Design and Implementation of a Scalable Membership Service for Supercomputer Resiliency-Aware Runtime*

Yoav Tock[1], Benjamin Mandler[1], José Moreira[2], and Terry Jones[3]

[1] IBM Haifa Research Laboratory, Haifa, Israel
[2] IBM T.J. Watson Research Center, Yorktown Heights, NY, USA
[3] Oak Ridge National Laboratory, Oak Ridge, TN, USA

Abstract. As HPC systems and applications get bigger and more complex, we are approaching an era in which resiliency and run-time elasticity concerns become paramount. We offer a building block for an alternative resiliency approach in which computations will be able to make progress while components fail, in addition to enabling a dynamic set of nodes throughout a computation lifetime. The core of our solution is a hierarchical scalable membership service providing eventual consistency semantics. An attribute replication service is used for hierarchy organization, and is exposed to external applications. Our solution is based on P2P technologies and provides resiliency and elastic runtime support at ultra large scales. Resulting middleware is general purpose while exploiting HPC platform unique features and architecture. We have implemented and tested this system on BlueGene/P with Linux, and using worst-case analysis, evaluated the service scalability as effective for up to 1M nodes.

1 Introduction

Current trends dictate increasing complexity and component counts on supercomputers and mainstream commercial systems alike [1]. This trend exposes weaknesses in the underlying clustering infrastructure needed for continuous availability, maximizing utilization, and efficient administration of such systems [2]. These issues can properly be tackled by having a resiliency supportive run-time for ensuring continuous availability and elastic run-time support for utilization maximization through proper jobs placement and load balancing. System elasticity can be an important factor also for the conservation of power which is a growing concern in the HPC world. The issue of resiliency has been identified as one of the big future challenges in the HPC world [3].

Current HPC execution environments do not provide the hosted parallel applications with a fault-free guarantee. Rather, developers need to specifically take appropriate actions in the presence of such faults (for example by using checkpoint-restart). Moreover, the most popular programming paradigm for HPC, MPI, assumes all interruptions, including single core failures, are fatal to the entire parallel application [4]. It has been identified that as systems grow, failure rates will reach a level that will render current resiliency models ineffective [5].

* This research received funding from the U.S. DoE under award No. DE-SC0002107; the European Community's FP7/2007-2013 Programme under grant agreement No. 317862; and used resources of the Oak Ridge Leadership Computing Facility at ORNL, which is supported by the U.S. DoE Office of Science under Contract No. DE-AC05-00OR22725.

F. Wolf, B. Mohr, and D. an Mey (Eds.): Euro-Par 2013, LNCS 8097, pp. 354–366, 2013.

We propose a step to alleviate these problems by providing a highly scalable clustering infrastructure for supporting a resiliency-aware elastic runtime and the hosted applications (see Fig. 1-A). The most important service a clustering infrastructure must provide is a membership service, which delivers to every node, or a select subset of the nodes, a view of the all the nodes that belong to the cluster. The membership service must detect failures of members, and support leave, join, and discovery functions. Although membership services were widely investigated, the characteristics of HPC pose unique requirements:

Ultra Large Scale: To deal with 1M nodes, we apply two principles: (1) use of a relaxed consistency model, namely eventual consistency, and (2) a hierarchical architecture.

Coupling: HPC workloads are mostly tightly coupled. That is, a single failure may cause a huge amount of other nodes to wait for recovery. Such parallel computations require *fast failure detection* in order for the failed computation to migrate to a new location with minimal disruption to the entire computation. To accommodate that we expedite failure notification to certain privileged "high priority monitors".

Churn: Current HPC workloads are characterized by static, a-priori defined groups. Even programming models that theoretically permit dynamic membership [6], are currently implemented in a way that assumes static a-priori membership. Dynamic membership is primarily needed to deal with faults, yet opens up the possibility to dynamically shrink and expand a computation. The expected churn rate in HPC is lower than in Internet or data-center settings. However, the dense integration of modern machines increases the likelihood of correlated failures, where a failure of one component (e.g. IO-node) causes cascading failures (e.g. of compute nodes). This requires a membership service that tolerates concurrent failures without significant loss of performance.

Failure Model: Recent advancements in HPC resiliency support include CIFTS and MPI3 [7], which are based on a failure model assuming fail-stop failures, no network partitions, and a perfect failure detector. Our system takes it a step further by removing these restrictions on the failure model.

In order to support resilient and elastic HPC runtime and applications we expose several services: (1) a membership service, (2) an attribute replication service, and (3) group communication services [8]. The runtime will be able to take advantage of this suite of services in order to achieve, among other aspects, adequate tasks placement, scheduling, load balancing, migration, and performance monitoring.

In this paper we concentrate on the membership and the attribute replication services, which form the foundation on which other group communication services are built. The contribution of this paper is centered around the design, implementation, and evaluation of these services. The system was evaluated to reach a higher scale than known methods (see Sec. 6), while achieving good results both for massive start-up as well as for failure detection, while assuming a general non restrictive failure model. Using worst case analysis, a full implementation of these services was evaluated as effective for up to a million nodes. Both the architecture and the membership and attribute replication services present innovations that facilitate this achievement.

In the remainder of the paper, Sec. 2 provides an overview of the system architecture, while Sec. 3 & 4 dive deeper into the major components of the system. Section 5 details

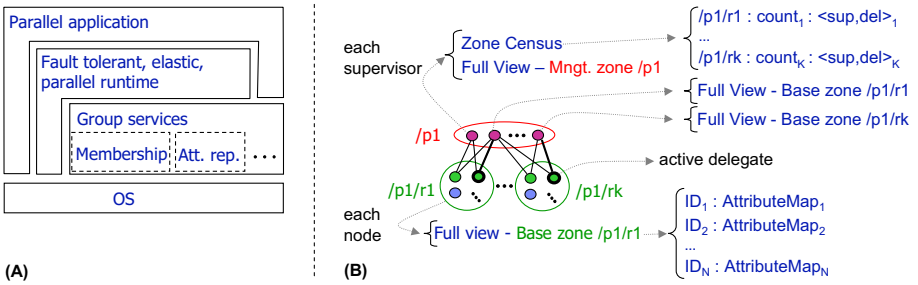

Fig. 1. (A) The software stack of a fault tolerant, elastic parallel runtime. (B) A fault tolerant hierarchy with redundant connectivity, and the data structure distribution along the two levels.

a thorough evaluation of the system, related work is in Sec. 6, and concluding remarks are presented in Sec. 7.

2 System Architecture

The requirements and characteristics of HPC systems, detailed in the previous section, led us to a hierarchical peer-to-peer design (see Fig. 1-B). The basic building block of the system is that of a zone. Within a zone, full membership is maintained with an eventual consistency semantic (detailed in Sec. 3). In addition, an attribute replication service allows each node to write a key-value table, which is then replicated to all other nodes. This allows each node to communicate its state to every other node in the zone. This design scales to zones consisting of approximately a few thousand members. In order to achieve the 1M nodes target, the membership and attribute replication services are used as building blocks of a two layers hierarchy, composed of base-zones federated by a management-zone (see Fig. 1-B). On Blue Gene/P [9] for example, zone affiliation be can derived from the "personality" of each node, organizing the compute nodes of a rack into a base-zone, selecting IO-nodes for the management zone, and assigning co-located IO-nodes to the respective base-zones. Each base zone elects, using the attribute service, delegates to connect to the management zone's appropriate supervisor. An active delegate sends the supervisor updates about the base-zone membership. A supervisor shares information with its peers, using the attribute replication service, publishing the names of guarded base-zones, the number of active delegates per such zone, and the view size of each base-zone (see Fig. 1-B). This enables each member of the management zone to have a "system census", which is an up-to-date eventually consistent summary view of the entire cluster. In addition, if further details are needed, there are mechanisms in place which enable every management node to obtain more specific information regarding each base zone, such as the detailed full membership of that zone. Since components, including delegates, supervisors, and the links between them can fail, the hierarchy contains redundancy in each of these elements. Whenever a component fails, it is immediately and automatically replaced, so that the integrity of the hierarchy is maintained at all times.

In order to ensure fast failure detection, the active delegate in each zone serves as the zone's "high priority monitor" (HPM). Failure detection information is sent to the HPM

node directly, although possibly using unreliable communication, in addition to the reliable yet relatively slow gossip-based dissemination mechanism. These notifications traverse the hierarchical structure as well. Thus, a single monitor or a hierarchy thereof can take fast actions such as re-spawning the failed computation.

3 Zone Membership

In this section we describe in detail the membership protocol implemented in both base and management zones. Each node n is associated with an *ID*, which carries the node name and network endpoints. Each *ID* is accompanied by a version $\mathit{Ver} = \langle inc, minor \rangle$, where: (1) inc identifies different incarnations of the same node, separated by crash failures or restarts, and is strictly increasing; and (2) $minor$ starts from 1 on every incarnation and is incremented by n according to the protocol described below. The local membership *View* is a map $\mathit{ID} \rightarrow \mathit{Ver}$, where $\mathit{View}_n[m]$ is the most current *Ver* of node m known to n. Changes to the view are delivered differentially: NOTIFY-JOIN($\langle p, v \rangle$) is invoked when node p with version v joins the overlay or after a network partition heals; NOTIFY-LEAVE($\langle p, v \rangle$) is invoked when $\langle p, v \rangle$ leaves, fails, or is behind a network partition. The semantic is that of eventual consistency: $\forall m, n$ that remain in the same group, $\mathit{View}_n = \mathit{View}_m$, some finite time after no more nodes join or leave, and the network is stable.

Bootstrap & History: When a node starts, it is given a set *B* of *ID*s to bootstrap from. The history map *H* contains the *ID*s of nodes which were recently removed from the *View*, along with their removal time and version when removed. If a node re-enters the view, it is removed from *H* (i.e. $H \cap \mathit{View} = \emptyset$). The history map is used to identify stale messages, that arrive after a node fails. In case the history map *H* grows beyond a certain limit, it is pruned by removing nodes older than some threshold.

Discovery: The discovery task selects target $m \in \{B \setminus \mathit{View} \cup H\}$ randomly, and sends a discovery-request message that contains its own *ID* and *Ver*, as well as a $boot$ flag that encodes whether the target m was selected from *B* or from *H*. The discovery-reply consists of the full view of the target m, and the flag $boot$. For both request and reply, in case $\neg boot$, the receiver p performs $\mathit{Ver}_p.min \leftarrow \mathit{Ver}_p.min + 1$, and then processes each pair of $\langle \mathit{ID}, \mathit{Ver} \rangle$ from the message using PROCESSALIVE($\langle \mathit{ID}, \mathit{Ver} \rangle$) (see Alg. 1). This process both heals partitions and discovers new peers. The discovery targets are not kept as permanent neighbors. The discovery task is performed frequently at bootstrap, and as time passes its frequency decreases.

Topology: As the view begins to fill up, the topology component starts choosing and connecting to long-term neighbors. The topology built from the view has two ingredients: (1) a robust ring where each node is connected to K_s successors (ring order based on SHA1 of *ID*), and (2) K_r random neighbors. The ring ensures that eventually all failed nodes will be discovered by their predecessor(s). This ensures eventual completeness of the view [10]. The random nodes are selected according to a protocol which approximates a K_r-connected random graph [11], yielding a robust and well connected overlay.

Algorithm 1. Processing of Alive, Leave, and Suspicion events, at node n

```
1:  procedure ProcessAlive(ID p,Ver v)
2:      if (p ∉ View_n) ∧ ((p ∉ H_n) ∨ (p ∈ H_n ∧ v > H_n[p].Ver)) then          ▷ join
3:          View_n[p] ← v; remove p from H_n; add ⟨p, v⟩ to Δ.A; NotifyJoin(⟨p, v⟩);
4:      if (p ∈ View_n) ∧ (v > View_n[p]) then                                  ▷ newer version
5:          add ⟨p, v⟩ to Δ.A;
6:          for all r ∈ S_n[p], S_n[p][r] < v do   ▷ prune refuted suspicions remove r from S_n[p];
7:          if View_n[p].inc = v.inc then View_n[p] ← v;
8:          else                                                                ▷ new incarnation
9:              NotifyLeave(⟨p, View_n[p]⟩); View_n[p] ← v; NotifyJoin(⟨p, v⟩);
10: procedure ProcessLeave(ID p,Ver v)
11:     if (p ∈ View_n) ∧ (v ≥ View_n[p]) then                                  ▷ in-view leave
12:         remove p from View_n and S_n; H_n[p] ← ⟨v, time⟩; add ⟨p, v⟩ to Δ.L;
13:         NotifyLeave(⟨p, v⟩);
14:     if (p ∉ View_n) ∧ (p ∈ H_n) ∧ (v > H_n[p].Ver) then                     ▷ out-of-view leave
15:         H_n[p] ← ⟨v, time⟩; add ⟨p, v⟩ to Δ.L;
16: procedure ProcessSuspicion(ID r,ID s,Ver v)
17:     if s = n then                                                           ▷ refute suspicion on self
18:         Ver_n.min ← Ver_n.min + 1; add ⟨n, Ver_n⟩ to Δ.A;
19:     else if s ∈ View_n ∧ v ≥ View_n[s] then
20:         if r ∉ S_n[s] ∨ S_n[s][r] < v then                                  ▷ valid, new suspicion
21:             S_n[s][r] ← v; add ⟨r, s, v⟩ to Δ.S;
22:             if |S_n[s]| ≥ Θ then   ▷ enough evidence! correction for small views omitted
23:                 remove s from View_n and S_n; H[s] ← ⟨v, time⟩; NotifyLeave(⟨s, v⟩);
```

Failure Detection and Orderly Leaves: Failure detection is based on neighbors monitoring each other using heartbeats. Node r creates a failure suspicion report on node s if (1) an established connection between r and s fails, or (2) a heartbeat timeout is reported on s, or (3) s is member of r's successor list, and a connect attempt from r to s fails. The last condition ensures the view's eventual completeness [10]. A suspicion report consists of the tuple $\langle r, s, v\rangle$ for the reporter's *ID*, the suspect's *ID*, and suspect's *Ver*. The reporter calls ProcessSuspicion$(r, s, View_r[s])$ in order to spread the report further (see Alg. 1). Adjacent to $View_n[m]$ is the suspicion repository 2D map $S_n[s][r] \rightarrow v$, which stores every unique suspicion report received. Note that suspicion reports with a version lower than the version of the suspect in the view are discarded. In order to decrease the false detection rate, a node is declared as "failed" only after the number of reporters suspecting the same node exceeds a threshold, Θ. To ensure eventual completeness, $K_s \geq \Theta$ must hold. When a node orderly leaves the overlay, it sends all its neighbors a leave message, which contains the node *ID* and *Ver* (and possibly an exit code). Upon reception leave messages are processed by ProcessLeave(*ID*, *Ver*) and added to the update database for further dissemination.

Membership Updates: Node membership information is disseminated over the $K_r + K_s$ long-term overlay links. When node n acquires a new neighbor m, it will send m a membership message that contains: (1) the entire $View_n$, and (2) all the current suspicions (all the tuples $\langle r, s, S_n[s][r]\rangle$). After the first "base-view message" a neighbor is

sent only "update" messages, which differentially capture the difference from the base-view. The update-database $\mathbf{\Delta}$ contains the following three sets: (1) $\mathcal{A}$ – The $\langle$*ID*,*Ver*$\rangle$ of nodes on which fresh *Alive* information was received (may include the current node); (2) $\mathcal{S}$ – received suspicion reports $\langle$r,s,v$\rangle$; (3) $\mathcal{L}$ – The $\langle$*ID*,*Ver*$\rangle$ of received *Leave* messages. The update database starts empty and accumulates alive, leave, and suspicion events (see Alg. 1). A membership update message is sent to all neighbors after a configurable aggregation interval (τ) from the first such event that hits an empty $\mathbf{\Delta}$. After $\mathbf{\Delta}$ is sent to all the neighbors, it is cleared. Upon receiving a membership message (base or update), every $\langle p, v\rangle \in \mathcal{L}$, every $\langle p, v\rangle \in \mathcal{A}$, and every $\langle r, s, v\rangle \in \mathcal{S}$ is processed by Alg. 1 #10,#1,#16, respectively. This order minimizes the chance of notifying a false suspicion.

High Priority Monitoring: In the protocol described above, membership updates are expected to propagate to all nodes in time proportional to τ and the overlay diameter, which is $O(\log_{K_r} N)$ [11]. In many applications there is only a single or a small number of monitors that take decisions based on membership events. It is possible to decrease the monitor's failure detection time by sending the original suspicion reports directly to the monitor (e.g. using UDP), in addition to the reliable propagation mechanism described in Alg. 1. We allow a small number of selected nodes to declare themselves (automatically or programmatically) as monitors using the attribute replication service. This lets every node in the zone know who the monitors are. When node n suspects the failure of a neighbor m with version v, it will immediately send the monitors a membership message containing suspicion $\langle n, m, v\rangle$. This message is processed just like any other suspicion (see Alg. 1). A monitor will receive up to $K_r + K_s$ such messages on every failure.

4 Attribute Replication Service

Each node n has an attribute map A_n of key-value pairs it can write to. Each key-value pair $\langle$k,t$\rangle$ is associated with a version number $u \in \mathbb{N}$, such that newer values of the same key carry larger version number. The goal is to replicate the attribute map A_n of node n to all other nodes. Let us denote by $M_n(m)$ the map replica of node m, that node n holds. Thus, node n holds one map it can write to directly $M_n(n) \equiv A_n$, and read-only replicas of the maps of every other node $M_n(m),\ \forall m \neq n$. Thus, the ultimate goal is to reach $M_n = M_m, \forall n, m$ (some finite time after the end of writes). When node n learns about the attribute changes of node p, it notifies the user by invoking NOTIFY-ATTCHANGE($M_n(p)$). The attribute dissemination protocol is inspired by Anti-Entropy (AE) protocols [12,13], where in every round a node reconciles its state with a randomly selected gossip peer. The disadvantage is that on every AE round, the entire membership of node n ($O(|View_n|)$) has to be transmitted. This step has to be repeated periodically even if no updates to the map were made, consuming bandwidth even in idle state. Moreover, running AE reconciliation with two peers in parallel carries the cost of potentially getting duplicate copies of the same data entries. In our topology each node has a stable set of neighbors, connected by reliable connections. Thus, remembering what was exchanged in the last round and transmitting just the difference can save a lot of bandwidth. We therefore designed an improved protocol, which: (1)

Algorithm 2. Attribute replication message handlers: received at n, sent from m

```
1:  procedure UPON-ATTUPDATE(Λ update)
2:      Λ_n ← ∅
3:      for all ⟨p, u⟩ ∈ update do
4:          if u > M_n(p).u ∧ u > M_n(p).u_pend then
5:              Λ_n ← Λ_n ∪ ⟨p, M_n(p).u⟩; M_n(p).⟨u_pend, trg_pend⟩ ← ⟨u, m⟩;
6:      if Λ_n ≠ ∅ then send AttRequest(Λ_n, false) to m;
7:  procedure UPON-ATTREQUEST(Λ request, push)
8:      AttData_n ← ∅;
9:      for all ⟨p, u⟩ ∈ request do
10:         if u < M_n(p).u then
11:             for all ⟨k, t, u'⟩ ∈ M_n(p) : u' > u do AttData_n ← AttData_n ∪ ⟨p, k, t, u'⟩;
12:         else if ¬push then AttData_n ← AttData_n ∪ ⟨p, ⊥, ⊥, ⊥⟩;
13:     send AttReply(AttData_n) to m
14: procedure UPON-ATTREPLY(AttData data)
15:     Λ_n ← ∅;
16:     for all d_m(p, k, t, u) ∈ data do
17:         if u = ⊥ then
18:             Λ_n ← Λ_n ∪ ⟨p, M_n(p).u⟩;
19:             M_n(p).⟨u_pend, trg_pend⟩ ← ⟨M_n(p).u, ∅⟩;
20:         else if M_n(p)[k] = ∅ ∨ M_n(p)[k].u < u then
21:             M_n(p)[k] ← ⟨t, u⟩; M_n(p).u ← max(M_n(p).u, u);
22:     if Λ_n ≠ ∅ then send AttRequest(Λ_n, true) to all neighbors;
23:     for all M_n(p).u > M_n(p).u_notified do
24:         NOTIFY-ATTCHANGE(M_n(p)); M_n(p).u_notified ← M_n(p).u;
```

avoids sending full $O(|\mathit{View}_n|)$ sized messages in each reconciliation; (2) lets traffic reduce to zero when there are no writes and the overlay is stable; and (3) minimizes the reception of duplicate data updates.

An entry in the map is $A_n[k] = \langle t, u\rangle$. The map itself has a version number $A_n.u \in \mathbb{N}$ which starts at 0 when the map is empty. The map is written one key at a time. Every time a key is written the version number of the map is incremented, and the version of the corresponding $\langle k, t\rangle$ entry is set to $A_n.u$. Thus no two entries carry the same version, and $A_n.u$ equals the maximal version number in the map. This versioning scheme permits a protocol with *per source sequential consistency*, meaning that writes to A_n are delivered in the same order in every other node p. The tables $M_n(p)$ are augmented with the following fields:

1. $M_n(p).u = \max\{u : \langle k, t, u\rangle \in M_n(p)\}$ – the last version of A_p known to n;
2. $M_n(p).u\_sent$ – the version sent by n to all its neighbors during the last round;
3. $M_n(p).pend\_trg$ – the neighbor *ID* to which a request was sent;
4. $M_n(p).u\_pend$ – a version known to $M_n(p).pend\_trg$ to which a request was sent;
5. $M_n(p).u\_notified$ - the last version delivered to the application.

In every round, node n will reconcile its state with its overlay neighbors, so that eventually $M_n = M_m$ for every neighbor m. Let a digest Λ_n be a list of identifier and table

version pairs $\langle p, M_n(p).u\rangle$; and let *AttData*$_n$ be list data entries $d_n(p,k,t,u)$, where each entry is a single key-value-version tuple from $M_n(p)$. The reconciliation protocol has three stages – *Update*, *Request*, and *Reply*. (1) In the *Update* stage, node n will prepare, at configurable periodic intervals, a differential digest of its state, containing tables that changed since the last round: $\Lambda_n \leftarrow \{\langle p, M_n(p).u\rangle : M_n(p).u > M_n(p).u\_sent\}$. If $\Lambda_n \neq \emptyset$, it will be sent to all n's neighbors, and the tables will be marked as sent: $\forall\langle p,u\rangle \in \Lambda_n, M_n(p).u\_sent \leftarrow u$. (2) In the *Request* stage (Alg. 2 #1), after processing an update digest from m, node n sends m a request containing a digest of its parts of the state that are less recent than those of m. The validity of the request w.r.t. the preceding update is given special care (Alg. 2 #12,17-19). (3) In the *Reply* stage (Alg. 2 #7), node n sends to node m the data entries that are more recent than what node m declared it knows and needs. (4) Finally, when node n receives the reply (Alg. 2 #14), it merges the incoming data into the existing tables, and notifies the application on the respective attribute changes.

When node n acquires a new neighbor m, the full digest $\Lambda_n \leftarrow \{\langle p, M_n(p).u\rangle, \forall p \in$ *View*$_n\}$ is sent to m. In case a neighbor m disconnects (fails, leaves, or changes neighbors), its *ID* is searched in all the $M_n(p).pend\_trg$ fields. If found, it means that a request sent to it will not be answered. Thus, the pending request (i.e. $\langle p, M_n(p).u\rangle$) will be resent to all neighbors. The attribute map of a node is valid to other nodes only when the node is "alive". Thus, when a node p leaves the overlay, all $M_n(p), \forall n \neq p$, are deleted. When a node joins the overlay, an empty replica is initialized in all other nodes. As it acquires new neighbors, the joining node will push its state digest to its neighbors, and vice versa. Key deletion is translated into a write $A_n[k] \leftarrow \langle \perp, A_n.u + 1\rangle$, which is a kind of "death-certificate" for the key (see [12]).

5 Evaluation

Developing hardware and software for much larger systems than presently available has always posed difficult challenges for those who must assess performance before full scale measurements are possible. Our design lends itself to encapsulated component performance testing even though supercomputers with one million nodes do not exist yet. We divide our entire system into three components: management zone, base zones, and the communication links between them (see Fig. 1-B). We isolated the required relevant performance metrics for each component. Then we are able to devise tests for each component at the required performance to achieve successful systems of one million nodes. We fully implemented the hierarchical membership and attribute services in C++. Our test bed was a rack of Blue Gene/P[1]. We used regular Linux on compute nodes, rather than CNK[2]. The version we used provides full TCP/IP functionality on all the networks, including the torus. We set out to test whether our system is up to the task of managing the target scale by first testing a single zone to its full scale. Then we test a hierarchical system that has the full number of base zones and management nodes although with "stub" base zones. Each "stub" base zone is represented by a single node, which injects in to the management zone the same traffic as a full base-zone. We used

[1] Which includes 1024/64 compute/IO nodes, 32 bit integer, 850MHz, 4GB RAM [9].

[2] Version 2.6.29.1 with IBM modifications for Blue Gene/P, available: `http://git.anl-external.org/bg-linux.repos/linux-2.6.29.1-BGP.git/`

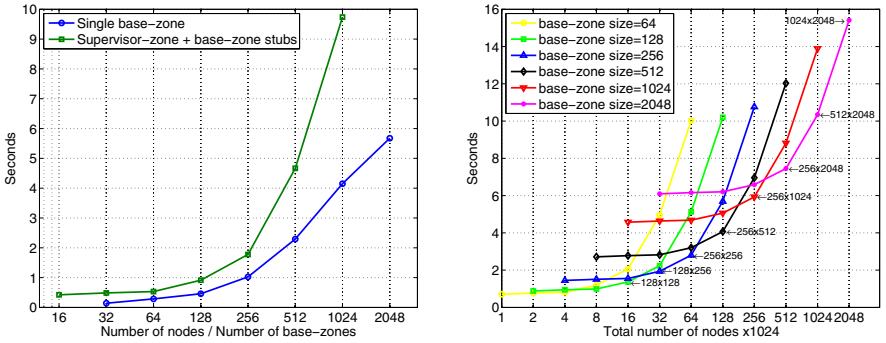

Fig. 2. Left: Boot time of (1) a single base zone with a variable number of nodes, (2) a management zone with a variable number of small base zones, one supervisor per base zone. **Right**: Projected boot time of a complete system as a function of total size and base zone size. Arrows indicate the optimal configuration for each size.

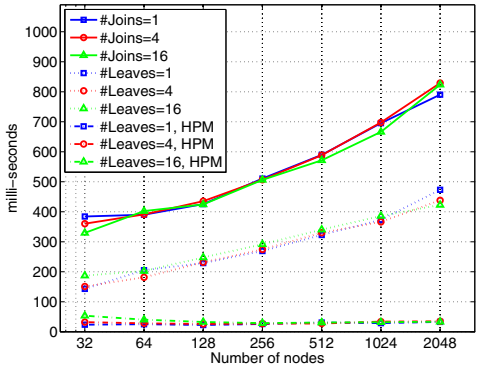

Fig. 3. Average time for a view with nodes joining and leaving, as a function of zone size and number of concurrent joins/leaves

$\tau = 200$ms, $\Theta = 1$, $K_r = 3$, $K_s = 1$ in all the experiments. The metrics chosen for measuring our system's performance are biased towards the HPC use case. The main difference between this use case and traditional (Internet and data-center) scenarios is the way in which the system is brought up, the frequency of nodes joining and leaving (churn), and the coupling between the nodes.

5.1 Boot Time

Unlike conventional Internet-scale or data-center based systems in which the nodes gradually join until the system gains size, a supercomputer or a partition of a supercomputer, usually boots all its nodes at once. Thus we want to make sure that the time it takes to form a stable view upon boot is reasonable, in line with the time it takes for the other processes that happen during boot (daemon startup, file system mount, etc).

First, we measure the time to a stable view of a single base zone, versus the number of nodes (32-2048), assuming all the nodes boot at once (Fig. 2-Left). Results indicate that a 2048-node zone yields a stable view in $T^{Boot}_{Base}(n = 2048) \approx 5.7$s (we used 2 processes per machine, with 1 process per machine results are approximately $\approx 30\%$ better). This tests the performance of the membership protocol. The results indicate that the boot time for a single zone is linear in the number of nodes. This is to be expected because every node has to get at least a single "alive" indication from every other node, directly or indirectly. Next, we measure the time to a stable view of a 2-layer setup with the number of base zones increasing from 16 to 1024, with 1 nodes in each base zone serving as a stub, and 1 supervisor per base zone (Fig. 2-Left). Results indicate that a 1024-node supervisor zone with 1024-base zones (1 stub node) boots in $T^{Boot}_{Sup}(m = 1024, n = 1) \approx 9.7$s. The boot stabilization time of the management zone includes: (1) stabilization of the management zone membership view and topology, (2) connecting with the base zone delegates, (3) receiving the views of the base-zones, and (4) distributing the summaries to all the members of the management zone using the attribute service. This takes longer than the stabilization time of a base zone with the same number of nodes, and shows linear scaling as well. These two measurements allow us to make a worst-case estimate of the full system stabilization time, for different combinations of management zone size (m) and base zone size (n), by making the following worst case assumptions: (1) that the stabilization time of a stub base zone is negligible, and (2) that the management zone starts after the base zones have stabilized. This results in $T^{Boot}_{Sup}(m, n) \lessapprox T^{Boot}_{Sup}(m, 1) + T^{Boot}_{Base}(n)$. Figure 2-Right shows that a 1M system with 2048-node base-zones and a 512-node management zone would boot in $\approx T^{Boot}_{Sup}(512, 1) + T^{Boot}_{Base}(2048) \sim 10.3$s. Figure 2-Right also shows what would be the optimal configuration of a management- and base-zones, for every system size, in terms of boot time.

5.2 Leave-Join Performance

We evaluate the leave-join performance by first booting a zone, and then forcing a number of nodes to fail concurrently. The victim nodes concurrently rejoin after a while. In both cases we measure the average time to a stable view; in case of leaves this includes failure detection time. The time to stable view (Fig. 3, top 6 curves) measures the propagation time of concurrent membership changes using the normal membership and attribute dissemination protocol. Membership stabilization time follows a logarithmic relation with zone size, since our topology creates a graph with logarithmic diameter and average path length. (We verified that $\forall N$ the diameter is $\leq \log_{K_r} N$). The number of nodes leaving or joining has hardly any effect, since as long as the ratio of transient nodes to total zone size is not too high, the topology retains its desirable robust "logarithmic" features (diameter, average path length) in the face of churn [11]. Join events take exactly 1 aggregation delay (τ) longer than leaves, since the joining node takes one round to discover peers before it spreads its identity, in an attempt to build a "good" topology right away. Membership events are propagated as node-census attribute events in the supervisor zone. The propagation time follows the same rule as in base zones, since it is influenced by the (identical) topology of the overlay and the aggregation time. Therefore we can estimate the stabilization time of a full system, as

in Sec. 5.1, by adding the stabilization times of the base- and supervisor-zones, according to their respective sizes. For example, in a 1M node system with a 512/2048 supervisor/base-zone configuration, a leave event would propagate to all the supervisors in ≈700ms.

5.3 High Priority Monitoring

Figure 3 (bottom 3 curves) displays the time it takes an HPM node to reach a stable view after leave events (same experiment as above). The delay is almost independent of zone size, and is ≈30ms for almost all cases. The round-trip delay between any node in Blue Gene/P is below 1ms; we therefore conclude that this time is mainly failure detection time, which is independent of size. When the ratio of failed nodes to the zones size is too large (e.g. 16 out of 32), detection time grows because of the likelihood that some failed node X would have all its neighbors failing as well. The failure of node X is then discovered by some surviving node that tries to connect to it as a successor, and fails.

6 Related Work

Membership services present a wide spectrum of semantics [14], which vary from consistent views like Virtual Synchrony [15], to eventual consistency [16] as implemented in Cassandra [17], and to partial views either randomized as in SCAMP [18] or structured as in Chord [19]. Scalability increases as semantics become less strict, from hundreds in VS, thousands in eventual consistency, and tens of thousands for partial views. An important class of these services relies on "gossip" protocols [20], where peers periodically exchange parts of their state with a random selection of peers. Gossip based protocols are extremely robust. However, they are slow to detect failures [10], and generate traffic even when no membership changes occur. Overlay networks in which peers have stable connections retain similar robustness by choosing peers such that the resulting overlay network remains well connected in the face of failures, as is Araneola [11] and Symphony [21]. The advantages of stable peers are (1) efficient distributed failure detection [10,22]; (2) the ability to minimize "OS jitter" [23]; and (3) the ability to implement additional functions like a publish-subscribe service [8], and a key-value store [19,17]. Hierarchical membership schemes were proposed by HiSCAMP [24], where the focus is on a dynamic self organizing hierarchy, and more recently by Census [25] where the focus is on self-organization reflecting the geographic distribution of the nodes, and on delivering consistent views. Our implementation focuses on scalability and is an order of magnitude greater than Census and HiScamp (1M vs. 10K and 50K, resp.); and fast failure detection, which at 1M nodes is less than 100ms for the HPM and takes around 800ms to form a consistent view (Census [25] chooses to provide a consistent view every 30 seconds for a system of 10K nodes). An early attempt to develop a membership service specifically for HPC introduced a flat tree-based membership algorithm for MPI environments, and was evaluated only up to 1024 nodes [4]. More recently the CIFTS project (e.g [7]) was devoted to fault-tolerance in HPC systems. Our approach adopts a much more general failure model than the one adopted by CIFTS, and therefore uses a different overlay topology (expander vs. tree) and different algorithms.

7 Conclusions

We demonstrated that membership services can scale effectively to upcoming HPC system sizes, supporting continuous availability, for a next generation of HPC run-time support, and system administration. We believe that ExaScale size deployments can be made resiliency aware by employing this work. We are currently working to integrate our membership and attribute replication services with Charm++ [6], in order to demonstrate a true fault tolerant, elastic, parallel runtime.

References

1. Kogge, P.M., Dysart, T.J.: Using the TOP500 to trace and project technology and architecture trends. In: SC 2011(2011)
2. Sarkar, V., et al.: ExaScale Software Study: Software Challenges in Extreme Scale Systems. Technical report, DARPA IPTO, AFRL (September 2009)
3. Cappello, F., Geist, A., Gropp, B., Kale, L., Kramer, B., Snir, M.: Toward Exascale Resilience. Int. J. High Perform. Comput. Appl. 23(4), 374–388 (2009)
4. Varma, J., Wang, C., Mueller, F., Engelmann, C., Scott, S.L.: Scalable, fault tolerant membership for MPI tasks on HPC systems. In: ICS 2006 (2006)
5. Ferreira, K., Stearley, J., Laros, J.H., Oldfield, R., Pedretti, K., Brightwell, R., Riesen, R., Bridges, P.G., Arnold, D.: Evaluating the viability of process replication reliability for exascale systems. In: SC 2011 (2011)
6. Charm++ website (2012), `http://charm.cs.uiuc.edu/`
7. Buntinas, D.: Scalable distributed consensus to support MPI fault tolerance. In: Cotronis, Y., Danalis, A., Nikolopoulos, D.S., Dongarra, J. (eds.) EuroMPI 2011. LNCS, vol. 6960, pp. 325–328. Springer, Heidelberg (2011)
8. Tock, Y., Mandler, B., Moreira, J., Jones, T.: Poster: scalable infrastructure to support supercomputer resiliency-aware applications and load balancing. In: SC 2011 Companion, pp. 9–10 (2011)
9. IBM-Blue-Gene-Team: Overview of the IBM Blue Gene/P project. IBM J. of Res. and Dev. 52(1/2), 199–220 (2008)
10. Gupta, I., Chandra, T.D., Goldszmidt, G.S.: On scalable and efficient distributed failure detectors. In: PODC 2001, pp. 170–179 (2001)
11. Melamed, R., Keidar, I.: Araneola: a scalable reliable multicast system for dynamic environments. In: NCA 2004, pp. 5–14 (2004)
12. Petersen, K., Spreitzer, M.J., Terry, D.B., Theimer, M.M., Demers, A.J.: Flexible Update Propagation for Weakly Consistent Replication. In: SOSP 1997, pp. 288–301 (1997)
13. van Renesse, R., Dumitriu, D., Gough, V., Thomas, C.: Efficient reconciliation and flow control for anti-entropy protocols. In: LADIS 2008 (2008)
14. Chockler, G., Keidar, I., Vitenberg, R.: Group communication specifications: a comprehensive study. ACM Computing Surveys 33(4), 427–469 (2001)
15. Birman, K., Joseph, T.: Exploiting virtual synchrony in distributed systems. In: SOSP 1987 (1987)
16. van Renesse, R., Minsky, Y., Hayden, M.: A gossip-style failure detection service. In: Middleware 1998, pp. 55–70 (1998)
17. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44(2), 35–40 (2010)
18. Ganesh, A., Kermarrec, A.M., Massoulie, L.: Peer-to-Peer Membership Management for Gossip-Based Protocols. IEEE Trans. Comput. 52(2), 139–149 (2003)

19. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: SIGCOMM 2001, pp. 149–160 (2001)
20. Kermarrec, A.M., van Steen, M.: Gossiping in distributed systems. SIGOPS Oper. Syst. Rev. 41(5), 2–7 (2007)
21. Manku, G.S., Bawa, M., Raghavan, P.: Symphony: distributed hashing in a small world. In: USITS 2003: Proc. USENIX Symp. on Internet Tech. and Sys., vol. 4, pp. 10–23 (2003)
22. Zhuang, S.Q., Geels, D., Stoica, I., Katz, R.H.: On failure detection algorithms in overlay networks. In: INFOCOM 2005, vol. 3, pp. 2112–2123 (2005)
23. Jones, T.: Linux Kernel Co-Scheduling and Bulk Synchronous Parallelism. The Int'l J. of High Performance Computing Applications 26(2), 136–145 (2012)
24. Ganesh, A.J., Kermarrec, A.M., Massoulié, L.: HiScamp: self-organizing hierarchical membership protocol. In: SIGOPS 2002 European Workshop, pp. 133–139 (2002)
25. Cowling, J., Ports, D.R.K., Liskov, B., Popa, R.A., Gaikwad, A.: Census: Location-Aware Membership Management for Large-Scale Distributed Systems. In: USENIX (2009)