Scalable, Fault-Tolerant Membership for MPI Tasks on HPC Systems [*]

Jyothish Varma[1], Chao Wang[1], Frank Mueller[1], Christian Engelmann[2], Stephen L. Scott[2]

[1] Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7534

[2] Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN 37831-6016

e-mail: mueller@cs.ncsu.edu

ABSTRACT

Reliability is increasingly becoming a challenge for high-performance computing (HPC) systems with thousands of nodes, such as IBM's Blue Gene/L. A shorter mean-time-to-failure can be addressed by adding fault tolerance to reconfigure working nodes to ensure that communication and computation can progress. However, existing approaches fall short in providing scalability and small reconfiguration overhead within the fault-tolerant layer.

This paper contributes a scalable approach to reconfigure the communication infrastructure after node failures. We propose a decentralized (peer-to-peer) protocol that maintains a consistent view of active nodes in the presence of faults. Our protocol shows response times in the order of hundreds of microseconds and single-digit milliseconds for reconfiguration using MPI over BlueGene/L and TCP over Gigabit, respectively. The protocol can be adapted to match the network topology to further increase performance. We also verify experimental results against a performance model, which demonstrates the scalability of the approach. Hence, the membership service is suitable for deployment in the communication layer of MPI runtime systems, and we have integrated an early version into LAM/MPI.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; D.4.8 [**Operating Systems**]: Performance—*Measurements,Modeling and prediction*

General Terms

Measurement, Performance, Reliability

Keywords

Reliability, high-performance computing, node failure, message passing, group communication, scalability

1. INTRODUCTION

As contemporary high-performance computing (HPC) systems are increasing in size to thousands of processors, such as IBM's BlueGene/L (BG/L), high availability is becoming a challenge [4]. While the reliability of a single node is often remarkably high, a job's chance to complete execution prior to *any* failures decreases as the number of nodes to parallelize the job increases. For the BG/L at Livermore with 130k processors, a dual-processor compute card is currently failing every other day forcing a 1024-processor midplane to be temporarily shut down to replace the card [29].

In such large-scale environments, high-performance applications commonly employ a checkpoint-and-restart methodology to tolerate failures. When a node fails, the current job is generally relinquished in favor of a new job whose nodes restart from the last checkpoint saved on stable storage [41, 11, 12, 3]. Such application-side fault tolerance imposes the burden on the programmer to explicitly and non-portably address the robustness of the code for large HPC systems.

Different approaches to provide programmers with support for fault tolerance have been studied in the context of high-performance systems, ranging from application-level [25] over communication-level [40, 39, 22, 10] to network-level [6]. While application-side techniques require significant modifications to programs, they potentially reduce the amount of state that needs to be saved. Techniques at the network layer provide reliability within the message layer and need to be complemented by additional techniques at higher abstraction levels. An implementation at the communication layer provides a compromise in that modifications to the application are minimal and application state can be captured adequately. Our work is aimed at an integration into the communication layer of MPI runtime systems, specifically that of LAM/MPI and Open MPI [40, 27].

Efforts in group communication have focused on providing services to a dynamically growing and shrinking set of members (or nodes) [26, 5, 30, 34, 8]. These services are often Internet services using high-level communication abstractions. Implementations range from client-server approaches to the peer-to-peer paradigm with hybrids of both in the middle. These approaches generally utilize an all-to-all communication paradigm, which is inherently un-

[*] The work at NC State University was supported in part by DOE grant DE-FG02-05ER25664 and NSF grants CAREER CCR-0237570 and CCF-0429653. The research at ORNL was supported by the Mathematics, Information and Computational Sciences Office, Office of Advanced Scientific Computing Research, Office of Science, U. S. Department of Energy, under contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

ICS06, June 28-30, Cairns, Queensland, Australia.
Copyright ©2006 ACM 1-59593-282-8/06/0006...$5.00

scalable. More recent work on group membership proposes fully decentralized or hybrid approaches, but the resulting restructuring overhead is still in the order of seconds [33, 46].

In this work, we contribute a scalable approach to reconfigure the communication infrastructure after node failures within the runtime system of the communication layer. Building on our past experience of scalable communication frameworks [19, 20, 21, 24], we propose a decentralized (peer-to-peer) protocol that maintains membership of MPI tasks in the presence of faults. Our protocol is primarily tailored to local area networks, specifically dedicated clusters, instead of wide area networks or Grid frameworks.

However, while existing approaches provide either scalability or small reconfiguration overhead, our protocol combines these features. Instead of seconds for reconfiguration, our protocol shows overheads in the order of hundreds of microseconds and single-digit milliseconds over MPI on BG/L and TCP on Gigabit Ether, respectively. Our protocol can be configured to match the network topology to increase communication throughput. We utilize radix trees to implicitly encode routing information into node IDs and additionally represent the tree structure as an array (dynamically resized upon node joins/failures) to provide access to the data structure of individual nodes in constant time. We also verify our experimental results against a performance model to assess the scalability of the approach and allow extrapolation for larger number of nodes.

Overall, our membership service for MPI tasks combines the best of both worlds, the scalability of a decentralized membership protocol and the performance of existing fault-tolerant mechanisms within high-performance runtime systems. Having implemented the protocol in the low-level communication layer of LAM/MPI, we are currently assessing the protocol's suitability for deployment within the MPI Component Architecture (MCA), specifically as an add-on to the Point-to-point Management Layer (PML) within Open MPI [27, 7, 43, 44]. Nonetheless, our approach is more general and can be applied for any membership service or in other frameworks that require scalable group communication, such as efficient multicast services, *e.g.*, in MRNet [38].

2. HIGH-LEVEL ASSUMPTIONS AND DESIGN

To tolerate faults for an MPI job, the set of individual MPI tasks represent a group within which they may communicate and coordinate execution and termination. Within the runtime system, MPI tasks have a consistent *view* about who is a member in such an abstract communication domain [28, 9, 14]. Fault tolerance requires a dynamic domain in which members can join and leave. The latter may be due to faults while the former may occur upon recovery from faults or when additional compute resources are required. Group communication, such as multicasting, can be based on membership properties within a domain.

Membership within a domain is implemented within a runtime-level membership service layer and used by an application layer that relies on this service. The *view* of the system is the set of currently active and connected (unpartitioned) processes. The application layer interacts with the membership service for communication and *view change* actions. The membership service maintains a consistent view of the system. It ensures that communication takes place only between processes that share the same view. In our model, every process starts with a *default view*. This view is internally represented as a tree. In the absence of faults, each node has a children, where a is constrained to be a power of two for reasons given below.

2.1 Assumptions and Safety Properties

We make the following assumptions about the overall framework:

Execution Integrity: We assume that no event occurs at a process between its crash and recovery. After a crash, the process is assumed to remember its unique ID (*e.g.*, derived from the IP address or the host name), but not necessarily the view since a view may change any time. The new view is obtained from the current root on recovery.

Message uniqueness: Each message contains a message type, the sender and the receiver information. The underlying communication stack guarantees reliable messaging, *i.e.*, neither will there be any duplications nor losses of messages. Given message uniqueness, our protocol ensures that any message be sent exactly once to a given destination.

The protocol should meet the following safety properties of communication and multicast services (see [14, 18]): **Self Inclusion:** The membership algorithm satisfies the self inclusion property, *i.e.*, if a process p establishes a view V, then p is a member of V. **Delivery Integrity:** For every receive, there is a preceding send. **No duplication:** At any process, two receive events can neither originate from the same send event, nor can they have identical message content. **Same view delivery:** If two processes p and q receive message m, they receive it in the same view.

The membership algorithm relies on the detection of faults by another layer of the software architecture. We specifically react to processor failures (crashes) and recoveries.

2.2 Fault Detection in the Execution Environment

Faults are detected by an external detection mechanism. Faults can be identified by hardware health monitoring, such as IPMI [1], detection of link failures or any other mechanism. The details are beyond the scope of this paper.

For the experiments in Section 7, we employ a fault detector based on a timeout mechanism. Excessive delay in response from any process to a message request leads to the assumption that the process has failed. Such a process is removed from the set of views in the *view change* event triggered by the above timeout. Link failures are handled similarly to node failures in this scenario, *i.e.*, different causes of failure need not be distinguished. The described protocol handles only single-path routing. An extension could handle multi-path routing through NACKs.

2.3 Processor Failure and Recovery

Within our execution environment, a fault-injecting application inquires the state of every other process randomly. This application is a micro-benchmark resembling the communication portion of real applications communicating *via* MPI over a runtime-supported membership service. A process failure should not cause the entire application to fail. Instead, each remaining node will update its membership view to obtain a new, consistent view in response to a message triggered within the tree structure, excluding failed nodes.

3. SCALABLE, LOW-LATENCY MEMBERSHIP SERVICE

In the following, the operational details of the membership algorithm, based on a radix tree, are detailed. The objective of the algorithm is to provide a new, consistent view of active nodes (members) in a scalable manner at very low overhead. The process of establishing a new view is called *tree stabilization* in the following.

3.1 Radix Tree Representation

Nodes participating in the membership service are internally represented in two data structures: a radix tree and a linear array of nodes. The former provides an efficient representation for collective communication while the latter supports point-to-point communication.

The radix tree provides a hierarchical representation that implicitly encodes routing information in the node ID, which reduces the overhead of algorithms that exploit the membership service. The radix encoding of a node ID can be used to determine the routing path of messages from the root to this node or to determine its position in the tree structure. To allow an efficient decoding of routing information, the number of children in the radix tree has to be a power of two. Hence, for a binary tree, the routing decision from one node to the next lower level is determined by a single bit indicating that one should follow the left (0) or right (1) child. In a tree with four children, such as in Figure 1, two bits indicate which link to follow to determine the location of a child in the tree.

In addition to the radix tree, an array of nodes provides access to arbitrary nodes at constant time, which can be utilized for point-to-point messages in a message-passing framework. This array is dynamically resized upon node joins and failures to accurately reflect view changes in a consistent manner.

3.2 Initialization

At the initialization phase, every node in the system is assumed to have knowledge of the number of children and the total number of nodes. Each node has a unique ID. These assumptions are consistent with MPI runtime environments. Communication between nodes is not required during the initialization phase, since the knowledge of the number of children and the total number of nodes is sufficient for nodes to locally form a hierarchical structure.

The hierarchical structure, *i.e.*, the radix tree, is organized such that the node with lowest ID is the root. Each node has a fixed number of children. The ID of each child of a node is determined as a function of the height of the node in the tree and the maximum number of children, as depicted in Figure 1. This is a constant-time operation due to the routing information encoded into the radix tree.

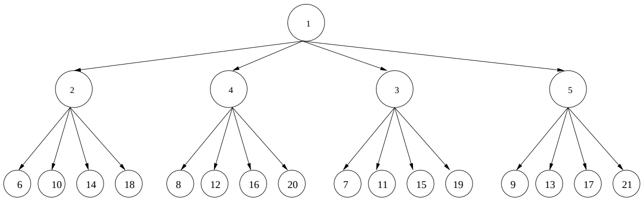

Figure 1: Stabilized Tree Structure

The radix tree is duplicated on each node and kept up-to-date with respect to a global view in a decentralized manner (consistent with other nodes). At startup, all nodes have the same initial view. Afterwards, any two nodes in the application layer may communicate at any time. This approach still allows for node failures during start-up, as discussed later. Overall, the system is scalable due to the fully decentralized initialization since no message exchange is required to form the hierarchy. The tree structure with a configurable number of children furthermore ensures that the system can be adapted to reflect a given network topology.

3.3 Fault Handling

A node is considered to have failed if indicated by the failure detector. For the experiments in Section 7, we detect a failure when a node does not respond within a timeout window to a query/message from another node. A node failure can be one of the following: Single node failure, multiple node failure, root failure and link failure. Upon detecting a failure, the root is informed of the failed node and initiates a view change (see Figure 3(a)).

A link failure is handled implicitly as if a node (and its subtree consisting of immediate children and their children etc.) is unreachable. Notice that partitions (subtrees) reorganize to form a new view (succinct from the view with the prior root). Applications may elect to continue or abort upon network partitioning, *e.g.*, depending on their ability to communicate with I/O nodes (such as in the BG/L model [4]).

3.4 Single Node Failure

This failure is the easiest to handle and requires very low communication bandwidth during the tree stabilization phase. The tree is assumed to be stabilized once the root receives an acknowledgment from all of its children affirming a stabilized tree in the lower layers, as depicted in Figure 3(a) and described below. Every failure detection message to the root will be acknowledged by a $FAILURE\_DET\_ACK$ message. When multiple nodes simultaneously detect the same failure, the root acknowledges each failure detection message but disregards all but the first failure detection message.

For simulation purposes, our application scenario lets nodes inquire the state of other nodes in the system at random intervals, which we used for fair testing and benchmarking. (As long as the application has regular communication, the protocol will be supported.) Assume that node 11 has sent a HOW_ARE_YOU message to a failed node 4 in Figure 1. On failure detection, it sends a $NODE\_FAILURE$ message to the root (assuming the failed node is not the root and all the nodes have a consistent view). The root recalculates its tree structure by eliminating the failed node from its list of nodes and updates corresponding links to its children in the tree, as depicted in Figure 2. The root node initiates the next step of the algorithm by sending a $FAILED\_NODE$ message to its children. Each child propagates the message down the tree after recalculating its local view (tree).

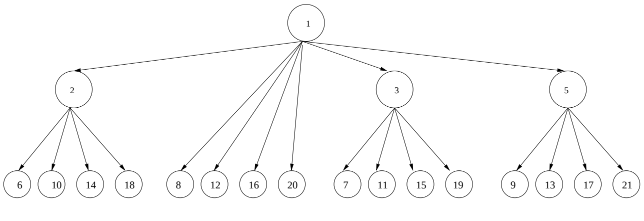

Figure 2: Tree Structure after node elimination

```
(a)Handling a node failure

On failure of a node (ID)
    if (ID == root)
        new_root = find_next_highest (ID);
        send $ROOT\_FAILURE$ (ID, self) message
        to new_root;
    else
        send $NODE\_FAILURE$ (ID, self) to root;

On receiving NODE_FAILURE(failed_node, detector)
by root
    send $FAILURE\_DET\_ACK$ to detector;
    Regroup (failed_node, flag);
        //$flag \in \{0, 1\}$, 0 = node failure, 1 = root failure
Regroup(failed_node, flag)
    recalc_tree_structure(failed_node);
    locate my children;
    send $FAILED\_NODE$(failed_node) message to children;

On receiving FAILED_NODE(failed_node) Message
in a child
    if(self $\neq$ leafnode)
        Regroup(failed_node, flag);
        locate my children;
        send $FAILED\_NODE$(failed_node) message
        to children;
    else
        Regroup(failed_node, flag);
```

```
(b)Handling a root failure

On receiving ROOT_FAILURE(ID, detector)
by new_root
    send $FAILURE\_DET\_ACK$ to detector;
    Regroup(ID, flag);
        //$flag \in \{2, 3\}$, 2 = recovery process, 3 = new node join
Regroup(failed_node, flag)
    recalc_tree_structure(failed_node);
    locate my children;
    send $ROOT\_DEAD$ message to children;

(c)Handling a node join

On receiving NEW_NODE(ID) by root
    Regroup(new_node, flag);
    send $JOIN\_DET\_ACK$ to new_node;

Regroup(new_node, flag)
    recalc_tree_structure(new_node);
    locate my children;
    send $NEW\_NODE\_JOIN$ message to children;

On receiving NODE_ALIVE(ID) by root
    Regroup(alive_node, flag);
    send $ALIVE\_NODE\_JOIN\_DET\_ACK$ to alive_node;

Regroup(alive_node, flag)
    recalc_tree_structure(alive_node);
    locate my children;
    send $ALIVE\_NODE\_JOIN$ message to children;
```

Figure 3: Pseudocode of the Membership Algorithm

The local tree recalculation procedure is as follows. Let D be the failed node, P(D) be its parent and C(D) the set of its children. Then, the new view is calculated by (1) assigning the parent of C(D) as P(D), (2) removing D from the list of children of P(D), (3) merging the list of children of D with the list of children of P(D) and (4) removing the list of children from D.

The tree structure will be consistent after each node has acknowledged to its parent a stable structure for the respective subtree. Once a $FAILED\_NODE$ message reaches a leaf node, the stabilization phase starts. Leaves respond with a $FAILURE\_ACK$ message to parents. Higher nodes acknowledge with a $FAILURE\_ACK$ to their parent once they have received the acknowledgments from their children. Failure to receive a $FAILURE\_ACK$ message will invoke another instance of the failure detector, as discussed in Section 3.5. The tree becomes stable once the root receives a $FAILURE\_ACK$ from all children.

3.5 Multiple Node Failures

This case is handled similarly to a single node failure. If multiple nodes fail simultaneously, the root receives a $NODE\_FAILURE(X)$ message from the detector process while the first phase of tree stabilization is in progress. The root acknowledges each failure detection message, and, if multiple nodes detect a failure of the same node, all but the first message are disregarded (although acknowledged). For multiple, distinct failed nodes, the root sends a list of dead nodes after recalculating the tree locally. To facilitate the presentation, the list is omitted in Figure 3(a); it simply extends the `failed_node` parameter to a set.

Example: Assume failures for nodes 4 and 5, and 11 has detected the failure of 4. The root sends $FAILED\_NODE(X)$ to its children and waits for an acknowledgment during the first tree stabilization phase. Since it does not receive an acknowledgment from node 5, it times out assuming that node 5 is dead. If this happens at lower layers of the tree, the node that fails to get an acknowledgment from the dead node informs the root through a $NODE\_FAILURE(Y)$ message. Then, the root propagates a list of failed nodes to its children. If a node failure has occurred at each level of the tree, it will take $H - 1$ initial tree stabilization phases for the tree to stabilize, where H is the tree height. A lower height can be achieved by choosing a larger number of children per node to speed up tree stabilization during multiple node failures. However, extremely low height (*e.g.*, a "flat" tree with just two levels) reduces performance as upper nodes become bottlenecks when propagating messages. Depending on the number of children (any power of two is legal), the height needs to be chosen accordingly, *i.e.*, by modeling stabilization time for different configurations.

3.6 Root Failure

Should the root fail, the detecting node sends a $ROOT\_FAILURE$ message to the next live node in the linear list (see Figure 3(b)), *i.e.*, a sequential scan suffices to designate a new root assuming the new root is alive. The algorithm proceeds in accordance with the single node failure recovery procedure explained above with following additions:

- The new root sends a $ROOT\_DEAD$ message to its children who transitively send it to their children.
- During the tree recalculation phase, each node also has to update its root to the new root.

The tree becomes stable after the new root has received acknowledgments from all of its children.

Consider the case where a root failure coincides with multiple node failures. To distinguish this case for a single root failure, a different message, $ROOT\_AND\_NODE\_FAILURE$, will be propagated down the tree indicating the new root and the set of failed nodes, followed by acknowledgments upwards. This new message allows children of the failed nodes that may be engaged in recalculations due to a prior failure to identify its proper parent and acquire a consistent overall view. Due to the similarity to handling $FAILED\_NODE$ messages, this detail is omitted in Figure 3.

3.7 Node Join

A new node may join a domain (the set of MPI tasks) by sending a $NEW\_NODE(ID)$ message to the root (see Figure 3(c)). The root adds it as a leaf to the bottom of the tree. This message then propagates in the same way as for a node failure. The root issues a $NEW\_NODE\_JOIN(ID)$ message to its children, which is propagated further down the tree by its children. The tree assumes a stabilized structure once each node in the hierarchy has received $NEW\_NODE\_JOIN\_ACK(ID)$ from all of its children. The leaves will eventually send an acknowledgment to their respective parent, and this message is propagated upwards to the root.

An implicit node join may occur when a node recovers from a failure. Recovered nodes may re-join with their original ID by maintaining an association between the host name and the ID of failed nodes. This mapping is maintained by all the nodes in the system. The recovered process issues a $NODE\_ALIVE(ID)$ message to the root, and the stabilization routine follows the same procedure as for a join of a new node.

Once the tree is stabilized, the root sends $JOIN\_DET\_ACK$ message to the recovered process or the new node welcoming it to the system. A failure to get a $JOIN\_DET\_ACK$ from the root triggers the new node or a recovered process to send a $NEW\_NODE(ID)$ or $NODE\_ALIVE(ID)$ message, respectively, to the next node in its sequential list of nodes. The time to join the system might increase if a considerable number of processes have failed in the top of the hierarchy and a node with a lower ID has assumed the status of the root. If a node join occurs when a system recovers from a failure, the root node sends a list of failed and (prior) joined nodes. The tree recalculation occurs locally. One message suffices for establishing a stabilized tree structure. The joiner has to find the current root through a linear scan of the list. Other schemes, such as random requests to other nodes to inquire about the root, are also possible. If the joiner happens to be the new root, every node agrees on this during the tree recalculation phase.

4. PERFORMANCE MODELING

In addition to the protocol design and implementation efforts, we attempted to model the performance of our protocol with a theoretical model. Initial efforts to measure the overall *time for stabilization*, Ts, in the presence of a single node failure within network simulators, such as the *network simulator 2* (Ns-2) [35], were considered inappropriate since such simulators generally do not allow computational overhead to be reflected in their models. We also observed practical challenges on clusters, as explained in the following, that cannot be accurately represented by simulation.

We derived a rudimentary performance model based on *communication overhead* (Ocm) and *computation overhead* (Ocp). Ocp captures the time for updating the tree structure on a local node and can simply be measured in wall-clock time on a target architecture. Ocm is based on the latency L of point-to-point connections of adjacent nodes in the tree.

Our *base model* assumes a single-hop connection between adjacent nodes with uniform latency measured as half the round-trip time in a ping-pong experiment. To measure Ocm for the entire tree, two times the latency is being considered between each node level, one per message, *i.e.*, to propagate a node failure down and another to receive a response. Let H be the height of the tree. Then, there are $H-1$ levels for communication between parents and children. Thus,

$$Ocm = 2 \times L \times (H-1) \tag{1}$$

The total tree stabilization overhead, Ts, is based on the overall communication overhead and the delay due to computational overhead within each level of the tree structure. Hence,

$$Ts = Ocm + Ocp * H \tag{2}$$

We next turn to experimental results to assess the performance of our protocol. The model is used as a reference to allow projections into larger number of processors if it fits the observed results. While found to be valid in principle, several refinements of the model were necessary due to machine-specific impacts on the latency, as discussed in the following. These refinements go beyond other models, such as LogP or its extensions [17].

5. EXPERIMENTAL FRAMEWORK

To assess the performance of our protocol, various tests were conducted on a number of test beds. We report the results for two of them in the following: a BlueGene/L (BG/L) machine and the eXtreme TORC (XTORC) cluster at Oak Ridge National Laboratory(ORNL). On BG/L, all executables run on the compute nodes atop a light, UNIX-like proprietary kernel, the compute node kernel (CNK) [2]. There are two midplanes (each with 512 nodes or 1024 embedded PowerPC processors), and each midplane has a three-dimensional (3D) torus interconnect for point-to-point messages besides other interconnects for selected collective communication. When the partition is smaller than a midplane, the interconnect is a 3D mesh, hence, we ensured that an entire midplane was allocated to our jobs. XTORC has 64 2Ghz Pentium 4 compute nodes connected by 1Gb/s Ethernet running RedHat 9.0 (Linux kernel-2.4.20-8). Of the 64 nodes, only 47 nodes were available for testing. The entire test environment was written in C in a single-threaded manner since we observed high variations for threading in prior implementations.

The memory requirement of the scheme is small and increases linearly per node. On each node, the tree has a space complexity of O(N), where N is the number of nodes in the tree structure. For BlueGene/L, each compute node has slightly less than 512MB of physical memory available for user programs. A tree structure that has 1024 nodes (using both midplanes of BlueGene/L) uses less than one MB of memory leaving ample memory space for the running applications. XTORC provides 768MB of physical memory, and the memory requirement of our protocol was only a few kilobytes for less than 64 nodes. We are currently assessing a variant of our protocol with localized views of the overall tree to limit the memory requirements to a constant size and, thereby, support scaling into tens of thousands of nodes and beyond.

On BlueGene/L, MPI_Send and MPI_Irecv primitives implement the communication of the protocol. The reason for using non-blocking receive calls was to eliminate threading since (a) threading is not supported on BG/L and (b) threading was shown to result in high overhead and variance in performance on Linux. The implementation on XTORC relies on TCP sockets.

6. FUNCTIONALITY TESTING

The implementation of the protocol was subjected to extensive functionality tests with single node failures, multiple simultaneous failures, single root and chained, simultaneous root and top node failures, the last of which requires linear selection of the next root node. Failures were injected to the testing environment and resemble non-responsiveness of nodes as commonly detected by timeouts during communication.[1] The protocol proved to be robust to allow functioning nodes to survive failures of other nodes while still retaining the capability to communicate and track the set of operational nodes.

We further implemented the protocol as part of LAM/MPI at the LAM daemon level as a new service module. Extensive tests show that the protocol can sustain injected faults and reconfigure. Observed measurements are similar to the results discussed below for TCP on a Linux cluster, albeit with a 10%-20% higher overhead due to integration into the costly process model of the low-level LAM infrastructure, and will be omitted due to space constraints. We have integrated our approach with the Berkeley Labs Checkpoint/Restart (BLCR) [22] facility such that one does not have to restart an MPI job when a node fails if (a) a failed node is recovered or (b) a spare node exists to assign the failed work to using the old MPI rank. We are also integrating transparent (no application modifications, no manual state re-distribution) and periodic, yet coordinated checkpointing. LAM/MPI lacks these capabilities; it requires a cold restart of the entire MPI job, which can be costly considering that most nodes still contain the process image, and it results in long response times for users that could be avoided if spare nodes were available.

7. PERFORMANCE EVALUATION

We assessed the performance of our protocol in terms of the time for stabilization, Ts, after a single node failure, which is the most common type of failure since, as will be shown, Ts is in the order of hundreds of microseconds or single-digit milliseconds and, thus, orders of magnitude smaller than the mean-time-to-failure (MTTF) in even the largest systems.

7.1 MPI on BlueGene/L

Figure 4 depicts the experimental results for assessing the stabilization time, Ts, on BlueGene/L over MPI for increasing numbers of nodes. A binary tree configuration was chosen with two children (a=2). Notice that the x-axis is on a log scale, which shows that our protocol scales logarithmically with increasing number of nodes. Furthermore, Ts is in the order of microseconds up to 1024 nodes. If we interpolate these results, this trend is likely to continue into the tens of thousands of processors on BG/L. The results were obtained from five samples with a confidence interval of $\pm 3\mu$s to $\pm 16\mu$s for smaller and larger node numbers, respectively, at a 99% confidence level.

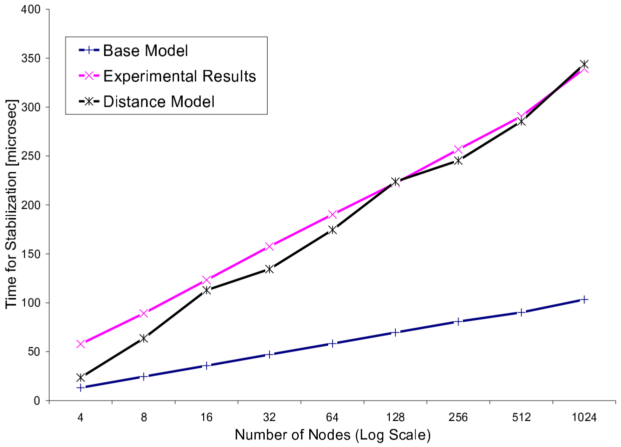

Figure 4: Ts over MPI for a=2 on BG/L

We also assessed the validity of our base model for a single hop, point-to-point latency of $L = 4.6\mu$s and a computational overhead of $Ocp = 2\mu$s on BG/L. The resulting base model diverges significantly from the experimentally obtained results. This can be attributed to the point-to-point communication topology of BG/L. We conducted our experiments on two midplanes with each midplane consisting of 512 processors, which have a 8x8x8 3D torus interconnect. When MPI tasks are mapped to nodes, adjacent nodes in the tree may have to communicate over varying number of hop counts (distances) within the torus. Each hop thereby imposes the cost of the base latency L. To consider this overhead, we refined our base model to account for the communication overhead, Ocm, using a distance-aware latency to derive a *distance model*. Here, the overall number of hops contributing to the latency is the sum over all levels in the tree of the maximum distance in hops at each level. Thus,

$$Ocm = 2 \times [\sum_{levels} max(\texttt{hops b/w nodes at level})] \times L \times (H-1)$$

This model considers the maximum latency between adjacent nodes (all parent/child pairs) at each level (in both directions) and aggregates the respective maximum for all levels in the tree. The hop count is determined as the sum of differences between each pair of x, y and z coordinates of nodes in the 3D-torus that are adjacent in the tree structure. As the results in Figure 4 show, this distance model closely matches the observed results. This underlines the benefits of simplicity and scalability of our protocol while delivering performance.

Figure 5 shows the stabilization time for a tree configuration with four children per parent (a=4). Again, the experimental results show that the protocol scales logarithmically with the number of nodes. The absolute overhead for Ts is slightly smaller than for the binary tree configuration (a=2), which can be attributed to the reduction of height in the tree. But the impact of hop counts reduces this benefit to some extent. The results were obtained from five samples with a confidence interval of $\pm 0.5\mu$s to $\pm 12\mu$s for smaller and larger node numbers, respectively, at a 99% confidence level.

The base model shows an interesting behavior in that it alternates between slight increases and no changes (flat line) in performance. A flat line occurs when the number of nodes is increased but the height of the tree remains unchanged, *i.e.*, the height of the tree changes only for powers of four. Once we consider the distance model that includes the hop counts for point-to-point communication in the tree, the model closely approximates the observed performance for each measurement point that is a power of four (or exceeds the height of the previous tree). In between, however,

[1] When a node times out but has not failed, it will still be treated as if it has failed since progress is hindered by this node. By excluding this node from further communication, other nodes can proceed in a timely manner, *e.g.*, by electing a replacement node within the MPI runtime system. Any messages from the excluded nodes pertaining to the old job are henceforth ignored by other nodes. If the node is fully responsive again, it may join the set of running nodes and can be assigned any work at that time, same as or different from the original work.

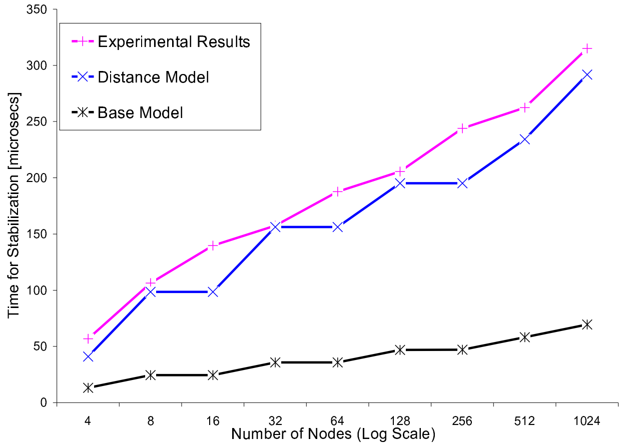

Figure 5: Ts over MPI for a=4 on BG/L

performance is underestimated. This artifact remains not fully explained, but we have eliminated system activity as a source. We will discuss network contention as a potential source in subsequent results. Nonetheless, the overall trends demonstrate the scalability of the protocol with a matching model for powers of four.

Notice that the protocol could alternatively have been implemented over the hardware tree interconnect utilized by some collective communications on BG/L, which would have resulted in shorter response times. However, the objective of this work was to assess the scalability of the protocol for large numbers of nodes assuming commodity interconnect topologies without special one-to-all support in hardware.

7.2 TCP over Ethernet

Figure 6 depicts the stabilization time observed in experiments on a dedicated Linux cluster (no background activity) with a single Gigabit switch using a TCP implementation of our protocol for a binary tree (a=2). Notice that the x-axis is on a linear scale. The experimental results show a step-curve of increasing stabilization time. Upon closer analysis, we observe that the protocol is scalable for TCP as well, *i.e.*, that its time complexity increases logarithmically with the number of nodes.[2] The results were obtained from five samples with a confidence interval of ±4μs to ±86μs for smaller and larger node numbers, respectively, at a 99% confidence level.

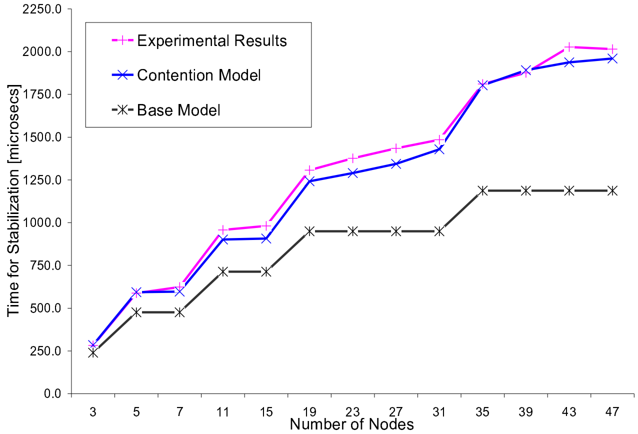

Figure 6: Ts over TCP for a=2 on Linux

We also observe that Ts increases linearly between any power-of-two node counts. This behavior is consistent with the experimental results in Figure 5. We further observe that the base model (with a TCP latency of $L = 118\mu$s and a computation overhead of $Ocm = 2\mu$s) does not resemble the experimental results. The hop count is not a factor as a single full-duplex switch allows direct communication between any pair of nodes without contention at the network fabric. The switch itself, however, may serialize packet processing.

The hypothesis of packet serialization within the switch was confirmed in a series of experiments where an increasing number of neighboring nodes communicated along a localized structure. Figure 7 presents the experimentally determined latency under contention for these configurations of (a) pairs of nodes, (b) a parent with two children and (c) a parent with four children communicating with one another, as depicted in order of increasing latency. We observe that point-to-point communication of pairs of nodes is handled well by the switch up to twelve nodes, after which the latency linearly rises with the number of nodes added. More significantly, a more complex internal structure, such as a binary tree, inflicts higher switch contention for the same number of nodes due to serialized communication with multiple nodes at the parent. The latency increases even more significantly for a tree with four children.[3]

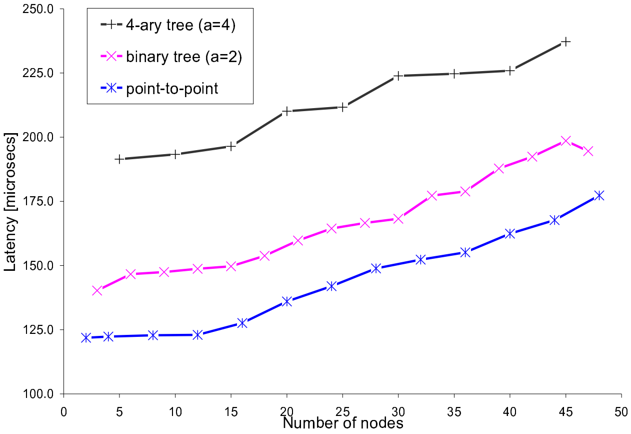

Figure 7: Contention-based Latency over TCP

The results obtained as contention latency in Figure 7 were subsequently used to substitute the base latency in Equation 1 with the contention latency in the figure corresponding to the respective number of nodes. The resulting contention-based model in Figure 6 resembles the the experimental results very closely. Moreover, we argue that contention latencies can be extrapolated for larger node

[2] A plot on a logarithmic x-axis for results of $2^n - 1$ nodes illustrates this behavior. The linear x-axis here is intentionally used to motivate the following analysis.

[3] Notice that these results could not be accurately be modeled by other models, such as LogP [16] with its account of send/receive overhead and the gap, since a linear increase with increasing number of nodes of any of the base parameters is not considered.

numbers, due to the near-linear behavior in single switches. When switches are hierarchically combined, contention latencies of each single switch can be aggregated in a manner reflecting the switch topology. This is subject of future investigation.

Figure 8 depicts the results for TCP over a tree with four children per parent. The overall results indicate scalability of our protocol in terms of its logarithmic complexity. The results were obtained from five samples with a confidence interval of $\pm 12\mu$s to $\pm 98\mu$s for smaller and larger node numbers, respectively, at a 99% confidence level.

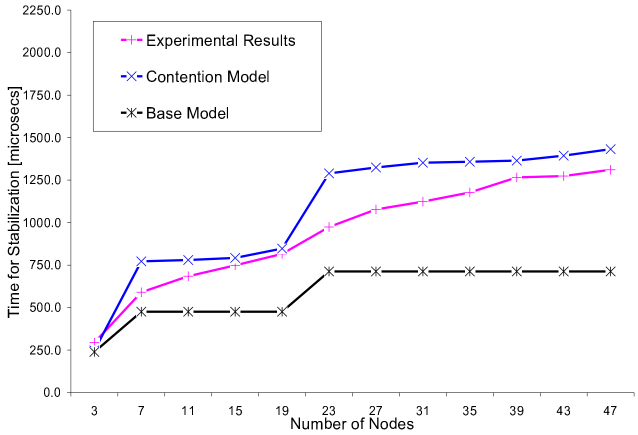

Figure 8: Ts over TCP for a=4 on Linux

The base model shows the typical step curve with increases in stabilization time when the number of nodes increases such that the tree height increases by one (above 5 and 21 nodes), but the base model does not resemble the actual measurements. When considering the latencies of Figure 7, the contention model resembles the experimental results just before the tree height is increased. More significantly, the contention model more accurately reflects the increased contention for larger number of nodes. The fact that the contention model tends to overestimate the experimental results is not fully understood but we observed that larger overestimations also tend to coincide with larger confidence intervals.

Overall, the experimental results confirm the scalability of our protocol and the refined models show a close resemblance of experiments, which should qualify them for the task of extrapolations for larger number of nodes.

8. RELATED WORK

Chockler *et al.* provide a set of rigorous specifications for the group membership service and discuss various systems where different properties are satisfied [14]. Most of the existing systems assign a view identifier for each new view installed in the system [26, 5, 31]. Our model does not require maintenance of a list of different views (*i.e.*, a view set with unique IDs per view) since the system stabilizes once the root node receives all acknowledgments from its children. Our approach of each process deciding its own view without exchange of any message with any other node is also found in Transis [34] and Consul [36]. We do not allow multiple disjoint views to exist concurrently. This property of primary component membership is implemented elsewhere as well [9, 42, 15]. A solution to the view-oriented partitionable membership problem is provided by R. Khazan [32, 33]. His approach is a hybrid of decentralized clients and more powerful servers with a leader at any given point in time, *i.e.*, it is not a fully decentralized (peer-to-peer only) model due to practical network connectivity issues.

The Coyote system [8] provides a group membership service based on a token-passing paradigm and uses 25 micro-protocols to implement each group membership property. Our algorithm keeps the interaction among different nodes simple, and stabilizes the hierarchical structure after each node receives just one message from its immediate parent node.

A topology-aware membership service for cluster-based Internet services is proposed by Zhou, Chu and Yang [45, 46]. It uses Time-To-Live in the IP packet header to form hierarchical groups that resemble the network topology. The reported time for tree stabilization for this model does not account for network latency, gap and over heads involved for sending and receiving data. In this protocol, the view convergence time is measured as the sum of failure detection time and the time to propagate the information along the hierarchical tree. The paper does not provide the tree stabilization time. Hence, we cannot make a fair comparison with our work.

Other prior work includes support for fault tolerance to the communication layer of MPI run time systems. Sankaran *et al.* [39] discuss a LAM/MPI checkpoint/restart framework where MPI applications can be check-pointed to disk and restarted later. They use the (Lawrence) Berkeley Labs Checkpoint/Restart (BLCR) mechanism [22, 23] to implement a lightweight and modular component-based architecture. It requires each MPI process to coordinate with other processes to reach a consistent global state in which the MPI job can be check-pointed. Bosilca *et al.* propose an uncoordinated checkpoint mechanism by saving the computation and communication contexts independently [10]. Each node stores the execution contexts in remote checkpoint servers and uses dedicated nodes (Channel Memory) to store in-transit messages. Chakravorty *et al.* [13] extended the runtime layer of Adaptive MPI (AMPI) beneath Charm++ to migrate objects in a proactive fault-tolerant manner. Collective communication structures, such as trees, were rebalanced after node failures. In contrast, our work is more general (any group communication structure), and their quantitative results include migration overhead, *i.e.*, no direct comparison can be given.

Prior work on distributed locking explored the scalability of tree structures [21]. This prior work focused on mutual exclusion protocols and reader/writer locks in the context of middleware such as CORBA. A fault-tolerant extension of such a locking protocol is developed as a ring-based topology, which limits its scalability [37]. Our membership algorithm, in contrast, provides consistent views among nodes in the presence of faults in a scalable manner. Furthermore, the approach is reconfigurable for a variable number of children (as a power of 2), natively encodes routing information due to its use of a radix tree, and it provides constant time access to the data structure for individual nodes.

9. CONCLUSION

This work presents a novel membership algorithm that combines scalability with low recalculation overhead in the order of hundreds of micro-seconds and single-digit milliseconds for MPI over BG/L and TCP over Linux, respectively. The protocol supports reconfiguration in terms of the communication structure, *i.e.*, the data structures can be adapted to match the network topology to further increase performance. The protocol utilizes a radix tree representation to implicitly encode routing information into node

IDs and additionally represent the tree structure as an array to provide access to the data structure of individual nodes in constant time. The protocol builds on prior experience of designing scalable communication frameworks by utilizing a fully decentralized protocol that maintains a coherent membership view of MPI tasks in the presence of faults. Experiments demonstrate high performance and scalability of our protocol over TCP on Gigabit Ether and over MPI on BG/L. Experimental results were also validated against a closely matching performance model to allow extrapolations to larger number of nodes. The membership service has been deployed in the communication layer of the LAM/MPI runtime system, and we are currently pursuing its integration into Open MPI and, independently, into LAM/MPI with BLCR to continue job execution in the presence of faults.

10. REFERENCES

[1] http://www.intel.com/design/servers/ipmi/index.htm.

[2] http://www.redbooks.ibm.com/redbooks/pdfs/sg246686.pdf.

[3] The ASCI purple benchmarks. http://www.llnl.gov/asci/purple/benchmarks, 2002.

[4] N. Adiga and et al. An overview of the BlueGene/L supercomputer. In *Supercomputing*, Nov. 2002.

[5] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, Nov. 1995.

[6] R. T. Aulwes, D. J. Daniel, N. N. Desai, R. L. Graham, L. D. Risinger, M. A. Taylor, T. S. Woodall, and M. W. Sukalski. Architecture of LA-MPI, a network-fault-tolerant MPI. In *International Parallel and Distributed Processing Symposium*, 2004.

[7] B. Barrett, J. M. Squyres, A. Lumsdaine, R. L. Graham, and G. Bosilca. Analysis of the component architecture overhead in Open MPI. In *Proceedings, 12th European PVM/MPI Users' Group Meeting*, Sorrento, Italy, September 2005.

[8] N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu. Coyote: a system for constructing fine-grain configurable communication services. *ACM Trans. Comput. Syst.*, 16(4):321–366, 1998.

[9] K. P. Birman and R. Van Renesse, editors. *Reliable distributed computing using the Isis Toolkit*. IEEE Computer Society Press, 1994.

[10] G. Bosilca, A. Boutellier, and F. Cappello. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Supercomputing*, Nov. 2002.

[11] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of MPI programs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2003.

[12] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Collective operations in an application-level fault tolerant MPI system. In *International Conference on Supercomputing*, June 2003.

[13] S. Chakravorty, C. Mendes, and L. Kale. Proactive fault tolerance in large systems. In *HPCRI: 1st Workshop on High Performance Computing Reliability Issues, in Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*. IEEE Computer Society, 2005.

[14] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study, Apr. 23 2001.

[15] F. Cristian. Reaching agreement on processor group membership in synchronous distributed systems, June 12 1991.

[16] D. Culler, R. Karp, D. Patterson, A. Sahay, E. Santos, K. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, Nov. 1996.

[17] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. 4th Symp. Principles and Practice of Parallel Programming*, pages 1–12. ACM, 1993.

[18] X. Defago, A. Schiper, and P. Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[19] N. Desai and F. Mueller. A log(n) multi-mode locking protocol for distributed systems. In *International Parallel and Distributed Processing Symposium*, Apr. 2003.

[20] N. Desai and F. Mueller. Scalable distributed conucrrency services for hierarchical locking. In *International Conference on Distributed Computing Systems*, pages 530–537, May 2003.

[21] N. Desai and F. Mueller. Scalable hierarchical locking for distributed systems. *Journal of Parallel Distributed Computing*, 64(6):708–724, June 2004.

[22] J. Duell. The design and implementation of berkeley lab's linux checkpoint/restart. Tr, Lawrence Berkeley National Laboratory, 2000.

[23] J. Duell, P. H. Hargrove, and E. S. Roman. Requirements for linux checkpoint/restart, May 20 2002.

[24] C. Engelmann, S. Scott, and G. Geist. Distributed peer-to-peer control in Harness. In *International Conference on Computational Science*, volume 2330, pages 720–728, 2002.

[25] G. E. Fagg and J. J. Dongarra. FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world. In *Euro PVM/MPI User's Group Meeting, Lecture Notes in Computer Science*, volume 1908, pages 346–353, 2000.

[26] R. Friedman and R. van Renesse. Strong and weak virtual synchrony in horus. Technical Report TR95-1537, Cornell University, Computer Science Department, Aug. 24, 1995.

[27] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of

a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[28] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[29] IBM T.J. Watson. Personal communications. July 2005.

[30] I. Keidar. Group communication, June 12 2000.

[31] I. Keidar, J. B. Sussman, K. Marzullo, and D. Dolev. A client-server oriented algorithm for virtually synchronous group membership in WANs. In *International Conference on Distributed Computing Systems (ICDCS)*, 2000.

[32] R. Khazan. Group membership: A novel approach and the first single-round algorithm. In *PODC: 23th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 2004.

[33] R. Khazan and S. Yuditskaya. Using leader-based communication to improve the scalability of single-round group membership algorithms. In *International Parallel and Distributed Processing Symposium*, 2005.

[34] D. Malki, D. Dolev, and R. Strong. A framework for partitionable membership service, Aug. 19 1995.

[35] S. McCanne and S. Floyd. VINT Network Simulator - ns (version 2). *http://www-mash.CS.Berkeley.EDU/ns/*, Apr. 1999.

[36] S. Mishra, L. L. Peterson, and R. D. Schlichting. Consul: a communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering*, 1(2):87–103, Dec. 1993.

[37] F. Mueller. Fault tolerance for token-based synchronization protocols. In *Workshop on Fault-Tolerant Parallel and Distributed Systems*, Apr. 2001.

[38] P. C. Roth, D. C. Arnold, and B. P. Miller. Mrnet: A software-based multicast/reduction network for scalable tools. In *Supercomputing*, pages 21–36, Washington, DC, USA, 2003. IEEE Computer Society.

[39] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proceedings, LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.

[40] J. M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

[41] G. Stellner. CoCheck: checkpointing and process migration for MPI. In IEEE, editor, *Proceedings of IPPS '96. The 10th International Parallel Processing Symposium: Honolulu, HI, USA, 15–19 April 1996*, pages 526–531, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.

[42] S. Toueg and T. D. Chandra. Unreliable failure detectors for reliable distributed systems, June 18 1996.

[43] T. Woodall, R. Graham, R. Castain, D. Daniel, M. Sukalski, G. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, and A. Lumsdaine. Open MPI's TEG point-to-point communications methodology: Comparison to existing implementations. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 105–111, Budapest, Hungary, September 2004.

[44] T. Woodall, R. Graham, R. Castain, D. Daniel, M. Sukalski, G. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, and A. Lumsdaine. TEG: A high-performance, scalable, multi-network point-to-point communications methodology. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 303–310, Budapest, Hungary, September 2004.

[45] T. Yang, J. Zhou, and L. Chu. An efficient topology-adaptive membership protocol for large-scale network services. Technical report, University of California, Santa Barbara, Computer Science, June 2004.

[46] J. Zhou, L. Chu, and T. Yang. An efficient topology-adaptive membership protocol for large-scale cluster-based services. In *International Parallel and Distributed Processing Symposium*, 2005.