Scalable Fault Tolerant Protocol for Parallel Runtime Environments

Thara Angskun[1], Graham Fagg[1], George Bosilca[1], Jelena Pješivac–Grbović[1], and Jack Dongarra[2]

[1] Department of Computer Science, The University of Tennessee, Knoxville
[2] University of Tennessee, Oak Ridge National Lab. and University of Manchester
{angskun, fagg, bosilca, pjesa, dongarra}@cs.utk.edu

Abstract. The number of processors embedded on high performance computing platforms is growing daily to satisfy users desire for solving larger and more complex problems. Parallel runtime environments have to support and adapt to the underlying libraries and hardware which require a high degree of scalability in dynamic environments. This paper presents the design of a scalable and fault tolerant protocol for supporting parallel runtime environment communications. The protocol is designed to support transmission of messages across multiple nodes with in a self-healing topology to protect against recursive node and process failures. A formal protocol verification has validated the protocol for both the normal and failure cases. We have implemented multiple routing algorithms for the protocol and concluded that the variant rule-based routing algorithm yields the best overall results for damaged and incomplete topologies .

1 Introduction

Recently, several high performance computing platforms have been installed with more than 10,000 CPUs such as Blue-Gene/L at LLNL, BGW at IBM and Columbia at NASA [5]. Unfortunately, as the number of components increases, so does the probability of failure. To satisfy the dynamic requirement of such a dynamic environment (where the available number of resources is fluctuating) a scalable and fault-tolerance framework is needed. Many large-scale applications are implemented on top of message passing systems for which the de-facto standard is the Message Passing Interface (MPI) [10]. MPI implementations require support of parallel runtime environments, which are extensions of the active operating system services, and provide necessary functionalities (such as naming resolution services) for both the message passing libraries and applications themselves. However, currently available parallel runtime environments are either not scalable or inefficient in dynamic environments. The lack of scalable and fault-tolerance parallel runtime environments motivates us to design and implement such a system. A scalable and fault-tolerant communication protocol that can be used as a basis for constructing higher level fault-tolerant parallel runtime environment is described in this paper. The basic ability of the designed

protocol is to transfer messages across multiple (multicast and broadcast rather than unicast) nodes efficiently, while protecting against recursive node or process failures.

The structure of this paper is as follows. The next section 2 discusses related work. Section 3 introduces the scalable and fault-tolerant protocol, while the section 4 presents the formal protocol verification. Experimental results are given in section 5, followed by conclusions and future work in section 6.

2 Related Work

Although there are several existing parallel runtime environments for different types of systems, they do not meet some of the major requirements for MPI implementations: scalability, portability and performance. Typically, distributed OS and single system image systems are not portable while the nature of Grid middle-wares has performance problems.

The MPICH implementation [8] uses a parallel runtime environment called Multi-purposed daemon (MPD) [3] for providing scalability and fault-tolerant through a ring topology for some operations and a tree topology for others. Runtime environments of other MPI implementations, such as Harness [1] of FT-MPI [6], Open RTE [4] of Open MPI [7] and LAM of LAM/MPI [2], do not currently provide both scalable and fault tolerance solutions for their internal communications.

The scalability and fault-tolerance issues have been addressed in several networking areas. However, those approaches could not be used or they are not efficient in the parallel runtime environments. Structured peer-to-peer networking based on distributed hash tables such as CAN [11], Chord [14], Pastry [13] and Tapestry [15] was designed for resource discovery. They are only optimized for unicast messages. Techniques used in sensor or large scale ad-hoc networking based on gossiping (or epidemic algorithm) [9] [12] mainly focus on information aggregation.

3 Scalable and Fault-Tolerant Protocol

The protocol in this paper is not aware of MPI implementation. It aims to support parallel runtime environments of various message passing implementations. However, currently work is in progress to integrate it in a fault-tolerance implementation of message passing interface called FT-MPI as well as in the modular MPI implementation called Open MPI.

The protocol is based on a *k-ary* sibling tree topology used to develop a self healing tree topology. The *k-ary* sibling tree topology is a *k-ary* tree, where k is number of fan-out ($k \geq 2$), and the nodes on the same level (same depth on the tree) are linked together using a ring topology. The tree is primary designed to allow scalability for broadcast and multicast operations that are typically required during MPI application startup, input redirection, control signals and termination. The ring is used to provide a well understood secondary path for

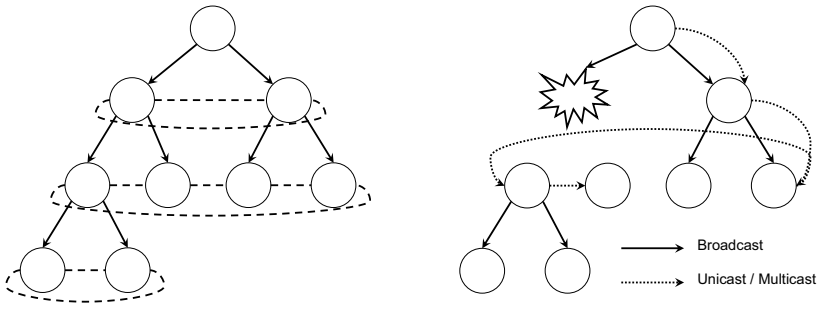

Fig. 1. (a) Binary sibling tree topology. (b) Message rerouting in case of failure.

transmission when the tree is damaged during failure conditions (simplest multi-path extension). In addition, typical k-ary tree only needs a single link or node failure to become bisectional, while the *k-ary* sibling tree can tolerate up to k failures. Fig. 1(a) illustrates an example of the binary (k=2) sibling tree. Each node needs to know the contact information of at most k+3 neighbors (i.e. parent, left, right and their children). The number of neighbors is kept to a minimum to reduce the state management load on each node. Both the tree and the ring topologies allow for neighbors addressing to be computed locally. Usually, we expect the k parameter to remain constant for the lifetime of the topology. The contact information of each node in some cases can be calculated locally for some tightly coupled systems or may be stored in an external directory service such as a name service of FT-MPI, a general purpose registry (GPR) of Open MPI or even a LDAP server for loosely coupled systems. The tree will automatically repair itself depending on an external recovery policy (i.e. when and how to repair it) specified by the user. The details of protocol is specified in section 3.1. The routing control of the protocol is discussed in section 3.2

3.1 Protocol Specification

Service Specification: The goal of the protocol is to deliver messages across multiple nodes while protecting against different types of node and/or process failures. The protocol currently provides two kinds of message delivery service, which are broadcast (1 to n) and multicast (1 to m, where $m \leq n$ [3]). The broadcast service uses the k-ary tree to send messages in normal circumstance. It will use the neighbor nodes to reroute the messages in the failure cases as shown in Fig. 1(b). The multicast service treats the k-ary sibling tree as a graph. It uses best effort to deliver messages with the shortest path from a source to destinations in both normal and failure situations.

Environment Assumption: The protocol assumes that any failures are Fail-stop rather than Byzantine i.e. if a process or a node crashes, it should be

[3] A unicast message is a special case of multicast where m=1

unreachable rather than pretend that it is still alive. After each failure, at least one neighbor of each node should be alive. Otherwise the *k-ary* tree will become bisectional, and no routing of messages between the two section of the tree will be possible. This assumption can be removed, if we allow each node to contact a directory service (considered as a stable resource) to overcome the orphan situation. The protocol also assumes that the transmission channel in which the protocol is executed can detect and recover from transmission errors (e.g. based on TCP and/or reliable UDP).

Protocol Vocabulary: There are 3 distinct kinds of messages: *hello* for the initialize message, which constructs the *k-ary* sibling tree; *mcast* for the multicast messages and *bcast* for the broadcast messages.

Message Format: The general message format of the protocol starts with a version number followed by a message type (i.e. the control fields *hello*, *mcast* and *bcast*). The *hello* message format consists of the above fields followed by an originator of the message indicated by *SrcID*. The *bcast* message format also contains *data* with the size DataSz. The *mcast* message consists of above mentioned fields followed by *#Dest*, *DestInd*, *DestList* and *TranList*. The *#Dest* is the number of destinations. The *DestInd* is an index, which points to the current destination in the *DestList*. The *TranList* is a transit list which contains the list of IDs of all the transit nodes in the tree to prevent looping and for back-tracking purposes.

Procedure Rules: The procedure rules can be separated into two steps: initialization and routing.

The initialization step of the procedure rules was described as follows: "Each node will register itself to the directory service (DS) and get its logical ID. It builds a logical topology and asks for the contact information of its neighbors from the DS. Once ready, it will start sending *hello* packet to its parent and its left neighbor. If the node is the right most in its level, it will also send *hello* to the left most node of the same level". After exchanging these *hello* messages, the communication channel between them will be established.

The procedure rules for routing a packet of the protocol were described as follows: "A node uses best effort to deliver messages following the shortest possible path. Sending a message procedure is dependent on the message type. If the message type is *bcast*, the node will send the message to all of its children. If a child died, it will reroute the message to all children of the child. This is done using an encapsulation technique. The node will encapsulate the broadcast message into a multicast message and send to its grandchildren. The grandchildren will decapsulate the multicast packet and continue to forward the broadcast message. However, if the message type is *mcast*, the next hop is chosen from a valid neighbor node which has the highest priority. [4] A node is said to be valid if and only if the node is not in the transit list and it is still alive. If there is no possible next hop, the message will be sent back to the previous sender (i.e. back-

[4] An implementation of the protocol may use a dynamic programming technique to improve performance by keeping the priority of neighbors for each destination in a look-up table.

tracking). When a node receives a message, it will first determine the header. If the message type is *hello*, it will do the initialization step. If the message type is *bcast*, it will forward to its children and handle node failure as mentioned above. If the message type is *mcast* and the node is not one of the destinations, it will add itself to the transit list and send it on to the next node. If the node is one of the destinations, but not the last one, it will remove itself from the destination list (*DestList*), decrease the destination count (*#Dest*), choose the next destination and update the destination index (*DestInd*), add itself to the transit list and send it to the next node."

Algorithm 1 Compute estimated cost

Procedure : Compute cost

```
 1: cost ⇐ 0 ; nextHop ⇐ srcID
 2: while nextHop ≠ destID do
 3:    if myLevel = destLevel then
 4:       Choose left or right
 5:    else if myLevel > destLevel then
 6:       nextHop ⇐ myParentID
 7:    else
 8:       if ChildID_i is an ancestor of destID then
 9:          nextHop ⇐ ChildID_i
10:       else
11:          Choose left or right, which one is closer to an ancestor of destID in myLevel
12:       end if
13:    end if
14:    cost ⇐ cost +1
15: end while
16: return cost
```

Procedure : Choose left or right

```
1: if (hopLeft ≤ hopRight) ∧ (destID ≠ myRightID) then
2:    nextHop ⇐ myLeftID
3: else
4:    nextHop ⇐ myRightID
5: end if
```

3.2 Routing algorithm

This section discusses the routing technique used for multicast messages (which is also used by broadcast routing during failures). The goal of the routing algorithm is to find the shortest possible route in both normal and failure situations with only local knowledge stored at each node. The next hop is chosen from the highest priority node of its valid neighbors. The first algorithm (as shown in Algorithm 1) uses a rule based method to estimate a cost from the current node to the destination. The highest priority node is a neighbor which has the lowest

cost (hop count). The rule is specified in such a way that a message will always go in a direction toward the destination. The second algorithm is a variant of the first algorithm, where it allows to go in a direction that does not directly route towards the destination if there is a shorter path to the destination from the current node. For example, instead of routing from left to right, it could be faster to go up a few levels, then go right and go down to the destination. The complexity of both algorithms is $O(\log_k n)$, where n is number of nodes and k is number of fan-outs. Routing with the shortest path may not be the best solution in a failure situation. The direction of the message may be changed too often such that the message is moving further from the destination. The third algorithm intends to prevent this situation by using knowledge of previously detected dead nodes from the header to compute the cost. The third method uses a graph-coloring technique of breath first search, which explores only alive neighbor nodes. However, this algorithm requires complexity $O(n + (k + 3))$, where n is number of nodes and k is number of fan-outs.

4 Protocol Verification

The main reason for the verification is to ensure that the design of the protocol did not exhibit any potential problems. The protocol has been modeled with the PROMELA [16] specification language, which is the input of the SPIN [17] verification tools. PROMELA (Process Meta Language) is a non-deterministic language, which provides a method for making abstractions of distributed system protocols. It supports dynamic creation of concurrent processes, both synchronous and asynchronous message passing via communication channels, message loss and duplicate simulation and several other features. SPIN is a model checker for asynchronous systems using an automata-theoretical. It checks for deadlocks, livelock (non-progress cycles) and non-reachable states in the entire state space. It can verify and simulate several correctness properties. If an error is found, SPIN will provide a counterexample to show a circumstance that can generate the erroneous state.

4.1 Specifying the Protocol in PROMELA

Due to the fact that the PROMELA language is based on point to point communication, there must be as many channels as nodes in order to model the broadcast system. Each node will exclusively receive messages only through this channel. They will use corresponding channel associated with the node to send messages. All the nodes will wait in a loop with the *do* repetition construct. The root node starts sending the initial messages. If a node gets a message, it will check the message type and execute portions of code corresponding to procedure rules in Section. 3.1. For simplicity reason, we use a new feature of SPIN version 4 which can include embedded C code fragments (with PROMELA's *c_code* construct) to compute node depth, neighbor IDs etc. The link failure is simulated with non-deterministic selection capability of the *if* selection construct.

The SPIN verifier and simulator will randomly choose the status (up or down) of links between a node and its neighbors while the node is trying to send a message on to the next hop. In order to speed up the verification process, we reduce the size of state space by using an *atomic* construct to atomically execute its code section which represents internal computation without interleaved execution with other processes.

4.2 Verification Results

The results were conducted on a PentiumIII 550MHz, with Spin 4.2.6 on Linux. The search depth bound was 10,000 and the memory limit was 512 MB. A deadlock was discovered from the original modeling. However, after closer examination, it turns out that TCP buffer size of the communication channel in the modeling was too small. When the deadlock problem was solved, no deadlock, livelock, invalid end state, unreachable codes and assertion violation were found during verification.

5 Experimental Results

The protocol performance was evaluated in both normal and failure modes. In the case of no failure, it is obvious that the average number of hops for multicast messages decreases when the number of fan-outs increases (i.e. closer to a flat tree). On the other hand, the average number of steps to complete the message transfer for broadcast increases when the number of fan-outs increases (except that 3-ary is better than 2-ary due to more parallelism).

During the failure mode, the dead nodes (D) are obtained from combinations of all possible nodes (N) i.e. $\binom{N}{D}$, where source node $\notin$ D. Fig 2(a) illustrates that both variant rule-based and dead node aware algorithms are scalable with unicast messages (multicast to one destination). The higher values of fan-out yields the worst performance, especially with the basic rule-based algorithm, because it has more chances to go in a direction toward a dead node. Fig 2(b) depicts that a dead node has only a small effect on the performance of a broadcast message. The results show that the basic and variant rule-based algorithms produce performance close to the dead node aware algorithm, but the rule-based algorithms are much simpler to the model e.g. a broadcast [5] with a single dead node on an AMD 2GHz machine, the simulation time of dead node aware is 15 minutes, while the basic and variant rule-based took only about 30 seconds.

6 Conclusions and Future Works

The scalable and fault tolerant protocol for parallel runtime environments was designed and developed to support runtime environments of MPI implementations. The design of the protocol has been formally proven to work under both

[5] 16K bcast, we model 16383 different network topologies

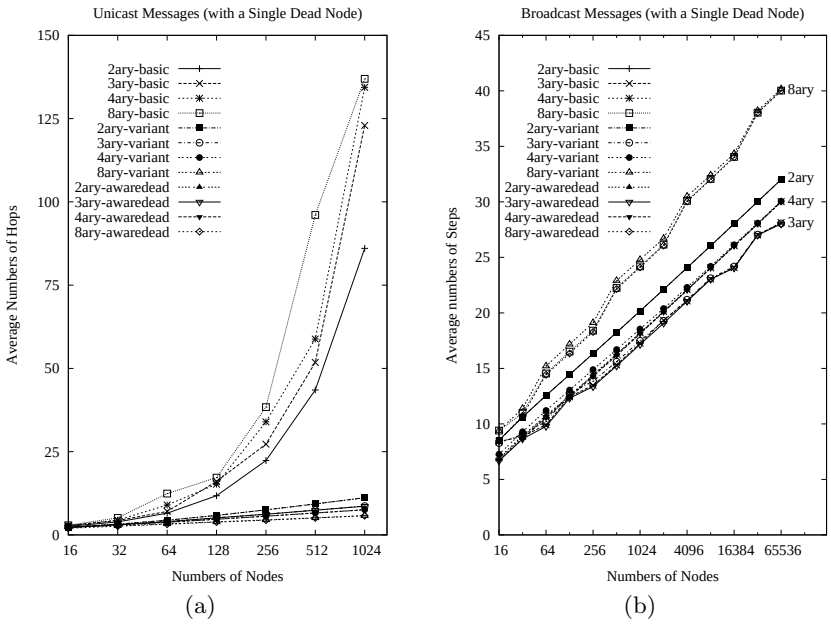

Fig. 2. Message transmission during failure situations. (a) Unicast (b) Broadcast

normal and failure modes. The performance results indicate that the variant rule-based algorithm is the best choice in terms of the shortest path (and simulation computation time as well).

There are several improvements that we plan for the near future. Making the protocol aware about the underlying network topology (in both LAN and WAN environments) will greatly improve the overall performance for both broadcast and multicast message distribution. This is equivalent to adding a function cost on each possible path and integrating this function cost to the computation of the shortest path. A faster and more accurate re-routing algorithm is in development. At a longer term, we expect this protocol to be the basic message distribution of the runtime environment within the FT-MPI and Open MPI runtime systems.

Acknowledgement. This material is based upon work supported by Los "Alamos Computer Science Institute (LACSI)", funded by Rice University Subcontract No. R7B127 under Regents of the University Subcontract No. 12783-001-05 49 and "Open MPI Derived Data Type Engine Enhance and Optimization", funded by the Regents of the University of California (LANL) Subcontract No. 13877-001-05 under DoE/NNSA Prime Contract No. W-7405-ENG-36

References

1. M. Beck, J. J. Dongarra, G. E. Fagg, G. A. Geist, P. Gray, J. Kohl, M. Migliardi, K. Moore, T. Moore, P. Papadopoulous, S. L. Scott, and V. Sunderam. HARNESS:

A next generation distributed virtual machine. *Future Generation Computer Systems*, 15(5–6):571–582, 1999.
2. G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings Supercomputing Symposium*, pages 379–386, 1994.
3. R. Butler, W. Gropp, and E. L. Lusk. A scalable process-management environment for parallel program. In *Proceedings of the 7th European PVM/MPI User's Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 168–175, London, UK, 2000. Springer-Verlag.
4. R. H. Castain, T. S. Woodall, D. J. Daniel, J. M. Squyres, B. Barrett, and G. E. Fagg. The open run-time environment (openrte): A transparent multi-cluster environment for high-performance computing. In *Proceedings 12th European PVM/MPI User's Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Sorrento(Naples), Italy, September 2005. Springer-Verlag.
5. J. J. Dongarra, H. Meuer, and E. Strohmaier. TOP500 supercomputer sites. *Supercomputer*, 13(1):89–120, 1997.
6. G. E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, Z. Chen, J. Pjesivac-Grbovic, K. London, and J. Dongarra. Extending the mpi specification for process fault tolerance on high performance computing systems. In *Proceedings of the International Supercomputer Conference (ICS) 2004*, Heidelberg, Germany, June 2006. Primeur.
7. 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 User's Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104, Budapest, Hungary, September 2004. Springer-Verlag.
8. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high - performance, portable implementation of MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.
9. I. Gupta, R. van Renesse, and K. Birman. Scalable fault-tolerant aggregation in large process groups. In *Proceedings of The International Conference on Dependable Systems and Networks (DSN)*, pages 433–442, 2001.
10. MPI Forum. MPI: A message-passing interface standard. Technical report, 1994.
11. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. Technical Report TR-00-010, Berkeley, CA, 2000.
12. R. V. Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. Technical Report TR98-1687, 28, 1998.
13. A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.
14. I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.
15. B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.
16. Holzmann, G.J.: Design and validation of computer protocols. Prentice Hall (1991)
17. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering **23** (1997) 279–295