Fermi National Accelerator Laboratory

FERMILAB-Conf-95/118

Fermilab DART Run Control

G. Oleynik, J. Engelfried, L. Mengel, C. Moore, V. Odell, R. Pordes
A. Semenchenko, D. Slimmer, L. Udumula and M. Votava

Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510

F. Prelz

INFN and Dipartimento di Fisica dell' Universit'a and INFN
Milano, I-20133 Milan, Italy

E. Van Drunen and G. Zioulas

University of California at Irvine
Irvine, California 92717

May 1995

Presented at the *Real Time 95 Conference*, East Lansing, Michigan, May 22-26, 1995

Disclaimer

Fermilab DART Run Control *

G. Oleynik, J. Engelfried, L. Mengel, C. Moore, V. O'dell, R. Pordes,
A. Semenchenko, D. Slimmer, L. Udumula, M. Votava
Fermi National Accelerator Laboratory, PO Box 500, Batavia, Il 60510, Tel: 708-840-2430

F. Prelz
INFN and Dipartimento di Fisica dell'Universit`a and INFN - Milano, I-20133 Milan, Italy

E. Van Drunen, G. Zioulas
University of California at Irvine, California 92717, U.S.A.

Abstract

DART is the high speed, Unix based data acquisition system being developed by Fermilab in collaboration with seven High Energy Physics Experiments [1 - 6]. This paper describes DART run control, which has been developed over the past year and is a flexible, distributed, extensible system for the control and monitoring of the data acquisition systems. We discuss the unique and interesting concepts of the run control and some of our experiences in developing it. We also give a brief update and status of the whole DART system.

I. INTRODUCTION

DART run control is a fully distributed system designed to be easily tailored and extended by individual experiments for their data acquisition (DA) needs [7, 8]. Though we provide for a high degree of customization, we also provide "turn-key" components suited to the needs of most experiments.

DART run control software provides the following functionality:

- Starting up DA applications in a distributed environment
- Support for storing and invoking multiple DA system configurations, e.g. normal running, photon calibration running
- Cleanly shutting down and restarting the DA
- Dynamically providing configuration parameters to the DA applications
- Controlling the DA applications via a single program with a command-line or Graphical User Interface (GUI)
- Graphical monitoring of the various DA attributes
- Logging a permanent history of relevant DA parameters
- Verifying that prerequisite applications are running & functioning

Control and configuration interfaces are required to be uniform and must operate on components in a distributed environment. Since DART experiments have groups of replicated intelligent components (e.g. filter processors, readout controllers), the operational portion of run control needs to deal with them efficiently and in a network transparent manner. This means one should be able to program without referring to node names. We also require run control to be highly portable so that tomorrow's favored computer can be easily incorporated.

DART run control must support many experiments with differing architectures, and experimenters must be able to easily integrate their specific applications with it. At the same time, it must provide turn-key functionality wherever possible. For these reasons, run control must be highly tailorable and extensible. In addition, in order for experiments to commission their detector components and integrate and commission their DA early on, we provide incrementally increasing levels of functionality in our software over time.

These requirements lead to trade-offs between providing a turn-key system versus a highly tailorable one, and providing a highly integrated system versus a "bottom-up" toolkit. We believe we have been successful in keeping a good balance between these competing requirements.

II. DATA ACQUISITION CONTROL

For run control command communication, we developed a group multicasting server using concepts from the ISIS Distributed Toolkit [9]. In this "group" communication paradigm, applications register as a member of one or more arbitrary groups.

* This work is sponsored by DOE contract No. DE-AC02-76CH03000

Other applications send messages to any of these groups. The messages get "multicast" to all members that have joined the group, while the sender waits for replies from all group members before continuing.

We felt that the group multicasting technique mapped very well onto data acquisition control and has a number of advantages over conventional rpc-like techniques. First, it allows parallel executions of commands where appropriate, while at the same time allowing for staged execution of commands where needed by sequentially multicasting a command to appropriate groups. This is important for the proper draining of buffers in a DA system; Figure 1 illustrates how this is done for the da_stop command. The paradigm also provides network transparency; applications deal with functionally mnemonic group names rather than IP addresses or process IDs. Finally, because communication proceeds through an intermediary server, applications can dynamically join or depart from groups, and are more loosely coupled to the system, which makes it more robust.

The commands that are multicast are formatted as TCL [10] verbs, which are basically text strings. We use the TCL interpreter to dispatch them in the receiving applications, which invoke a state change or execute a maintenance function. While communication is ultimately based on BSD socket connections to the server, we have an option to distribute commands locally on a node with our buffer manager software [11]. This allows applications to simultaneously wait for run control commands and data buffers.

Because commands are sent to applications in the proper sequence by multicasting them to groups in the appropriate chronological order, complicated techniques such as assigning a per application priority to a particular state transition are not necessary. DART uses a global state-diagram, though the paradigm only requires a state changes be consistent across a group.

We have based our operator control program (**ocp**) [12] on the TCL interpreter for command line processing, and TK [10] for graphical control. We chose TCL because of its extensible interpretive procedures. For graphics, we chose TK because it is well integrated with TCL, is extensible, and our experience has been that interfaces can be built more quickly with TK than from X and its toolkit or Motif. We provide TCL interpreter bindings for all of our run control communication and information functions. All run control commands are implemented as TCL procedures which invoke these bindings to multicast commands or fetch parameters. Since they are procedure based, they can be easily modified to add in a multicast to a new group or add a new command. The operator control program itself is merely an engine which reads in a file defining the TCL procedures, then loops executing them on user input.

We make the interface highly configurable and extensible in a number of other ways:

1. providing higher level procedures to bind DA commands to TK buttons and place them in the window
2. providing procedures to display parameters for modification
3. providing a tailorable start-up script that defines all of these procedures, plus standard command aliases and option flags.

It is very easy to add a new command procedure and bind it to a button, or extend an existing procedure without any compilation. These features make the operator control very flexible but suitable for most experiments as-is.

The **ocp** GUI can be customized in appearance and layout, as well as in run control functionality. We sat down with a couple of experiments to customize the layout of the graphical interface for them. This took on the order of 1/2-1 hour with resultant happy experimenters! We feel this is a big success of the TCL/TK approach.

The figure below, which is a code fragment of the da_stop TCL procedure taken from the **ocp** start-up file, illustrates how well the group multicasting concept maps onto DA control, as well as illustrating how simple it is to tailor or add a command. The da_stop procedure loops, multicasting the stop command sequentially to the groups listed in the da_stopList in the order of their listing, assuring that proper draining of buffers occurs. If a send to any group in the list fails, an abort is sent to all groups preceding it in the list. As is quite apparent, different groups can be added to the list, or existing ones removed.

```
set da_startList {logger filter gateway
   front-end trigger-manager}

set da_stopList {trigger-manager front-
   end gateway filter logger}

foreach daGroup  $da_stopList {
       lappend da_abortList $daGroup
       drc_cmd_send $daGroup "da_stop"
           sendStatus
       if { [ MUR_MSG_FAIL $sendStatus ]
   } {
            ocp_error_Handler $sendStatus
            mur_sIet_status $sendStatus
             reset_abortlist
             return
       }
}
```

Figure 1: Code fragment of OCP da_stop TCL procedure

III. DATA ACQUISITION INFORMATION SERVICES

*A. Introduction - **dis** and murmur*

We developed a distributed information services system, called **dis** [13], to provide the distributed framework needed

in three data acquisition areas: dynamically providing parameters to the various DA applications, recording a run by run history, and providing a repository for DA rates and statistics used by monitoring display programs (Figure 2). Information is stored and served by key-names, and the keyed "database" can be located on disk or in-memory. The latter is used for storage of transitory monitoring parameters.

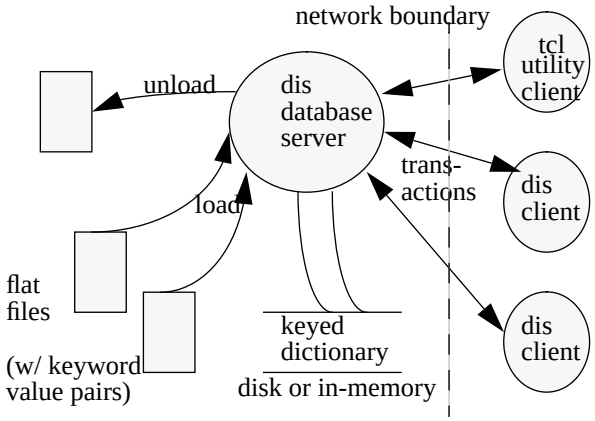

Figure 2: Distributed information services, **dis**

Parameters and their keys can be loaded/unloaded from/to flat ASCII files. Wildcarded (regular expression) loads or unloads are also possible. A variety of parameter types are supported, and atomic operations, such as addition, can be performed on them. In addition, the key name-space can be split up into node-specific or global name-spaces, and the server supports automatic searches across these spaces.These features, combined with an hierarchical name-space similar to the Unix directory structure, provide a powerful information organization tool.

Each of the three DA application areas - configuration parameters, run history, and monitoring - use a separate **dis** server that manages a separate database. For discussion purposes, they are given names to distinguish them: **disc**, **dish**, and **damp**, and each will be discussed in turn below. All use the **dis** interface to access their servers.

Another tool which is used for monitoring the DA is the distributed error messaging and display system, **murmur** [14, 15]. All applications, including run control, signal their error messages to **murmur**. **murmur** keeps a time-ordered log of such information, displays the messages on X windows, and can optionally run scripts based on the message type or content.

*B. Configuration Parameters Services - **disc***

We call the **dis** server and database that serves parameters to DA applications "**disc**". Each DA software product maintains a template **disc** file that contains the default values for parameters which it uses. This file is customized by the user and loaded into their **disc** database. The associated application connects to the **disc** server and fetches parameters when appropriate; we follow the convention that parameters can be read when an application first starts, and are usually reread only upon receiving the da_initialize run control command.

*C. Run History Services - **dish***

We call the **dis** server and database that records permanent run-by-run information from the various DA applications "**dish**". Parameters stored in **dish** are per run counters such as number of events read out, number accepted, number logged, calibration files used, etc. The current run number is also stored in the **dish** database.

*D. Monitoring Services and Display - **damp***

We call the **dis** server and database that records monitoring rates and statistics from the various DA applications "**damp**". Information such as the current event rate, logging rate, acceptance ratio, etc., are stored in **damp**.

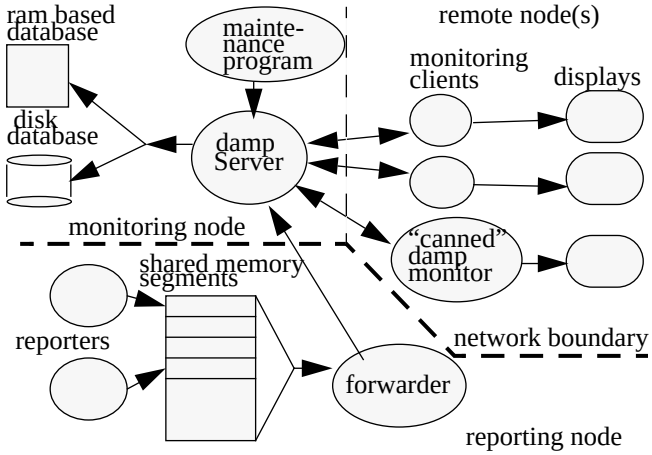

Figure 3: **damp** DA monitoring system

damp is different from **disc** and **dish** in two ways. First, the recorded information does not require permanent storage, so it is kept in an "in-memory" database. This provides for quicker access with no disk operations. Secondly, **disc** and **dish** are usually accessed only when data is not being acquired, while applications using **damp** need to write statistics periodically while data is being taken. If the applications were directly connected to the **damp** server, they would be susceptible to hangs from server crashes. For this reason, applications using **damp** are decoupled from the server using shared memory (see above). They dump their statistics into shared memory, and a special forwarder application, that runs on each node, periodically wakes up and sends the memory segment to the **damp** server. Applications dynamically define parameter names for their statistics, allocate space for them, then allocate a segment of shared memory to store them. Applications also can have direct **dis** socket connections to the **damp** server.

The statistics in the **damp** server are accessed by display programs through normal **dis** fetches. Wildcarded **dis** fetches are important for this application as they attain a much higher level of throughput to display programs. We supply a template display program that displays statistics from the tape logger and event gathering applications. The display uses TK and consists of strip-chart widgets built out of **blt** [16] to display running histories, and **blt** bar chart and label widgets to display instantaneous values.

IV. DATA ACQUISITION START-UP AND ORGANIZATION

We provide an rlogin session multiplexor, called **dbs** [17], which is used to start up applications on the DA network from a single program while capturing any terminal output they produce into a logfile or Xterm display. This program can be driven from a single script containing the information required to start up all DA applications. It has such features as sending shell commands to wildcarded sessions so they are simultaneously sent to matching sessions (not to be confused with the run control command multicasting). Commands can be sent to any session without "attaching" to them. Optionally, the client can "attach" to a remote rlogin session as if there were no **dbs**, interactively issue commands, receive the session output, then detach again by typing an escape character. Buffered output from all sessions, along with a session identifying stamp, is dumped to a file or Xterm periodically. Also provided is a mechanism to register DA processes with a session, and killing all processes started in a given session (or all sessions) so that the DA can be cleanly restarted.

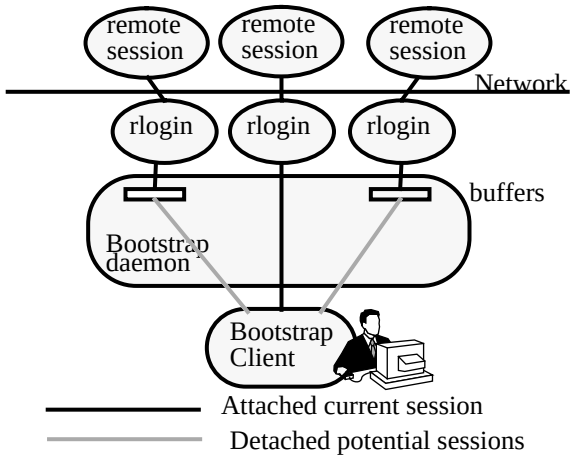

Figure 4: DART bootstrap service (**dbs**)

We provide an automated way to start up all DA applications called fresh_start. Fresh_start is a script that uses **dbs** and files located in standard directories in the DA account (Figure 5). Fresh_start starts up the servers on the host node, then reads a file of target nodes to start up, creates sessions on all of these nodes, then executes **dbs** scripts locally, and shell scripts remotely on the nodes. These scripts are fully user defined and are placed in standard directories. The directory structure permits definition of multiple DA "system" configurations, each of which can have their own target node list, host and target scripts, **disc** database, and application start-up scripts (e.g. **dbs** and **ocp**). Using fresh_start in this manner completely hides **dbs** details and focuses the experimenter on the configuration issues he is concerned with.

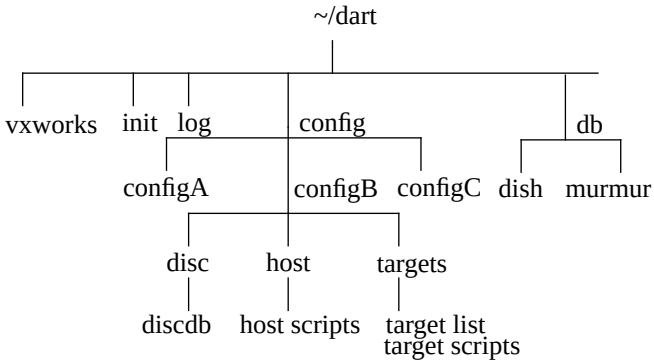

Figure 5: DA account directory layout

V. TOOLS USED AND EXPERIENCES

All of our servers are written in C++. We use the tools.h++ [18] class library extensively. It has hash-dictionary classes we use for name look-ups, file management classes we use for managing **dis** databases on disk, and, built into these classes, regular expression matching we use for **dbs** session wildcarding and **dis** wildcarded parameter fetching. Use of the tools.h++ class library significantly reduced the production time of our server software.

TCL/TK freeware proved to be a powerful combination for providing a highly extensible and configurable run control interface, enabling easy modifications and customizations with absolutely no code re-compilation required. A student was able to code the basic **ocp** API in less than a summer. We also found the freeware **blt** TK widget package a good basis to build upon for providing **damp** monitoring displays.

We did not use ISIS since we wished to develop in-house expertise and required more control of the interface, and ISIS was not available under the VxWorks [19] operating system. However, we found its multicasting communications paradigm to map very well to the requirements of run control, and the network transparency a great improvement over rpc-like implementations.

We developed our own toolkit, dart_tools, to present a more integrated interface to the user. Tools are provided which simplify programming, such as a single routine which initializes run control communications, information services, and buffer management for a process.

VI. STATUS, PLANS AND CONCLUSIONS

Efforts over the last six months have been on “shrink wrapping” the DART system into a “template” suitable for most experiments with little tailoring, providing **damp** and fresh_start, and freezing all user interfaces. The target date for completion of this effort is June. Major remaining DART work to be done includes extensions to our logger to support tape switching, providing a mechanism to verify all required DA applications are up and active, extending fresh_start to start up subsets of the DA, and simplifying the distribution and installation of the entire system. The target for this additional functionality is October 1995, about six months before first beam is scheduled to be delivered to the experiments.

Two experiments have already taken data with portions of the DART software. Other experiments that are not running yet are using DART to acquire commissioning data, have integrated run control into their applications, and are just starting to read out multiple data streams (DART is a multi-stream hardware readout architecture).

We believe we have achieved a highly flexible and extendible run control system that meets the requirements we specified and which experiments can easily integrate into their data acquisition systems. All components that we could identify as turn-key have been made so, but with a careful eye to making them easily extensible to meet specific requirements of individual experiments. Based on our observation of unexpectedly long lifetimes of previous DA systems which we have built, we designed the system for a long lifetime, using portable and ubiquitous paradigms and tools that we expect to live long. We believe this system will serve Fermilab experiments well into the future. The collaborative effort between computer professionals and experimenters was a key ingredient for this success.

VII. REFERENCES

[1] R. Pordes V. White, DART Project Definition, Fermilab Computing Division publication (CD) PN-467

[2] G. Oleynik R. Pordes, DART System Requirements, Fermilab CD PN-468

[3] G. Oleynik R. Pordes, DART System Architecture, Fermilab CD PN-469

[4] G. Oleynik et al, DART - Data acquisition for the next Generation Fermilab Fixed Target Experiments, IEEE Transactions on Nuclear Science, Vol 41, No 1.

[5] R. Pordes et al, Fermilab’s DART DA System, Proceedings of CHEP94

[6] DART documents are available at url http://www-dart.fnal.gov:8000.

[7] G. Oleynik, L. Udumula, M. Votava, DART Run Control Requirements, Fermilab CD PN-471

[8] G. Oleynik. L. Udumula, M. Votava, DART Run Control Design, Fermilab CD PN-474

[9] ISIS, Isis Distributed Systems Inc., 111 S. Cayuga Street, Ithaca, NY 14850, info@isis.com. Also see URL http://www.cs.cornell.edu/Info/Projects/ISIS/ISIS.html

[10] Tcl and the Tk Toolkit, J. Ousterhout, Addison Wesley Computing Series.

[11] D. Berg et al, Data Flow Manager for DART, Proceedings of CHEP94

[12] C. Moore et al, Operator Control Program (ocp) User’s Guide, Fermilab CD PN-497

[13] **dis** - L. Appleton (now Mengel) et al, DART Information Services Design, Fermilab CD DS-233

[14] G. Oleynik et al, Murmur - A Message Generator and Reporter for Unix, VMS, and VxWorks, IEEE Transactions on Nuclear Science, Vol 41, No 1.

[15] **murmur** is available through Fermitools, url: http://www-fermitools.fnal.gov.

[16] **blt** TK widget toolkit freeware

[17] L. Udumula et al, DART Bootstrap Services (dbs), Fermilab CD PN-483

[18] tools.h++ class library, Rogue Wave Software, Inc., 260 SW Madison, Corvallis, Oregon, 97333 USA

[19]] VxWorks is a registered trademark of Wind River Systems, Inc., 1010 Atlantic Avenue, Alameda, CA 94501-1147