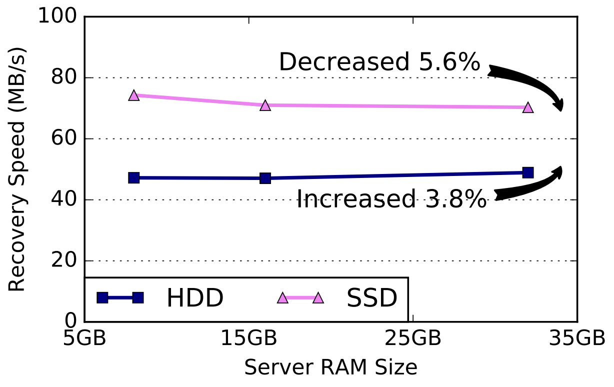
The results from the RAID recovery experiments show that **adding more DRAM does not always improve ZFS' recovery performance**. Figure 7.37b shows that using more DRAM for an HDD-based array improves the recovery speed by 3.8% on average, while the performance on an SSD-based array drops by 5.6%. In order to explain this phenomenon, we decompose the resilvering process of ZFS into three steps: 1) reading data and parity blocks, 2) computing the data that is lost, and 3) writing the recovered data to a spare. The performance of the second stage (data recovery computation) significantly influences the overall resilvering speed. We can compare the resilvering process with the *rewrite* operations that Bonnie++ performs, which also consists of reading, modifying, and writing blocks of data. From Figure 7.37c, we can see that the **recovery is a time-consuming process and the CPU performance is a dominant factor**. Although larger DRAM accommodates more metadata and file caching for regular I/O operations, the resilvering process rarely uses cached files. Hence, the DRAM size does not significantly affect the recovery performance of ZFS.

7.7.3.5. Does the Storage Utilization Affect ZFS' Performance?

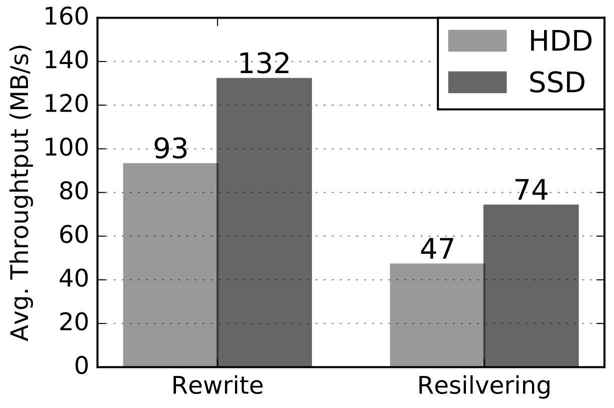
ZFS employs *copy-on-write* (COW), that is a new copy is created only when the data is modified. It efficiently shares duplicate data to avoid unnecessary resource consumption. However, COW causes more fragmentations as the utilization of the storage increases. Moreover, when the storage usage increases to approximately 80%, ZFS switches to a space-conserving (rather than speed-oriented) mode to preserve working space on the volume. In our experiment, we profile ZFS and measure its throughput as we ramp up zpool's utilization from 0% to 90%. Figure 7.38a presents the experimental results. The solid curve shows the average throughput of ZFS under the RAIDZ configuration, and the dashed curve is the corresponding trendline using a linear regression. From the figure, we can see that **as the utilization of zpool increases, the throughput of ZFS decreases significantly in the RAID array using HDDs**. As a zpool becomes fully used, ZFS' throughput drops



(A) Recovery performance is not monotonically decreasing when stripe width are increasing.



(B) With the RAM size increases, rebuild performance only shows negligible differences.



(C) Rewrite performance compare with recovery performance.

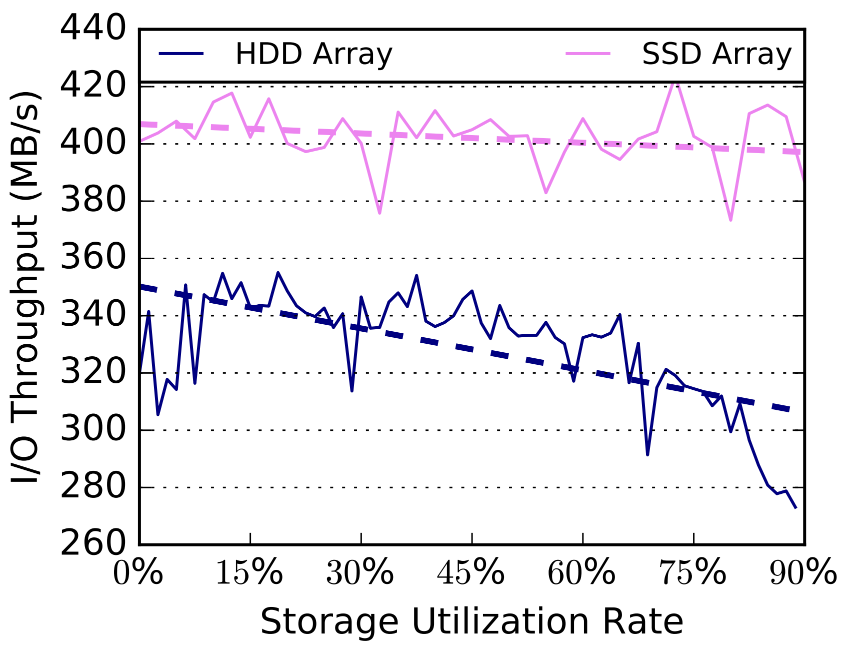
FIGURE 7.37. The I/O and recovery performance of ZFS.

by up to 25%.

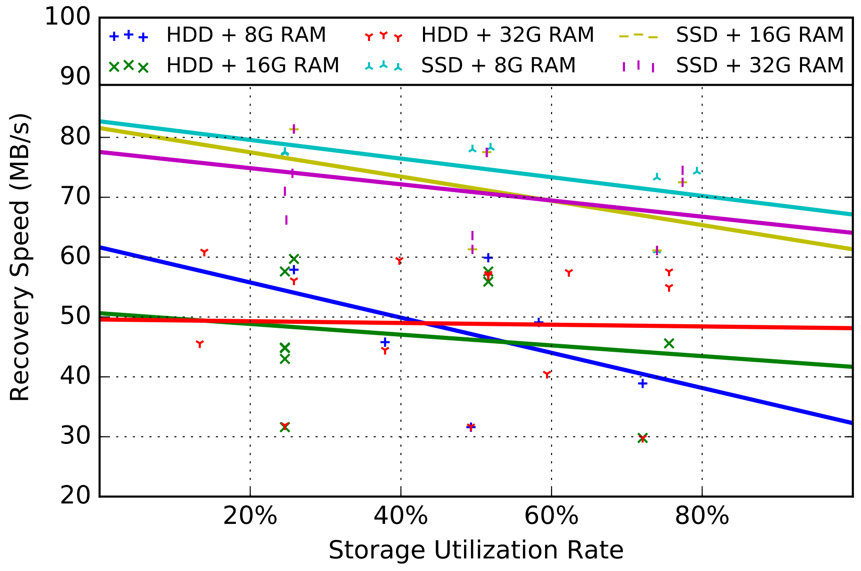
Similar to the preceding results, **the recovery throughput degrades as the zpool utilization increases**. Figure 7.38b shows the recovery throughput under different zpool utilization for both HDDs and SSDs based array. The least-squares linear regression model fits the results the best. We observe that the recovery performance decreases as the zpool utilization increases. When the storage array is close to a full utilization, the recovery speed is about 37% slower than the average throughput, and 47% slower than the peak speed. The first and last steps of the resilvering process consist of reads and writes. Therefore the performance degradation due to increased zpool utilization plays a major role for the reduced recovery speed.

7.7.4. Proactive Data Protection

Disk access speed has been outpaced by the increasing capacity. The stripe width of RAID arrays have grown to fill the gap between disk speed and capacity. Unfortunately, the probability of having double and even triple failures also increases as the disk recovery time is significantly prolonged. Although ZFS supports RAID 6 and RAID 7 (triple parity) to handle disk failures, the longer performance degradation and even unavailability of disk arrays compromise the overall system performance and users' satisfaction. Complementary to post-failure disk rebuilds, data can be rescued proactively prior to disk failures. This is enabled by disk failure prediction techniques [83] [111] [22] which forecast when failures will happen on which drives with promising accuracy. We aim to incorporate disk failure prediction methods in ZFS so that ZFS becomes capable of replacing a failing drive before the failure actually happens. Data on the failing drive (in contrast to failed drive) can be moved to a spare drive without the expensive disk rebuild process, thereby avoiding service disruption of the storage system. In addition, proactive data protection can be scheduled to perform during off-peak hours, which can further improve storage availability and achieve service-level objectives (SLO). Disk failure prediction remains an active research topic. In this work, we leverage the existing techniques of prediction disk failures and explore them in ZFS to develop proactive data protection. Research on the prediction methods is not the



(A) Throughput vs. utilization rate.



(B) Recovery Speed vs. utilization rate.

FIGURE 7.38. Storage utilization influences on ZFS' performance.

focus of this paper. Table 7.16 lists the variables that we use in the following discussion and analysis.

Table 7.16. Variables Used in the Analysis of Proactive Data Protection and Strategy Selection.

Variables	**Description**
t_l	Lead time of a failure prediction
t_i	Time when a failure prediction is performed
t_f	Time when a disk failure is predicted to happen
t_v	Validation period of a prediction
T	Duration of the data rescue process
S	Data rescue speed
A	Amount of data to be rescued
W	Wasted time due to failure misprediction
p	Precision of failure prediction

7.7.4.1. Proactive Strategies for Handling Disk Failures

A disk failure prediction model (FPM) uses monitored, real-time status data of disk drives to compute the probability at which the drives will fail in the future. At time t_i, if FPM predicts that a disk is going to fail, it reports the predicted failure occurrence which will happen at time t_f. We use lead time t_l, i.e., the length of time between the point when FPM makes prediction and the predicted failure occurrence time, to represent the *urgency* of a failure. We can calculate the lead time using $t_l = t_f - t_i$. Proactive data rescue from the failing disk to a spare or available disk also takes time. We use T to denote the time that a proactive action uses to rescue data. If $t_l > T$, then proactive data rescue can ensure the safety of the data on the failing drive(s). In this paper, we discuss several proactive strategies to handle disk failures. To proactively rescue data on a failing disk, we need to calculate the estimated recovery time T for each proactive strategy and compare it with the lead time t_l.

(1) **P**roactive **Dis**k **Clo**ning (P-DISCO)

(2) **P**roactive **A**ctive-**da**ta **R**ecovery (P-ADAR)

(3) **P**roactive Active-**da**ta **Clo**ning (P-DACO)

The *Proactive Disk Cloning*, or P-DISCO, migrates data on a predicted, failing drive to a hot spare using disk cloning. In the conventional RAID rebuilds, data reconstruction involves massive data movement and intensive parity computation using data and parity blocks from all of the other disks in an array. CPU and interconnect become the performance bottleneck as they determine the recovery throughput. Disk cloning, on the other hand, only migrates the data from the failing disk to a spare one without involving other disks or calculating parities. Theoretically, proactive disk cloning can achieve a much higher throughput. The additional workload might accelerate the death of the already failing drives, but other disk drives in the same RAID array stays intact. Moreover, the recovery speed is independent of the disk space utilization. In contrast, the performance of ZFS' resilvering process is affected by disk utilization as shown in Section 3.2. Fig. 7.39a compares the two recovery strategies, i.e., ZFS' resilvering and proactive disk cloning, for a fully used drive. From this figure, we can see P-DISCO completes data rescue in 17.59 hours for an 8TB HDD, and in 4.17 hours for a 3.2TB SSD. Most importantly, data rescue happens before a disk failure happens, thereby preventing a storage system from operating in a degraded state and nested disk failures. In contrast, after the disk fails, the default ZFS data recovery process takes over two times longer period of time to rebuild the same disk. During this period, the data reconstruction process competes with regular I/O workloads for system resources, which prolongs the recovery time and the vulnerable period spent in a degraded state. The volume manager masks data locations to make the recovery process transparent to the system. Without knowing which disk sectors contain useful data, disk cloning has to copy everything from every sector to the spare drive. For a nearly empty disk, P-DISCO may not be time and resource efficient.

$$T_{P-DISCO} = \frac{A_{disk\ capacity}}{S_{cloning}} \tag{7.10}$$

ZFS combines a file system and volume manager. It keeps tracking "active data" and provides the opportunity to recover only those tagged data areas when a disk fails. This results in an efficient data rescue strategy. *Proactive Active-data Recovery*, or P-ADAR, enables ZFS to proactively rescue only active-data instead of the entire disk to the spares. Although resilvering is computationally intensive, recovering the minimum amount of necessary data can save time. The amount of active data on a disk is no more than the disk's overall capacity, that is $A_{active\ data} < A_{disk\ capacity}$. Additionally, ZFS' resilvering is safe to interrupt. If power outage or reboot occurs during data rescue, the resilvering process resumes at the exact location where it is interrupted without human intervention. However, the resilvering process is more complex than cloning, which involves a full disk scan to gather active data locations, followed by compute-intensive data reconstruction. Therefore, $S_{resilvering} < S_{cloning}$. The actual recovery time T equals

$$T_{P-ADAR} = \frac{A_{active\ data}}{S_{resilvering}} \tag{7.11}$$

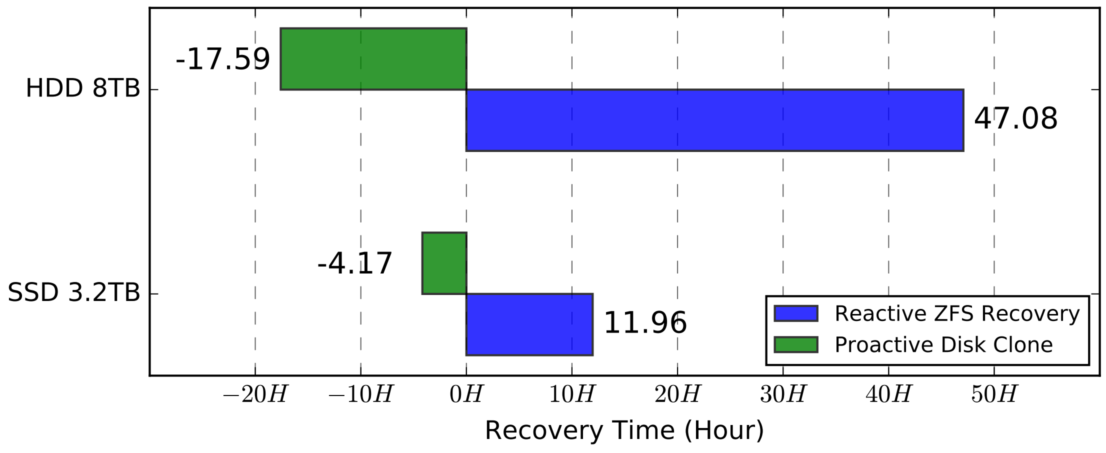
To further speed data recovery, we leverage computation-light cloning to rescue the necessary data tagged by ZFS. The proposed *Proactive Active-data Cloning*, or P-DACO, only moves the active data during recovery, and avoids parity computation. It combines the advantages of the preceding two strategies. Thus the recovery time equals to

$$T_{P-DACO} = \frac{A_{active\ data}}{S_{cloning}} \tag{7.12}$$

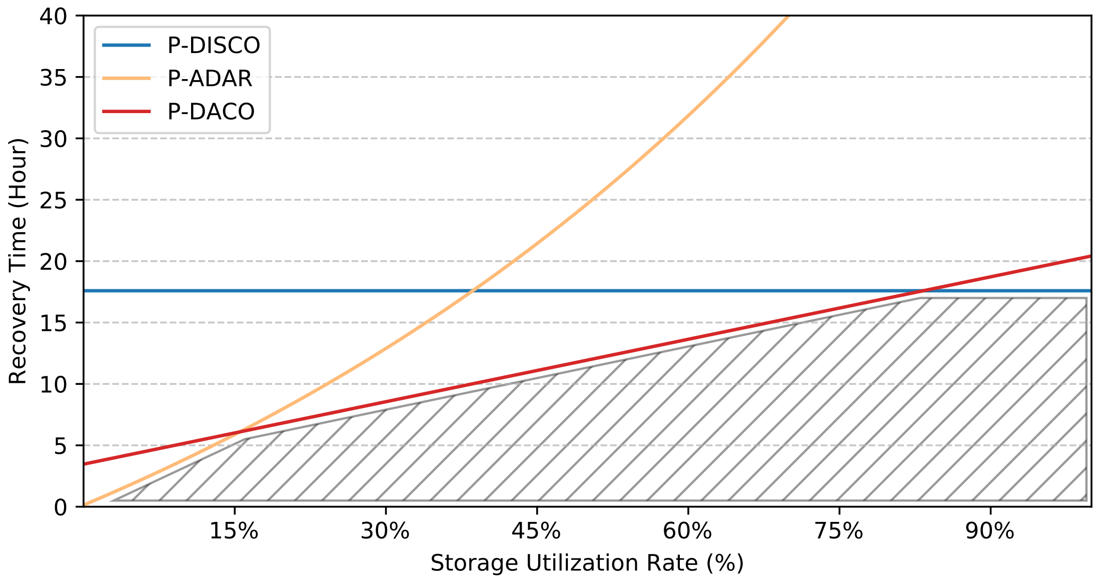
To obtain the value of S and A, a set of daemons frequently check disk's SMART (Self-Monitoring Analysis and Reporting Technology) data and zpools' utilization. A daemon process also runs a set of micro-benchmarks to determine the I/O performance and data recovery speed S at runtime. The amount of data to be rescued A is determined when the proactive action starts to rescue data. Meanwhile, the SMART data of each disk in the system is transferred to FPM for failure prediction and disk health analysis.

7.7.4.2. Selecting the Best Data Rescue Strategy

From the preceding discussion, we can see each proactive data resue strategy is only suitable for some specific situations. How to select the appropriate strategy to handle disk



(A) Time used for proactive disk cloning and post-failure disk recovery for a fully-utilized zpool. ***0H*** denote the disk failure occurrence.



(B) Time used by each proactive actions for data rescue.

FIGURE 7.39. Effects of storage utilization on ZFS' performance.

failures is critical. Figure 7.39b illustrates the performance of each proactive data rescue strategy to recover data from an 8TB enterprise-grade HDD with different disk utilization. From the figure, we find that P-ADAR is more efficient when the zpool is lightly used (i.e., below 15%). As more active data is stored, P-DISCO becomes a better choice. P-DACO is a promising strategy that yields better results than P-ADAR and P-DISCO most of the time. However, P-DACO cannot tolerate interrupts during data rescue, which needs trade-off between dependability and performance.

To select the best data rescue strategy systematically and accurately to handle disk failures, we design two constraints for the selection process: *urgency* and *dependability*. Specifically, the urgency indicates how fast a data rescue strategy can complete data recovery, while the dependability measures to what extent the data rescue strategy is tolerant to interruptions. For example, to minimize storage downtime caused by disk failures, we can prioritize *urgency* over *dependability*. In this case, if failure predictions provide enough lead time, where $t_l > min(T_{P-DISCO}, T_{P-ADAR}, T_{P-DACO})$, then we can select the strategy with shortest data rescue time T. For a storage system that prioritizes *dependability* over *urgency*, we adopt a simpler approach to narrow down the strategy selection set, since *active-data resilvering* is the only one that is safe to interruptions. Therefore, if FPM provides enough lead time for P-ADAR, we choose *active-data resilvering* as the proactive data rescue strategy to handle disk failures. Otherwise, the default reactive data rescue, or R-ADAR, is employed.

7.7.4.3. Analysis of Data Rescue Cost

In practice, each FPM only performs well for certain model and type of disk drives. For drives of a different model or type, it may generate many false alarms that incur unnecessary data rescues, and false negatives which need the expensive disk rebuild. In this section, we analyze the effectiveness and cost of proactive data protection strategies.

When FPM predicts a failure to occur at time t_f, it also provides a valid period t_v, which indicates the failure occurrence is likely within $t_f \pm \frac{1}{2}t_v$. We can calculate the adjusted lead time as $t_l' = t_f - \frac{1}{2}t_v - t_i$. If the disk is still healthy beyond $t_f + \frac{1}{2}t_v$, we say the prediction is a false positive. For proactive data protection, the cost of a false positive is a replacement of a healthy disk, in addition to the wasted resources used during data rescue. If the failure actually happens before $t_f - \frac{1}{2}t_v$, or FPM does not report that failure, we say the prediction is a false negative. In our discussion, we assume the worst case of false negative is that there is no proactive action to rescue data. Since proactive data protection should be complemented by reactive data rescue (R-ADAR) to address mis-predictions, the cost of false negatives becomes the cost from performing R-ADAR. The default R-ADAR and proactive strategy P-ADAR take the same time to do data rescue ($T_{P-ADAR} = T_{R-ADAR}$). The difference is their start time. At time t_i, when FPM predicts that a failure will occur by time t_f, P-ADAR starts data rescue and complete it by $t'_l - T_{P-ADAR}$. This is equivalent to $t_f - \frac{1}{2}t_v - T_{P-ADAR}$. For R-ADAR, the rescue process starts only after the actual disk failure occurrence at t'_f, and completes by $t'_f + T_{R-ADAR}$. Assume the probability that $t_f = t'_f$ is p. Once FPM provides enough lead time for proactive data rescue, i.e., $t'_l > T$, we say that the proactive data rescue strategy can be completed before the reactive action starts. With a probability of $(1-p)$ that FPM makes a wrong prediction, the cost is no higher than that of the default reactive strategy. Therefore, proactive data rescue improves the overall storage reliability by reducing the risk of data loss risk and the cost of data recovery.

We note that proactive data rescue strategies are not to replace the conventional resilvering process in ZFS. They are complementary to resilvering, aiming to enhance the storage reliability further. When failure predictions are sufficiently accurate, that is the majority of disk failures can be handled by proactive data rescue strategies prior to failure occurrences, the data recovery time is significantly shortened and the storage availability is improved. The probability of nested/multiple disk failures can also be reduced, which mitigates the risk of data loss.

7.7.5. Conclusions

Storage reliability imposes a major challenge to big data systems and applications. In this paper, we characterized storage performance under disk failures with a variety of ZFS configurations. We propose a proactive data protection scheme that leverages promising disk failure prediction techniques and rescues data prior to disk failure occurs. We explore the findings from our experiments to design an analytic model that aims to find the optimal data rescue strategy. Our analytic model uses zpool utilization and the configuration of a storage system to select the best strategy that minimizes the data rescue cost and maximize storage availability.

7.8. An Empirical Study of Quad-Level Cell (QLC) NAND Flash SSDs for Big Data Applications [135][7]

As the SSD technology develops, quad-level cell (QLC) NAND based SSD is gradually being introduced to the market. And as such, we evaluate the QLC technology's impact on the landscape of modern data centers. Since a large number of applications and workloads in the modern data centers have far more read requests than they write requests, QLC SSD provides a promising solution. For example, real-time analytics and big data, machine and deep learning, and read-intensive AI applications are all read hungry perfectly suited for the QLC. Its favorable performance (especially in reads), high capacity and density greatly help modern data centers to provide more efficient services to their customers. At the same time, the low cost of QLC SSD also helps to lower the cost of operation for data centers. Additionally, we explore the state-of-art QLC SSD from the system architecture point of view to shows its key advancements from previous technologies. By conducting a comprehensive performance evaluation of QLC SSD, we are able to compare it with other types of SSD and analyze factors that impact its performance.

7.8.1. Introduction

The booming onset of Big Data analysis and data-driven Artificial Intelligence tasks requires fast data access and data storage. Faster storage facilitates the data-intensive tasks to keep up with the ever-growing data sets. Solid state drives (SSD), or flash storage, are now the mainstay for mission-critical workloads such as business intelligence (BI) and content distribution. Since computers can only "learn" as quickly as it can read and analysis data, flash storage dramatically improve the machine learning efforts. But, as flash-based block device has limited per drive capacity and a much higher price-tag, they are commonly designated as a caching layer, which bridges memory and HDD-based permanent storage.

Ever since SSD were introduced to the enterprise market many years ago, its density

[7] Section 7.8 is reproduced in its entirety from Shuwen Liang, Zhi Qiao, Sihai Tang, Jacob Hochstetler, Song Fu, Weisong Shi, and Hsing-Bung Chen, An empirical study of quad-level cell (QLC) NAND flash SSDs for big data applications, 2019 IEEE International Conference on Big Data (Big Data), pp. 3676-3685, with permission from IEEE.

continues to improve thanks to the advances of semi-conductor technology. DRAM based and 3D XPoint based SSD offers higher performance, but NAND flash are still the most commonly used technology in SSD due to its lower cost. Depending on the number of bits stored in each flash cell, there are four basic types of NAND flash used in an SSD. Each type have its own distinct performance, cost, endurance, and density trade-offs. Single-level cell (SLC) requires 2 voltage levels (i.e., 0 and 1) to store 1 bit of data, offering highest write performance and endurance at the cost of price and density. Multilevel cell (MLC) requires 4 voltage levels to represent 2 bits of data (i.e., 00, 01, 10, and 11). Triple-level cell (TLC) and Quadruple-level cell (QLC) requires 8 and 16 levels of voltage to store 3 and 4 bits of data, respectively. As the data density increase, the price per bit lowers, but the endurance and write performance decreases as a result.

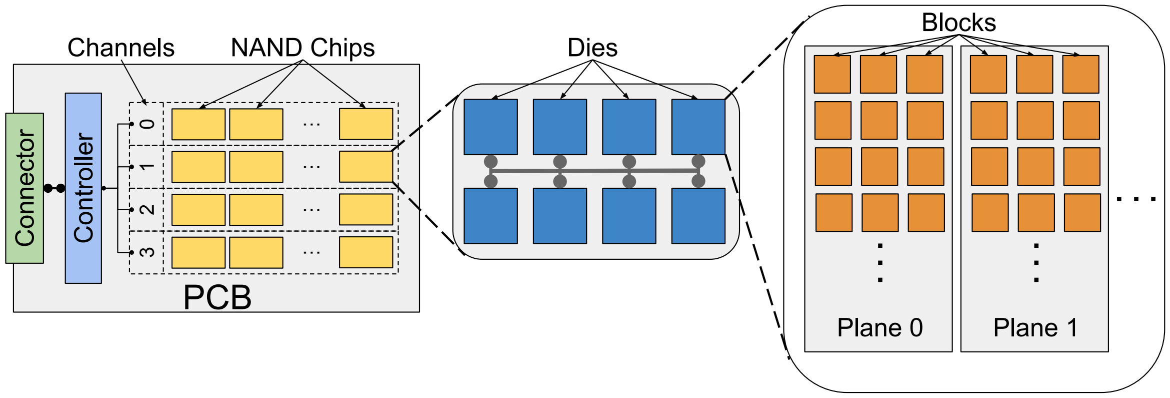


FIGURE 7.40. Example of an SSD Physical Layout.

Because endurance is such a vital component to the SSD, it is the primary concern to many data centers that are adopting SSDs. As SSD technology uses NAND flash, the inherent wear-out characteristics of the NAND flash directly effects the durability of the SSD. NAND flash chips are non-volatile, meaning they retain data without a constant power supply. Furthermore, because NAND flash based SSDs does not have moving parts involved during operation, they are more resistant to sudden shocks and extreme environments than their HDD counterparts. On the flip side, the NAND flash will eventually wear out as data writes accumulate overtime. If the amount of data written to the device exceeds its life span, NAND flash based SSD will gradually lose the ability to retain charge and the ability to retain data integrity. SSD employs several techniques to improve endurance, e.g., wear leveling, garbage collection, and chip-level RAID.

While the write endurance of NAND flash generally decreases as the number of bits stored per cell increases, a large number of applications and workloads in modern data centers have far more reads than writes. Typical read-intensive workloads such as streaming, content distribution, guided navigation, user profile access, and business intelligence/analysis are all capable of taking advantage of the cost-effective QLC SSD for its immense storage density compared with other types of SSD. In addition to the storage benefits of QLC SSD, the high performance of flash storage also supersede the legacy HDD based storage. For real-time analytics and big data applications such as Spark or Flink, QLC based storage uplift the performance of HDFS (Hadoop Distributed File System) and delivers massive data sets with high capacity storage that other types of SSD cannot provide.

QLC based storage allows deeper queries to build more detailed analytic and better insights for decision support systems. It also provides the low latency that machine/deep learning algorithms depend upon at an affordable price point. In addition, increased storage density consolidates the platform, reducing the total cost of ownership (TCO).

In this paper, we comprehensively evaluate the QLC technology's impact to the landscape of modern data centers. In Section 7.8.2 and Section 7.8.3, we study the latest QLC technology, which improves the SSD economics and fills the much-needed gap between MLC/TLC SSD and legacy HDD storage, from an architecture level to evince its key advancements over previous technologies. This storage paradigm transition enables more read-focused workloads to be migrated from dated HDD based storage to flash, thus releasing the expensive and limited MLC/TLC resources for write-centric applications. We evaluate two real-world QLC SSDs performance and compare them against state-of-art SSDs using MLC and TLC technology in Section 7.8.4. Our evaluation gives IT professionals and system designers valuable insight towards identifying the appropriate use cases of QLC SSD and how they can modernize data centers while reducing the TCO.

7.8.2. SSD Architecture

This section discusses the general architecture of SSD. From physical to logical, this section mainly focuses on the construction of the SSD and the main procedures of the data process in SSD.

Referencing Figure 7.40, an SSD is composed of, from left to right:

- A Connector interface.
- A Controller.
- NAND flash memory chips.
- Integrated Circuits (ICs).
- Resistors, inductors, and capacitors.
- Printed Circuit Board (PCB) substrate

7.8.2.1. Connector

The connector mediates data transfer between the SSD and the host computer. The universal connectors are SATA and NVMe. SATA has three major revisions. Most of the consumer-graded SSD supports SATA 3.0 or above. SATA 3.0 can support a burst throughput up to 6 Gbit/s [231]. In contrast, enterprise-graded and high-end SSD usually embeds the NVMe connector. It is a logical device interface to access NAND flash memory via a PCIe bus. Theoretically, the throughput of NVMe based SSD can be up to 32 Gbit/s [230].

7.8.2.2. Controller

An SSD controller works like a central manager that in-charge of all the NAND chips in the drives. The modern SSD controller is also a powerful "brain" as it is capable of managing different kinds of jobs; it executes firmware-level code, manages I/O requests, and ensures data integrity and storage efficiency. In particular, it manages bad blocks, enforce wearing leveling, monitors disk health, and handles garbage collection. Table 7.17 summarizes universal features that are supported by most SSD controllers. Each SSD manufacturer also has its own unique tweaks or features to the SSD controllers that boost the performance and reliability.

Table 7.17. SSD Controller Features

Feature[*]	**Description**
Flash Translation Layer (FTL)	Map logical address (LBA) to physical address in the flash memory [139].
Bad Block Mapping	Map the bad block's logical sector to reserved sector when bad block is detected [233].
Wear Leveling	The mechanism that maps re-writed/updated data to a new location and marks the previous location as "invalid". Meanwhile, cold data will also be moved around periodically to provide evenly wear among each cell [29].
Error Correction Code (ECC)	Detect and correct memory bit errors (soft errors). Hamming code and parity bit error detection schemes are widely used in SSD [233].
S.M.A.R.T.	Monitor and report SSD health status. Some SSD S.M.A.R.T. attributes are pre-defined, some are manufacturer defined [232].
Encryption/ Decryption	Controller support encryption/decryption to ensure data security; it typically uses the 256 AES encryption. Encryption can be applied to partial or whole drive [233].
Garbage Collection	The controller erases invalid data blocks periodically to ensure the available space for new write request. Garbage collection typically runs in the background during idle time [233].
I/O Caching	Store frequently used or recently used data to exploit spatial and temporal locality.
Data scrubbing	The mechanism that verifies the integrity of each memory block periodically. If a bit error is detected, controller will invoke ECC to correct in the same memory location. Data scrubbing is usually operated in disk idle time [233].
Power Management	To improve power efficiency for SSD, especially when used in mobile devices, the controller manages power consumption for the SSD in different states: active, idle, and slumber.
Trim Support	Enables the operating system to notify SSD on which data blocks can be erased [234].
Thermal Throttling	With the internal thermal sensors monitoring the environment temperature, SSD controller will reducing the I/O speed when overheating.

[*]Proprietary controller features not included.

7.8.2.3. NAND Flash Memory Chip

NAND chips are important components to the SSD. Thus far, an SSD can have 4-16 NAND chips [139]. An individual NAND chip can be decomposed from top to down in the following order (Also see Figure 7.40):

- **Die**: Each NAND chip can contain several NAND memory dies.
- **Plane**: Each die can contain 1-4 planes [139].
- **Block**: Each plane has thousands of flash blocks:
  - **Page/Wordline**[1]: Contains hundreds to thousands of rows of pages (horizontal).
  - **String/Bitline**[2]: Contains hundreds to thousands of columns of strings (vertical).
- **Cell**: Each page or string contains thousands of flash cells.

In 2D NAND, A flash block is the cluster of wordlines and bitlines. Figure 7.41 shows the details on the architecture of a block. A row is called a wordline, and a column is called a bitline. In the block level, transistors are neatly arranged.

The *Page* is the smallest data storage unit that can be read and wrote to, while the *Block* is the smallest data storage unit that can be erased. A page can typically contain 2K, 4K, 8K or 16KB data. And, the size of a block can vary between 256KB and 4MB [139]. But as technology develops rapidly, we can expect these numbers update quickly.

7.8.2.4. PCB, ICs and Other Components

All components, including controller, NAND chips, and connector, are on the printed circuit board (PCB), connecting with integrated circuits (ICs). Besides, there are sensors and counters tied to the PCB such as the thermal sensor and physical event counters. All components work together to make the SSD work appropriately as a whole.

7.8.2.5. How SSD work with Data

NAND flash SSD supports three possible data operations: write, read, and erase.

[1] "page" is preferred when referring to data, while "wordline" is used when referring to architecture.

[2] "string" is preferred when referring to data, while "bitline" is used when referring to architecture.

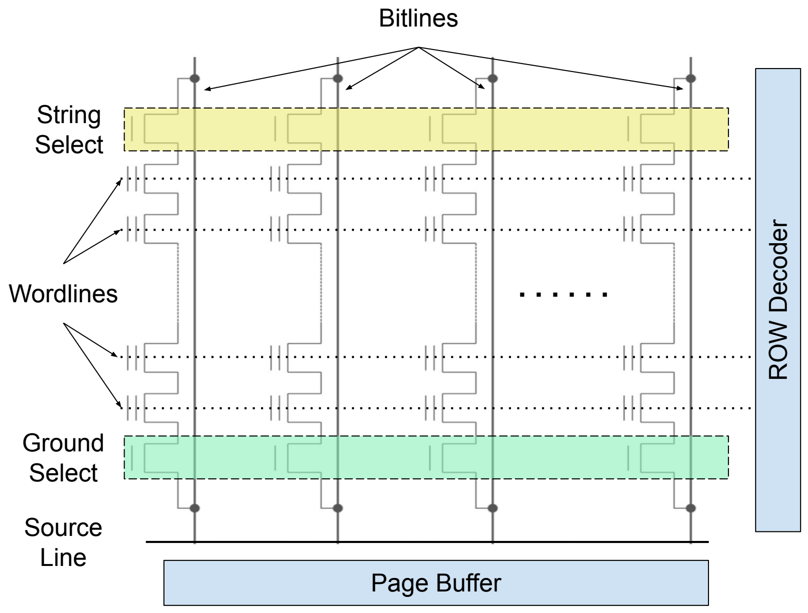


FIGURE 7.41. Example of an SSD Block Layout.

7.8.2.6. Write

Data from the file system goes through the SSD connector before arriving at the controller. Since data can only be written to empty blocks, the controller maintains a pool of empty blocks. If the drive runs out of empty blocks, the controller will perform garbage collection to reclaim "invalid" data blocks before writing data to the NAND memory. Otherwise, the FTL runs the address mapping algorithm and determines the physical addresses in NAND chips – the FTL maps the logical data blocks into the NAND page and then written into a block. Specifically, the controller applies a high positive voltage to responsive NAND pages and strings. Voltages of selected cells will be changed to logical "0". After related cell voltages are updated, the ECC will verify the written data before return the "success" signal to the OS.

7.8.2.7. Read

Reading data from SSD is similar to the writing process. The file system issues the read request. The request goes through the connector and enters the SSD controller. The controller processes the request and communicates with the NAND interface. FTL locates the physical addresses of the request data. Then, the controller applies the read voltage (intermediate positive voltage) to related NAND pages and strings. Since medium positive voltage won't change the logical representation of NAND cells, the selected NAND cells will respond with the corresponding stored logical 0 and 1. Raw data then goes through the NAND decoders. After decoding and verifying, data stream sends back to the file system.

7.8.2.8. Erase

Erasing data in SSD is quite different from erasing data on a HDD. In SSD, the erase process can only be performed in block-level while writing and reading process can perform at page-level. An erase operation is the process of removing electrons from the storage layer to change the state of the cell to logical 1. Typically, delete request sent from the file system won't immediately remove the data from flash memory; the controller only marks the "erased data" as "invalid." The garbage collection algorithm run in the background decides when to issue a large negative voltage to erase the whole block.

7.8.2.9. Data Placement

In general, SSD has two storage areas: main area and spare area. Main area stores user data and the spare area contains bad block marker, ECC and may have some metadata. Usually, the spare area is reserved and user cannot get access to it.

Data in SSD, including data placement, are specified and managed by the controller. SSD only write to one page each time and block marked as "bad block" will not be used. But, determining which page will be written and how to skip the bad block are defined by the related algorithms embedded in the SSD controller. The controller also defines other jobs related to data placement. For example, wear-leveling algorithms and I/O algorithms in the controller will divide large files and store each part in different flash chips. The data integrity algorithm writes data parities to a separate flash chip.

7.8.3. State of Art of Quad-Level SSD Architecture

The quad-level cell(QLC) technology was first introduced by NEC in 1996 [229]. Compared with prior generation, i.e., SLC, MLC, and TLC, QLC technology enables larger per bit capacity while lowering the cost. This technology was initially developed for the

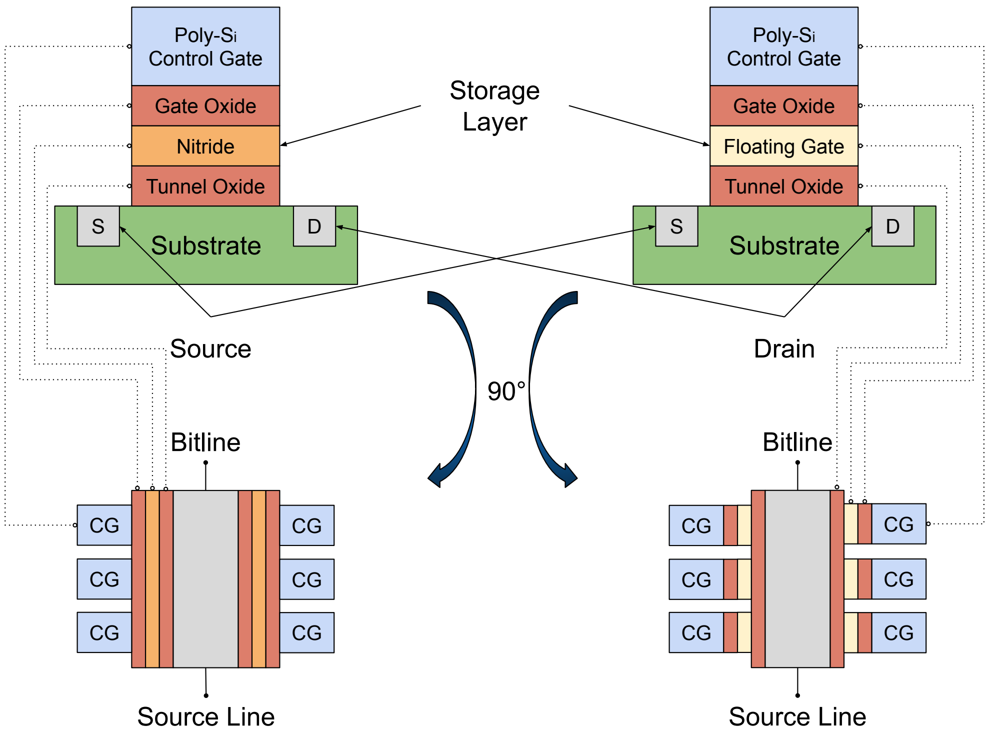


FIGURE 7.42. Charge Trap (CT) cell vs. Floating Gate (FG) cell.

dynamic random-access memory(DRAM) chip. In the 2000s, QLC technology applied to NOR flash memory cells and NAND flash chips. Later on in the winter of 2018, the first commercial QLC NAND-based SSD became available [229]. In this section, we present the key advancements of QLC technology that uplift the storage density, and bridge the gap of performance and cost between flash and legacy HDD storage. The main difference between QLC and other types of SSD lies at the cell level and block level. We highlight the main features of QLC technology in Table 7.18, then explore them in detail and compare them against competitive SSD technologies.

7.8.3.1. Cell Level Architecture

7.8.3.2. Physical Architecture

NAND-based SSD generally employs one of the two transistor technologies: Charge Trap (CT) MOSFET (Metal Oxide Semiconductor Field Effect Transistor) and Floating Gate (FG) MOSFET. Manufactures like Samsung, Toshiba, SanDisk, and Western Digital

TABLE 7.18. Specifications of SLC, MLC, TLC, and QLC

Types	**SLC**	**MLC**	**TLC**	**QLC**
P/E Cycle	90-1000k	8-30k	3-5k	500-1k
Bit per Cell	1	2	3	4
Reliability	★★★★★	★★★★	★★★	★★
Endurance	★★★★★	★★★★	★★★	★★
Cell Density	★	★★	★★★	★★★★
Power*	0.1-3.6W	0.6-2.6W	0.7-3.6W	1.5-3.6W
Voltage Levels	2	4	8	16
NAND Architecture	2D/3D	2D/3D	3D	3D
		Latency / QoS [237]		
Read	25μs	50μs	75μs	≈100μs
Write	200-300μs	600-900μs	900-1350μs	≈1500μs
Erase	1.5-2ms	3ms	5ms	≈6ms

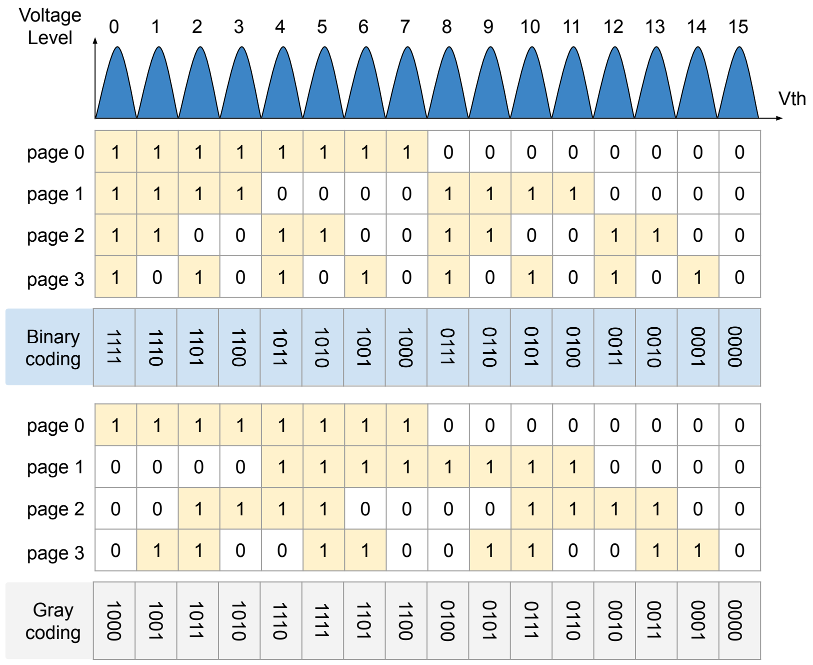
*Ranging from *idle* to *active* power consumption.

develop their SSD architecture using the CT MOSFET, while Micron and Intel adopt the FG MOSFET. Figure 7.42 illustrates the difference between CT cell and FG cell. The storage layer of CT cell uses the silicon nitride while the FG cell uses floating gate [199]. Additionally, the charged storage layer in CT is shared among all cells while in FG they are isolated. According to Micheloni's study [188], the majority of SSD relying on CT cells but FG cell also have its market share.

7.8.3.3. Data Programming

One QLC cell can store four bits of information, which is 33% more than TLC technology. As illustrated in Figure 7.43, it requires 2^4 different electric voltage levels to program a QLC cell (represented by 0000 - 1111). A logical QLC cell data (4 bits) are mapped to four pages. Thus, it takes four cycles to raise the voltage level to the desired state.

There are many QLC programming algorithms. Traditional method named Binary Code is illustrated in the upper portion of Figure 7.43. For a narrower and tighter voltage



Voltage Level	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
page 0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
page 1	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
page 2	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
page 3	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
Binary coding	1111	1110	1101	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001	0000
page 0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
page 1	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
page 2	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0
page 3	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
Gray coding	1000	1001	1011	1010	1110	1111	1101	1100	0100	0101	0111	0110	0010	0011	0001	0000

FIGURE 7.43. QLC Voltage Levels and Data Mapping in Binary Code vs. Gray Code.

range, traditional data mapping based on Binary Code becomes inefficient and can easy introduce data error [137]. Thus, more sophisticated programming algorithms have been proposed. Since QLC SSD is designed for read-intensive workloads, data reading based on Gray Code method provides a better solution (refer to lower portion of Figure 7.43). The most obvious benefit to this is that two successive values differs in only one bit [228]. Thus, if voltage shift and data retention error occurs, Binary Coding may encounter up to 4 bits data error (i.e., voltage level 7 to 8 in Figure 7.43), Gray coding only yields 1-bit data error. There are many Gray Code variants [137], but all of them retain the 1-bit differ rule for sibling values.

7.8.3.4. Block Level Architecture

Starting from the TLC technology, block-level design shifted from 2 to 3 dimensional. Each generation of 2D NAND architecture shrinks the size and increases the number of transistors in the chips to accommodate the Moore Law. However, 2D NAND design has reached the lithographic limitation [188]. The new 3D flash architecture extends the vertical space to achieve a higher density and capacity. Thus, QLC SSD continues the trend of using

3D NAND architecture in block level design.

Each manufactures has its proprietary 3D NAND designs. Toshiba developed a 3D NAND technology named Bit-Cost Scalable NAND technology (BiCS) then later improved to Pipe-shaped Bit Cost Scalable (P-BiCS). Samsung introduced several vertical gate (with either horizontal or vertical channels) designs, namely V-NAND architecture. The V-NAND family includes the Vertical Recess Array Transistor (VRAT), the Vertical Stacked Array Transistor (VSAT), and the Terabit Cell Array Transistor (TCAT) [188]. SK Hynix proposed designs based on FG cell, named Dual Control-gate with Surrounding Floating-gate (DC-SF) and its variant called Advanced DC-SF [225].

3D NAND design is an extension of 2D NAND that adds the vertical dimension. Figure 7.44 illustrates a typical way to convert a 2D string to a 3D string. Imagine when all the strings in a 2D block are folded over then stood vertically, essentially transitioning the 2D planar block into a 3D block. Samsung's TCAT and SK Hynix's DC-SF uses different approaches. In a nut shell, TCAT and DC-SF don't fold over the string, but add a "z" axle to the 2D planar instead. Figure 7.45 shows this type of 3D NAND block construction. Both CT and FG cells presented in Figure 7.42 can be applied to this type of 3D architecture.

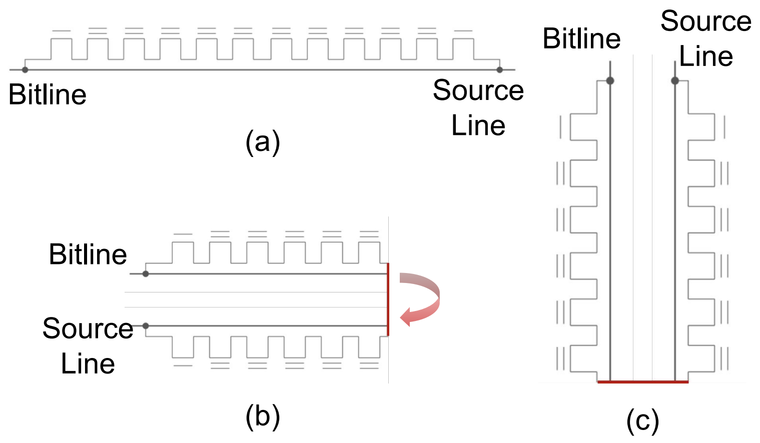


Figure 7.44. P-BiCS Converts a 2D NAND String to 3D NAND String: (a) The 2D NAND string (b) 2D string is stretched out in the middle and folded over (c) Make it stand vertical and it becomes a 3D string.

Transitioning from 2D to 3D block, we encounter a new concept called "layer." The number of layers defined by the number of vertical Control Gates (CG). In QLC, the number

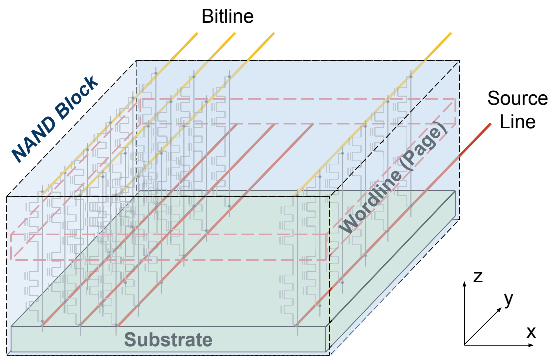


FIGURE 7.45. TCAT and DC-SF Construct a 3D NAND Block.

of layers are usually the multiple of 4. The common construction of 3D NAND flash uses 32, 36, 48, 56 and 96 layers. More layers usually means higher storage density. In the year of 2019, 96 layer 3D NAND dominates the QLC market share. Meanwhile, the 3D NAND flash of over 100 layers (e.g., 128 layers) is just around the corner.

7.8.4. Evaluating Performance and Value of Qlc Ssd for Modern Data Centers

Historically, performance-sensitive and read-centric workloads have relied on parallel arrays of HDDs to deliver the required capabilities that service-level agreement (SLA) demand. With the advance of QLC technology, can these new SSD achieve the storage performance and capacity requirement? In this section, we evaluate the real-world performance of the latest QLC SSDs and compare its performance with state-of-art SSDs using MLC and TLC technologies. We try to answer the question: *Can QLC SSD offers flash storage performance at more approachable price?* Table 7.19 highlights the main features of each SSD used in our experiment.

7.8.4.1. Experiment Setup

Our evaluation comprises of several factors that might impact SSD performance. All of our experiments are performed on two HP Proliant ML110 G6 Storage servers with identical configuration. Each server is equipped with an eight core Intel Xeon (3 GHz), 8 GB DRAM, and Ubuntu 18.04 LTS. All HDDs and SSDs are physically attached to the sever machine via SATA 3.0 connectors. We use the `fio` (aka., Flexible I/O) synthetic trace to simulate various types of workloads. During the experiment, `fio` was set to use

Table 7.19. Types of SSD in This Evaluation

Brand Name*	**Cell Type+**	**Architecture**	**Capacity (GB)**	**Cost per bit**
Brand A	eMLC	2D	240	$$$
Brand B	TLC	3D	480	$$
Brand C	QLC	3D	480	$
Brand D	eQLC	3D	1920	$

* For each brand, the logical sector size are 512 byte; physical sector size may vary.

+ The "e" in front of a cell type denotes the enterprise grade drive.

asynchronous engine for non-buffered I/O, and the I/O depth were set to 64 to saturate the bandwidth. The broad range of factors that might affect the performance of SSD in production environment includes read-write ratio, data access patterns, block size, garbage collecting operations, bad block managements and reserved block replacement policy, etc. The total workload size exceeds available memory to ensure a storage-centric workload. We repeat each test five times then report the average.

Note that new SSD needs to break-in before the experiments. Since brand new SSDs shipped with empty flash blocks, I/O latency measured at empty blocks will differ from non-empty blocks. The break-in process fills the new drive with nonzero data. I/O performance measured from non-empty block represents real-world results from production environment.

7.8.4.2. Performance Evaluation

In production storage systems, different applications exhibit distinct I/O patterns and characteristics. We can categorized them into two types: small reads/writes and large reads/writes. The former is typically measured by IOPS, while the latter is evaluated by throughput. In our preliminary testing, we adopted the widely used benchmark configuration and procedures to evaluate the performance of an SSD. We selected the following three metrics to quantitatively measure the performance. The objective of each metric is highlighted as follows.

(1) **Sequential Write/Read with 1MB block size.** This test measures I/O band-

width for large I/O requests. In this test, sequential write/read are performed in multiple parallel streams, using 1MB I/O size to simulate large data writes/reads.

(2) **Write/Read IOPS with 4KB block size.** This test measures the ability of a block device to handle small I/O requests. Following the industrial best-practice, we set I/O size to 4KB. Write/read are only performed in single stream, so the number of concurrent request is adjusted to a larger number to generate sufficient requests before they saturate the I/O bandwidth.

(3) **Write/Read Latency with 4KB block size.** This test evaluates the latency of a block device completing a I/O request. The write/read are performed in single stream and I/O array size is set to a small number, so the number of concurrent request is adjusted down to prevent reaching the maximum bandwidth or maximum IOPS.

TABLE 7.20. Benchmark Results

Metrics	**Brand A**	**Brand B**	**Brand C**	**Brand D**
		Write		
Throughput (MB/s)	247	250	191	231
IOPS	9663	10400	6853	8852
Latency (μs)	406.1	378.9	574.7	446.3
		Read		
Throughput (MB/s)	210	249	241	192
IOPS	4468	5337	2441	3338
Latency (μs)	896.9	763.4	1631.9	1195.1

Table 7.20 shows the preliminary results in these experiments. Overall, the write/read of both QLC drives performs worse than that of other drives. The brand C QLC drive has the lowest writing speed at 191 MB/s. Brand D QLC drive performs better at 231 MB/s. However, the write IOPS of both QLC drives can only achieve 66% - 90% of its MLC and TLC competitor, while the write latency are also 40μs-200μs higher than Brand A and Brand

B drives. Similarly, the sequential read IOPS of QLC drives are 25% - 55% lower than its competitors and the read latency almost doubles the MLC and TLC drives. We cannot simply conclude that all QLC drives performs worse than TLC or MLC drives, but from the result we can extrapolate that SSD performance will degrade when the data bits per cell are increased. This result is intuitive as the increase of bit per cell require advanced architecture design and complicated electron level controls. In addition, QLC only have around 500 to 1000 P/E cycles. To prolong its lifespan, some QLC SSDs throttle the write performance by design.

On the other hand, QLC SSD packs 33% more data per cell (4 bits rather than 3) and adopts more sophisticated algorithms to encode data. Hence, they might exhibit different I/O characteristic than industry best-practice for MLC and TLC drives. In the following sections, we explore a range of factors to optimize the storage configurations that yield better read performance for QLC SSDs.

7.8.4.3. Block Size Matters

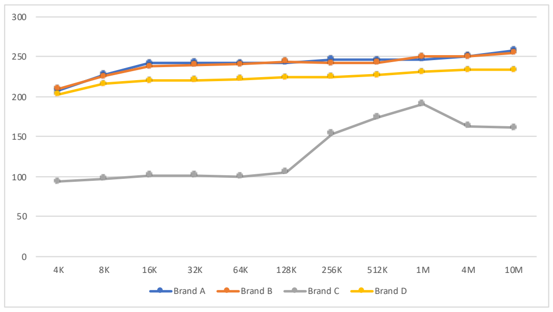
Our preliminary test uses 4KB data block size. However, this block size may artificially inflate the total I/O number that the drives are capable of handling [212], and the I/O patterns in real-world scenarios are also more complicated. We may encounter different data block sizes with a mix of reads and writes requests. To better understand the QLC I/O characteristics for read-centric workloads, we test various block sizes with the read/write ratio at 75/25. The read and the write operations are also randomly mixed to simulate real-world scenarios. Figure 7.46 - 7.48 shows the experiment results. We increase the block sizes from 4KB to 10MB, and tested both sequential and random I/O operations. From the results, we have the following observations.

- **I/O speed increases with the block size.** Overall, the performance of sequential and random write/read operations of QLC SSD increases with the block size. We observed a similar trend in SLC, MLC and TLC SSDs. Since the logical block size of each drive is 512KB, write requests can only be handled per block unit, we believe the writing performance degradation, when block size exceed 1 MB, is due to data buffering or write aggregation. *We conclude that for the write process, SSD logical block size positively impacts the I/O performance.*

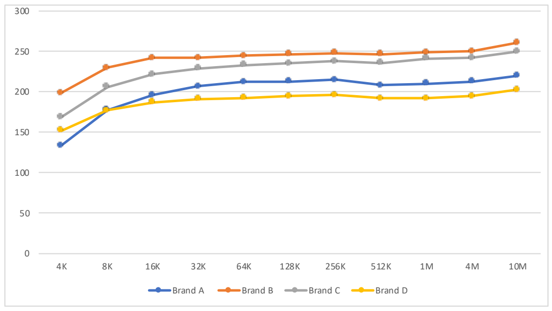
- **I/O speed increases faster when block sizes are smaller.** Its obvious that sequential I/O speed increases faster when the block size is less than 16KB. The random read performance increases rapidly until the block size reaches 1MB. However, for random write request, each drive have a different optimal block size. The experiment results also indicate that too many tiny files or massive files may hurt SSD performance. *The optimal block size should be in the range of 16KB to 1MB.*
- **Performance of enterprise-grade QLC SSD is more stable and predictable than consumer-grade QLC SSD.** In this experiment, we evaluated a consumer-grade QLC SSD (Brand C) and an enterprise-grade QLC SSD (Brand D). In both write and read tests, Brand D QLC SSD strictly follows the increasing trend as other MLC and TLC drives. But the I/O performances of Brand C SSD has more fluctuations, especially during the write tests. For the sequential write test, the throughput of Brand C SSD peaks at 1MB block size before it starts to degrade. For a random write test, the throughput of Brand C SSD rapidly grows at first but then degrades after the block size exceeds 64KB. We still need further study to fully explain the fluctuating performance of Brand C SSD, but *we believe the SSD controller design has a major impact on I/O performance.*

7.8.4.4. Garbage Collection Matters

In SSDs, the garbage collection (GC) process releases the blocks that were occupied by invalid data. Recall that GC is usually performed at background when the drive is idle, so it minimize the performance impact while ensures the available drive capacity. Such a strategy is typically useful for consumer environment as they tend to have more idle time. However, enterprise environment have a much more intensive storage usage, causing the GC procedure to lack having sufficient time to perform its task in the background. When GC is eventually forced to run in the foreground alone with the application I/O payload, it imposes a significant performance and endurance impact to the system, especially for the

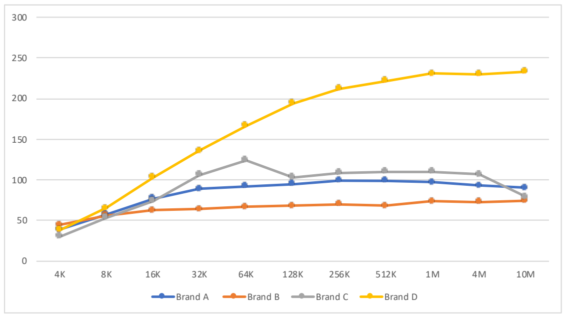


(A) Sequential Write

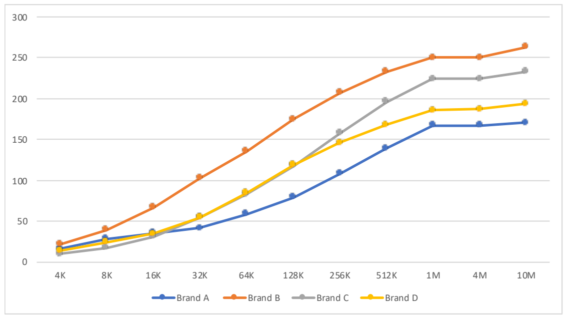


(B) Sequential Read

FIGURE 7.46. Sequential Write and Read: Throughput (MB) vs. Block Size.

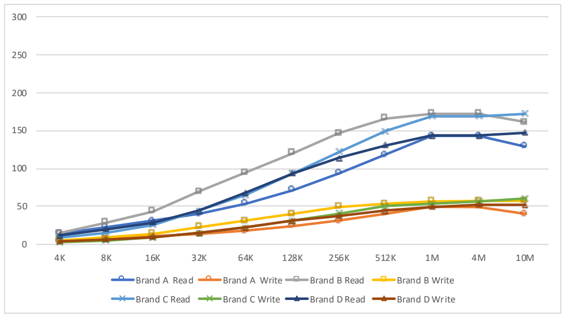


(A) Random Write

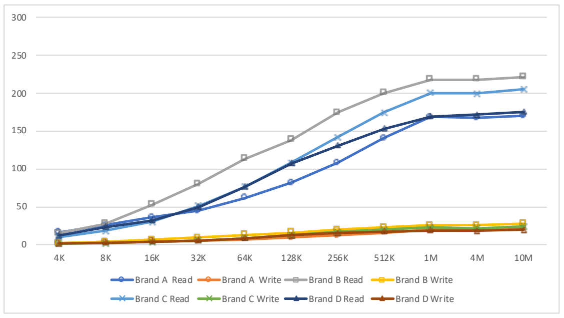


(B) Random Read

FIGURE 7.47. Random Write and Read: Throughput(MB) vs. Block Size.



(A) Read/Write = 75/25

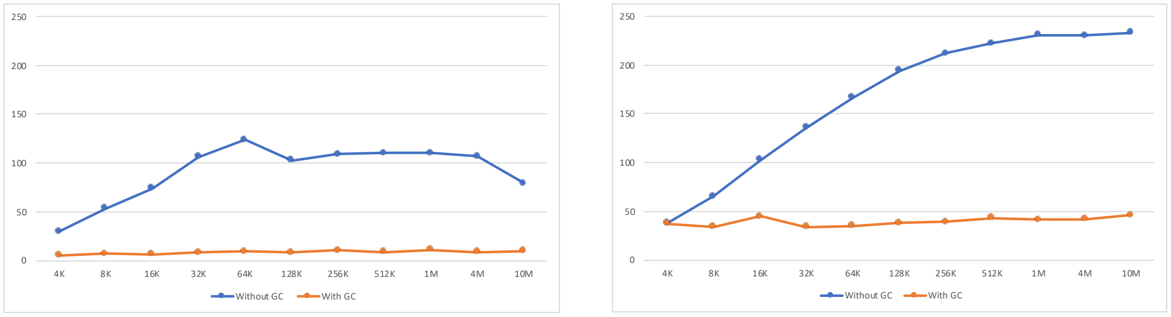


(B) Read/Write = 90/10

FIGURE 7.48. Read and Write in Mixed Ratios: Throughput (MB) vs. Block Size.

write performance.

GC activities may have distinct performance impact for different SSDs, as it is effected by the embedded GC algorithm, the wear-leveling algorithm, the SSD controller policy, the amount of SSD empty blocks, capacity, block size, and other factors. Theoretically, QLC



(A) Brand C QLC SSD (B) Brand D QLC SSD

FIGURE 7.49. Garbage Collection Effects on Random Write: Throughput (MB) vs. Block Size.

SSD needs to spend more efforts on GC than other types of SSD due to the complexity of NAND cells design. To measure the performance impact of GC for our QLC SSDs, we first fill up the SSD with random data then immediately issue burst I/O workloads. This will invoke the GC to release the invalid data blocks before handling new write request. Fig 7.49 shows the I/O performance difference between QLC SSD with GC and without GC in different I/O size. On average, Brand C SSD random write performance drops 90% when garbage collection onset, while Brand D SSD drops about 76%. Garbage collection activity not only impact write performance, but also affects read operations. Recent studies [205] found that read performance also degrades significantly when garbage collection is engaged; the read request will also be blocked until garbage collection process finish.

7.8.4.5. Bad Block and Reserved Block Matters

SSD is a masterpiece of complex industrial products that comprises of thousands of sub-components. So, besides the block sizes and garbage collection, SSD performance might also affected by the number of bad blocks and reserved blocks. A block is marked as a bad block when its P/E cycle reaches a preset threshold or it becomes inaccessible. When a block is marked as a bad block, the SSD controller will map its logical address to a spare block from the reserved block area. Data in the inaccessible bad block is considered as lost since read and write requests cannot be completed. However, if the SSD supports block level redundancy such as Erasure coding or internal RAID, then the SSD controller can initiate the recovery procedure that reconstruct the lost data to the spare blocks. The recovery procedure will impact I/O performance as it requires additional I/O resources. Moreover, as the number of bad blocks accumulates, the number of reserved blocks will be exhausted. As a result, the available capacity of the SSD eventually shrinks. QLC SSD is more likely to encounter this problem as it has much lower P/E cycles.

7.8.4.6. Environment Matters

Like other types of SSD, environment factors such as power surge, radiation and operating temperature also impact the QLC SSD performance.

- Power: QLC SSD is a NAND flash based drive that are non-volatile. It can retain data even without power supply. However, study [253][215] shows that sudden power outage (and the associated power surge) will flip the bits and cause data corruption. Even in data center environment where power supply is generally stable, long power-on hours might lead to electron discharge and cause NAND flash to lose its data retention ability. When any bit errors are detected, the SSD controller will engage built-in error correction code (ECC) mechanisms to resolve the silently corrupted data. As a result, I/O performance will also be significantly affected by the ECC activity.
- Radiations: Cosmos radiation can disrupt the NAND flash cell energy level and causing soft errors. It can also permanently damage the semiconductor, leading to malfunction. Radiation problems not only occurs to SSD drives used in space flights but also impact data centers at higher altitude.
- Temperature: Temperature has significant impact on the physical characteristics of NAND flash cell, hence indirectly affects the SSD's I/O performance. As the working temperature increasing, the oxide tunnel in NAND gates loses the ability to retain its charge level. Electrons will be able to escape from the tunnels much easier, which leads to bit flipping and soft errors. To tackle this problem, most SSD controllers implement thermal throttling mechanisms that artificially decrease the I/O resource quota when it detects temperature increase nearing the set threshold. This gives

ECC mechanisms more time to correct the increasing number of corrupted bits. However, thermal throttling significantly degrades the SSD performance. When storage systems on heavy workloads or storage rack are placed at areas that has bad air circulation or ventilation, thermal throttling might be triggered much more frequently and will greatly impact the overall performance of the storage system.

7.8.4.7. Economic Analysis

With the ever-increasing data load and read-centric requests, data centers are pressed to meet the increasing storage and service demands. When upgrading the storage infrastructure from the existing one or building a new one, IT professionals are constrained to the binary options of: performance-oriented 2.5-inch 10K RMP HDDs that offers higher performance but smaller capacity (e.g., 2.4TB), or 3.5-inch 7200 RPM HDDs that have higher capacity (e.g, 14TB) but lower read IOPS. Therefore, we can transitioning the standard twelve-bay 2U storage server into high-performance node that have 29TB raw capacity, or high-capacity node that have 168TB of raw capacity. QLC SSDs from major manufactures offers up to 7.6TB capacity (available in Brand D) per 2.5-inch drive that transitioning the same 2U server into 184TB raw capacity (i.e., 10% more than capacity tier HDDs). New QLC SSD offers a higher-density and higher-performance storage for the same data center footprints. In addition, QLC SSD has the following advantages that makes it beneficial to replace HDDs for read-intensive workloads (data listed in Table 7.21).

- **Power efficient**: the HDD power consumption is around 8-11 watts while QLC SSD is around 3 watts (3X less). However, the read IOPS per watts shows QLC SSD has 38X higher power efficiency for the real-world power consumption.
- **Reliable**: the maximum data bytes that passed through the HDD drive interface (both read and write) is typically 2.5 - 3.5 petabyte (PB) while the QLC SSD have estimated 450 TB of total write byte (TBW). Since QLC SSD is targeted at read-intensive workloads (e.g., 90%+ reads), it can endure up to 4.5PB data bytes that passed through the SSD interface within life-cycle, that is 28.5% higher than HDDs.

- **Cost-effective**: HDDs are still the most affordable storage solution in terms of the dollar per GB of storage (i.e., \$ / GB). But in order to support the quality of service (QoS) demand that are essential to data center operation, a large number of HDD arrays is required to achieve the desired read performance. Our previous study [175] indicates that a RAID-5 array comprises of five HDDs or a RAID-6 array comprises of seven HDDs provide similar read performance to a single QLC SSD. Therefore, QLC SSD and HDD ended up having similar investment per GB to reach same level of performance. Consider the cost per GB for management and maintenance such as cooling and rack space, QLC SSD becomes more cost-effective than HDD, which leads to a higher investment gain.

TABLE 7.21. Per Drive Cost vs. Performance

Characteristics	**HDD**	**eMLC**	**TLC**	**eQLC**
Random I/O MB/s	50	167	250	186
Read IOPS	189	4468	5337	2441
\$ / GB	0.02	0.67	0.2	0.12

7.8.5. Conclusions

In this paper, we study the QLC SSD performance, as well as its economic effects on the landscape of data centers. Our research indicates that QLC SSD is a promising contender when comparing against other types of SSDs and HDDs. While QLC has the worst write/read IOPS when compared to other SSDs, it does not detract from the fact that QLC SSD is more suitable for large data workloads compared with small data workloads. QLC SSD can also provide a favorable solution to data centers with its high capacity and density. In the cost-effective analysis, we concluded that one QLC SSD is equivalence in performance as a 5-7 HDDs RAID array at a similar cost. Also, QLC SSD has lower power consumption while retaining higher reliability than HDD.

CHAPTER 8

RELATED WORK

8.1. Cloud Computing

8.1.1. SSD Reliability

Many existing works study the reliability of raw flash chips. Their evaluations are performed in controlled lab environments with only a limited number of models and devices. In general, they use synthetic benchmarks to stress individual flash components, and identify error symptoms and sources. For example, [23][122][117][149][28][26] found that flash reliability is attributed to *read disturb error* and *program disturb error* which are caused by the tunneling effect where data in the untouched blocks are affected by read or program operations in the surrounding blocks. *Data retention error* is caused by detrap current that erratically changes the data at threshold voltage[241][242][13][131]. The error prediction and recovery methods are discussed in [148]. The reliability of flash cells deteriorate over a number of P/E cycles [33][201]. In [92], the cost, performance, capacity, and reliability trend of flash memory are studied. In the controlled environment, tests focusing on certain aspects of flash memory aim to eliminate unwanted effects. Results from these works provide a knowledge base on flash reliability, and are complementary to our work.

The aforementioned studies provide insights to chip-level flash reliability. It is also urgent to understand flash reliability in large-scale data centers under real-world workloads. Recent works from Facebook[147], Google[195], and Microsoft[153] study field datasets. Their studies discover important differences in the field compared with those in the controlled environments. SSDs in their studies are used as permanent storage devices. In contrast, SSDs in the data center that we study are used as caching devices, which resembles the burst buffer as used in HPC systems. In addition, our dataset includes six months of detailed SSD-specific SMART records, which are valuable for characterizing SSD reliability at the device level with rich semantic information.

8.1.2. Proactive Data Protection in ZFS

To improve storage reliability, some researches utilizes erasure coding based redundancy scheme for cost-effective error tolerant. Although we have seen its application in Windows Azure storage [110] and NEC HYDRAstor [60], the usage of erasure coding in primary storage, which requires high throughput and low latency, has not been widely adopted yet. To address such issue, our previous work [39] [40] leverages the parallelism to improve Jerasure's [168] coding performance for storage systems. That been said, erasure code is still a reactive solution to improve the storage reliability.

Other researches focus on RAID related technologies. For example, early works for tolerating multiple failures in an RAID array include [18], [6] and [52]. Researchers have also investigated remediation mechanisms to mitigate performance degradation caused by RAID recovery. For example, in [214], the authors presented a new RAID organization called multi-partition RAID to reduce the performance degradation during RAID rebuilds. In [238], the workload that targeting degraded RAID sets were outsourcing to surrogate RAID sets, hence improving the overall availability of a storage system. Parity declustering [108] [37] recently gained attention as it could reduce the reconstruction time of RAID 5 and 6. However, as we mentioned in Section 7.6.1, most of the problems these researches tries to addressed can be eliminated if we can proactively rescue data prior to disk drives failure.

8.1.3. Cost-Effective Disk Failure Data Rescue Schemes

To improve storage reliability, existing research mainly focus on RAID related technologies. For example, early works for tolerating multiple failures in an RAID array include [18], [6] and [52]. Goel and Corbett proposed a RAID technique that supports triple parity in [80]. Works in [112], [136], and [141] presented methods such as disk scrubbing to improve storage reliability. In [238], the workload that targeting degraded RAID sets were outsourcing to surrogate RAID sets, hence improving the overall availability of a storage system. Researchers have also investigated remediation mechanisms to mitigate performance degradation caused by RAID recovery. For example, in [214], the authors presented a new RAID organization called multi-partition RAID to reduce the performance degradation during RAID rebuilds. As a result, system performance was improved during the disk rebuild process, thereby improving the I/O throughput of user applications. Parity declustering [108] recently gained attention as it could reduce the reconstruction time of RAID 5 and 6. In [37], a new layout method for declustered RAID was presented. Improving disk reliability also attracts much attention, as it provides a solution to confine disk failures to an isolated region instead of causing the entire disk to be unavailable. Wan et al. proposed a data rescue scheme where data was migrated from bad sectors to a buffer zone on disk drives [223]. The Intelligent Storage Element (ISE)[211] implemented a similar mechanism by which malfunctioned disk drives were kept in service but at a reduced capacity. ISE requires proprietary software and hardware which affects its openness and wide adoption.

8.1.4. An Empirical Study of Quad-Level Cell SSDs

QLC SSD is a new product targeted at read-intensive workloads for use in data centers. As far as we know, there are only two related research papers focus on this aspect. The research conducted by Yoshiki et al. [208] focuses on QLC NAND flash memory power consumption and performance analysis on different heterogeneous SSD configurations. Their research points out that SCM(Storage Class Memory)/TLC configuration is optimal for cold workloads; while SLC/QLC configuration is recommended for hot workloads. Another research is purposed by Liu et. [137]. This paper studies efficient coding methods for QLC NAND flash. Their paper presents four enhanced Gray codings to QLC NAND to improve efficiency for read operations and data error correction. To distinct our work from the previous researches, our paper emphasize the performance evaluation of QLC SSD as a contender for HDD and other types of SSDs in data center storage systems. Our evaluation also compares QLC SSD against MLC or TLC SSD in terms of the economic aspects and analyzes how QLC SSD will change the landscape of modern data centers.

The 3D NAND QLC is by far the most promising solution to achieve the “high capacity, high reliability , low cost” goal in SSD storage. But it is not the only solution. Another famous storage technology is called *3D XPoint* [233] by Intel. Intel integrated this technology in its *Optane memory*, as well as applying this technology to its SSDs, namely

Optane SSD. Micron also has its own 3D XPoint brand, named *QuantX*. But Micron does not has any SSDs available that come embedded with this technology. The performance evaluation shows that *3D XPoint* SSD achieve better write latency and I/O speed than most 3D NAND SSDs in the market, according to the research [227]. However, the price of *Optane* SSD is still 4-5X greater.

8.2. Edge Applications

8.2.1. Popular Mutation Testing Tools

There are mutation tools for every major language. Some of widely used tools for Java include *muJava*[140] and *PIT*[51]. While *muJava* is not under active development, both tools have been used extensively in the industry. Python has *MutPy*[58], although it is only for Python 3.3 and above. For C and C++, there exists a number of tools. *MUSIC* was recently introduced in 2018 and was tested against complex industry software and a large open-source program [165]. *MuCPP* was introduced in 2014 [57], and uses similar methods in mutation testing as our work, namely abstract syntax tree traversal. The popular Javascript mutation testing tool *Stryker* [156] just reached version 2.0 and supports a wide variety of JS frameworks and language features. The same group that produced *Stryker*, also make *Stryker.NET*, which is a mutation tool for C# (.NET Core and .NET Framework projects). All of the tools suffer from expensive testing time on larger testsuites due to their longer preprocessing, compilation and linking phases.

8.2.2. Smart Police Patrolling

There has been a recent explosion in Smart City research. The novelty of the field lends itself to vastly different topics and solutions, while still under the “umbrella” of Smart City issues. Many safety-related Smart City research has been focused on mass surveillance and big data analysis, and not predictive/preemptive policing strategies. While many cities have heavily invested in surveillance networks, without combining this technology with “smarter” strategies, it provides both challenges and opportunities for smart city planning.

8.2.2.1. Predictive Policing

R. van Brakel and P. De Hert surveyed some of the work in predictive policing in their 2013 paper [221]. Their goals were to both determine the latest developments and technology in use for policing, namely the huge increase in surveillance and the viewpoint that using such technology can predict crime. Additionally, they sought to discover the inadvertent consequences that came about because of these "preemptive policing" programs.

Starting in July 2011 [12], the Santa Cruz police department implemented the first law-enforcement predictive policing program in the United States. The program processed burglary hotspots within the city, and published a "Top 10" list each day for units to patrol. The goal was to prevent crime, not to increase overall arrests, and 13 arrests were made during the program. While in six months they only showed a 4% reduction in burglaries compared to the same time-period the previous year, the program was innovative in the way it profiled the city for crime-prediction.

One of the main problems with both predictive and proactive policing strategies is how to measure "success". Since overall crime has been statistically on the decline in the United States, it is hard to empirically say a certain policing strategy has had a positive or even negative effect.

8.2.2.2. Entropy-Based Resource Consumption

In addition to the work on smarter water-distribution [46], many other researchers have explored resource planning based on entropy [164]. Since 100% coverage of resource utilization is infeasible due to both environmental and monetary factors, information-gain systems are a popular way to find optimal solutions.

J. H. Lee describes an entropy-based system for placing water-quality sensors in sewer systems [132]. After gathering data from monitoring points, a genetic algorithm was used to select from 80 points based on information gain. These 80 points represented a real sewer network in a small-subsection of Seoul, South Korea.

A similar technique [55] was used by Xiaoting et al. in Fujian province, China, to timely and accurately classify electric power marketing. Their results showed that using their entropy-based method was a better predictor of usage than the current system in place.

The requirements and results of entropy systems should be very attractive to smart city ventures. For city-planners, entropy systems have basic inputs that are easy to understand, and notably these inputs may already be available for collection. For policy-makers, entropy-based algorithms show a clear and reasonable path to their outcomes, as opposed to the complex neural networks of “deep learning.”

8.3. Connected Autonomous Vehicles

8.3.1. Embedded Deep Learning for Vehicular Edge Computing

Recent benchmarks/surveys on embedded systems has narrowed to specialized accelarators like VPUs. Recent research by Qasaimeh et al. [172] compared the ARM57 CPU, Nvidia Jetson TX2 GPU and Xilinx ZCU102 FPGA, using their own platform’s vision kernels for deep-learning. Work by Karki et al. [121] created *Tango*, a deep neural network benchmark suite. *Tango* supports any platform that can run CUDA or OpenCL, and includes simulators for server-GPU, mobile-GPU, and a mobile FPGA.

An extensive survey by Reuther et al. [187] covers everything from very low power learning accelerators, to high-power data center systems. Their work focused on Operations/Watt as their metric, and selected two commercially available low size, weight, and power (SWaP) accelerators to benchmark: the Google Edge TPU and the Intel® Movidius™ X Neural Compute Stick 2 (a successor to the VPU I benchmarked in Section 7.3).

8.3.2. Low-latency Data Sharing with L3

Object detection failures and visual obstructions are both core difficulties that all autonomous vehicle must face. Techniques such as cooperative perception (COOPER) [43] and others address this problem from a fundamental level through fusion. While detection results are improved, the wireless bandwidth available for V2X communications is too limited to support huge amount of data transmission among vehicles.

Currently-known fusion methods for connected and autonomous vehicles are categorized into three types: low-level, middle-level and high-level fusions. Low-level fusion is also called raw data fusion in which the original sensing data produced by vehicles are transmitted and shared among vehicles [128, 43]. While middle-level fusion methods make use of the extracted features from raw data to conduct fusion [118], high-level fusion mainly combines the sensing results processed by individual vehicles [2]. Other approaches like [240] and [173], marry the different sensors from the same vehicle to improve their object detection accuracy.

With the current works detailing the ground work, we know that communication in between vehicles plays an important limiting role based on the type of fusion methods being utilized. Taking COOPER [43] for example, while this method improves detection by merging point cloud data, it is limited by the narrow bandwidth available in vehicular networks. Not only does using higher quality sensors increase the amount of data that gets generated, using higher quality data also posses the risk that the data being generated will be too big to be transferred efficiently. Works exploring the sharing data between autonomous vehicles such as [103], discusses the uses of implementing V2X and identifies the requirements for doing so. The fundamental issue here is that existing vehicular network standards are design to exchange short messages among vehicles, rather than sensing data which could potentially be very large.

CHAPTER 9

SUMMARY

9.1. Conclusion

The Connected Autonomous Vehicle ecosystem is one of the most complicated environments in all of computing. Not only is the hardware scaled all the way from 16 and 32-bit microcontrollers, to multi-CPU Edge nodes, and multi-GPU Cloud servers, but the networking also encompasses the gamut of modern communication transports. Hypothetically, camera video recorded on a CAV could be transferred through 4G to an Edge server in a telecom cell site. From there it is transcoded to a 'human' appropriate color palette, and transferred over microwave to another cell site, where it's sent over fiber to a backbone and ingested into a specialized Cloud service queue for training machine learning models. From there, a customer can use their mobile phone to access and download this video. This scenario spans multiple manufacturers and many formats and protocols. For them all to work together, a CAV-focused approach needs to be applied. This includes major facets of extensibility and compatibility, as CAV development is increasingly accelerating, so versionless compatibility is paramount for platforms that may be on the road for a decade or more. Manufacturers will also be opening up their CAVs to deployment of applications they never designed or envisioned. This is a natural progression from current vehicle entertainment systems utilizing web APIs to access services like Spotify, or even fully embedding Apple CarPlay. Future applications will no doubt need direct access to local CAV data and manufacturers will need a safe, isolated environment to run these 3$^{\text{rd}}$-party applications in.

9.1.1. Major Contributions

In this dissertation my major contribution is the creation of a framework for CAV communication that can be used today, by developers at vehicle manufacturers, the network Edge, and cloud service providers. This design uses industry standards and 'battle-proven' components, deployed at the largest and smallest scale of computing. As detailed in Chapter 3, this framework is composed of a data format, protocol and application orchestration layer to overcome the challenges listed in Chapter 2. As shown in Chapters 4, 5 and 6, this framework is both technically superior and meets the other architecture requirements of the CAV system. Those chapters contain component benchmarks using modern tools, and explain integrations into developer workflows. I also provided detailed comparisons to other competing components and explained why those technologies do not meet the requirements of the CAV ecosystem.

9.1.2. Example Platform

Using this framework, an example edge management platform is shown in Figure 9.1. This platform uses the data format and protocol in Chapters 4 and 5. This management plane is deployed to any Cloud provider supporting Kubernetes, and controls Edge nodes built upon the orchestration platform described in Chapter 6. On top of the edge manager is a job queue, appropriate for distributing job requests across the clusters. An example job order flow is shown in Figure 9.3. A detailed view of a roadside Edge node is shown in Figure 9.2. Each Edge node is running Kubernetes operating system and includes a userspace job queue and scheduler for receiving job requests from the control plane. These road-side nodes also have an event bus, collecting events from passing vehicles, aggregating/processing them if needed, and passing the data back to the node manager. While communication within the platform is using gRPC from Chapter 5, a gRPC-Gateway is deployed for both the UI and also direct API end-users to consume. This Gateway uses the protobuf service definitions and generates a reverse-proxy server which translates the RESTful HTTP/JSON API calls into gRPC methods, as previously discussed in Section 5.3, allowing legacy clients an API entry point.

9.2. Published Work

My prior published work is focused on dependable computing, from hardware surveys, to predictive storage monitoring to produce reliable storage systems, and a mutation testing framework to produce complete and correct application tests. Other work has been focused on Smart City developments, like using data mining to create optimal police patrols based

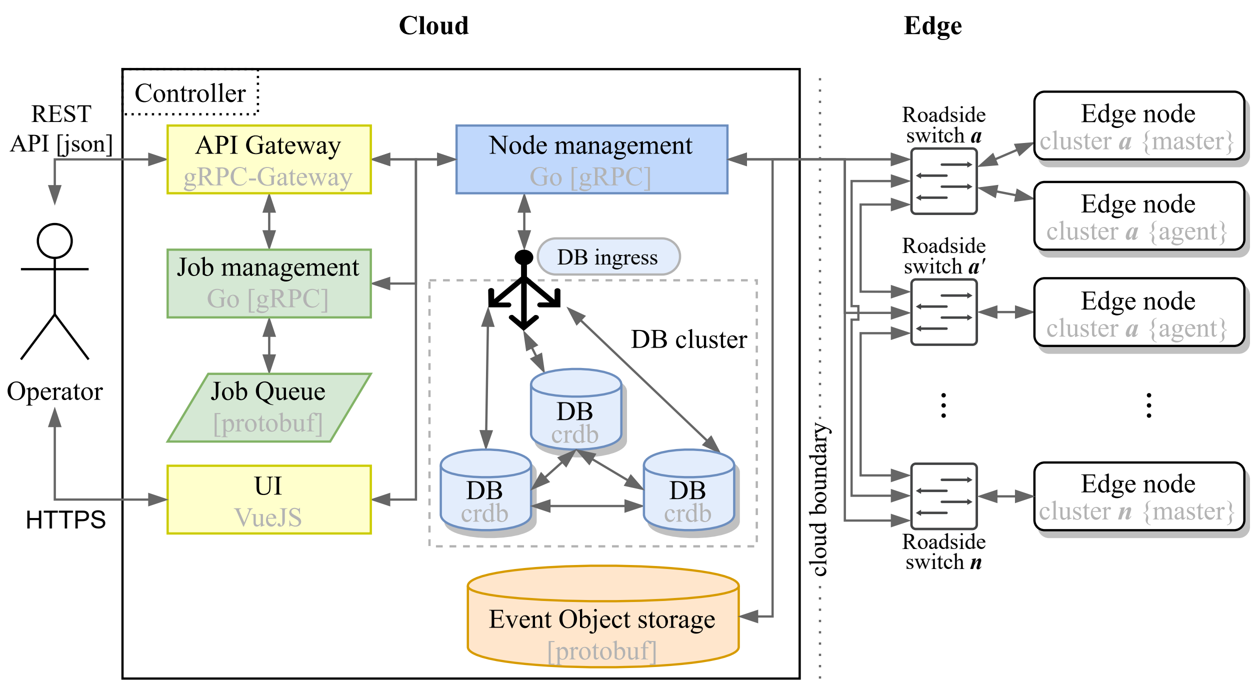


FIGURE 9.1. Example Edge management platform to control roadside Edge nodes. Overview of the roadside device is shown in Figure 9.2. Each component is isolated into a container-based replicaset. User-facing components are shown in yellow, internal services are blue, and the job engine is in green.

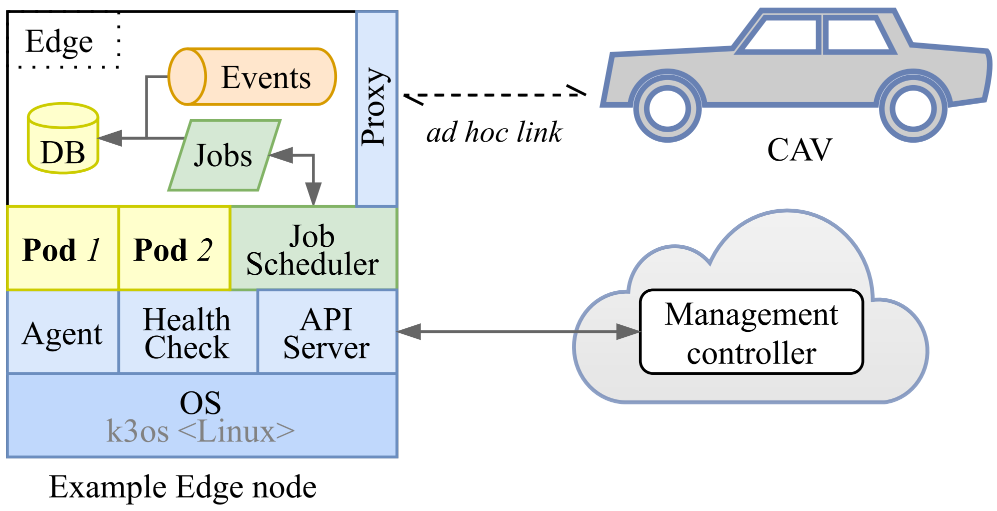


FIGURE 9.2. Example roadside Edge node for processing jobs from a cloud management framework. Host services are in blue, while job services are in green and userspace applications are in yellow. An event bus is in orange, collecting events from passing CAVs, processing them locally if possible, and aggregating them for shipment to cloud services if needed.

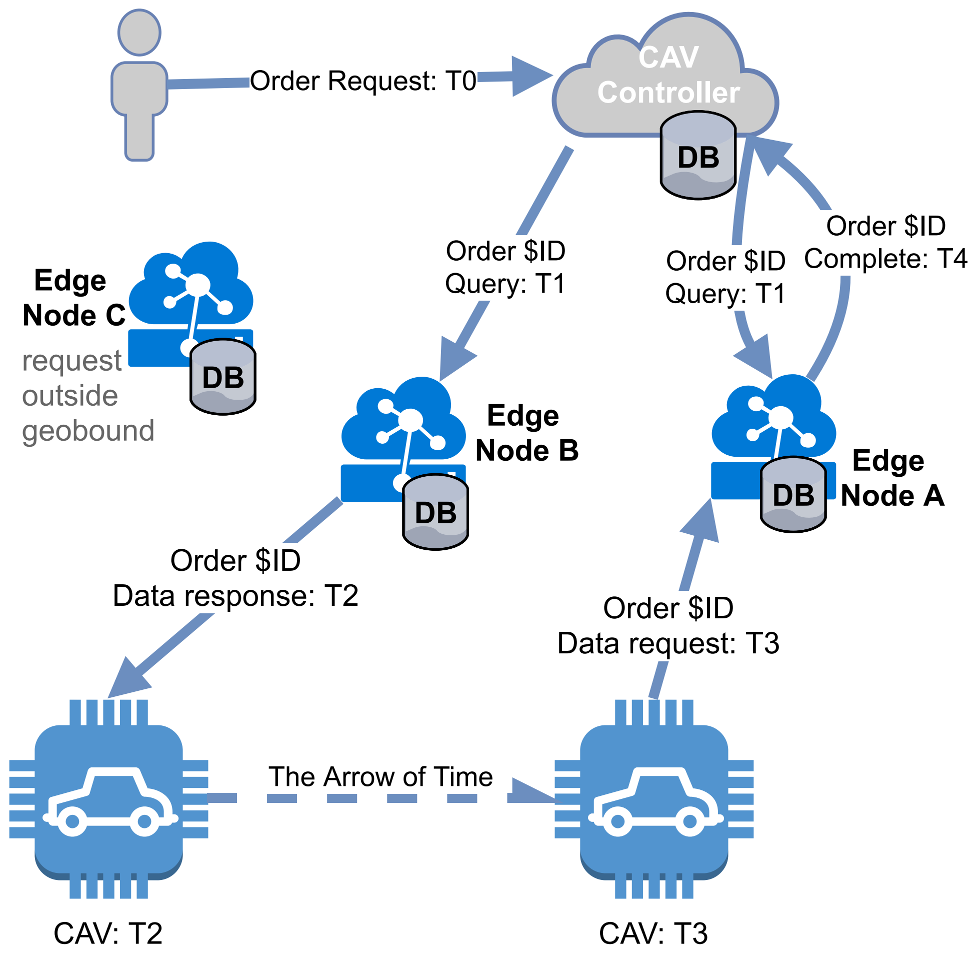


FIGURE 9.3. Example job order system flow using the Edge management platform and roadside Edge nodes. Edge clusters would dynamically be created automatically based on data locality, to ensure efficient processing respecting data gravity.

on information entropy. Finally, my latest work has been researching edge devices to further machine learning on embedded systems.

9.3. Intended Future Work

With the arrival of *k3os*, easily deploying clusters of single-board computers (SBC) is feasible for large-scale testing of edge clusters. Combined with redundant switches and power-over-Ethernet (PoE), a 100 m of roadside can be covered by five SBCs (one every 10 m), and three PoE switches. Combined with the lower cost of administration, real-life experiments with smaller, dynamic clusters of edge devices are possible using off-the-shelf components. With container-enabled Edge nodes immediately available, more research can focus on actually developing experimental applications, and not spend time on how to deploy or configure them.

End-user privacy is a major concern for CAV applications, and I will be researching the creation of an Edge-focused Trust Management System. This system will be based on security features found in the CoAP/Thread protocol, as the ‘Project Connected Home over IP’ has been established with a focus on security, and faces many of the same vulnerabilities and attack surfaces that CAVs/Edge devices do.

9.4. List of Prior Publications

- Jacob Hochstetler, Rahul Padidela, Qi Chen, Qing Yang, and Song Fu, *Embedded deep learning for vehicular edge computing*, 2018 IEEE/ACM Symposium on Edge Computing (SEC), IEEE, 2018
- Jacob Hochstetler, Lauren Hochstetler, and Song Fu, *An optimal police patrol planning strategy for smart city safety*, 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPC-C/SmartCity/DSS), IEEE, 2016
- Qi Chen, Shihai Tang, Jacob Hochstetler, Jingda Guo, Yuan Li, Jinbo Xiong, Qing Yang, and Song Fu, *Low-latency high-level data sharing for connected and autonomous vehicular networks*, 2019 IEEE International Conference on Industrial Internet (ICII),IEEE, 2019
- Shuwen Liang, Zhi Qiao, Jacob Hochstetler, Song Huang, Song Fu, Weisong Shi, Deesh Tiwari, Hsing-Bung Chen, Bradley Settlemyer, and David Montoya, *Reliability characterization of solid state drives in a scalable production datacenter*, 2018 IEEE International Conference on Big Data (Big Data), IEEE, 2018
- Shuwen Liang, Zhi Qiao, Sihai Tang, Jacob Hochstetler, Song Fu, Weisong Shi, and Hsing-Bung Chen, *An empirical study of quad-level cell (QLC) NAND flash SSDs for big data applications*, 2019 IEEE International Conference on Big Data (Big Data), IEEE, 2019

- Zhi Qiao, Jacob Hochstetler, Shuwen Liang, Song Fu, Hsing-bung Chen, and Bradley Settlemyer, *Incorporate proactive data protection in ZFS towards reliable storage systems*, Proceedings of IEEE International Conference on Big Data Intelligence and Computing(DataCom), 2018
- Zhi Qiao, Jacob Hochstetler, Shuwen Liang, Song Fu, Hsing-bung Chen, and Bradley Settlemyer, *Developing cost-effective data rescue schemes to tackle disk failures in data centers*, International Conference on Big Data, Springer, 2018

REFERENCES

[1] Randy Abernethy, *Programmer's guide to Apache Thrift*, Manning Publications, April 2019.

[2] Michael Aeberhard and Nico Kaempchen, *High-level sensor data fusion architecture for vehicle surround environment perception*, Proc. 8th Int. Workshop Intell. Transp, 2011.

[3] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa, *Firecracker: Lightweight virtualization for serverless applications*, 17th {usenix} symposium on networked systems design and implementation ({nsdi} 20), 2020, pp. 419–434.

[4] Petteri Aimonen, Kyle Manna, Oliver Lee, and Sébastien Morin, *Nanopb - Protocol Buffers for Embedded Systems*, 2021, `https://github.com/nanopb/nanopb`.

[5] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle, *Elasticity in Cloud Computing: State of the Art and Research Challenges*, IEEE Transactions on Services Computing 11 (2017), no. 2, 430–447.

[6] Guillermo A Alvarez, Walter A Burkhard, and Flaviu Cristian, *Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering*, Proceedings of the 24th annual international symposium on Computer architecture, 1997, pp. 62–72.

[7] Paul Ammann and Jeff Offutt, *Introduction to Software Testing*, Cambridge University Press, 2008.

[8] James M. Anderson, Kalra Nidhi, Karlyn D. Stanley, Paul Sorensen, Constantine Samaras, and Oluwatobi A. Oluwatola, *Autonomous vehicle technology: A guide for policymakers*, Rand Corporation, 2014.

[9] JENSC Arnbak and Wim Van Blitterswijk, *Capacity of slotted ALOHA in Rayleigh-fading channels*, IEEE Journal on Selected Areas in Communications 5 (1987), no. 2, 261–269.

[10] K3OS Project Authors and the Cloud Native Computing Foundation, *The Kubernetes Operating System*, `https://k3os.io/`.

[11] K3s Project Authors and the Cloud Native Computing Foundation, *K3s: Lightweight Kubernetes*, `https://k3s.io/`.

[12] Stephen Baxter, *Modest gains in first six months of Santa Cruz's predictive police program*, Santa Cruz Sentinel.—2012.—Retrieved (2015), 05–26.

[13] H. P. Belgal, N. Righos, I. Kalastirsky, J. J. Peterson, R. Shiner, and N. Mielke, *A new reliability model for post-cycling charge retention of flash memories*, Proc. 40th Annual (Cat. No.02CH37320) 2002 IEEE Int Reliability Physics Symp, April 2002, pp. 7–20.

[14] M. Belshe, R. Peon, and M. Thomson, *Hypertext transfer protocol version 2 (http/2)*, RFC 7540, RFC Editor, May 2015, `http://www.rfc-editor.org/rfc/rfc7540.txt`.

[15] Michael Bernstein, *5 Reasons to Use Protocol Buffers Instead of JSON for Your Next Service*, June 2014, `https://codeclimate.com/blog/choose-protocol-buffers/`.

[16] Giuseppe Bianchi, Luigi Fratta, and Matteo Oliveri, *Performance evaluation and enhancement of the CSMA/CA MAC protocol for 802.11 wireless LANs*, Proceedings of PIMRC'96-7th International Symposium on Personal, Indoor, and Mobile Communications, vol. 2, IEEE, 1996, pp. 392–396.

[17] Mike Bishop, *Hypertext transfer protocol version 3 (http/3)*, Internet-Draft draft-ietf-quic-http, Akamai, February 2021, `https://tools.ietf.org/html/draft-ietf-quic-http-34`.

[18] Mario Blaum, Jim Brady, Jehoshua Bruck, and Jai Menon, *EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures*, IEEE Transactions on Computers 44 (1995), no. 2, 192–202.

[19] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum, *The Zettabyte File System*, Proceedings of the 2nd Usenix Conference on File and Storage Technologies, vol. 215, 2003.

[20] Carsten Bormann, Angelo P Castellani, and Zach Shelby, *CoAP: An Application Pro-*

tocol for Billions of Tiny Internet Nodes, IEEE Internet Computing 16 (2012), no. 2, 62–67.

[21] Jon Bosak, *XML, Java, and the future of the Web*, World Wide Web Journal 2 (1997), no. 4, 219–227.

[22] Mirela Madalina Botezatu, Ioana Giurgiu, Jasmina Bogojeska, and Dorothea Wiesmann, *Predicting Disk Replacement towards Reliable Data Centers*, Proceedings of the 22nd International Conference on Knowledge Discovery and Data Mining ACM SIGKDD, ACM, 2016, pp. 39–48.

[23] A. Brand, K. Wu, S. Pan, and D. Chin, *Novel read disturb failure mechanism induced by FLASH cycling*, Proc. 31st Annual Reliability Physics 1993, March 1993, pp. 127–132.

[24] CW Bruce, *Police strategies and tactics: What every analyst should know*, International Association of Crime Analysts: 11 (2008), 1.

[25] Kevin Butler, *FMV 1.3 is now available!*, February 2016, `https://www.esri.com/arcgis-blog/products/product/announcements/fmv-1-3-is-now-available/`
Esri grants the recipient of the Esri information contained within the esri.com Web site the right to freely reproduce, redistribute, rebroadcast, and/or retransmit this information for personal, noncommercial purposes, including teaching, classroom use, scholarship, and/or research, subject to the fair use rights enumerated in sections 107 and 108 of the Copyright Act (Title 17 of the United States Code).

[26] Y. Cai, Y. Luo, S. Ghose, and O. Mutlu, *Read Disturb Errors in MLC NAND Flash Memory: Characterization, Mitigation, and Recovery*, Proc. 45th Annual IEEE/IFIP Int. Conf. Dependable Systems and Networks, June 2015, pp. 438–449.

[27] Y. Cai, Y. Luo, E. F. Haratsch, K. Mai, and O. Mutlu, *Data retention in MLC NAND flash memory: Characterization, optimization, and recovery*, Proc. IEEE 21st Int. Symp. High Performance Computer Architecture (HPCA), February 2015, pp. 551–563.

[28] Y. Cai, O. Mutlu, E. F. Haratsch, and K. Mai, *Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation*, Proc. IEEE 31st Int. Conf. Computer Design (ICCD), October 2013, pp. 123–130.

[29] Yu Cai, Saugata Ghose, Erich F. Haratsch, Yixin Luo, and Onur Mutlu, *Errors in Flash-Memory-Based Solid-State Drives: Analysis, Mitigation, and Recovery*, CoRR abs/1711.11427 (2017), 11427, `https://dblp.org/rec/bib/journals/corr/abs-1711-11427`.

[30] Yu Cai, Erich F Haratsch, Onur Mutlu, and Ken Mai, *Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis*, Proceedings of the Conference on Design, Automation and Test in Europe, EDA Consortium, 2012, pp. 521–526.

[31] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F Haratsch, Adrian Cristal, Osman S Unsal, and Ken Mai, *Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime*, Computer Design (ICCD), 2012 IEEE 30th International Conference on, IEEE, 2012, pp. 94–101.

[32] Canonical, *Jujucharms — Juju*, August 2018, `https://jujucharms.com/`.

[33] P. Cappelletti, R. Bez, D. Cantarelli, and L. Fratin, *Failure mechanisms of flash cell in program/erase cycling*, Proc. IEEE Int. Electron Devices Meeting, December 1994, pp. 291–294.

[34] Francois Caron, Emmanuel Duflos, Denis Pomorski, and Philippe Vanheeghe, *GPS/IMU data fusion using multisensor Kalman filtering: introduction of contextual aspects*, Information fusion 7 (2006), no. 2, 221–230.

[35] Nicolas Chaillan, *How the Department of Defense Moved to Kubernetes and Istio*, 2019.

[36] Bruno Chatras, *Spécification des règles d'encodage spécifiques pour la signalisation (SER)*, Tech. report, NT/DAC/SSR/12, Centre National d'Étude des Télécommunications (CNET, France), 1997.

[37] Siu-Cheung Chau and Ada Wai-Chee Fu, *A gracefully degradable declustered RAID architecture*, Cluster Computing 5 (2002), no. 1, 97–105.

[38] Feng Chen, David A Koufaty, and Xiaodong Zhang, *Understanding intrinsic characteristics and system implications of flash memory based solid state drives*, ACM SIGMETRICS Performance Evaluation Review, vol. 37, ACM, 2009, pp. 181–192.

[39] Hsing-Bung Chen and Song Fu, *Improving Coding Performance and Energy Efficiency of Erasure Coding Process for Storage Systems-A Parallel and Scalable Approach*, Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on, IEEE, 2016, pp. 933–936.

[40] ———, *Parallel Erasure Coding: Exploring Task Parallelism in Erasure Coding for Enhanced Bandwidth and Energy Efficiency*, Networking, Architecture and Storage (NAS), 2016 IEEE International Conference on, IEEE, 2016, pp. 1–4.

[41] Qi Chen, Bellows Brendan, Mike Wittie, Patterson Stacy, and Yang Qing, *MOVESET: MOdular VEhicle SEnsor Technology*, IEEE Vehicular Networking Conference (VNC) (2016), 1.

[42] Qi Chen, Shihai Tang, Jacob Hochstetler, Jingda Guo, Yuan Li, Jinbo Xiong, Qing Yang, and Song Fu, *Low-Latency High-Level Data Sharing for Connected and Autonomous Vehicular Networks*, 2019 IEEE International Conference on Industrial Internet (ICII), IEEE, 2019, pp. 287–296.

[43] Qi Chen, Sihai Tang, Qing Yang, and Fu Song, *Cooper: Cooperative Perception for Connected Autonomous Vehicles based on 3D Point Clouds*, arXiv preprint arXiv:1905.05265 (2019), 1.

[44] Xiaozhi Chen, Huimin Ma, Ji Wan, Bo Li, and Tian Xia, *Multi-View 3D Object Detection Network for Autonomous Driving*, The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), July 2017.

[45] Tong Cheng, Guangchi Liu, Qing Yang, and Jianguo Sun, *Trust assessment in vehicular social network based on three-valued subjective logic*, IEEE Transactions on Multimedia 21 (2019), no. 3, 652–663.

[46] Symeon E Christodoulou, *Smarting Up Water Distribution Networks With An Entropy-Based Optimal Sensor Placement Strategy*, Journal of Smart Cities 1 (2015), no. 1, 47–58.

[47] City of Los Angeles, *Los Angeles County GIS Data Portal*, April 2016, `http://egis3.lacounty.gov/dataportal/`.

[48] City of Los Angeles Personnel Department, *The LAPD Career Ladder*, January 2011, `http://www.joinlapd.com/career_ladder.html`.

[49] Adrian Cockcroft, *State of the Art in Microservices*, December 2014, DockerCon Europe.

[50] Adrian Cockcroft, *SIMulate Interactive Actor Network VIsualiZation (formerly Simulate Protocol Interactions in Go - spigo)*, December 2016, `https://github.com/adrianco/spigo`.

[51] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque, *PIT: A Practical Mutation Testing Tool for Java (Demo)*, Proceedings of the 25th International Symposium on Software Testing and Analysis (New York, NY, USA), ISSTA 2016, ACM, 2016, `http://doi.acm.org/10.1145/2931037.2948707`, pp. 449–452.

[52] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar, *Row-diagonal parity for double disk failure correction*, Proceedings of the 8th USENIX Conference on File and Storage Technologies, 2004.

[53] Council of European Union, *Council regulation (EU) no 269/2014*, 2014, `http://eur-lex.europa.eu/legal-content/EN/TXT/?qid=1416170084502&uri=CELEX:32014R0269`.

[54] Douglas Crockford, *The JSON Saga*, `https://www.microsoft.com/en-us/research/video/the-json-saga`.

[55] Xiao-ting Dai, Rong-si Chen, and Bing Xiao, *Application of decision tree mining algorithms based on information entropy in the intelligent electric power marketing [J]*, Journal of Zhengzhou University of Light Industry (Natural Science) 3 (2012), 1.

[56] R. Degraeve, F. Schuler, B. Kaczer, M. Lorenzini, D. Wellekens, P. Hendrickx, M. van Duuren, G. J. M. Dormans, J. Van Houdt, L. Haspeslagh, G. Groeseneken, and G. Tempel, *Analytical percolation model for predicting anomalous charge loss in flash memories*, IEEE Transactions on Electron Devices 51 (2004), no. 9, 1392–1400.

[57] Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Juan José Domínguez-Jiménez, Antonio García-

Domínguez, and Francisco Palomo-Lozano, *Class mutation operators for C++ object-oriented systems*, Annales des Télécommunications 70 (2014), no. 3-4, 137–148, `https://doi.org/10.1007/s12243-014-0445-4`.

[58] Anna Derezinska and Konrad Hałas, *Improving mutation testing process of python programs*, Software Engineering in Intelligent Systems, Springer, 2015, pp. 233–242.

[59] Michael Droettboom, *Understanding JSON Schema – Understanding JSON Schema 7.0 documentation*, `https://json-schema.org/understanding-json-schema/`.

[60] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki, *HYDRAstor: A scalable secondary storage.*, FAST, vol. 9, 2009, pp. 197–210.

[61] Olivier Dubuisson, *ASN.1 Communication Between Heterogeneous Systems*, Morgan Kaufmann, 2000.

[62] Prabal Dutta, Stephen Dawson-Haggerty, Yin Chen, Chieh-Jan Mike Liang, and Andreas Terzis, *Design and evaluation of a versatile and efficient receiver-initiated link layer for low-power wireless*, Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems, ACM, 2010, pp. 1–14.

[63] Ben Eckart, Xin Chen, Xubin He, and Stephen L Scott, *Failure prediction models for proactive fault tolerance within storage systems*, IEEE MASCOTS, 2008, pp. 1–8.

[64] Envision America, *10 U.S. Cities Selected to Kickoff Envision America Smart Cities Acceleration Initiative*, April 2016, `http://www.dallasinnovationalliance.com/news/?offset=1449593130557`.

[65] Rob Evans and Ayke van Laëthem, *TinyGo - A Go Compiler for Smalle Places*, 2021, `https://tinygo.org/`.

[66] F. Randall Farmer and Bryce Glass, *Building Web Reputation Systems*, O'Reilly Media, March 2010.

[67] Federal Aviation Administration, *GLOBAL POSITIONING SYSTEM (GPS) STANDARD POSITIONING SERVICE (SPS) PERFORMANCE ANALYSIS REPORT*, Tech. Report #107, William J. Hughes Technical Center, October 2019, page 22.

[68] Roy T Fielding, Richard N Taylor, Justin R Erenkrantz, Michael M Gorlick, Jim Whitehead, Rohit Khare, and Peyman Oreizy, *Reflections on the REST Architectural Style and "Principled Design of the Modern Web Architecture" (Impact Paper Award)*, 11$^{\text{th}}$ Foundations of Software Engineering (2017), 4–14.

[69] Roy Thomas Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, Ph.D. thesis, University of California, Irvine, 2000.

[70] The Apache Software Foundation, *Apache Avro^{TM} 1.10.1 Specification*, `https://avro.apache.org/docs/current/spec.html#schema_primitive`.

[71] Martin Fowler, *Continuous Integration - Keep The Build Fast*, April 2006, `http://martinfowler.com/articles/continuousIntegration.html`.

[72] ———, *BlueGreenDeployment*, March 2010, `https://martinfowler.com/bliki/BlueGreenDeployment.html`.

[73] ———, *Microservices and the First Law of Distributed Objects*, August 2014, `https://martinfowler.com/articles/distributed-objects-microservices.html`.

[74] Sadayuki Furuhashi, *MessagePack: It's like JSON. but fast and small.*, `https://msgpack.org`.

[75] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al., *An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems*, Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019, pp. 3–18.

[76] Jesse James Garrett, *Ajax: A new approach to web applications*, 2005, `http://adaptivepath.org/ideas/ajax-new-approach-web-applications/`.

[77] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun, *Vision meets Robotics: The KITTI Dataset*, International Journal of Robotics Research (IJRR) (2013), 1.

[78] Andreas Geiger, Philip Lenz, and Raquel Urtasun, *Are we ready for autonomous driving? The KITTI vision benchmark suite*, 2012 IEEE Conference on Computer Vision and Pattern Recognition, IEEE, 2012, pp. 3354–3361.

[79] Garth A. Gibson and David A. Patterson, *Designing Disk Arrays for High Data Reliability*, Journal of Parallel and Distributed Computing 17 (1993), no. 1-2, 4–27.

[80] Atul Goel and Peter Corbett, *RAID triple parity*, ACM SIGOPS Operating Systems Review 46 (2012), no. 3, 41–49.

[81] Aniruddha Gokhale, Bharat Kumar, and Arnaud Sahuguet, *Reinventing the wheel? CORBA vs. Web services*, Proceedings of 11th International World Wide Web Conference, 2002.

[82] Charles F Goldfarb, *The roots of SGML: A personal recollection*, Technical communication 46 (1999), no. 1, 75.

[83] Moises Goldszmidt, *Finding Soon-to-Fail Disks in a Haystack*, Proceedings of the HotStorage, 2012.

[84] Google, Inc., *Frequently Asked Questions (FAQ)*, `https://golang.org/doc/faq#What_is_the_purpose_of_the_project`.

[85] ———, *The Go Programming Language*, `https://github.com/golang/go/`.

[86] ———, *The Go Programming Language Specification*, `https://golang.org/ref/spec#Operators`.

[87] ———, *Google Maps Distance Matrix API*, April 2016, `https://developers.google.com/maps`.

[88] ———, *Cloud APIs Versioning*, February 2019, https://cloud.google.com/apis/design/versioning.

[89] ———, *Protocol Buffers Version 3 Language Specification*, 2020, https://developers.google.com/protocol-buffers/docs/reference/proto3-spec.

[90] Robert M Gray, *Entropy and information theory*, Springer Science & Business Media, 2011.

[91] Ilya Grigorik, *Making the web faster with HTTP 2.0*, Communications of the ACM 56 (2013), no. 12, 42–49.

[92] Laura M Grupp, John D Davis, and Steven Swanson, *The bleak future of NAND flash memory*, Proceedings of the 10th USENIX conference on File and Storage Technologies, USENIX Association, 2012, pp. 2–2.

[93] Daniel Gruver, Pierre-Yves Droz, Gaetan Pennecot, Anthony Levandowski, Drew Eugene Ulrich, Zachary Morriss, Luke Wachter, Dorel Ionut Iordache, Rahim Pardhan, William McCann, Bernard Fidric, and Samuel William Lenius, *Vehicle with multiple light detection and ranging devices (LIDARs)*, April 2017, US Patent 9,625,582.

[94] ———, *Vehicle with multiple light detection and ranging devices (LIDARs)*, January 2018, US Patent 9,864,063.

[95] Jingda Guo, Xianwei Cheng, and Qing Yang, *Detection of Occluded Road Signs on Autonomous Driving Vehicles*, 2019 IEEE International Conference on Multimedia and Expo (ICME), IEEE, 2019.

[96] D. Springmeyer H. Butler, C. Schmidt and J. Livni, *Spatial Reference*, April 2016, https://spatialreference.org.

[97] Mohammad Haghighat, Mohamed Abdel-Mottaleb, and Wadee Alhalabi, *Discriminant correlation analysis: Real-time feature level fusion for multimodal biometric recognition*, IEEE Transactions on Information Forensics and Security 11 (2016), no. 9, 1984–1996.

[98] Randy Heffne, *How to Avoid Building a Microservices Death Star*, August 2020, https://www.brighttalk.com/webcast/17917/428437/how-to-avoid-building-a-microservices-death-star-featuring-forrester.

[99] Dominique A Heger, *Workload Dependent Performance Evaluation of the Btrfs and ZFS Filesystems*, Proceedings of the International conference of CMG, 2009.

[100] Thomas R Henderson, Mathieu Lacage, George F Riley, Craig Dowell, and Joseph Kopena, *Network Simulations with the ns-3 Simulator*, SIGCOMM demonstration 14 (2008), no. 14, 527.

[101] Michi Henning, *The Rise and Fall of CORBA*, Queue 4 (2006), no. 5, 28–34.

[102] Munenori Hikita, *Automotive sensor MA58MF14-7N*, https://www.murata.com/en-us/products/productdetail.aspx?cate=cgsubUltrasonicSensors&partno=MA58MF14-7N.

[103] Laurens Hobert, Andreas Festag, Ignacio Llatser, Luciano Altomare, Filippo Visintainer, and Andras Kovacs, *Enhancements of V2X communication in support of cooperative autonomous driving*, IEEE communications magazine 53 (2015), no. 12, 64–70.

[104] Jacob Hochstetler, Lauren Hochstetler, and Song Fu, *An Optimal Police Patrol Planning Strategy for Smart City Safety*, 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), IEEE, 2016, pp. 1256–1263.

[105] Jacob Hochstetler, Rahul Padidela, Qi Chen, Qing Yang, and Song Fu, *Embedded deep learning for vehicular edge computing*, 2018 IEEE/ACM Symposium on Edge Computing (SEC), IEEE, 2018, pp. 341–343.

[106] Kevin Hoffman, David Zage, and Cristina Nita-Rotaru, *A survey of attack and defense techniques for reputation systems*, ACM Computing Surveys (CSUR) 42 (2009), no. 1, 1–33.

[107] Bernhard Hofmann-Wellenhof, Herbert Lichtenegger, and James Collins, *Global positioning system: theory and practice*, 5th ed., Springer Science & Business Media, December 2012.

[108] Mark Holland and Garth A Gibson, *Parity declustering for continuous operation in redundant disk arrays*, ACM SIGPLAN Notices 27 (1992), no. 9, 23–35.

[109] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam, *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*, CoRR abs/1704.04861 (2017), 1704.04861, `http://arxiv.org/abs/1704.04861`.

[110] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, Sergey Yekhanin, et al., *Erasure Coding in Windows Azure Storage*, Usenix ATC, Boston, MA, 2012, pp. 15–26.

[111] Song Huang, Song Fu, Quan Zhang, and Weisong Shi, *Characterizing Disk Failures with Quantified Disk Degradation Signatures: An Early Experience*, IEEE International Symposium on Workload Characterization (IISWC), IEEE, 2015, pp. 150–159.

[112] Ilias Iliadis, Robert Haas, Xiao-Yu Hu, and Evangelos Eleftheriou, *Disk scrubbing versus intra-disk redundancy for high-reliability raid storage systems*, ACM SIGMETRICS Performance Evaluation Review 36 (2008), no. 1, 241–252.

[113] Intel® Corporation, *Movidius™ Neural Compute SDK (ncsdk)*, `https://movidius.github.io/ncsdk/`, 7 2018, version V2.05.00.

[114] ———, *Movidius™ neural compute stick product page*, `https://developer.movidius.com/`, 2018.

[115] ISO, *ISO/IEC 14882:2017 Information technology — Programming languages — C++*, fifth ed.,

International Organization for Standardization, Geneva, Switzerland, December 2017, "`https://www.iso.org/standard/68564.html`".

[116] Audun Jøsang and Jennifer Golbeck, *Challenges for Robust Trust and Reputation Systems*, Proceedings of the 5th International Workshop on Security and Trust Management (SMT2009), Saint Malo, France, European Research Consortium in Informatics and Mathematics (ERCIM), October 2009, p. 52.

[117] Tae-Sung Jung, Young-Joon Choi, Kang-Deog Suh, Byung-Hoon Suh, Jin-Ki Kim, Young-Ho Lim, Yong-Nam Koh, Jong-Wook Park, Ki-Jong Lee, Jung-Hoon Park, Kee-Tae Park, Jang-Rae Kim, Jeong-Hyong Lee, and Hyung-Kyu Lim, *A 3.3 V 128 Mb multi-level NAND flash memory for mass storage applications*, Proc. ISSCC 1996 IEEE Int. Solid-State Circuits Conf.. Digest of TEchnical Papers, February 1996, pp. 32–33.

[118] Nico Kaempchen, Matthias Buehler, and Klaus Dietmayer, *Feature-level fusion for free-form object tracking using laserscanner and video*, Intelligent vehicles symposium, 2005. Proceedings. IEEE, IEEE, 2005, pp. 453–458.

[119] David Kahle and Hadley Wickham, *ggmap: Spatial Visualization with ggplot2*, The R journal 5 (2013), no. 1, 144–161.

[120] Gorkem Kar, Shubham Jain, Marco Gruteser, Jinzhu Chen, Fan Bai, and Ramesh Govindan, *PredriveID: Pre-trip Driver Identification from In-vehicle Data*, Proceedings of the Second ACM/IEEE Symposium on Edge Computing (New York, NY, USA), SEC '17, ACM, October 2017, `http://doi.acm.org/10.1145/3132211.3134462`, pp. 2:1–2:12.

[121] Aajna Karki, Chethan Palangotu Keshava, Spoorthi Mysore Shivakumar, Joshua Skow, Goutam Madhukeshwar Hegde, and Hyeran Jeon, *Tango: A deep neural network benchmark suite for various accelerators*, 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE, 2019, pp. 137–138.

[122] M. Kato, N. Miyamoto, H. Kume, A. Satoh, T. Adachi, M. Ushiyama, and K. Kimura, *Read-disturb degradation mechanism due to electron trapping in the tunnel oxide for low-voltage flash memories*, Proc. IEEE Int. Electron Devices Meeting, December 1994, pp. 45–48.

[123] Timothy H Keitt, *rgdal: Bindings for the Geospatial Data Abstraction Library, R package version 0.6-28*, CRAN (2010), 1, `http://cran.r-project.org/package=rgdal`.

[124] John B Kenney, *Dedicated short-range communications (DSRC) standards in the United States*, Proceedings of the IEEE 99 (2011), no. 7, 1162–1182.

[125] Lawrence A Klein and Lawrence A Klein, *Sensor and data fusion: a tool for information assessment and decision making*, vol. 324, SPIE press Bellingham, 2004.

[126] István Koren and Ralf Klamma, *The exploitation of OpenAPI documentation for the generation of web frontends*, Companion Proceedings of the The Web Conference 2018, 2018, pp. 781–787.

[127] Dan Kortschak, Vladimír Chalupecký, and Brendan Tracey, *Gonum*, Oct 2017, `https://www.gonum.org/`.

[128] Raphaël Labayrade, Cyril Royere, Dominique Gruyer, and Didier Aubert, *Cooperative fusion for multi-obstacles detection with use of stereovision and laser scanner*, Autonomous Robots 19 (2005), no. 2, 117–140.

[129] Redis Labs, *Redis Cluster Specifications*, `https://redis.io/topics/cluster-spec`.

[130] Olaf Landsiedel, Federico Ferrari, and Marco Zimmerling, *Chaos: Versatile and efficient all-to-all data sharing and in-network processing at scale*, ACM Embedded Networked Sensor Systems, November 2013.

[131] Jae-Duk Lee, Jeong-Hyuk Choi, Donggun Park, and Kinam Kim, *Degradation of tunnel oxide by FN current stress and its effects on data retention characteristics of 90 nm NAND flash memory cells*, Proc. 41st Annual 2003 IEEE Int. Reliability Physics Symp, March 2003, pp. 497–501.

[132] Jung Ho Lee, *Determination of optimal water quality monitoring points in sewer systems using entropy theory*, Entropy 15 (2013), no. 9, 3419–3434.

[133] Steven Legg, *RFC3641: Generic String Encoding Rules (GSER) for ASN. 1 Types*, 2003.

[134] Shuwen Liang, Zhi Qiao, Jacob Hochstetler, Song Huang, Song Fu, Weisong Shi, Devesh Tiwari, Hsing-Bung Chen, Bradley Settlemyer, and David Montoya, *Reliability characterization of solid state drives in a scalable production datacenter*, 2018 IEEE International Conference on Big Data (Big Data), IEEE, 2018, pp. 3341–3349.

[135] Shuwen Liang, Zhi Qiao, Sihai Tang, Jacob Hochstetler, Song Fu, Weisong Shi, and Hsing-Bung Chen, *An Empirical Study of Quad-Level Cell (QLC) NAND Flash SSDs for Big Data Applications*, 2019 IEEE International Conference on Big Data (Big Data), IEEE, 2019, pp. 3676–3685.

[136] Junping Liu, Ke Zhou, Zhikun Wang, Liping Pang, and Dan Feng, *Modeling the Impact of Disk Scrubbing on Storage System*, Journal of Computers 5 (2010), no. 11, 1629–1637.

[137] Shijun Liu and Xuecheng Zou, *QLC NAND Study and Enhanced Gray Coding Methods for Sixteen-level-based Program Algorithms*, Microelectron. J. 66 (2017), 58–66.

[138] Los Angeles Open Data, *Information, Insights, and Analysis from the City of Los Angeles*, April 2016, `https://data.lacity.org/`.

[139] Yixin Luo, *Architectural Techniques for Improving NAND Flash Memory Reliability*, Ph.D. thesis, School of Computer Science Carnegie Mellon University, 2018.

[140] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon, *MuJava: A Mutation System for Java*, Proceedings

of the 28th International Conference on Software Engineering (New York, NY, USA), ICSE '06, ACM, 2006, `http://doi.acm.org/10.1145/1134285.1134425`, pp. 827–830.

[141] Farzaneh Mahdisoltani, Ioan A. Stefanovici, and Bianca Schroeder, *Proactive error prediction to improve storage system reliability*, USENIX Annual Technical Conference, 2017.

[142] Mugur Marculescu, *Introducing gRPC, a new open source HTTP/2 RPC Framework*, February 2015, `https://developers.googleblog.com/2015/02/introducing-grpc-new-open-source-http2.html`.

[143] F. Margaglia and A. Brinkmann, *Improving MLC flash performance and endurance with extended P/E cycles*, Proc. 31st Symp. Mass Storage Systems and Technologies (MSST), May 2015, pp. 1–12.

[144] Carl Mastrangelo, *Visualizing gRPC Language Stacks*, December 2018, `https://grpc.io/blog/grpc-stacks/`.

[145] Anne McCrory, *Ubiquitous? Pervasive? Sorry, they don't compute*, Computerworld (2000), 1.

[146] Peter Mell and Tim Grance, *Special Publication 800-145: The NIST definition of Cloud Computing*, Tech. report, National Institute of Standards and Technology, Gaithersburg MD, 20899, September 2011.

[147] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu, *A large-scale study of flash memory failures in the field*, ACM SIGMETRICS Performance Evaluation Review, vol. 43, ACM, 2015, pp. 177–190.

[148] C. Miccoli, J. Barber, C. M. Compagnoni, G. M. Paolucci, J. Kessenich, A. L. Lacaita, A. S. Spinelli, R. J. Koval, and A. Goda, *Resolving discrete emission events: A new perspective for detrapping investigation in NAND Flash memories*, Proc. IEEE Int. Reliability Physics Symp. (IRPS), April 2013, pp. 3B.1.1–3B.1.6.

[149] N. Mielke, T. Marquart, Ning Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L. R. Nevill, *Bit error rate in NAND Flash memories*, Proc. IEEE Int. Reliability Physics Symp, April 2008, pp. 9–19.

[150] Rick Mohr and Paul Peltz Jr, *Benchmarking SSD-based Lustre file system configurations*, Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment, ACM, 2014, p. 32.

[151] Joseph F Murray, Gordon F Hughes, and Kenneth Kreutz-Delgado, *Hard drive failure prediction using non-parametric statistical methods*, Proceedings of the ICANN/ICONIP, 2003.

[152] ———, *Machine learning methods for predicting failures in hard drives: A multiple-instance application*, Journal of Machine Learning Research 6 (2005), no. May, 783–816.

[153] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid, *SSD failures in datacenters:*

What? When? and Why?, Proceedings of the 9th ACM International on Systems and Storage Conference, ACM, 2016, p. 7.

[154] Paolo Neirotti, Alberto De Marco, Anna Corinna Cagliano, Giulio Mangano, and Francesco Scorrano, *Current trends in Smart City initiatives: Some stylised facts*, Cities 38 (2014), 25–36.

[155] H Michael Newman, *BACnet [R] explained: Part one*, ASHRAE Journal 55 (2013), no. 11, B2–B2.

[156] Simon de Lang Nico Jansen and Alex van Assem, *Announcing Stryker 2.0*, `https://stryker-mutator.io/blog/2019-05-17/announcing-stryker-2-0`.

[157] NOAA and National Weather Service, *NOWData - NOAA Online Weather Data*, April 2016, `http://w2.weather.gov/climate/xmacis.php`.

[158] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta, *Comparison of JSON and XML data interchange formats: a case study.*, ISCA 22$^{\text{nd}}$ International Conference on Computer Applications in Industry and Engineering (CAINE) 9 (2009), 157–162.

[159] A. Jefferson Offutt, *A Practical System for Mutation Testing: Help for the Common Programmer*, Proceedings IEEE International Test Conference 1994, TEST: The Next 25 Years, Washington, DC, USA, October 2-6, 1994, 1994, `https://doi.org/10.1109/TEST.1994.528535`, pp. 824–830.

[160] A. Jefferson Offutt and Roland H. Untch, *Mutation 2000: Uniting the Orthogonal*, 2001, pp. 34–44.

[161] Jeff Offutt, *A Mutation Carol: Past, Present and Future*, Information & Software Technology 53 (2011), no. 10, 1098–1107, `https://doi.org/10.1016/j.infsof.2011.03.007`.

[162] Andy Oram and Greg Wilson, *Beautiful Code: Leading Programmers Explain How They Think*, O'Reilly Media, Inc, December 2008.

[163] Fredrik Österlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt, *Cross-Level Sensor Network Simulation with COOJA*, First IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2006), 2006.

[164] Maria Papadopoulou, Benny Raphael, Ian FC Smith, and Chandra Sekhar, *Hierarchical sensor placement using joint entropy and the effect of modeling error*, Entropy 16 (2014), no. 9, 5078–5101.

[165] Duy Loc Phan, Yunho Kim, and Moonzoo Kim, *MUSIC: Mutation Analysis Tool with High Configurability and Extensibility*, 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE, 2018, pp. 40–46.

[166] Veerapat Phromchana, Natawut Nupairoj, and Krerk Piromsopa, *Performance Evaluation of ZFS and LVM (with ext4) for Scalable Storage System*, Computer Science and Software Engineering (JCSSE), 2011 Eighth International Joint Conference on, IEEE, 2011, pp. 250–253.

[167] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso, *Failure Trends in a Large Disk*

Drive Population, Proceedings of the 8th USENIX Conference on File and Storage Technologies, 2007.

[168] James S Plank, Scott Simmerman, and Catherine D Schuman, *Jerasure: A library in C/C++ facilitating erasure coding for storage applications-Version 1.2*, Tech. Report CS-08-627, University of Tennessee, 2008.

[169] Meghan Pleticha, *Silicon Valley*, May 2017.

[170] Eugenia Politou, Efthimios Alepis, and Constantinos Patsakis, *Forgetting personal data and revoking consent under the GDPR: Challenges and proposed solutions*, Journal of Cybersecurity 4 (2018), no. 1, tyy001.

[171] Tom Preston-Werner, *Semantic Versioning 2.0.0*, `https://semver.org/`.

[172] Murad Qasaimeh, Kristof Denolf, Alireza Khodamoradi, Michaela Blott, Jack Lo, Lisa Halder, Kees Vissers, Joseph Zambreno, and Phillip H Jones, *Benchmarking vision kernels and neural network inference accelerators on embedded platforms*, Journal of Systems Architecture 113 (2021), 101896.

[173] Charles R Qi, Wei Liu, Chenxia Wu, Hao Su, and Leonidas J Guibas, *Frustum PointNets for 3D Object Detection from RGB-D Data*, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 918–927.

[174] Zhi Qiao, Jacob Hochstetler, Shuwen Liang, Song Fu, Hsing-bung Chen, and Bradley Settlemyer, *Developing Cost-Effective Data Rescue Schemes to Tackle Disk Failures in Data Centers*, Big Data – BigData 2018 (Cham) (Francis Y. L. Chin, C. L. Philip Chen, Latifur Khan, Kisung Lee, and Liang-Jie Zhang, eds.), Springer International Publishing, 2018, pp. 194–208.

[175] ———, *Incorporate Proactive Data Protection in ZFS Towards Reliable Storage Systems*, 2018 IEEE 16th International Conference on Dependable, Autonomic and Secure Computing, 16th International Conference on Pervasive Intelligence and Computing, 4th International Conference on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/-CyberSciTech), IEEE, 2018, pp. 904–911.

[176] Krzysztof Rakowski, *Learning Apache Thrift*, Packt Publishing Ltd, 2015.

[177] Raspberry Pi Foundation™, *Raspberry Pi 3 Model B product page*, `https://www.raspberrypi.org/products/raspberry-pi-3-model-b/`, 2018.

[178] Jerry H Ratcliffe, Travis Taniguchi, Elizabeth R Groff, and Jennifer D Wood, *The Philadelphia foot patrol experiment: A randomized controlled trial of police patrol effectiveness in violent crime hotspots*, Criminology 49 (2011), no. 3, 795–831.

[179] ITU-T Recommendation, *X.690 Information technology - ASN.1 encoding rules: Specification of Basic*

Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER), Tech. report, Technical report, International Telecommunication Union (ITU-T), 2015.

[180] ———, *X.691 Information technology - ASN.1 encoding rules: Specification of Packed Encoding Rules (PER)*, Tech. report, Technical report, International Telecommunication Union (ITU-T), 2015.

[181] ———, *X.693 Information technology - ASN.1 encoding rules: XML Encoding Rules (XER)*, Tech. report, Technical report, International Telecommunication Union (ITU-T), 2015.

[182] ———, *X.696 Information technology - ASN.1 encoding rules: Specification of Octet Encoding Rules (OER)*, Tech. report, Technical report, International Telecommunication Union (ITU-T), 2015.

[183] ———, *X.697 Information technology - ASN.1 encoding rules: Specification of JavaScript Object Notation Encoding Rules (JER)*, Tech. report, Technical report, International Telecommunication Union (ITU-T), 2017.

[184] Red Hat, Inc., *What's a Linux container?*, 2019, `https://www.redhat.com/en/topics/containers/whats-a-linux-container`.

[185] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun, *Faster R-CNN: Towards real-time object detection with region proposal networks*, Advances in neural information processing systems, June 2015, pp. 91–99.

[186] Shaoqing Ren, Kaiming He, Ross Girshick, Xiangyu Zhang, and Jian Sun, *Object detection networks on convolutional feature maps*, IEEE transactions on pattern analysis and machine intelligence 39 (2016), no. 7, 1476–1481.

[187] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, *Survey and benchmarking of machine learning accelerators*, 2019 IEEE High Performance Extreme Computing Conference (HPEC), 2019, pp. 1–9.

[188] Cristian Zambelli Rino Micheloni, Luca Crippa and Piero Olivo, *Architectural and Integration Options for 3D NAND Flash Memories*, Computers 6 (2017), no. 27, 1, `https://doi.org/10.3390/computers6030027`.

[189] Arnon Rotem-Gal-Oz, *Fallacies of distributed computing explained*, 2006.

[190] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei, *ImageNet Large Scale Visual Recognition Challenge*, International Journal of Computer Vision (IJCV) 115 (2015), no. 3, 211–252.

[191] Louis Ryan, *gRPC Motivation and Design Principles*, September 2015, `https://grpc.io/blog/principles/`.

[192] SAE International, *Taxonomy and Definitions for Terms Related to Driving Automation Systems for*

On-Road Motor Vehicles [standard J3016], June 2018, `https://www.sae.org/standards/content/j3016_201806/`.

[193] M. Satyanarayanan, *Pervasive computing: vision and challenges*, IEEE Personal Communications 8 (2001), no. 4, 10–17.

[194] Onur Savas and Julia Deng, *Big data analytics in cybersecurity*, Auerbach Publications, 2017.

[195] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant, *Flash Reliability in Production: The Expected and the Unexpected*, FAST, 2016, pp. 67–80.

[196] Kim Bartel Sheehan, *Toward a typology of Internet users and online privacy concerns*, The Information Society 18 (2002), no. 1, 21–32.

[197] Z. Shelby, K. Hartke, and C. Bormann, *The Constrained Application Protocol (CoAP)*, RFC 7252, RFC Editor, June 2014, `http://www.rfc-editor.org/rfc/rfc7252.txt`.

[198] W. Shi and S. Dustdar, *The Promise of Edge Computing*, Computer 49 (2016), no. 5, 78–81.

[199] Xuecheng Zou Shijun Liu, *Analysis of 3D NAND technologies and comparison between charge-trap-based and floating-gate-based flash devices*, Journal of Universities of Posts and Telecommunications 24 (2017), 75–96.

[200] Jérôme Siméon and Philip Wadler, *The Essence of XML*, ACM Sigplan Notices 38 (2003), no. 1, 1–13.

[201] Peter Desnoyers Simona Boboila, *Write Endurance in Flash Drives: Measurements and Analysis*, 2010.

[202] Preston M Smith, Jason St. John, and Stephen Lien Harrell, *There and Back Again: A Case Study of Configuration Management of HPC*, Proceedings of the HPC Systems Professionals Workshop, 2017, pp. 1–7.

[203] Joel Spolsky, *The Law of Leaky Abstractions*, Joel on Software, Springer, 2004, pp. 197–202.

[204] Beth Anne Steel, *Securing Smart TVs*, FBI (2019), 1, `https://www.fbi.gov/contact-us/field-offices/portland/news/press-releases/tech-tuesdaysmart-tvs`.

[205] Bo Mao Suzhen Wu, Yanping Lin and Hong Jiang, *GCaR: Garbage Collection aware Cache Management with Improved Performance for Flash-based SSDs*, Proceedings of ACM International Conference on Supercomputing (ICS) (2016), 1–12, `https://doi.org/10.1145/2925426.2926263`.

[206] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich, *Going Deeper with Convolutions*, CoRR abs/1409.4842 (2014), 1409.4842, `http://arxiv.org/abs/1409.4842`.

[207] Naomi Tajitsu, *On the radar: Nissan stays cool on lidar tech, siding with Tesla*, May 2019, `https://www.reuters.com/article/us-nissan-lidar-autonomous-idUSKCN1SM0W2`.

[208] Y. Takai, M. Fukuchi, R. Kinoshita, C. Matsui, and K. Takeuchi, *Analysis on Heterogeneous SSD*

Configuration with Quadruple-Level Cell (QLC) NAND Flash Memory, Proceedings of IEEE International Memory Workshop (IMW), 2019.

[209] K. Takeuchi, S. Satoh, T. Tanaka, K. Imamiya, and K. Sakui, *A negative Vth cell architecture for highly scalable, excellently noise immune and highly reliable NAND flash memories*, Proc. Symp. VLSI Circuits. Digest of Technical Papers (Cat. No.98CH36215), June 1998, pp. 234–235.

[210] R Core Team et al., *R: A language and environment for statistical computing*, R Core Team (2013), 1.

[211] X-IO Technology, *Disk Failure Isolation*, 2017.

[212] The SAN Guy, *Storage performance benchmarking with fio*, 2019, `https://thesanguy.com/tag/block-size/`.

[213] Ian Thibodeau, *Ford profits down more than 50 percent in 2018*, The Detroit News (2019), 1.

[214] W-J Tsai and S-Y Lee, *Multi-partition RAID: A new method for improving performance of disk arrays under failure*, The Computer Journal 40 (1997), no. 1, 30–42.

[215] H. Tseng, L. Grupp, and S. Swanson, *Understanding the impact of power loss on flash memory*, Proceedings of ACM/IEEE Design Automation Conference (DAC), 2011.

[216] Ishaq Unwala, Zafar Taqvi, and Jiang Lu, *Thread: An iot protocol*, 2018 IEEE Green Technologies Conference (GreenTech), IEEE, 2018, pp. 161–167.

[217] U.S. Air Force Air Combat Command Public Affairs, *U-2 Federal Lab achieves flight with Kubernetes*, October 2020, `https://www.af.mil/News/Article-Display/Article/2375297/u-2-federal-lab-achieves-flight-with-kubernetes/`.

[218] U.S. Department of Health and Human Services, Food and Drug Administration, *Compliance guide for laser products*, Center for Devices and Radiological Health, Food and Drug Administration (DHHS), FDA (1992), 4, 86-8260.

[219] U.S. Senate. 115th Congress, 1st Session., *American Vision for Safer Transportation through Advancement of Revolutionary Technologies Act*, September 2017, `https://www.congress.gov/bill/115th-congress/senate-bill/1885`.

[220] U.S. White House, *Administration Announces New Smart Cities Initiative to Help Communities Tackle Local Challenges and Improve City Services*, September 2015, `https://www.whitehouse.gov/the-press-office/2015/09/14/fact-sheet-administration-announces-new-smart-cities-initiative-help`.

[221] Rosamunde Van Brakel and Paul De Hert, *Policing, surveillance and law in a pre-crime society: Understanding the consequences of technology based strategies*, Technology-led policing 20 (2011), 165.

[222] Velodyne LiDAR, Inc., *Alpha Puck™ Product Data Sheet*, 2019, Rev-3.

[223] Jiguang Wan, Jibin Wang, Jianzong Wang, Zhihu Tan, and Maliang Liu, *RSA: RAID system with Self-healing and Active data migration*, IEEE International Conference on Intelligent Computing and Intelligent Systems (ICIS), vol. 3, IEEE, 2010, pp. 582–586.

[224] Yan Wang, Wei-Lun Chao, Divyansh Garg, Bharath Hariharan, Mark Campbell, and Kilian Q Weinberger, *Pseudo-LiDAR from Visual Depth Estimation: Bridging the Gap in 3D Object Detection for Autonomous Driving*, CVPR (2019), 1.

[225] SungJin Whang, KiHong Lee, DaeGyu Shin, BeomYong Kim, MinSoo Kim, JinHo Bin, JiHye Han, SungJun Kim, BoMi Lee, YoungKyun Jung, SungYoon Cho, ChangHee Shin, HyunSeung Yoo, SangMoo Choi, Kwon Hong, Seiichi Aritome, SungKi Park, and SungJoo Hong, *Novel 3-dimensional Dual Control-gate with Surrounding Floating-gate (DC-SF) NAND flash cell for 1Tb file storage application*, International Electron Devices Meeting (2010), 29.7.1–29.7.4, `https://doi.org/10.1109/IEDM.2010.5703447`.

[226] Hadley Wickham and W Chang, *ggplot2*, 2012, `http://ggplot2.org`.

[227] Wikipedia, *3D XPoint*, 2019, `https://en.wikipedia.org/wiki/3DXPoint`.

[228] ______, *Gray Code*, 2019, `https://en.wikipedia.org/wiki/Graycode`.

[229] ______, *Multi-level cell*, 2019, `https://en.wikipedia.org/wiki/Multi-levelcell`.

[230] ______, *NVME*, 2019, `https://en.wikipedia.org/wiki/NVMExpress`.

[231] ______, *Serial ATA*, 2019, `https://en.wikipedia.org/wiki/SerialATA`.

[232] ______, *S.M.A.R.T.*, 2019, `https://en.wikipedia.org/wiki/S.M.A.R.T`.

[233] ______, *SSD*, 2019, `https://en.wikipedia.org/wiki/Solid-statedrive`.

[234] ______, *Trim*, 2019, `https://en.wikipedia.org/wiki/Trim(computing)`.

[235] Emma Woollacott, *UK Government Hands NHS Data To Amazon For Free*, December 2019, `https://www.forbes.com/sites/emmawoollacott/2019/12/09/uk-government-hands-nhs-data-to-amazon-for-free/#20d1a2e17f59`.

[236] Austin Wright, Henry Andrews, Ben Hutton, and Greg Dennis, *JSON Schema: A Media Type for Describing JSON Documents*, Internet-Draft draft-handrews-json-schema-02, Internet Engineering Task Force, September 2019, WIP `https://datatracker.ietf.org/doc/html/draft-handrews-json-schema-02`.

[237] S. Wu, H. Li, B. Mao, X. Chen, and K. Li, *Overcome the GC-Induced Performance Variability in SSD-Based RAIDs With Request Redirection*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 38/5 (2019), 822–833.

[238] Suzhen Wu, Hong Jiang, Dan Feng, Lei Tian, and Bo Mao, *Improving availability of RAID-structured storage systems by workload outsourcing*, IEEE Transactions on Computers 60 (2011), no. 1, 64–79.

[239] Kevin Balke Xiaosi Zeng and Praprut Songchitruksa, *Potential Connected Vehicle Applications to Enhance Mobility, Safety, and Environmental Security*, Tech. report, Southwest Region University Transportation Center (US), Texas Transportation Institute, 2012.

[240] Danfei Xu, Dragomir Anguelov, and Ashesh Jain, *PointFusion: Deep Sensor Fusion for 3D Bounding Box Estimation*, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 244–253.

[241] R. Yamada, Y. Mori, Y. Okuyama, J. Yugami, T. Nishimoto, and H. Kume, *Analysis of detrap current due to oxide traps to improve flash memory retention*, Proc. 38th Annual (Cat. No.00CH37059) 2000 IEEE Int Reliability Physics Symp, April 2000, pp. 200–204.

[242] R. Yamada, T. Sekiguchi, Y. Okuyama, J. Yugami, and H. Kume, *A novel analysis method of threshold voltage shift due to detrap in a multi-level flash memory*, Proc. Symp. VLSI Technology. Digest of Technical Papers (IEEE Cat. No.01 CH37184), June 2001, pp. 115–116.

[243] Qing Yang, Alvin Lim, Shuang Li, Jian Fang, and Prathima Agrawal, *ACAR: Adaptive Connectivity Aware Routing for Vehicular Ad Hoc Networks in City Scenarios*, Mob. Netw. Appl. 15 (2010), no. 1, 36–60.

[244] Qing Yang and Honggang Wang, *Towards Trustworthy Vehicular Social Network*, IEEE Communication Magazine 53 (2015), no. 8, 42–47.

[245] Qing Yang, Binghai Zhu, and Shaoen Wu, *An architecture of cloud-assisted information dissemination in vehicular networks*, IEEE Access 4 (2016), 2764–2770.

[246] François Yergeau, Tim Bray, Jean Paoli, CM Sperberg-McQueen, and Eve Maler, *Extensible markup language (xml) 1.0 w3c recommendation*, Tech. report, World Wide Web Consortium, 2004.

[247] Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal, *Wireless sensor network survey*, Computer networks 52 (2008), no. 12, 2292–2330.

[248] Junko Yoshida, *Counter Argument: 3 Reasons We Need V2X*, EE Times (2013), 88.

[249] Tom Yu, *Yu2000potentialpo*, Internet-Draft yu-asn1-pitfalls-00, MIT, March 2000, `https://tools.ietf.org/id/draft-yu-asn1-pitfalls-00.txt`.

[250] Jikai Zhang and Liu Zhengyang, *gRPC-swagger*, `https://github.com/grpc-swagger/grpc-swagger`.

[251] Qingyang Zhang, Yifan Wang, Xingzhou Zhang, Liangkai Liu, Xiaopei Wu, Weisong Shi, and Hong Zhong, *OpenVDAP: An Open Vehicular Data Analytics Platform for CAVs*, 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS) (2018), 1.

[252] Qingyang Zhang, Quan Zhang, Weisong Shi, and Hong Zhong, *Enhancing AMBER Alert Using Collaborative Edges: Poster*, Proceedings of the Second ACM/IEEE Symposium on Edge Computing (New York, NY, USA), SEC '17, ACM, October 2017, `http://doi.acm.org/10.1145/3132211.3132459`, pp. 27:1–27:2.

[253] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge, *Understanding the Robustness of SSDs under Power Fault*, Proceedings of USENIX Conference on File and Storage Technologies (FAST), 2013.

[254] Yin Zhou and Oncel Tuzel, *VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection*, The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), June 2018.

[255] Zigbee Alliance, *Project Connected Home over IP*, 2021, `https://www.connectedhomeip.com/`.