Routing and Resource Discovery in Phoenix Grid-Enabled Message Passing Library

Kenji Kaneda
University of Tokyo / PRESTO, JST
kaneda@yl.is.s.u-tokyo.ac.jp

Kenjiro Taura
University of Tokyo / PRESTO, JST
tau@logos.t.u-tokyo.ac.jp

Akinori Yonezawa
University of Tokyo
yonezawa@yl.is.s.u-tokyo.ac.jp

Abstract

We describe design and implementation of a "Grid-enabled" message passing library, in the context of Phoenix message passing model. It supports (1) message routing between nodes not directly reachable due to firewalls and/or NAT, (2) resource discovery facilitating ease of configuration that allows nodes without static names (e.g., DHCP nodes) to participate in computation without specific efforts, and (3) nodes dynamically joining/leaving computation at runtime. We argue that, in future Grid environments, all of the above functions, not just routing across firewalls, will become important issues of Grid-enabled message passing systems including MPI. Unlike solutions commonly proposed by previous work on a Grid-enabled MPI, our system runs a distributed resource discovery and routing table construction algorithm, rather than assuming all such pieces of information are available in a static configuration file or alike. Experimental results using 400 nodes in three LANs indicate that our algorithm is able to dynamically discover participating peers, connect them each other, and calculate a routing table. The elapsed time of our algorithm is only approximately twice as long as that of offline route calculation that just connects nodes based on a fully given configuration.

1. Introduction

Message passing model is a dominant programming model for high performance parallel computation involving a large number of (e.g., > 100) nodes. It may be even more so in future multi-clusters and/or the Grid environment, where the programmer carefully needs to optimize communication of applications. Thus it is natural for researchers on HPC to seek a message passing library suitable for such environments.

There have been a great deal of work with this end, most of which aim at building "Grid-enabled" MPI libraries [2, 4, 6, 10, 11]. A primary design/implementation issue is how to deal with the fact that nodes may not be directly reachable in the underlying communication layer (e.g., TCP). This issue arises due to IP filtering as well as NAT/DHCP.

As far as we know, all existing systems essentially let a user specify the routes offline (e.g., in a configuration file). Most typically, a configuration file groups nodes and specifies a gateway node (either for each group or for the entire nodes) via which messages between directly unreachable nodes are routed. This solution is simple to implement and feasible for a small number of nodes distributed over a couple of clusters. The solution, however, must be generalized and extended in several ways for future environments, as discussed below.

One obvious issue is a scalability limitation due to gateways. A user should be able to specify as many gateways as permitted by a network administrator, rather than just one for each cluster. More important, having more resources spread over the Grid implies that resource selections tend to become more dynamic and adaptive. It will thus quickly become impractical for the user to maintain a *complete* resource description, which works for all possible set of resources that might be selected. Note that in the Grid setting, a complete description not only involves a list of resources, but also specifies routing (i.e., connectivity between nodes). Nodes that have dynamic IP addresses are more troublesome. Though they are able to participate in computation with a suitable resource manager support, it would be difficult for the MPI user even to specify such nodes in what would be called a "complete" configuration file.

All in all, neither routing nor the names of participating nodes should be completely specified by the user; communication libraries must learn them whatever resources are selected by the scheduler.

To this end, we have developed a Grid-enabled communication library called Phoenix [24]. This paper describes design and implementation of its enhanced routing and peer discovery facility only briefly addressed in [24]. Specifically, it allows nodes to connect each other without initially knowing all the peer names participating in computation. Then the nodes build a routing table according to the resulting graph of connections. The initial knowledge of the nodes is only the names of a small (arbitrary) number of “hub” nodes, through which nodes learn names of other participating nodes and bootstrap the entire connection graph.

The mechanism is implemented in a fully dynamic fashion, in that it allows nodes to join and leave at an arbitrary point of execution. Such a fully dynamic peer discovery and routing table construction is mandatory if a parallel programming model supports dynamic processes (e.g., Phoenix, Dyn-MPI [25], PVM [8], and MPI2 [15]). In addition, the mechanism is a natural facility even if the model, *per se*, only supports static processes. This is because, as we have mentioned, dynamic and adaptive resource schedulers, and/or even a very primitive form of fault tolerance (e.g., a mechanism that avoids initially dead nodes) makes selected resources not completely predictable by a user. Migration of MPI jobs and fault-tolerant MPIs [3–5, 23] also need such mechanism since it enables nodes and networks to change dynamically.

Technically, our system consists of routing table construction and resource discovery. Thus, we borrowed basic ideas from a body of work on routing [20] and resource discovery [1, 9, 12, 13]. Specifically, our routing table construction algorithm is based on the Destination Sequenced Distance Vector (DSDV) routing algorithm [17], originally proposed in the context of mobile ad-hoc network routing. Our experiments indicate, however, that naively adopting the algorithm for our purpose does not scale because, in our setting, the connection graph is dense and/or the connection graph sometimes changes very rapidly (e.g., at start up). By carefully engineering propagation and scheduling of routing events, we dramatically improved its performance. We also show this is achieved even when nodes initially know only a small number of other processes. That is, resource discovery does not affect the performance.

The remainder of this paper is organized as follows. Section 2 reviews existing Grid-enabled MPIs. Section 3 describes our problem setting. Section 4 gives the details of the routing and resource discovery algorithm. Section 5 presents experimental results. Section 6 discusses related work. The final section summarizes the paper.

2. Grid-Enabled MPIs

2.1. Requirements

We summarize requirements on Grid-enabled communication systems , especially focusing on MPI. First, message forwarding routes must be shortest. For high performance communication, nodes must be able to transmit messages directly if possible. Second, they must work on various network topologies where many restrictions are imposed. Third, in many contexts, it is highly desirable for them to allow dynamic changes of the connection topology. This is true even if the computation model only allows a static number of processes. For example, many fault tolerant MPIs such as MPI/FT [3] have been developed. In these systems, crashed processes may be restarted on different machines. There are also systems that balance system loads adaptively (e.g., Dyn-MPI [25], dynamic load balancing on LAM/MPI [14]). These systems dynamically change the allocation of MPI ranks or the number of nodes that participate in computation. To support such systems, the routing mechanism must adapt to dynamic changes of nodes and connections.

2.2. Existing Systems

MPICH-V [4] is an automatic volatility tolerant MPI based on uncoordinated checkpoint/rollback and distributed message logging. It provides fault-tolerance as well as a communication mechanism that enables nodes to communicate across firewalls. To bypass firewalls, MPICH-V prepares Channel Memories (CMs), which must be globally reachable from all the nodes. The system enables nodes to communicate with one another by relaying messages via CMs. This indirect communication of MPICH-V has three drawbacks. First, every node always needs to communicate with each other via CMs even if they can communicate with each other directly. Second, the network topology that MPICH-V supports is limited. It requires as least one globally reachable node in networks. Third, it cannot tolerate dynamic changes on network topology since CMs are assumed to be fixed.

Stampi [10] and PACX-MPI [7] provide unified MPI interfaces for heterogeneous networks. They can exploit multiple clusters that may belong to different private networks. Stampi creates a message routing process that relays messages between them when machines in different clusters cannot communicate directly with each other through IP. PACK-MPI provides a similar facility by having proxies that handle inter-cluster communication. They cannot tolerate dynamic changes to connection topologies. In addition, they support only limited connection topologies; routing processes/proxies must be globally reachable.

MPICH/MADIII [2] offers a forwarding mechanism for inter-cluster communication. It automatically calculates forwarding routes for every machine using manually given information about the entire network. Since the forwarding routes are calculated statically at start up, MPICH/MADIII cannot tolerate dynamic changes of the connection topology.

3. Problem Setting

We assume the system assigns each participating node one or more *application level* names, or simply *logical* names. Node rank in MPI is an example of a logical name. The programmer uses logical names to specify message destinations. In contrast, a *physical* name refers to a name used to communicate in the underlying communication layer. For example, if we build MPI on top of TCP, a physical name is a pair $\langle$*hostname*, *port number*$\rangle$. The basic job of the communication library is to route messages with their destinations specified by logical names to the right destination node, even though it may not be directly reachable in the underlying communication layer.

As we mentioned in the introduction, we generalize the problem as follows. First, the communication library allows network connectivity to change at runtime. It changes routing accordingly. Second, it allows nodes to be added at runtime without initially knowing their (either physical or logical) names. When a new node joins a computation, the new node must know, of course, at least one physical name of an already participating node. On the other hand, any participating node does not have to know the new node in advance. Nodes may also be deleted at runtime. As mentioned previously, they are slight generalization of a minimally dynamic process model where resources are selected by the scheduler at a job start up depending on the availability and loads of resources.

Our system has been implemented in the context of Phoenix message passing model [24] developed by the authors. It facilitates writing scalable parallel algorithms accommodating dynamic processes. However, none of the algorithms described in this paper depend on the specifics of Phoenix model.

4. Routing and Resource Discovery Algorithm

4.1. Overview

We briefly sketch the behaviour of our algorithm using an example network illustrated in Figure 1. It consists of two subnets X and Y. Node a and e are gateways of X and Y respectively, and they have fixed names. Node b, c, and d are configured with DHCP in subnet X. Node f and g are also DHCP clients in subnet Y. They do not have any static names. Firewalls are installed on both subnets. They block connections between non-gateways belonging to different subnets. The only allowed connection across the firewalls is SSH [21] connection (port 22) between a and e.

The system roughly works as follows.

Step 1: Initial setup A set of processes bring up on resources chosen by a scheduler or a user. Each process knows physical names of *some* (not necessarily all) participating nodes. Let us assume in Figure 1, nodes only know the physical names of the two gateway nodes. This is a small piece of configuration information comfortably kept in each node or even passed upon command submission. The system currently supports three underlying communication protocols: direct TCP, OpenSSL [16], and SSH tunneling. SSH is useful in many cases where the only inbound connection allowed is SSH port (22).

Step 2: Overlay network construction Each node tries to establish connections to machines it knows. In Figure 1, the DHCP clients will succeed in establishing direct TCP connections to one of the gateways. The gateways also establish SSH connections between them.

Step 3: Resource discovery and routing table construction Each node constructs its routing table by exchanging messages on the overlay network. By looking up the routing table, each node determines which neighbor a message should be transmitted to. On the receipt of messages destined for other nodes, a node also looks up the table to determine a forwarding route.

Each node also learns new machines it initially does not know from messages it receives. When a node finds physical names it does not know, it retraces **step 2** and **step 3**. The steps are repeated until each node knows all the participating machines and all the possible connections are established. This mechanism can minimize the number of hops each message travels. For example, in Figure 1, the DHCP clients in the same subnet can eventually communicate with each other directly even if they initially do not know each other.

The system guarantees that each node eventually knows all the available machines if the graph is connected after the first execution of **step 2**.

Note that **step 2** and **step 3** interleave. Thus the system can route messages before the routing table is fully stabilized. Whenever the connection topology changes, the overlay network and the routing tables are updated. For example, suppose machine h is added to subnet Y, and h initially knows e. In this case, the overlay network is re-constructed and finally h becomes directly connected to e, f, and g.

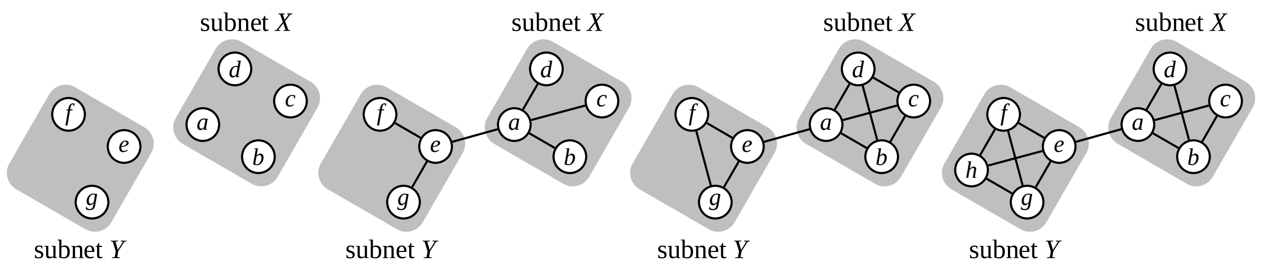

Figure 1. Process of route calculation. Nodes indicate machines, and solid lines indicate established connections between machines. (a) shows initial configurations. (b) shows an overlay network constructed only by the initial configurations. (c) shows an overlay network completely constructed when the system stabilizes. (d) shows a re-constructed network when h is added to the network.

4.2. Destination-Sequenced Distance-Vector Routing

Our routing algorithm is based on Destination-Sequenced Distance-Vector Routing (DSDV) [17] proposed for mobile ad-hoc networks. It gives us a good starting point because it adapts to changes of the connection topology and consumes a relatively small amount of memory compared to other schemes based on distance-vector. In DSDV, each routing table, at each node, lists all available destinations. Specifically, the entry for destination node v consists of:

- fwd: a node to which messages destined for v are forwarded.
- $nhops$: the number of hops of the route from the local node to destination v.
- seq: a sequence number that implies the freshness of the entry, as will be explained later.

Hereafter $R_u[v]$ is used to denote the entry for destination node v in u's routing table. $R_u[v].fwd$, $R_u[v].seq$, and $R_u[v].seq$ are used to describe fwd, $nhops$, and seq of $R_u[v]$ respectively.

To maintain the consistency of the routing table in dynamically varying topology, each node transmits (a subset of) its routing table to update its neighbor's routing table. Each node basically broadcasts when its routing table is updated by significant new information (e.g., discovery of a shorter path, break of a connection). On the receipt of a message, each node updates its routing table by the following rules: routes with larger sequence numbers are always preferred as the basis for making forwarding decisions; and of the path with the same sequence number, shorter routes are chosen.

To calculate the shortest paths without any loops, the sequence number is maintained in such a way that the most recently updated entry has the largest sequence number among all nodes. For example, when a node finds a broken link, the entries of which route depends on the broken link become obsolete. In such a case, the node broadcasts these entries with incrementing their sequence number to update the other nodes' routing tables correctly.

Note that the receipt of update messages may cause another transmission of update messages to make the routing table of all the nodes consistent. The message transmission is repeated until all the nodes in the network have received a copy of the update message with a corresponding metric.

4.3. Resource Discovery Algorithm

As described in Section 4.1, each node needs to discover available machines that it does not know in the beginning. Each node needs to collect information about available machines by exchanging messages with other nodes.

The node discovery is performed as follows. Initially each node only knows a part of machines participating in the application. When a node transmits a routing table message to update u's entry, it attaches u's physical name. On the receipt of this message, the receiver learns u's physical name, and tries to establish a connection to it.

4.4. Performance of the Naive DSDV

As we will show in Section 5, performance of a naively implemented DSDV is just poor when the number of nodes becomes large (> 50). We investigated this and found there are two primary reasons.

- Sending routing update messages to every neighbor result in many useless/redundant messages when the network is dense, as is usually the case in our problem setting.

- When the application brings up or when many nodes are simultaneously added to the application, the set of known node names as well as the topology of the graph change very rapidly. Naively running DSDV update protocol per each small change in the graph turns out to be a very inefficient way of calculating the final routing table, as we will see below.

For the first bullet, the topology of the overlay network in our problem setting is typically dense since many nodes can, and would like to, directly communicate with each another via the underlying communication layer. For example, the topology of the network consisting of multiple clusters is usually a collection of cliques. These dense networks cause a large number of redundant message transmissions. Suppose that one node updates its routing table. The minimum number of messages required to update all the routing tables is $N-1$ where N is the current number of nodes. This is because a message must be transmitted to each node at least once to deliver new information. On the other hand, the number of the messages that are transmitted until the naive DSDV stabilizes is $O(E)$ where E is the number of edges. This is because DSDV propagates update messages via all the edges. In dense networks, E is much larger than N (e.g., $E=\Omega(N^2)$). Thus the number of exchanged messages becomes large compared to the minimum $N-1$. We should point out this will not be a big issue in the context of mobile ad-hoc networks because the networks are typically sparse; neighbors of a node are limited to those physically close to the node. In contrast, connections are established between every allowed pairs of nodes in our problem setting.

For the second bullet, consider what will happen when many nodes are simultaneously added to the network (or when an application brings up). When a node accepts a connection from a new node or receives an update from a neighbor, it updates its routing table and sends update messages to neighbors. In general, this must be done promptly to propagate the new piece of information as fast as possible. When many nodes join an application almost simultaneously, however, updating neighbor nodes too eagerly result in many small messages that could have been merged when we know there will be subsequent updates.

4.5. Optimizations

We optimize the naive DSDV based on the two observations discussed above.

Eliminating redundant updates Suppose that node u transmits an update messages to its neighbors N_u. The algorithm guarantees that nodes in N_u do not propagate the update message to each other, as they get it from u anyways.

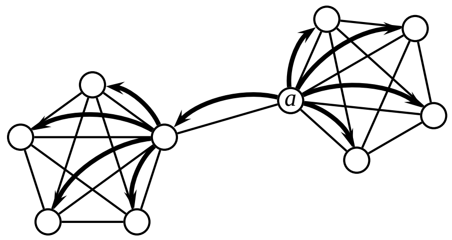

Figure 2. An example of elimination of redundant message transmission

This is implemented simply by adding two fields $trans$ and $rcpt$, to each entry of the routing table. $R_u[v].trans$ is the collection of nodes to which u has already transmitted $R_u[v]$, whereas $R_u[v].rcpt$ the collection of nodes that received or will soon receive the entry corresponding to $R_u[v]$ from some node [1].

Thus $w \in R_u[v].trans \cup R_u[v].rcpt$ indicates that u does not need to transmit a message to w; w has already received $R_u[v]$ or will soon receive $R_u[v]$. Node u must transmit $R_u[v]$ to u's neighbor w only if w does not belong to $R_u[v].trans \cup R_u[v].rcpt$.

Fields $trans$ and $rcpt$ are maintained as follows. When u sends $R_u[v]$ to a set of nodes V, V is added to $R_u[v].trans$. When w receives an entry e from u, $R_w[v]$ is updated as follows. If either $R_w[v].fwd$, $R_w[v].nhops$, or $R_w[v].seq$ is updated by this message, $R_w[v].trans$ becomes an empty set and $R_w[v].rcpt$ becomes a singleton that only contains sender u. Otherwise sender u and $e.trans$ are added to both $R_w[v].trans$ and $R_w[v].rcpt$.

Figure 2 shows an example of the elimination of redundant message transmission. Let us consider entries for destination v. Suppose that for all u both $R_u[v].trans$ and $R_u[v].rcpt$ are initially empty. Then a broadcasts $R_a[v]$, which is freshest among all the nodes. As the arrows in Figure 2 indicate, only a needs to broadcast update messages to make all the nodes' entry fresh.

Merging clustered updates Our second optimization tries to merge messages for many routing table updates that occur almost simultaneously. Simply buffering messages for a fixed period would sacrifice performance when an update occurs in isolation. Thus we address the problem by the following scheduling policy of events related to routing.

1. If a node knows another node but does not have a connection to it, it connects to the node with the highest priority.

[1]To identify each node uniquely, we basically use an IP address. When IP addresses are not unique among nodes, random bits are added to each node's identifier.

Table 1. Experimental environment

	CPU	# of procs
subnet A	UltraSPARCIII 750MHz	2CPU x 112 nodes
subnet B	Xeon 2.40GHz	2CPU x 64 nodes
subnet C	PentiumIII 800MHz	2CPU x 16 nodes
	PentiumIII 1.4GHz	1CPU x 16 nodes

2. If a node connects to all its acquaintance, but has some unprocessed messages updating its local routing table, it processes these messages.

3. Otherwise, it sends update messages to its neighbors.

In short, we give priorities to routing-related events in the following order. (1) making connections, (2) updating the local table, and (3) propagating updates to the neighbors' tables. Here, all updates that have not been propagated to a neighbor are merged into a single message.

5. Experiments

Table 1 summarizes the experimental environment. Machines in the same subnet can communicate directly with each other. The inter-subnet communication is restricted: each subnet has a gateway, which is the only machine that can accept inbound connections, at SSH port (22).

First we measured the elapsed time of routing table construction without node discovery. That is, we give all node names to each node offline (via a configuration file). We conducted the experiment on a single subnet (A) and the three subnets.

To begin with, let us confirm that the naive DSDV performs poorly, as shown in Figure 3. It does not scale at all when the number of processes become > 50. Thus we removed it from further investigation. Our interest is the price we pay for supporting the general, fully dynamic process model. So we compared our algorithm with two "easier" cases where processes are assumed to be static, or the process configuration is completely given offline.

Figure 4 shows the result. The upper graph is for the single subnet experiment and the lower graph for the three subnets case. For the latter, we proportionally mixed CPUs from the three subnets [2]. The curve labeled "offline" is the simplest and the easiest setting. A configuration file describes a complete process configuration (which process should connect to which) and processes simply follow it. Thus, the elapsed time is mostly of just establishing connections. The curve labeled "master" assumes processes are static and their names known to every process, but connectivities between nodes are not known. It also assumes

[2]UltraSPARCIII 750MHz : Xeon 2.40GHz : PentiumIII 800MHz : PentiumIII 1.4GHz $= 14 : 8 : 2 : 1$)

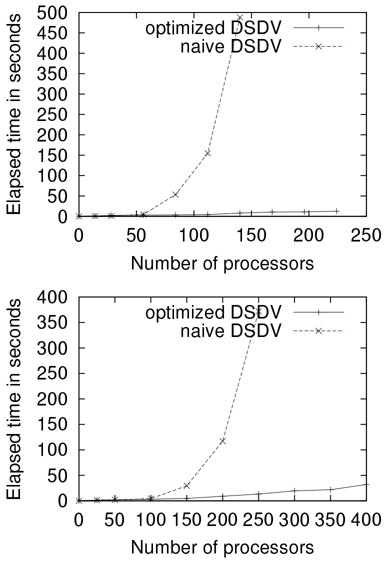

Figure 3. Naive DSDV compared with our optimized DSDV in a single subnet (upper) and three subnets (lower)

there is a master node and every process knows the path to the master node. Under this assumption, each process tries to connect to all other processes, learns its neighbors, and sends their names to the master. The master collects the messages and then calculates the all-to-all shortest paths. It finally sends the result to all processors.

When the number of processors is less than 100, all three cases have the approximately equal elapsed time. That is, our dynamic routing table construction has almost no overhead. Up to 400 processors, the elapsed time of our algorithm is within a factor of 2.3 of the offline case, and 1.5 of the master case.

Recall that our routing algorithm is fully dynamic, which means message send/receive can take place before the routing table is completely stabilized. Figure 5 gives us a sense of how much of the routes become "ready" at which point of calculation. The upper graph shows how much node pairs out of all the possible N^2 pairs are reachable at each moment (either directly or indirectly). All nodes are in a single cluster. As we can see, although it took 14 seconds to completely stabilize the routing table, more than 90% of node pairs become reachable at 10 second. The lower graph shows the average number of hops between reachable pairs at each moment. It should converge to one, and we almost get there at 10 second. In summary, it is fair to say most of the work has been done much earlier than the completion of the routing table construction. Figure 6 shows how routes become ready on a dynamically changing network. In this experiment, we began with 224 processes and killed a half

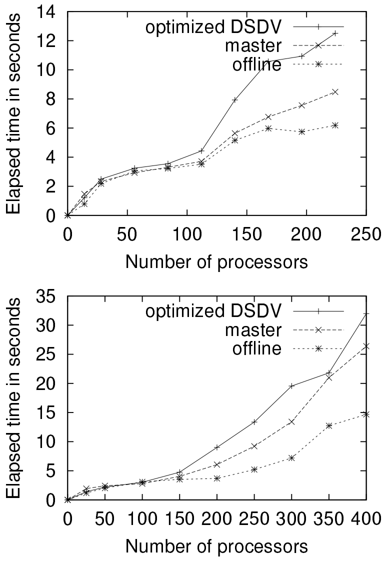

Figure 4. Comparison of our fully dynamic DSDV with two static cases (see text) in a single subnet (upper) and three subnets (lower)

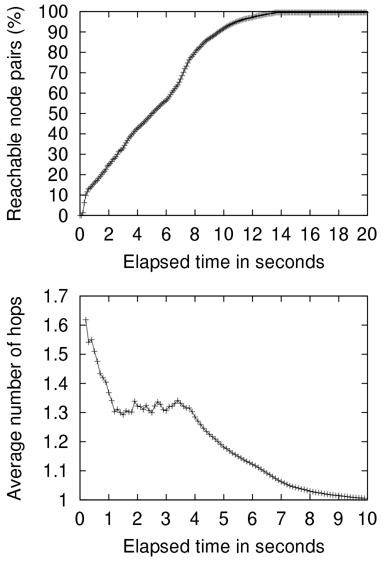

Figure 5. The fraction of reachable node pairs (upper) and the average number of hops between reachable node (lower)

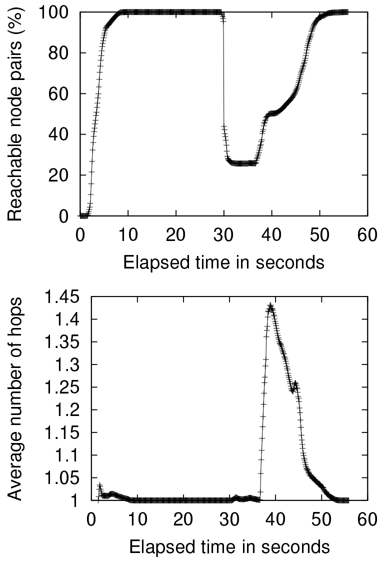

Figure 6. The fraction of reachable node pairs (upper) and the average number of hops between reachable node (lower) on a dynamically changing network

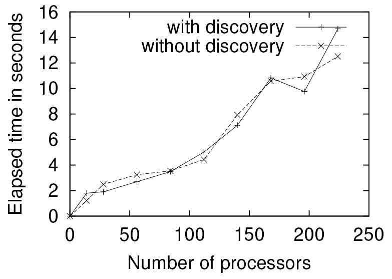

Figure 7. The elapsed time of the routing table construction with/without node discovery

of the processes after 30 seconds have passed. Then after another 5 seconds have passed, we restarted the killed 112 processes. The result shows that the routing tables can be updated rapidly according to the addition/deletion of processes.

Finally, Figure 7 compares cases with or without node discovery. The result shows that the overhead of node discovery is negligible.

6. Related Work

6.1. Peer-to-Peer Information Sharing Systems

Peer-to-Peer information sharing systems such as Pastry [19], Tapestry [26], Chord [22] and CAN [18] provide a distributed shared hash table. These systems are completely decentralized and self-organizing: each node automatically adapts to arrival, departure, and failure of nodes. They proposed efficient routing algorithms for looking up and inserting items in the hash table. For example, Pastry assigns each node a unique ID. Then it routes a message to the node with a node ID that is a numerically closest to the destination address of the message. This algorithm notably reduces the size of the routing table and the number of routing table update messages.

However, their algorithms assume that every node can communicate directly with one another and that nodes can always forward messages to appropriate nodes. Thus they cannot work on environments where direct communication may be prohibited by security policies (e.g., firewalls). In addition, these systems require forwarding messages via multiple nodes (e.g., in Pastry, $O(\log N)$ hops where N is the number of nodes in its network) even if nodes can communicate with one another directly. Since this unnecessary forwarding degrades communication performance heavily, their algorithms are not feasible for high-performance computing that involves dense communication.

6.2. Resource Discovery

A resource discovery problem introduced by Harchol-Balter, Leighton and Lewin in [9] is relevant for our algorithm. The resource discovery problem is to efficiently discover all the nodes that currently exist in the systems when each node initially knows only a small number of nodes. Though several algorithms for the resource discovery problems are proposed [1, 12, 13], they do not suffice for our system. This is because they focus on only discovering node names and do not care routing.

7. Summary

We have described a communication subsystem for message passing systems for the Grid. It provides routing and resource discovery that tolerate dynamic changes of connection topologies. We evaluated the performance of the algorithm by running the system on 400 nodes in three LANs. When the number of processors is less than 100, our dynamic routing table construction adds almost no overhead to the case where the network connectivity is completely given offline. In all cases, the elapsed time of our algorithm is within a factor of 2.3 of the offline case. Furthermore, 90% of node pairs become reachable much earlier than completion and messages can be routed when the routing table is being constructed.

References

[1] I. Abraham and D. Dolev. Asynchronous Resource Discovery. In *Proc. of PODC*, pages 143–150, 2003.

[2] O. Aumage and G. Mercier. MPICH/MADIII: a Cluster of Clusters Enabled MPI Implementation. In *Proc. of CCGrid*, pages 26–33, 2003.

[3] R. Batchu, J. Neelamegam, Z. Cui, M. Beddhu, A. Skjellum, Y. Dandass, and M. Apte. MPI/FT^{TM}: Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance-Portable Parallel Computing. In *Proc. of CCGrid*, pages 26–33, 2001.

[4] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, and F. Magniette. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *Proc. of SC*, pages 1–18, 2002.

[5] E. Fagg and J. J. Dongarra... FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world. In *Lecture Notes in Computer Science*, 2000.

[6] I. Foster and N. Karonis. A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. In *Proc. of SC*, page 46, 1998.

[7] E. Gabriel, M. Resch, T. Beisel, and R. Keller. Distributed Computing in a Heterogeneous Computing Environment. In *Proc. of EuroPVM/MPI*, pages 180–187, 1998.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[9] M. Harchol-Balter, T. Leighton, and D. Lewin. Resource Discovery in Distributed Networks. In *Proc. of PODC*, pages 229–237, 1999.

[10] T. Imamura, Y. Tsujita, K. Koide, and H. Takemiya. An Architecture of Stampi: MPI Library on a Cluster of Parallel Computers. In *Proc. of EuroPVM/MPI*, pages 200–207, 2000.

[11] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *Proc. of PPoPP*, pages 131–140, 1999.

[12] S. Kutten. Asynchronous Resource Discovery in Peer to Peer Networks. In *Proc. of SRDS*, pages 224–231, 2002.

[13] S. Kutten, D. Peleg, and U. Vishkin. Deterministic Resource Discovery in Distributed Networks. In *Proc. of SPAA*, pages 73–83, 2001.

[14] M. Matsubara, K. Suzuki, and A. Katsuno. Dynamic Load Balancing in HPC applications for Autonomic Computing (in Japanese). In *Proc. of SACSIS*, pages 349–356, 2003.

[15] MPI-2: Extensions to the Message-Passing Interface. http://www-unix.mcs.anl.gov/mpi/.

[16] Open SSL. http://www.openssl.org/.

[17] C. Perkins and P. Bhagwat. Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. In *Proc. of SIGCOMM*, pages 234–244, 1994.

[18] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. of SIGCOMM*, pages 161–172, 2001.

[19] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of Middleware*, pages 329–350, 2001.

[20] E. Royer and C.-K. Toh. A review of current routing protocols for ad-hoc mobile wireless networks. *IEEE Personal Communications Magazine*, 6:46–55, 1999.

[21] Secure Shell. http://www.ssh.com/.

[22] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications. In *Proc. of SIGCOMM*, pages 149–160, 2001.

[23] Y. Takamiya and S. Matsuoka. Towards MPI with user-transparent fault tolerance (in Japanese). In *Proc. of JSPP*, pages 217–224, 2002.

[24] K. Taura, K. Kaneda, and T. Endo. Phoenix: a Parallel Programming Model for Accommodating Dynamically Joininig/Leaving Resources. In *Proc. of PPoPP*, pages 216–229. ACM, 2003.

[25] D. B. Weatherly, D. K. Lowenthal, M. Nakazawa, and F. Lowenthal. Dyn-MPI: Supporting MPI on Non Dedicated Clusters. In *Proc. of SC*, 2003.

[26] B. Y. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. UCB Tech. Report UCB/CSD-01-1141, University of California Berkeley, 2001.