A Scalable and Efficient Self-Organizing Failure Detector for Grid Applications

Yuuki Horita
University of Tokyo
horita@logos.ic.i.u-tokyo.ac.jp

Kenjiro Taura
University of Tokyo / JST
tau@logos.ic.i.u-tokyo.ac.jp

Takashi Chikayama
University of Tokyo
chikayama@logos.ic.i.u-tokyo.ac.jp

Abstract— **Failure detection and group membership management are basic building blocks for self-repairing systems in distributed environments, which need to be scalable, reliable, and efficient in practice. As available resources become larger in size and more widely distributed, it is more essential that they can be easily used with a small amount of manual configuration in Grid environments, where connectivities between different networks may be limited by firewalls and NATs.**

In this paper, we present a scalable failure detection protocol that self-organizes in Grid environments. Our failure detectors autonomously create dispersed monitoring relationships among participating processes with almost no manual configuration so that each process will be monitored by a small number of other processes, and quickly disseminate notifications along the monitoring relationships when failures are detected. With simulations and real experiments, we showed that our failure detector has a practical scalability, a high reliability, and a good efficiency. The overhead with 313 processes was at most 2-percent even when the heartbeat interval was set to 0.1 second, and accordingly smaller when it was longer.

I. Introduction

Failure detection and group membership management are basic components to support distributed applications that can autonomously recover from faults (crash) of participating processes. With available Grid resources becoming larger in size and more widely distributed, efficient and scalable systems for failure detection and group membership management are becoming more important. In today's practice, such systems are used both for parallel programming libraries such as PVM [1] and MPI [2], [3] and resource monitoring services [4], [5]. Desirable features of such systems include low overhead, (semi-)automatic self-organization, absence of a single point of failure, scalability, detection accuracy and speed. Many of existing implemented systems either use a simple all-to-one or all-to-all heartbeating schemes that lack scalability [2], [3] or require complex manual configuration [4], [5]. Our system described herein is based on recent advances in scalable failure detectors and group membership protocols [6], [7] and addresses several issues of practical importance. Algorithms described in the literature of this category rarely come with evaluation of real implementation, and often make simplifying assumptions that become issues in practice. For example, they typically assume messages (connections) are never blocked between live nodes. This is not true in the presence of firewalls and NAT routers, both of which are common in real Grid environments. For another example, they often ignore the fact that in practice, sending a message to a node for the first time generally involves establishing a TCP connection, which is much more costly than sending subsequent messages over established connections. TCP is necessary, again because UDP is more often blocked between administrative domains than TCP is. See discussion in Section IV for some other reasons. Therefore practical systems need to pay attention not only to the total traffic per unit time, but also to how many nodes each node should *ever* send a message to. Systems based on simple gossiping are not enough in this respect [8].

This paper presents a scheme that overcomes these issues and reports both simulation and real experimental results. Our system can be concisely described as follows.

- It follows the basic idea of [6] for failure detection, in which each node is monitored by a randomly chosen small number of nodes (typically, 4 or 5).
- A node is monitored by TCP connections and heartbeats on them. A process crash is detected by a connection reset, and a machine crash by absence of heartbeats.
- Once process or machine faults are detected, notifications are quickly disseminated along the established TCP connections by simple flooding.
- It works with almost no manual configuration about network connectivity (firewalls and NATs).

Overall, our system will be useful as a library for supporting parallel applications that tolerate process/machine crashes and/or use dynamically changing set of resources [9]. This is the primary intended purpose of this work. We however believe it will also be useful as a basic building block for flexible and easy-to-use resource monitoring services.

The remainder of this paper is as follows. Section II discusses what failure detection should suffice to help fault-tolerant distributed applications and reviews the existing techniques in Section III. We propose our basic protocol for local-area networks in Section IV, and extend it for wide-area networks in Section V. We state simulation evaluation and experimental results in Section VI. Finally, Section VII presents summary and future work.

II. Background

We consider processes communicating with each other via point-to-point asynchronous messages. We do not assume the underlying communication layer's support for multicast. Each

0-7803-9493-3/05/$20.00 © 2005 IEEE

process may fail only by crashing (a process crash / a node crash).

Failure detectors generally detect failures by the absence of *heartbeats*, messages periodically sent by the processes they monitor. We use this basic approach, along with the operating system's support of automatically closing a TCP connection upon a process death. Heartbeating has a few known limitations, such as that detection latencies must be long enough to reduce false positives, that heartbeats are delayed by many reasons that have nothing to do with faults (e.g., network delays, scheduling delays, paging, etc.), and that the presence of hearbeats does not necessarily imply the correct functioning of the remote processes (e.g., it may be producing wrong outputs). We nevertheless use this approach because it is simple to implement and seems a practical way to detect both common application-level errors that lead to process deaths and OS/hardware faults which are difficult to avoid the application-level efforts.

Desirable properties of failure detectors under this assumption are summarized as follows.

- **Demand 1 (Completeness):** Any failure (process or machine crash) is eventually detected by all normal processes.
- **Demand 2 (Accuracy):** The probability of false positives is low.
- **Demand 3 (Consistency):** All processes that are not declared as failed obtain consistent failure information including false positives.
- **Demand 4 (Detection Latency):** The latency between a failure and its detection is low.
- **Demand 5 (Scalability):** CPU and network loads are low for a large number of participating processes.

In addition, we demand practical failure detectors to work in the presence of blocked edges due to firewalls and NATs. We call this property *flexibility*.

- **Demand 6 (Flexibility):** Various network settings are tolerated. In particular, firewall or NATs may restrict connectivity between some live node pairs.

Finally, they must be accomplished without tedious and error-prone manual configuration.

- **Demand 7 (Adaptiveness):** Systems should bring up with a small amount of manually-supplied information about network settings.

Our system proposed in this paper achieves these demands in a practical sense.

For completeness, however, the algorithm described in this paper misses a failure in a certain probability, which is considered very rare in practice. In order to assure completeness, we can always resort to have a backup that detects it with accordingly high detection latency or low accuracy.

Our algorithm in this paper depends on the simple heartbeat strategy in which heartbeat parameters (T_{hb}/T_{to}) are defined statically. There have been more refined heartbeat methods, in some of which the parameters are dynamically configured according to network condition [10], [11]. We can easily integrate these approaches into our algorithm to enhance accuracy. Further discussion of this is beyond the scope of this paper.

III. Related Work

There have been efforts to integrate a failure detector in PVM/MPI library. PVM [1] and MPI/FT [3] adopt a fixed central manager to monitor all processes, which makes it simple to achieve consistency. This approach, on the other hand, clearly has an issue in scalability and completeness (i.e., because of a single point of failure).

In the context of resource monitoring services which have a more stringent demand for scalability, a hierarchical approach is often taken [4], [5], [12], [13]. In this approach, processes are divided into some groups reflecting proximity, with each group typically having a single monitoring process. This can reduce the scalability problem to a large extent while retaining ease of maintaining consistency. It however still suffers from the single point of failure in each group. Having two or more monitoring processes in each group is a possible remedy to this problem, but it brings up all the issues (specifically, consistency, accuracy, and low detection latency) that need an accordingly complex protocol to resolve. Moreover, this approach generally imposes a burden of configuring an appropriate group of processes on the user or the maintainer. Thus, this approach is not practically applicable to parallel programming libraries where participating resources may vary from one invocation to another. It may be amenable to permanent services such as resource monitoring systems that can justify the cost of deployment. We also believe, however, reducing the burden of deployment is important for such services to scale in practice.

Many advanced protocols in the literature are based on some form of Gossip protocol. Gossip is a probabilistic multicast protocol originally invented for maintaining consistency of large distributed databases [14]. [8] is the first to apply the protocol to failure detection. In [8], each process maintains a gossip list. A gossip list of process P is composed of a list of process identifiers, each along with the timestamp (*heartbeat counter*) on which P most recently listened from the process. Each process periodically sends its own gossip list to a randomly selected process. On receiving a gossip list, the receiver merges its own list with the received list by taking the maximum heartbeat counter for each entry. [8] showed that any piece of information will get through to all processes with high probability after each processes' gossiping $O(\log n)$ times. A member will be suspected as failed on each process if the corresponding heartbeat counter has not been updated for some timeout period.

However, this simple protocol has problems in scalability, detection latency, and consistency. Though gossip protocol can reduce the number of messages, it constantly causes a large network traffic because the size of a single gossip message is proportional to the number of live processes (i.e., *Scalability*). To keep the false positive frequency less, the timeout has to be sufficiently larger than $O(\log n)$ times as long as the

gossip interval. This leads to a longer detection latency than our approach (i.e., *Detection Latency*). In addition, since each process judges failure independently, a false positive will cause inconsistency in failure information (i.e., *Consistency*).

SWIM group membership protocol [6], [7] improves these shortcomings. It uses separate mechanisms for detecting failures and disseminating failure notifications. Each process periodically sends a ping message to a randomly chosen process and requests it to send a reply back. A failure of a process is thus detected by another process that first notices the absence of a reply from the process. That process disseminates notifications to all processes by a gossip protocol. Thus it can keep both CPU load and network traffic small in the steady state. SWIM also has a consensus mechanism to keep consistency of failure information, called *suspicion mechanism*.

When applied to real environments, it will suffer from the following problems which we address in this paper. First, it assumes ping messages to random targets are never blocked by firewalls or NATs (i.e., *Flexibility*). Second, it lets every process send ping messages to all processes over a period of time. In practice, both for pinging and dissemination of failure notifications, we would like to maintain a target that changes only slowly if at all. This is because sending a message to a node may need to establish a TCP connection between the sender and the receiver, which is much more costly than sending a message.

Directional Gossip [15] is a protocol that tries to address the flexibility problem. Each gossip server dynamically calculates the number of link-disjoint paths to each other gossip server that it can communicate with directly, with trace information of a message piggybacked on itself. If the number of link-disjoint paths to a process p is smaller than K, it always sends a message to p when it receives a new message. For the other processes, it gossips to them probabilistically.

They have not reported experimental results in real environment, and we note two potential issues that might arise when we deploy this protocol in real environments. First, in the beginning of the algorithm where processes have not yet learned many paths to other nodes, the behavior of the system is close to simple flooding to all neighbors, which may make scalability an issue. This is especially important in the context of parallel programming libraries where application startup time should be small. Second, this method requires a fairly involved calculation on every receipt of gossip messages, which may cause a large overhead.

IV. Basic Protocol

To simplify the exposition, this section describes our protocol assuming no edges between processes are blocked. That is, each process can communicate with any processes directly via the underlying transport. In practice the protocol in this section works within a single cluster. Section V describes its extension to the general setting. Our protocol is similar to [6], [7] being composed of two phases, *failure-detection phase* and *information-propagation phase*.

```
/* The algorithm run by a process p */
variable I : a set of the processes given manually in advance
variable M : a set of the processes that p monitors
variable H : a set of the processes that monitor p
variable A : a set of the processes that p is requesting
variable N : a set of the processes that p knows
variable F : a set of the processes that p considers failed
init(I) :
  A = M = H = F = ϕ
  N = {p}
  for q in I :
    call add_node(q)

procedure add_node(q) :
/* called when the process learns a new process q */
  With probability of k/|N| :
    send monitor_req(p) to q
    A = A ∪ {q}
  N = N ∪ {q}

procedure detect_failure(f, q) :
/* called when the process detects or learns a process f's death */
  if f ∉ F :
    call handle_failure(f)
    call flood(failure(p, f), q)

procedure handle_failure(f) :
  if f ∈ (H ∪ A) :
    pick q' randomly from (N − H − A)
    send monitor_req(p) to q'
    A = A ∪ {q}
  F = F ∪ {f}
  N = N − {f}
  M = M − {f}
  H = H − {f}
  A = A − {f}

procedure flood(m, q) :
/* forward m on all connections, except to its source, q */
  for q' in ((M ∪ H) − {q}) :
    send m to q'

Recv monitor_req(q) :
  send node_info(p, N) to q
  send monitor_ack(p) to q
  M = M ∪ {q}
  if q ∉ N :
    call add_node(q)
    call flood(node_info(p, {q}), q)
Recv monitor_ack(q) :
  H = H ∪ {q}
  A = A − {q}
  if |H| > k :
    pick q' randomly from H
    send monitor_cancel(p) to q'
    H = H − {q'}
Recv monitor_cancel(q) :
  M = M − {q}
Recv nodes_info(q, N') :
  N'' = N' − N
  if N'' ≠ ϕ :
    call flood(nodes_info(p, N''), q)
    for q' in N'' :
      call add_node(q')
Every T_hb :
  for q in H :
    send heartbeat(p) to q
Recv failure(q, f) :
  call detect_failure(f, q)
T_to has passed since p sent monitor_req(p) to q ∈ A :
  call detect_failure(q, ϕ)
T_to has passed since p received the last heartbeat(q) (q ∈ M) :
  call detect_failure(q, ϕ)
```

Fig. 1. Basic algorithm (run by a process p)

- *failure-detection phase* : process failures are detected by some other processes
- *information-propagation phase* : the failure information detected by some processes is propagated all over the non-faulty processes

We establish a graph of TCP connections among processes and use it both for failure detections and information propagation. Each process randomly selects a small number of (e.g., 4 or 5) processes, establishes a connection between itself and each of them, and asks each of them to monitor itself. A failure of the process is detected by these processes. A process detecting a failure propagates notifications on all the established TCP connections, which include those it monitors and those being monitored. The entire algorithm is shown in Figure 1. We describe the details of the two components below.

A. Failure Detection

Our protocol tries to maintain that each process is monitored by k other processes, with k being a configurable integer constant which is typically as small as 4 or 5. When a process crashes, one of k processes that monitor it, *a monitoring process*, will detect the failure (unless they crashed simultaneously), and propagate a notification to all normal processes as described in Section IV-B. The goal of the failure detection phase is to establish and maintain such monitoring relationships (i.e., connections) among processes.

When a new process joins the system, we assume each process eventually learns the process name (**add_node**). Our algorithm is independent from how this is achieved, but the simple flooding on the established connections will be a natural choice and hence described in Figure 1. The main points of this algorithm are as follows.

- When a process p learns that a new process q joined, p will ask q to monitor itself with probability $k/|N|$ where N is the current p's view of participating processes (**add_node**). If the number of processes that monitor p becomes larger than k, p will randomly pick a monitoring process and ask it to stop monitoring p (**send monitor_cancel**). This is necessary to balance monitoring load over all participating processes.
- The process that receives the request replies its acknowledgement and starts monitoring the source process (**Recv *monitor_req***). If a process p fails to connect to a process q or receives no response from q (i.e., q is dead), p will suspect q as failed and try another request. Otherwise (i.e., p receives the acknowledgement), p begins to send heartbeats to q (**Recv *monitor_ack***).
- When a process p learns that one of its monitoring processes crashed, p will try to repair the relationships by randomly selecting a sufficient number of processes and asking them to monitor it (**handle_failure**).

Figure 2 shows an example of monitoring relationships created by our algorithm. Every process will be monitored by k other processes. Because of random selection of monitoring processes, the monitoring relationships will be dispersed.

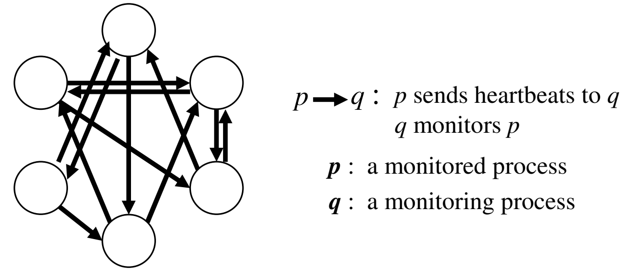

Fig. 2. Monitoring relationships : Each process is monitored by k other processes ($k = 2$ in this case). Since the monitoring processes are chosen randomly, the relationships will be well-dispersed like this.

Each process sends heartbeats to its k monitoring processes every T_{hb}, and the monitoring process will consider it failed if no heartbeat comes for a certain period of time T_{to} ($> T_{hb}$). Because each process is monitored by k processes, a failure of p is detected unless its k monitoring processes fail almost at the same time (i.e., before p repairs its set of monitoring processes).

In this protocol, we assume the use of TCP as a transport protocol as opposed to UDP that seems more commonly used for monitoring. We however believe there are a number of situations where TCP is more advantageous or is the only choice.

- Using TCP, regular process deaths, such as segmentation faults or exceptions due to programming errors (not hardware/OS failures), can be quickly detected by errors delivered by OS.
- UDP packets tend to be more often blocked by firewalls than TCP in wide-area network (WAN). In particular, packets on *established* TCP connections are rarely blocked. This means that even when a connection from p to q is blocked, p can still send a message to q as long as q can connect to p.
- There are a number of user-level tools to secure TCP connections such as SSL or SSH tunneling, which makes TCP more attractive for Grid setting.

Downside is that maintaining many TCP connections or updating (adding/removing) them frequently are resource consuming. Our algorithm tries to reduce the number of connections that must be maintained and does not update them unless the membership changes. Existing protocols [7], [8], if implemented naively on top of TCP, would end up with keeping connections between many node pairs over time, or closing and opening connections very frequently. This is because in these protocols, each process repeats sending messages to processes randomly chosen from all participating processes, even when membership does not change.

B. Information Propagation

Once a process p that monitors q detects the failure of q by the absence of heartbeats from q or connection reset by q, p propagates notifications that q is dead by flooding on all the established connections (**detect_failure**)). Since each process

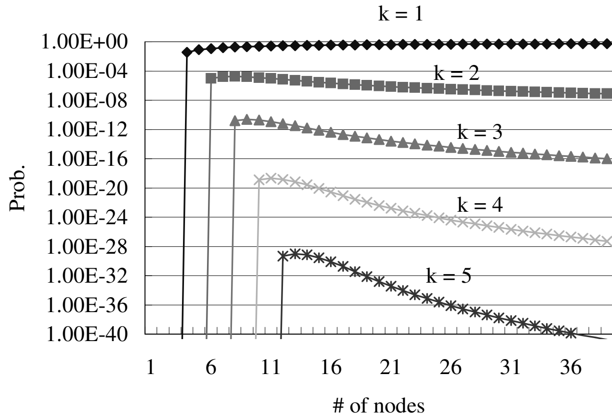

Fig. 3. The probability that the monitoring network is not connected in a complete graph

initiates connections to k processes, and floods on connections it initiated *as well as those it accepted*, the total traffic is $2kn$ messages. Since a broadcast takes $n-1$ messages in minimum, this is approximately a factor of $2k$ of the optimal in terms of the number of messages.

The reliability of this approach depends on the probability that the notifications get through to all processes. To partially assess it, we simulated the probability that the graph created above is not connected. Figure 3 shows the result with various values of k and the number of processes n. When $k \geq 3$, the highest probability occurs at $n = 9$ and it is $< 10^{-10}$. It also steadily decreases as n increases. As k becomes larger, the probability quickly goes down. Section VI-B assesses the reliability when multiple processes are crashed simultaneously.

Our approach is similar to some of the previous proposals based on random gossiping [7], [8]. [8] does not propagate failure notifications at all, but instead relies entirely on propagating heartbeats indicating *lives* of processes. [7] disseminates failure notifications using a Gossip protocol, but each process does not maintain a subset of processes to which they are sent. Instead, failure notifications are piggybacked with ping messages that are periodically sent to a randomly chosen process. In both of these approaches, the speed of a dissemination is less predictable than our approach and critically depends on intervals at which gossiping occurs.

Our dissemination method is similar to SCAMP [16], [17], a reliable multicast protocol. In [17], they showed that, if each process forwards a message to $\log n + c$ random neighbors, the probability that a multicast message gets through to all processes converges to $e^{-e^{-c}}$ (with $n \to \infty$). Our problem setting is similar to multicast, but different in an important point. In our setting, connections initiated by p and accepted by q are used in both directions. That is, they are used both for p to send notifications to q and for q to send notifications to p, whereas in conventional multicast settings, p's forwarding messages to q does not imply q's forwarding messages to p. This slight difference makes a difference in the number of connections we need. In fact, the non-vanishing probability of disconnections, $1 - e^{-e^{-c}}$, despite $\log n + c$ connections, is the probability that a *single* process is isolated from all the rest, having no processes forwarding messages to it. This never occurs in our protocol because the connections are used in both directions. In more general, since each process initiates k connections, a group of isolated processes, if any, is necessarily larger than k. The probability of having an isolated component in our setting is thus much smaller than that of having a non-reached node in conventional settings of multicast.

At this point, our propagation algorithm does not use any consensus algorithm to reduce false positives. That is, if any monitoring process declares a process to be dead, all other processes simply believe the notifications. Should this be an issue, our algorithm can readily incorporate a consensus mechanism, such as the one proposed in [7].

V. Extension to Wide-area Network

The reliability of the protocol presented in the previous section depends on the connectivity of the random graph, which in turn depends on the fact that a connection can be established to any randomly chosen process. This is typically the case within local area networks, but not across them. If we simply let each process select its monitoring processes *uniformly among those it can connect to*, chosen edges would be highly biased to intra-cluster edges. This may result in multiple isolated clusters. Consider for example a worst-case scenario in which we have two clusters each having a single gateway node and $m - 1$ internal nodes, and inter-cluster connections are allowed only between the two gateways. If each process choose their k monitoring processes among them, it is not likely that these gateway nodes happen to choose the other gateway as one of its monitoring processes. Its probability is:

$$\frac{\binom{m}{1}}{\binom{m+1}{k}} = \frac{k}{m+1}.$$

This section presents a simple approach to this problem, which is solely based on readily available information and thus needs no manual configuration specific to our algorithm. The basic idea is that in addition to each processes' choosing k monitoring processes within the cluster it belongs to, members of a cluster cooperatively establish at least k connections to each of the other cluster. The algorithm is concisely shown in Figure 4:

- A process behaves according to the basic algorithm shown in Figure 1 within the cluster it belongs to.
- If the number of connections between the cluster c_p a process p belongs to and another cluster c is less than k, p tries to connect to a process q, which is randomly chosen from those belong to c (**ensure_connectivity**).
- When a process p established a connection to a process q in another cluster (**Recv *monitor_req* / Recv *monitor_ack***), a notification about this connection (p, q) is broadcast to the cluster members by flooding within the cluster (**add_wconn / Recv *wconn_info***). This way,

```
/* The algorithm run by a process p */
c_x : a cluster that a process x belongs to
(x, y) : a connection between a process x and a process y
basic.func : a function (or a message event) func of basic protocol in Figure 1
variable F : a set of the processes that p considers failed
variable C : a set of the clusters
variable N(c) : a set of the processes belonging to a cluster c
variable W(c) : a set of wide-area connections between c_p and c
variable H_w : a set of the processes in other clusters that p monitors

init() :
  basic.init(N(c_p)) /* start basic protocol within this cluster */
  H_w = φ
  for c in (C − {c_p}) :
    ensure_connectivity(c)

procedure ensure_connectivity(c) :
  /* ensure there are at least k connections to a cluster c */
  if |W(c)| < k : /* there are less than k connections to a cluster c */
    pick q randomly from (N(c) − H_w)
    send monitor_req(p) to q

Recv monitor_req(q) :
  if c_q = c_p : /* within the cluster */
    basic.monitor_req(q)
  else : /* from another cluster */
    send monitor_ack(p) to q
    H_w = H_w ∪ {q} /* start monitoring this connection */
    call add_wconn(p, q, p)

Recv monitor_ack(q) :
  if c_q = c_p : /* within the cluster */
    basic.monitor_ack(q)
  else :
    H_w = H_w ∪ {q} /* start monitoring this connection */
    call add_wconn(p, q, φ);
    ensure_connectivity(c_q)

procedure add_wconn(p', q', q) :
  /* called when the process learns a new wide-area connection between
two processes p' and q' from a process q */
  W(c_q') = W(c_q') ∪ {(p', q')} /* add a connection entry */
  call basic.flood(wconn_info(p, p', q'), q) /* notify within this cluster */

Recv wconn_info(q, p', q') :
  /* notified of a new wide-area connection between p' and q' from q
  if (p', q') ∉ W(c_q') /* this is a new wide-area connection information */
    call add_wconn(p', q', q)

procedure detect_failure(f, q) :
  /* called when the process detects or learns a process f's death from a
process q */
  if f ∉ F : /* this is the first time to hear failure of f */
    if c_q = c_p : /* within the cluster */
      call basic.handle_failure(f)
    else :
      call handle_failure(f)
    call flood(failure(p, f), q)

procedure handle_failure(f) :
  H_w = H_w − {f}
  N(c_f) = N(c_f) − {f}
  F = F ∪ {f}
  for c in C : /* remove broken connections and ensure connectivity */
    W(c) = W(c) − {(*, f)} − {(f, *)}
    ensure_connectivity(c)

procedure flood(m, q) :
  /* forward m on all connections, except to its source, q */
  for q' in (H_w − {q}) : /* flood on wide-area connections */
    send m to q'
  call basic.flood(m, q) /* flood within this cluster */

Every T'_hb :
  for q in H_w :
    send heartbeat(p) to q

T'_to has passed since p sent monitor_req(p) to q (c_q ≠ c_p) :
  call ensure_connectivity(c_q)

T'_to has passed since p received the last heartbeat(q) (q ∈ H_w):
  call detect_failure(q, φ)
```

Fig. 4. The algorithm extended for wide area (Note : node discovery parts of the algorithm (like add_node in Figure 1) are omitted for simplicity; in other words, this algorithm is written on the assumption that each process knows all participating processes in advance)

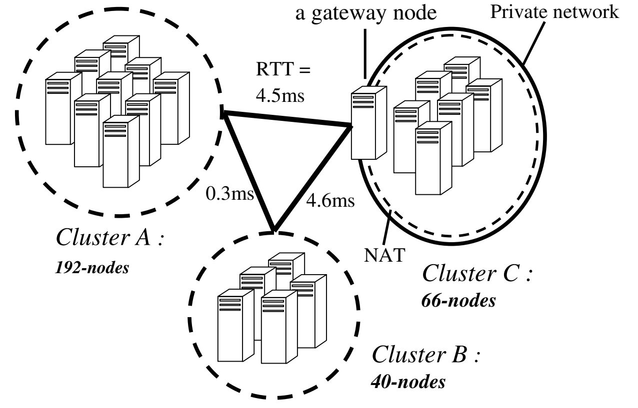

Fig. 5. Experimental environment (3-clusters)

members of a cluster share an approximate count of connections between the cluster it belongs to and each of the other clusters.
- A process stops attempting connections when it learns sufficient number ($\geq k$) of connections have been made from/to each cluster (**ensure_connectivity**).
- The connections established between clusters are monitored by both end processes with a heartbeat interval T'_{hb} and a timeout period T'_{to}.

This protocol is admittedly not scalable in the number of clusters, but seems a good and practical solution in typical Grid computing scenarios where a large-scale computation uses a modest number (e.g., < 10) of clusters each having a large number (e.g., several hundreds) of nodes.

A practical question that needs to be addressed is how to define a cluster and how processes know the cluster membership. For the purpose of our algorithm, a cluster can be any group of processes within which any process can connect to any other. To this end, we simply define a cluster to be a set of nodes having the same subnet mask. In actual implementation, when a message carries an endpoint name of a process (IP address and port number), it is augmented with its subnet mask. Each node can judge if two processes belong to the same cluster by examining their subnet masks.

VI. Experiments and Evaluation

In this section, we present results of experiments in the actual environment and simulations about the performance and the reliability of our protocol. We used three clusters as shown in Figure 5 for the experiments:

- ***Cluster-A*:** 191 processors (Xeon2.4GHz / Xeon2.8GHz, dual CPU)
- ***Cluster-B*:** 40 processors (Xeon2.4GHz / Xeon3.06GHz, single CPU)
- ***Cluster-C*:** 66 processors (Xeon2.4GHz, dual CPU)

In *Cluster-A* and *Cluster-B*, every node has a global IP address and is free to communicate with the outside processors (no firewall). On the contrary, *Cluster-C* utilizes NAT, in which all processors except one can communicate with the outside

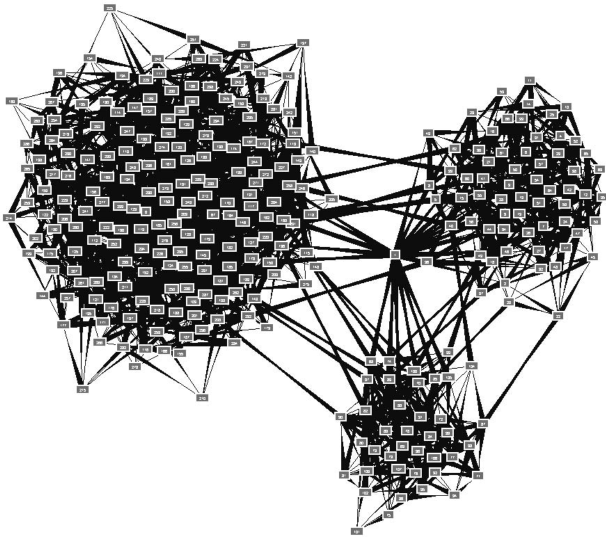

Fig. 6. Monitoring relationships constructed on 297-nodes in 3-clusters (k=5)

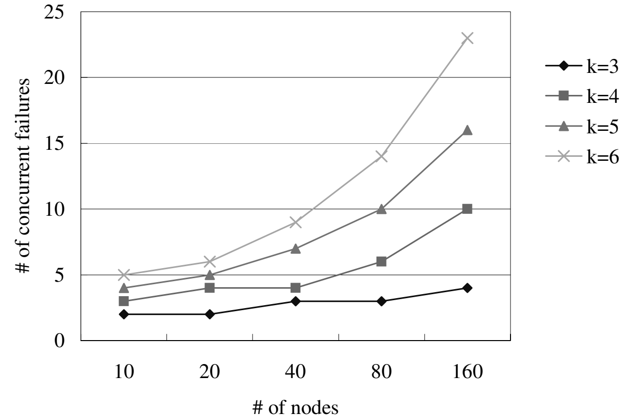

Fig. 7. Reliability of monitoring network : the maximum number of concurrent node-failures when the probability that monitoring network is not connected is less than 0.0001

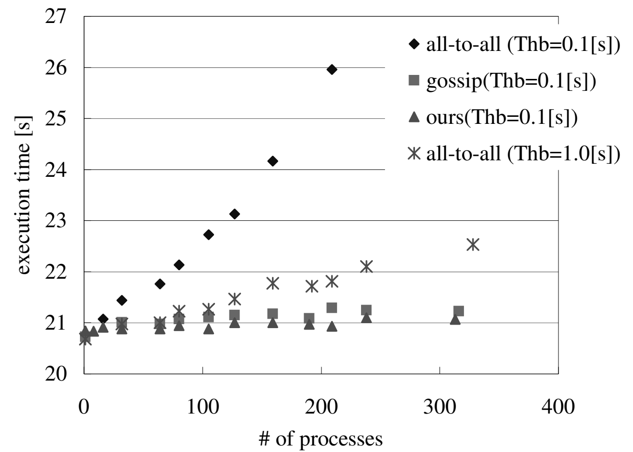

Fig. 8. Execution time of a compute-bound program under failure detection services

only from the inside. *Cluster-A* and *Cluster-B* are located in almost the same place, while *Cluster-C* is comparatively away from them (a round-trip time of about 4.5[ms]).

A. Construction of Monitoring Relationships

We ran our failure detectors, with k=5, on 297-nodes of three clusters. We gave to each process *three process names (one in each cluster)* as start-up information. This means that every process needed to learn other participating processes dynamically.

Figure 6 illustrates the monitoring network eventually constructed. A rectangle represents a process, and an edge represents a monitoring relationship that the thick-side process monitors the thin-side one. Within each cluster, each process was monitored by k other processes, and monitoring relationships were well-dispersed. It can be also seen that a little more than k (=5) connections were established between any pair of clusters. In the center of this figure, one process was connected with around k processes in each cluster. It corresponds to the process running on the gateway node of *Cluster-C*, which has two IP addresses, a global one and a private one for NAT. According to our definition of a cluster, it would belong to two clusters *C1 (global)* and *C2 (private)*. *C1* includes only the gateway node while *C2* includes all nodes of *Cluster-C*, and thus the gateway process came to establish k connections with each cluster.

B. Reliability of the Monitoring Network

We simulated the connectivity of the monitoring network when some node-failures occur simultaneously. We calculated the probability of disconnection of the monitoring network with 3.6×10^9 trials for each setting of parameters n and k.

Figure 7 displays the maximum number of node-failures when the probability of disconnection is less than 0.0001. In this graph, we can see that the reliability increases as n grows. The larger k is, the more quickly the reliability goes up. Large k may cause an additional overhead on applications, but within moderate bounds (e.g., $k < 10$) the penalty will be slight, because a process needs to handle only about $2k$ messages per unit time irrespective of n (See Section VI-C). Therefore it is reasonable to obtain high reliability according to need by setting k to a comparatively large value.

C. Overhead

We implemented two basic failure detectors, based on all-to-all heartbeating (*all-to-all*) and gossip protocol (*gossip*), besides our failure detectors (*ours*). We executed in *Cluster-A* a compute-bound program that finishes in about $20[s]$ under the condition that one of those kinds of failure detectors is running. The execution time was measured varying the number of processes n and heartbeat interval T_{hb}. In *ours*, we used 4 as the value of k.

Figure 8 depicts the results of the execution time. Each result represents the mean execution time of more than 30 runs. In the results of *all-to-all*, the execution time became longer as n increases. The overhead went up above 5-percent with 159-

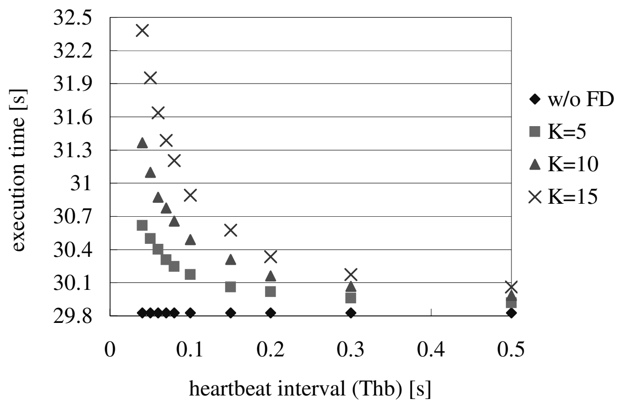

Fig. 9. System overhead

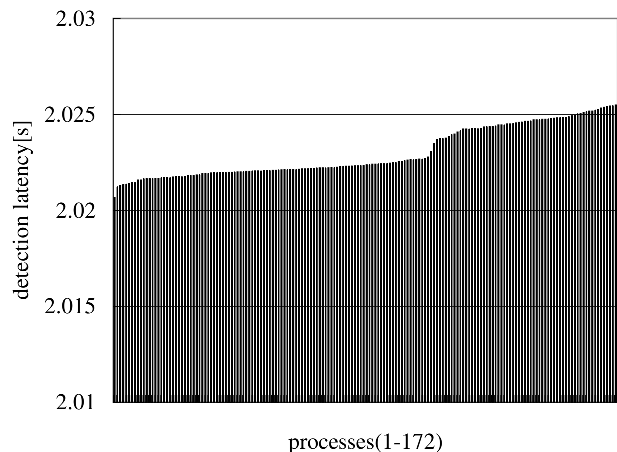

Fig. 10. Detection latency (k=3, T_{hb}=0.1[s], T_{to}=2.1[s], n=173 (one process crashed))

processes when T_{hb}=1.0[s], and reached 10-percent with 127-processes when T_{hb}=0.1[s]. All-to-all heartbeating imposes sending $O(n)$ messages every T_{hb} time on each process. This result indicates that lots of message handling will produce considerable CPU overhead. On the other hand, both *gossip* and *ours* had at most around 2-percent overhead regardless of n, even when T_{hb}=0.1[s] and n=313. In both methods, a process has to send only a small constant number of messages, and therefore showed scalability in CPU load.

In *gossip*, however, every message includes sets of an identifier and a heartbeat counter of all joining processes. We used 2-bytes for each element, so the message payload size was $4n$ bytes. When n=200 and T_{hb}=0.1[s], it would occupy 12.8[Mbps] bandwidth collectively (800[bytes] $\times$ 200 $\div$ 0.1 $\times$ 8=12.8[Mbps]). In *ours*, the message payload size was 1-byte regardless of n, for an identifier of message kinds. Since each process sends k messages every T_{hb}, it would occupy only $16k$[Kbps] in the same situation above. Thus our protocol is scalable in both CPU load and network traffic in fact.

Next, to investigate the effect of k, we measured CPU overhead of our failure detectors with different T_{hb} and k. Figure 9 shows the execution time of a compute-bound program on a 2.4GHz processor in *Cluster-B*, which terminates in about $30[s]$ without load, with our failure detectors running.

As we shortened T_{hb}, the overhead went up drastically, especially pronounced in larger k. If we chose double value of k, the overhead also came to approximately double. This clearly demonstrates that communication frequency determines the amount of the overhead. In practice, however, it is less meaningful to send heartbeats too frequently (e.g., $T_{hb} < 0.1[s]$), because T_{to} always needs to be at least several hundred milliseconds due to message delay by network congestion or high CPU load. In reasonable T_{hb}, the slight difference of k will be not so critical to the amount of overhead.

D. Detection Latency

We executed our failure detectors on two clusters (108-processes in *Cluster A* and 65-processes in *Cluster C*), with k=3, T_{hb}=0.1[s] and T_{to}=2.1[s], and killed an arbitrary process. We show in Figure 10 one of the results of the detection latency of all of the normal processes, sorted in ascending order.

All non-faulty processes detected the failure with latency in range from 2.020[s] to 2.026[s]. This result corresponds to the analysis that a monitoring process should detect a failure within T_{to}. Propagation of the failure information was achieved in about 5[ms], which is fast enough considering the number of processes and a 4.5[ms] round-trip time between the two clusters.

The number of messages generated for the information propagation was in range from 600 to 900. Since the monitoring network would be composed of about kn (=519) edges, the results agree with the property of flooding protocol that the number of messages should be less than $2kn$. Besides, because the monitoring relationships would be dispersed enough, each process only has to handle a constant number of flooding messages regardless of n.

VII. Conclusion

In this paper, we presented a scalable and efficient failure detection protocol for supporting self-repairing distributed systems. It autonomously creates uniformly-dispersed monitoring relationships so that each process is monitored by k other processes within LAN, and tries to establish at least k connections between any pair of clusters with almost no manual configuration. If a failure of a participating process is detected, the notification will be propagated to the other members by flooding along the established connections.

In our experiments in the actual environment and simulations, we showed the following:

- Our system was autonomously constructed in a widely distributed environment that included NAT in practice
- Our system endured concurrent node-failures with high probability
- Our system was scalable in both CPU load and network traffic. When we chose k of 4 and the heartbeat interval of 0.1[s], the system overhead with 313-processes was

less than 2-percent and total network traffic with 200-processes was $64[kbps]$, which was $\frac{1}{n}$ as much as that of Gossip protocol in the same situation.
- The notification of failures was quickly achieved with less than $2kn$ messages.

Our future work includes a study of light consensus techniques on membership connectivity for addressing catastrophic crashes and network partitioning. In addition, we have to integrate adaptive heartbeat techniques such as [10], [11] into our systems. Finally, we need to execute our failure detectors on larger-scale and more complex environments to confirm their usefulness.

ACKNOWLEDGMENT

This work is partially supported by "Precursory Research for Embryonic Science and Technology" of Japan Science and Technology Corporation and "Grand-in-Aid for Scientific Research" of Japan Society for the Promotion of Science.

REFERENCES

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

[2] G. E. Fagg and J. J. Dongarra, "Building and using a Fault Tolerant MPI implementation," *International Journal of High Performance Applications and Supercomputing*, 2004.

[3] R. Batchu, Y. Dandass, A. Skjellum, and M. Beddhu, "MPI/FT: A Model-Based Approach to Low-Overhead Fault Tolerant Message-Passing Middleware," *Cluster Computing*, pp. 303–315, Oct. 2004.

[4] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman, "Grid information services for distributed resource sharing," in *Proc. of Intl. Symposium on High Performance Distributed Computing*, 2001.

[5] R. Wolski, N. T. Spring, and J. Hayes, "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing," *Journal of Future Generation Computing Systems, 15(5-6)*, pp. 757–768, Oct. 1999.

[6] I. Gupta, T. D. Chandra, and G. S. Goldszmidt, "On scalable and efficient distributed failure detectors," in *Proc. of 20th Annual ACM Symp. on Principles of Distributed Computing*, 2001, pp. 170–179.

[7] A. Das, I. Gupta, and A. Motivala, "Swim: Scalable weakly-consistent infection-style process group membership protocol," in *Proc. of Intl. Conf. on Dependable Systems and Networks (DSN'02)*, June 2002, pp. 303–312.

[8] R. van Renesse, Y. Minsky, and M. Hayden, "A Gossip-Style Failure Detection Service," in *Middleware '98: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, 1998, pp. 55–70.

[9] K. Taura, T. Endo, K. Kaneda, and A. Yonezawa, "Phoenix : a Parallel Programming Model for Accommodating Dynamically Joining/Leaving Resources," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP2003)*, 2003.

[10] W. Chen, SamToueg, and M. K. Aguilera, "On the quality of service of failure detectors," *IEEE Transactions on Computers*, 2002.

[11] C. Fetzer, M. Raynal, and F. Tronel, "An adaptive failure detection protocol," in *Proc. 8th IEEE Pacific Rim Symposium on Dependable Computing (PRDC-8)*, Dec. 2001, pp. 146–153.

[12] P. Stelling, I. Foster, C. Kesselman, C. Lee, and G. von Laszewski, "A fault detection service for wide area distributed computations," *Cluster Computing*, vol. 2, no. 2, pp. 117–128, 1999.

[13] R. Wolski, N. T. Spring, and C. Peterson, "Implementing a performance forecasting system for metacomputing: The network weather service," in *In proceedings of Supercomputing 1997*, Nov. 1997.

[14] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, 1987.

[15] M.-J. Lin and K. Marzullo, "Directional gossip: Gossip in a wide area network," in *European Dependable Computing Conference (EDCC)*, 1999, pp. 364–379.

[16] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulie, "Scamp: Peer-to-peer lightweight membership service for large-scale group communication," in *Proc. Third Intl. Workshop Networked Group Communication*, Nov. 2001, pp. 44–55.

[17] ——, "Peer-to-peer membership management for gossip-based protocols," *IEEE Transactions on Computers*, vol. 52, no. 2, pp. 139–149, Feb. 2003.