A Gossip-Style Failure Detection Service

*Robbert van Renesse, Yaron Minsky, and Mark Hayden**
Dept. of Computer Science, Cornell University
4118 Upson Hall, Ithaca, NY 14853

Abstract

Failure Detection is valuable for system management, replication, load balancing, and other distributed services. To date, Failure Detection Services scale badly in the number of members that are being monitored. This paper describes a new protocol based on gossiping that does scale well and provides timely detection. We analyze the protocol, and then extend it to discover and leverage the underlying network topology for much improved resource utilization. We then combine it with another protocol, based on broadcast, that is used to handle partition failures.

1 INTRODUCTION

Accurate failure detection in asynchronous (non-realtime) distributed systems is notoriously difficult. In such a system, a process may appear failed because it is slow, or because the network connection to it is slow or even partitioned. Because of this, several impossibility results have been found (Chandra, Hadzilacos, Toueg & Charron-Bost 1996, Fischer, Lynch & Patterson 1985).

In systems that have to make minimal progress even in the face of process failures, it is still important to try to determine if a process is reachable or not. False detections are allowable as long as they are reasonable with respect to performance. That is, it is acceptable to report an exceedingly slow process, or a badly connected one, as failed. Unfortunately, when scaled up to more than several dozens of members, many failure detectors are either unreasonably slow, or make too many false detections. Although we are not aware of any publications about this, we know this from experiences with our own Isis, Horus, and Ensemble systems (van Renesse, Birman, Hayden, Vaysburd & Karr 1998, van Renesse, Birman & Maffeis 1996, Vogels 1996), as well as from experiences with Transis (Amir, Dolev, Kramer & Malkhi 1992).

*This work is supported in part by ARPA/ONR grant N00014-92-J-1866, ARPA/RADC grant F30602-96-1-0317 and AFOSR grant F49620-94-1-0198. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.
The current address of Mark Hayden is DEC SRC, 130 Lytton Ave., Palo Alto, CA.

In this paper, we present a failure detector based on random gossiping that has, informally, the following properties:

1. the probability that a member is falsely reported as having failed is independent of the number of processes.
2. the algorithm is resilient against both message loss (or rather, message delivery timing failures) and process failures, in that a small percentage of lost (or late) messages or small percentage of failed members does not lead to false detections.
3. if local clock drift is negligible, the algorithm detects all failures or unreachabilities accurately with known probability of mistake.
4. the algorithm scales in detection time, in that the detection time increases $O(n \log n)$ with the number of processes.
5. the algorithm scales in network load, in that the required bandwidth goes up at most linearly with the number of processes. For large networks, the bandwidth used in the subnets is approximately constant.

The protocol makes minimal assumptions about the network. There is no bound on message delivery time, and indeed messages may be lost altogether. We *do* assume that most message are delivered within some predetermined, reasonable time. Processes may crash or become unreachable. We assume a fail-stop rather than a Byzantine failure model, which basically means that processes do not lie about what they know. (The actual implementation uses checksums and other sanity checks on messages to approximate the fail-stop model as closely as possible.) We also assume that the clock rate on each host is close to accurate (low drift), but do not assume that they are synchronized. The basic version of the failure detector will place a limit on how many hosts may become unreachable. The final version will drop this limitation, and will tolerate arbitrary crashes and network partitions.

In our protocol, each host in the network runs a failure detector process that executes the protocol and offers continuous process failure and recovery reports to interested clients. Output from one of the failure detector processes in our department is used to generate and automatically update the web page http://www.cs.cornell.edu/home/rvr/mbrship.html, which gives an overview of our department's network status, and a log of recent activity on it. We view the failure detector as a middleware service. Other middleware services, such as system management, replication, load balancing, group communication services, and various consensus algorithms that rely on reasonably accurate failure detection (Chandra, Hadzilacos & Toueg 1992), can use the failure detection service to optimize their performance in the face of failures.

Approaches based on hierarchical, tree-based protocols have the potential to be more efficient than the ones described in this paper. However, we are not aware of publications describing any such protocols. Having tried to build one, we believe this is because tree-based failure detector protocols require complex agreement protocols to construct the tree and repair it as failures occur. Gossip protocols do not have this problem, and in fact one of their important advantages is their simplicity.

In this paper, we first present a simple protocol that assumes a uniform network topology, as well as a bounded number of host crashes. We analyze this protocol in Section 3. In Section 4, we extend the protocol to discover some information about the actual network topology and to use this information to optimize network resources. Section 5 presents another protocol that, when combined with gossiping, deals with arbitrary host failures and partitions.

2 BASIC PROTOCOL

The basic protocol is simple. It is based on gossiping as pioneered in the Clearinghouse project (Demers, Greene, Hauser, Irish, Larson, Shenker, Sturgis, Swinehart & Terry 1987). Long before that, Baker and Shostak (Baker & Shostak 1972) describe a gossip protocol, using *ladies* and *telephones* in the absence of the widespread availability of computers and networks. In gossip protocols, a member forwards new information to randomly chosen members. Gossip combines much of the efficiency of hierarchical dissemination with much of the robustness of flooding protocols (in which a member sends new information to all of its neighbors). In the case of Clearinghouse directories, it was used to resolve inconsistencies. Our protocol gossips to figure out whom else is still gossiping.

Each member maintains a list with for each known member its address and an integer which is going to be used for failure detection. We call the integer the *heartbeat counter*. Every T_{gossip} seconds, each member increments its own heartbeat counter, and selects one other member at random to send its list to. Upon receipt of such a *gossip* message, a member merges the list in the message with its own list, and adopts the maximum heartbeat counter for each member.

Each member occasionally broadcasts its list in order to be located initially and also to recover from network partitions (see Section 5). In the absence of a broadcast capability, the network could be equiped with a few *gossip servers*, that differ from other members only in that they are hosted at well known addresses, and placed so that they are unlikely to be unavailable due to network partitioning.

Each member also maintains, for each other member in the list, the last time that its corresponding heartbeat counter has increased.* If the heartbeat counter has not increased for more than T_{fail} seconds, then the member is considered failed. T_{fail} is selected so that the probability that anybody makes an erroneous failure detection is less than some small threshold $P_{mistake}$.

*It is possible that the heartbeat counter overflows, so it is necessary to use a *window* such as used in sliding window protocols.

After a member is considered faulty, it cannot immediately be forgotten about. The problem is that not all members will detect failures at the same time, and thus a member A may receive a gossip about another member B that A has previously detected as faulty. If A had forgotten about B, it would reinstall B in its membership list, since A would think that it was seeing B's heartbeat for the first time. A would continue to gossip this information on to other members, and, in effect, the faulty member B never quite disappears from the membership.

Therefore, the failure detector does not remove a member from its membership list until after $T_{cleanup}$ seconds ($T_{cleanup} \geq T_{fail}$). $T_{cleanup}$ is chosen so that the probability that a gossip is received about this member, after it has been detected as faulty, is less than some small threshold $P_{cleanup}$. We can make $P_{cleanup}$ equal to P_{fail} by setting $T_{cleanup}$ to $2 \times T_{fail}$.

To see why, let B be some failed member, and consider another member A that heard B's last heartbeat at time t. With probability P_{fail}, every other member will have heard B's last heartbeat by $t + T_{fail}$, and so every process will fail B by time $t + 2 \times T_{fail}$. Thus, if we set $T_{cleanup}$ to $2 \times T_{fail}$, then with probability P_{fail}, no failed member will ever reappear on A's list of members once that member has been removed.

Note that this protocol only detects hosts that become entirely unreachable. It does not detect link failures between hosts. In particular, if hosts A and B cannot talk to each other, but they can both communicate with some other host C, A will not suspect B nor the other way around. When these kinds of partial connectivity occur commonly throughout the systems, the protocol may cycle through adding and removing hosts. However, the dynamic Internet routing protocols should eventually take care of the problem.

In Section 4 we will extend this protocol to scale well in the Internet. This requires that part of the network topology is discovered, which we accomplish through gossiping as well. First, we will analyze the basic protocol to come up with a reasonable value for T_{fail}.

3 ANALYSIS

In order to find suitable parameters for the basic protocol, in particular the rate of gossiping and T_{fail}, it is necessary to understand how these parameters affect the probability of a false detection. In this section, we analyze the protocol for a system in which the population of processes is static (that is, no new processes are added), and known to all members. The discovery of new members only poses a problem when a large number of new members join at once, since that would lengthen the time it takes for the dissemination of information, and would potentially cause processes that have not yet learned of the enlarged membership to suspect other processes falsely. Section 5 discusses the behavior of the system in situations with sharp changes in membership.

The probabilities involved in gossiping can be calculated using epidemic theory (Bailey 1975, Demers et al. 1987, Golding & Taylor 1992, Birman, Hayden, Ozkasap, Budiu & Minsky 1998). A typical way of doing this is using stochastic analysis based on the assumption that the execution is broken up into synchronous rounds, during which every process gossips once. A member that has new information is called *infected*. For analysis, initially only one member is infected, and this is the only source of infection in the system, even though in reality new information may be introduced at multiple processes at once. At each round the probability that a certain number of members is infected given the number of already infected members is calculated. This is done for each number of infected members. Allowing all members to gossip in each round makes this analysis stochastically very complex, and usually approximations and/or upper bounds are used instead.

This section suggests a simplified analysis that we believe may be more accurate to boot. In practice, the protocols are not run in rounds. Each member gossips at regular intervals, but the intervals are not synchronized. The time for propagation of a gossip message is typically much shorter than the length of these intervals. We therefore define a different concept of round, in which only one (not necessarily infected) member is gossiping. The member is chosen at random and chooses one other member to gossip to at random. Thus, in a round, the number of infected members can grow by at most one, which is what simplifies the analysis.

Let n be the total number of members, k_i be the number of infected members in round i, and $P_{arrival}$ be the probability that a gossip successfully arrives at a chosen member before the start of the next round. (Actually, the probability can be chosen less conservatively, since all that is required is that the gossip arrives at the member before it itself gossips next.) The probability $P_{arrival}$ is independent for each message. Note that if the system behaves better than $P_{arrival}$ (for instance if no messages are dropped), then the protocol becomes more efficient. Initially, only one member p is infected. We assume that no more than f members fail.

For simplicity, we assume, conservatively, that all f members have failed at the beginning of the protocol. They are gossiped to, but they cannot pass gossips on. This way the number of infected members cannot shrink. We also assume that the initially infected member does not fail.* If k out of n members are infected already, then the probability that the number of infected members is incremented in a round is

$$P_{inc}(k) = (k/n) \times \frac{n-f-k}{n-1} \times P_{arrival} \tag{1}$$

*We believe this assumption will not affect the outcome by more than one round. For it to be more than one, the initially infected member would have to gossip, then crash, and the next infected member would have to do the same thing again.

Therefore, the probability that the number of infected members in round $i+1$ is k ($0 < k \leq n-f$) is

$$P(k_{i+1} = k) = P_{inc}(k-1) \times P(k_i = k-1) + (1 - P_{inc}(k)) \times P(k_i = k) \quad (2)$$

(with $P(k_i = 0) = 0$, $P(k_0 = 1) = 1$ and $P(k_0 = k) = 0$ for $k \neq 1$).

Then, the probability that any process does not get infected by p after r rounds is $P_{mistake}(p, r) = 1 - P(k_r = n - f)$. To find $P_{mistake}(r)$, which is the probability that any process is not infected by any other process, we need to range over all members. For any two processes p and q, $P_{mistake}(p, r)$ and $P_{mistake}(q, r)$ are not independent. We will bound $P_{mistake}(r)$ using the Inclusion-Exclusion Principle (Kozen 1991): $Pr(\cup_i A_i) \leq \sum_i Pr(A_i)$. Using this principle, we can bound $P_{mistake}(r)$ as follows:

$$P_{mistake}(r) \leq (n-f) P_{mistake}(p, r) = (n-f)(1 - P(k_r = n-f)) \quad (3)$$

From this we can calculate how many rounds are needed to achieve a certain quality of detection. (The calculation is best done not recursively, such as suggested here, but iteratively starting at the first round. Even then, it is costly. In (van Renesse, Minsky & Hayden 1998) we present an alternative analysis that works well if the number of members is at least about 50, and that can be calculated more easily.)

Next, we calculate how often members should gossip so that worst-case bandwidth use is linear in the number of members. Say that each member is allowed to send B bytes/second, and that 8 bytes per member are necessary in each gossip message (6 bytes for the address, and 2 bytes for the heartbeat counter). From this, the interval between gossips, T_{gossip} has to be $8n/B$ seconds. If n is small and B is large, then we may wish to place a minimum on T_{gossip}, since we assume that the common communication delay is less than T_{gossip}. (In the case of a pure switched network, as opposed to a broadcast-based network such as an Ethernet, the bandwidth per link will only be 2B bytes per second, independent of the number of members. Still, the total load on the switches increases with the number of members.) Note that members will receive, on average, B bytes/second worth of gossip information as a result, limiting the incurred overhead of the protocol.

We are interested in the time that it takes to reach a certain low probability p of a false failure detection being made. By iterating $P_{mistake}$, we can find how many rounds this takes, and multiply this by T_{gossip}. We have plotted this for different values of n, $P_{mistake}$, and $f = 1$, in Figure 1. The near-linearity in n can be confirmed visually. In actuality, the growth of the curve is $O(n \log n)$, because the number of rounds increases logarithmically with the number of members and the spacing between rounds linearly.

Readers familiar with other presentations on gossip protocols may be disappointed, and expected to see much better than linear degradation in detection

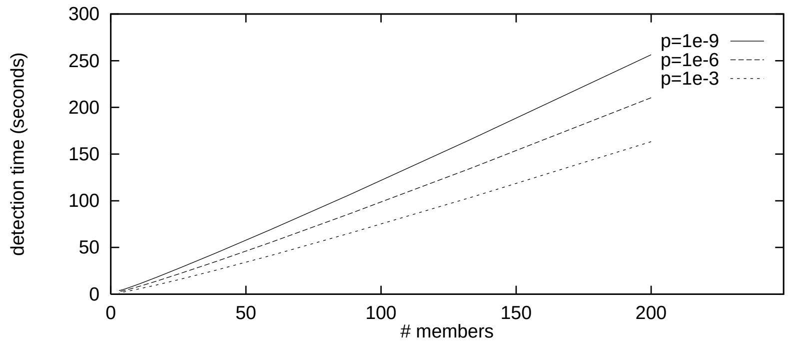

Figure 1 This graph depicts the failure detection time for three different mistake probabilities ($p = P_{mistake}$). The bandwidth per member is 250 bytes per second, and one member has failed. The protocol has $O(n \log n)$ behavior, which appears as being approximately linear in the graph.

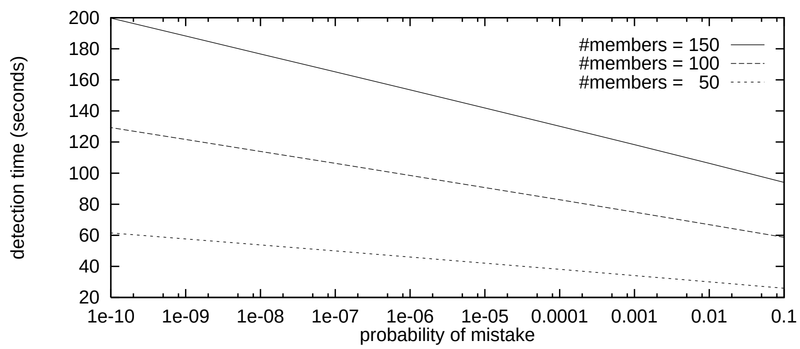

Figure 2 This graph shows the cost of quality of detection. Note that the x scale is logarithmic, thus the cost for better quality is excellent.

time. The reason for this is that much other gossip work either disseminated information that did not grow linearly with the number of members, or were only interested in message complexity, not in the bandwidth taken up by the messages. In our work, we decided to slow gossiping down linearly with the number of members because of the growing message size, which introduces a linear factor into our detection time.

There is a trade-off between minimizing the probability of making a false detection, and the failure detection time. In Figure 2 we show that this trade-off is satisfactory, in that little extra time is necessary for better quality.

Next, we look into the resilience of the protocol against process failures. Obviously, if many members fail, many gossips will be wasted, and it will take longer to distribute new heartbeat counters. We show this effect in Figure 3.

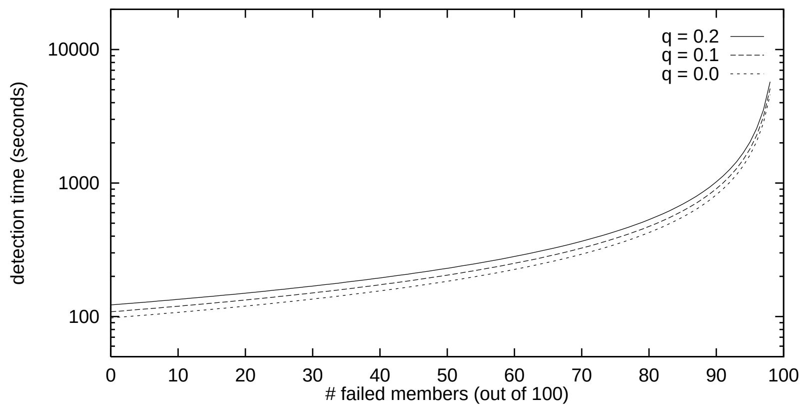

Figure 3 This graph shows, for three different probabilities of message loss q, the effect of failed members on detection time. It can be observed that the algorithm is quite resilient to process failures up until about half of the members have failed.

Notice that even if we assume that half of the members can fail, it takes less than twice as long to detect a failure than if we had assumed that no members would fail. This is because if half of the processes fail, then half of the gossips will be wasted, which is roughly equivalent to gossiping half as often. In general, it is easy to show that allowing for the failure of a fraction p of the members will increase the detection time by approximately $1/(1-p)$. Thus, the detection time begins to rise rapidly as p becomes larger than one half (note that the y-axes in this graph is logarithmic).

Therefore, in case of a network partition in which a small subgroup of members can get partitioned from a large group, the basic gossip algorithm does not perform satisfactorily (from the viewpoint of the small subgroup). For any reasonable detection time, the members of the subgroup will presume each other faulty because they are unlikely to gossip to each other. Section 5 presents a solution for this.

Finally, we investigate the impact of message loss. In Figure 4 we show for three different group sizes the effect of message loss on failure detection time. Again, we see satisfactory resilience. Even if we assume that 10% of the gossips get lost, we pay only a small price in detection time.

4 MULTI-LEVEL GOSSIPING

In a large system, the basic protocol has an inefficiency that can be corrected fairly easily, with an important improvement in scalability. The problem is that members choose other members at random, without concern to

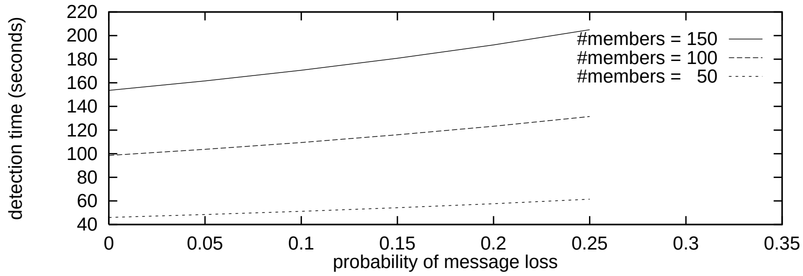

Figure 4 This graph shows the effect of message loss on detection time ($P_{mistake} = 10^{-6}$). It can be observed that the algorithm is quite resilient to message loss.

the topology of the network through which they are connected. As a result, the bridges between the physical networks are overloaded with much redundant information. In this section, we will show how members can correct this by automatically detecting the bounds of Internet domains and subnets, and reducing gossips that cross these bounds.

Before we present our modified gossiping protocol, we will review the structure of Internet domains and subnets. Internet addresses are 32-bit numbers subdivided into three fields. We call these fields the *Domain number*, *Subnet number*, and *Host* number. For example, here at the Cornell University campus the Domain number for all hosts is the same, and, roughly, each Ethernet (or other LAN) has its own Subnet number. The Host number identifies individual hosts on a subnet. The three fields form a hierarchy that reflects fairly accurately the underlying physical topology of networks.

The fields are variable length. The length of the Domain number can be determined by looking at the first few bits of it. The length of the Subnet and Host numbers cannot be determined that way, and can be different for any particular domain. * In case of Cornell, it happens that both the Subnet and Host numbers are 8 bits, but this is not visible externally. In fact, Cornell may decide to change this overnight without notifying anybody outside of Cornell, and this need not have any effects on routers outside of Cornell.

We will now present our modified protocol. There are two aspects to this protocol. First, the lengths of Subnet and Host numbers for each domain are gossiped along with the heartbeat counters of each host. Secondly, gossips are mostly done within subnets, with few gossips going between subnets, and even fewer between domains.

As for the first aspect, hosts now maintain two lists. One is the same as in the basic protocol, and contains the hosts and their heartbeat counters. The

*Subnets may be *subsubnetted* further to add another level to the hierarchy. We do not currently take this into account, which is not optimal, but does not affect the correctness of our protocol.

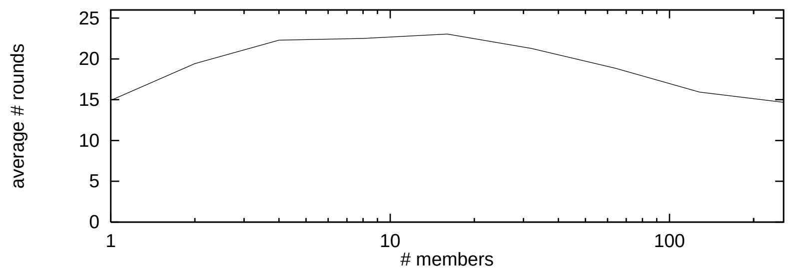

Figure 5 This graph shows the result of simulating the protocol for 256 members evenly distributed over a number of subnets.

second list contains the Domain numbers and their so-called *Subnet Masks* that determine the length of the Subnet and Host numbers. An invariant of our protocol is that for every host in the first list, there has to be an entry in the second that corresponds to that host's domain and subnet. (Since typically many hosts are in the same subnet, the second list will be much shorter.) The second list is gossiped along with the first, and merged on arrival, which maintains this invariant.

Within subnets, we run the basic protocol as described in Section 2. For gossips that cross subnets and domains we run a modified version of the protocol. This modified protocol tunes the probability of gossiping so that every round, on average one member per subnet will gossip to another subnet in its domain, and one member per domain will gossip to another domain. Thus, the bandwidth use at each level will be proportional to the number of entities contained at the next level down. So, for example, the cross-subnet bandwidth in a given domain will depend only on the number of subnets in that domain.*

To achieve this average behavior, every member tosses a weighted coin every time it gossips. One out of n times, where n is the size of the subnet, it picks a random (other) subnet in its domain, and within that subnet, a random host to gossip to. The member then tosses another weighted coin, but this time with probability $1/(n \times m)$, where m is the number of subnets in its domain, it picks a random (other) domain, then a random subnet within that domain, and then a random host within that subnet to gossip to.

This protocol significantly reduces the amount of bandwidth that flows through the Internet routers, since the gossiping is concentrated on the lower levels of the hierarchy. The protocol also allows for accelerated failure detection times within subnets, and is more resilient against network partitions. On the down side, it has a negative influence on the number of rounds needed

*We have chosen here to keep the bandwidth use per subnet and per domain equal to the bandwidth use per member at the subnet level. The protocol can trivially be generalized.

for information to be disseminated through the system, and hence on failure detection times across subnets and domains.

To show this negative effect, we simulated the protocol, and have plotted the average of the number of rounds it takes to gossip a new heartbeat counter to all members in a group of 256 members in Figure 5. We have spread the members evenly across a number of subnets, ranging from one subnet of 256 members to 256 subnets of one member each (all within a single domain). As we can see, we may pay as much as approximately 50% for certain subdivisions. We appear to pay most when the number of subnets and the sizes of the subnets are equal (16 in this case). But at this point we also are most efficient in bandwidth use, and use 16 times less bandwidth than we would have on a flat broadcast based network.

In fact, we may conclude that bandwidth use is approximately constant for large networks. To see this, let's see what happens when we double the number of members in a large network. We would not do this by doubling the number of members on each subnet, but by doubling the number of subnets, or, if there are many of those already, doubling the number of domains. When we double the number of members, the size of gossip messages doubles, but at the same time gossiping slows down by a factor of two. Therefore, within subnets and within domains, the bandwidth used remains the same. The only increase in bandwidth use would be at the highest level of the hierarchy, consisting of messages between domains. Fortunately, the Internet is designed to accommodate a constant amount of traffic per domain.

We can make calculations much like in (van Renesse, Minsky & Hayden 1998) (generalizing those of Section 3 becomes too complex) for any particular division of members into subnets and domains to figure failure detection times for members in particular subnets. The failure detection times for members in the same subnet will improve, while the other times will be longer than before, but generally not more than 50% longer. If this is a problem, we may choose to use some of the gain in bandwidth to increase the gossip frequency and hence reduce failure detection times.

In previous work, Agarwal et al. (Agarwal, Moser, Melliar-Smith & Budhia 1995) have shown that it is possible to exploit the network topology in membership and total ordering protocols. In their work, called Totem, it is necessary to run special software on the gateway machines between networks, while our protocol automatically discovers the boundaries of subnets and does not require special software other than on end-host machines. The intention of Totem is not providing rigorous guarantees about failure detections, but efficient membership and totally-ordered multicast in a local area network consisting of multiple physical networks.

Vogels (Vogels 1996) suggests multiple confidence levels of failure reports, and node failure monitors on each subnet in order to enhance scalability of failure detection. Our work implicitly follows the latter suggestion. (Vogels 1996) does not provide an analysis of scalability nor of quality of detection.

5 CATASTROPHE RECOVERY

Gossip algorithms do not work well if a large percentage of members crash or become partitioned away (see Figure 3). The problem is that too many gossips get wasted by sending them to unreachable processes. In order to deal with this, the failure detector does not immediately report members whom it has not heard from for time T_{fail}. Instead, it waits longer, but it does stop gossiping to these members.

In the meantime, we use a broadcast protocol to attempt to restore the connections to remaining members.* Since we know the *Subnet Masks* of each subnet (these were found using the protocol of Section 4), we can determine the broadcast addresses of each subnet (using either all zeroes or all ones for the Host Number).

The protocol below will guarantee, with high probability, that a broadcast is generated at least once every 20 seconds. In a failure-free environment, however, it will generate on average one broadcast every 10 seconds, imposing only a low load on the hosts in the network. (We have chosen exact parameters here for ease of explanation, but these parameters can be chosen more-or-less arbitrarily.)

Each member decides every second, probabilistically, on whether it should send a broadcast or not, based on how recently it has received a broadcast. If it was recent, then the member broadcasts with very low probability. If the member received the last broadcast almost 20 seconds ago, then it will broadcast with very high probability. The probability is also chosen so that the expected time that somebody first sends a broadcast is after 10 seconds. A function that appears to work well for this is $p(t) = (t/20)^a$. $p(t)$ is the probability that a member sends t seconds after receiving the last broadcast, and a is a constant chosen so that the expected first broadcast sent by any member is after $\mu = 10$ seconds. To find a, we first have to calculate the probability q(t) that anybody sends a broadcast after t seconds. This is $q(t) = 1 - (1 - p(t))^n$. Thus the probability $f(t)$ that the first broadcast is sent at time t is

$$f(t) = q(t) \times \prod_{i=0}^{t-1}(1 - q(i)) \tag{4}$$

Finally, the expected time μ at which the first broadcast is sent is

$$\mu = \sum_{t=0}^{20} t \cdot f(t) \tag{5}$$

*In the absence of a broadcast capability, a small set of hosts at well known addresses may be used to gossip through. The gossip servers should be placed strategically so they remain available in the case of partitions. We have not yet analyzed this approach.

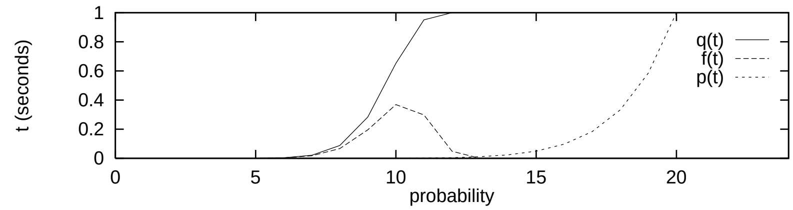

Figure 6 This graph shows $p(t)$, the probability that a particular member sends a broadcast after t seconds, $q(t)$, the probability that any member does, and $f(t)$, the probability that the first broadcast is sent after t seconds. The number of members is 1000 here.

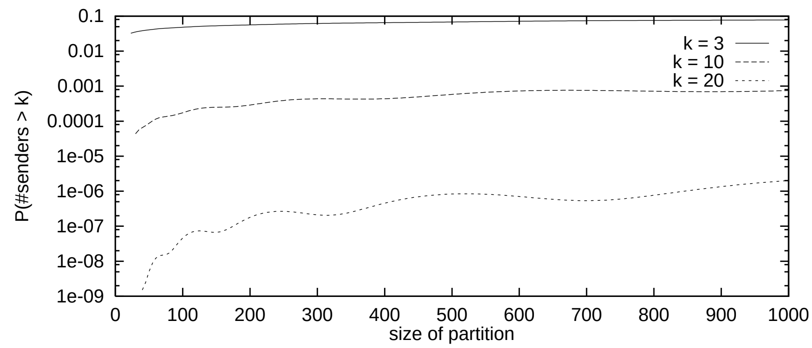

Figure 7 This graph shows, for every size of a partition of a group of 1000 members, the probability that more than k members broadcast at the same time for $k = 3, 10, 20$. Note that the y-axes is logarithmic.

For example, for $n = 1000$, we find that a has to be approximately 10.43 to make $\mu \approx 10$. We have plotted these functions for this example in Figure 6, and it can be seen that with high probability the next broadcast is around 10 seconds after the last one. The expected number of senders around 10 seconds is $1000 \times p(10)$, which is only 0.7, making it very unlikely that more than one member broadcasts at a time.

But what if a large fraction of members becomes unreachable? Clearly, with high probability, the next broadcast is before 20 seconds. The more members are unavailable, the closer the expected time gets to 20 seconds. The concern is that, as the expected time gets closer to 20 seconds, the probability that any member sends a broadcast, $q(t)$, goes up rapidly. Therefore, too many members may broadcast at the same time, leading to a storm of broadcasts. It turns out that the probability that more than 20 members broadcast at a time, in a group of 1000 members, is less than 10^{-5} (see Figure 7 and (van Renesse, Minsky & Hayden 1998)). The reason is that the smaller the partition becomes, the smaller the number of prospective senders, which offsets the growing probability with which the members do send.

This broadcast protocol may be made to scale better by using the hierarchy determined by the gossip protocol. Each subnet would run an instance of the broadcast protocol among the hosts, as well as each domain among the subnets, and the domains among each other. This is an improvement, because most partitions are not arbitrary, but occur along the boundaries of subnets and domains. Each host would determine three values for a to decide whether to broadcast in only its subnet, in the entire domain, or everywhere. (With three levels in the hierarchy, this would increase the total number of broadcasts by a factor of three, which may be corrected by adjusting the protocol's parameters.)

Now that the time to the next broadcast is bound, we can calculate how quickly the members within a new partition can find each other. (It should be taken into account that broadcast messages may get lost and that therefore the bound is probabilistic and may be somewhat larger than 20 seconds, but we are ignoring this issue here.) After the first broadcast in the new partition, all its members know just the originator of that broadcast, and will gossip to only it. Within $8/B$ seconds (where B is the bandwidth assigned to each member), the originator will know the entire membership, and gossip this information on (again, ignoring the probability of message loss). We can calculate the amount of time that is necessary to gossip the entire membership using the same analysis of Section 3. The maximum size of such a partition is $n - f$ members (f chosen as in Section 3). The maximum time to the next broadcast should be chosen so that the members within a partition have enough time to locate each other before they report each other as failed.

A particular instance of a catastrophic situation is at a cold start of one or more hosts, which do not know any other members to gossip to. A problem in this situation is that the domains and subnets are not initially known. Without this information, we can only determine the broadcast address of our own subnet. Currently, we extract a list of subnets in our own domain (for which we know the Subnet Mask) by scanning the *Hosts* data base that most computer networks maintain. Knowledge about subnets in other domains has to be manually configured.

6 CONCLUSION

In this paper we have presented a failure detection service based on a gossip protocol. The service provides accurate failure detection with known probability of a false detection, and is resilient against both transient message loss and permanent network partitions, as well as host failures. The service uses two separate protocols that automatically take advantage of the underlying network topology, and scale well in the number of members.

We have implemented a system based on these ideas and have run it within the `cs.cornell.edu` domain for the last three weeks. Over that period, it has measured over 100 failures and recoveries, without missing any failures and without making a single false failure detection. The output of the failure detector can be observed at http://www.cs.cornell.edu/home/rvr/mbrship.html. We have also implemented the basic protocol in the Ensemble system (van Renesse, Birman, Hayden, Vaysburd & Karr 1998).

The failure detection service can be used by distributed applications directly, or support other middleware services such as system management, replication, load balancing, and group communication and membership services. As such, failure detection is a valuable extension to current O.S. services.

Acknowledgments

The authors wish to thank Ken Birman, Nancy Lynch, Dexter Kozen, and the anonymous referees for helpful comments.

7 REFERENCES

Agarwal, D. A., Moser, L. E., Melliar-Smith, P. M. & Budhia, R. K. (1995), A Reliable Ordered Delivery Protocol for Interconnected Local-Area Networks, *in* 'Proc. of the International Conference on Network Protocols', Tokyo, Japan, pp. 365–374.

Amir, Y., Dolev, D., Kramer, S. & Malkhi, D. (1992), Transis: A communication subsystem for high availability, *in* 'Proc. of the Twenty-Second Int. Symp. on Fault-Tolerant Computing', IEEE, Boston, MA, pp. 76–84.

Bailey, N. T. J. (1975), *The Mathematical Theory of Infectious Diseases and its Applications (second edition)*, Hafner Press.

Baker, B. & Shostak, R. (1972), 'Gossips and telephones', *Discrete Mathematics* **2**(3), 191–193.

Birman, K. P., Hayden, M., Ozkasap, O., Budiu, M. & Minsky, Y. (1998), Bimodal multicast, Technical Report 98-1665, Cornell University, Dept. of Computer Science.

Chandra, T. D., Hadzilacos, V. & Toueg, S. (1992), The weakest failure detector for solving consensus, *in* 'Proc. of the 11th Annual ACM Symposium on Principles of Distributed Computing'.

Chandra, T. D., Hadzilacos, V., Toueg, S. & Charron-Bost, B. (1996), On the impossibility of group membership in asynchronous systems, *in* 'Proc. of the 15th Annual ACM Symposium on Principles of Distributed Computing', Philadelphia, PA.

Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D. & Terry, D. (1987), Epidemic algorithms for replicated database maintenance, *in* 'Proc. of the Sixth ACM Symp. on

Principles of Distributed Computing', ACM SIGOPS-SIGACT, Vancouver, British Columbia, pp. 1–12.

Fischer, M. J., Lynch, N. A. & Patterson, M. S. (1985), 'Impossibility of distributed consensus with one faulty process', *Journal of the ACM* **32**(2), 374–382.

Golding, R. & Taylor, K. (1992), Group membership in the epidemic style, Technical Report UCSC-CRL-92-13, UC Santa Cruz, Dept. of Computer Science.

Kozen, D. (1991), *The Design and Analysis of Algorithms*, Springer Verlag.

van Renesse, R., Birman, K. P., Hayden, M., Vaysburd, A. & Karr, D. (1998), 'Building adaptive systems using Ensemble', *Software—Practice and Experience* .

van Renesse, R., Birman, K. P. & Maffeis, S. (1996), 'Horus: A flexible group communication system', *Comm. of the ACM* **39**(4), 76–83.

van Renesse, R., Minsky, Y. & Hayden, M. (1998), A gossip-style failure detection service, Technical Report 98-1687, Cornell University, Dept. of Computer Science.

Vogels, W. (1996), World wide failures, *in* 'Proc. of the 7th ACM SIGOPS Workshop', Connemara, Ireland.

8 BIOGRAPHY

Robbert van Renesse is a Senior Research Associate at Cornell University, vice-president of Reliable Network Solutions, Inc., and editor for IEEE Transactions on Parallel and Distributed Systems. His research interests include fault tolerance and security. Currently he works on the Ensemble group communication project, and on the Tacoma mobile code project.

Yaron Minsky is a graduate student at Cornell University. His research interests include fault tolerance and security in distributed systems and cryptography. Currently he works on the Tacoma project.

Mark Hayden is a research staff member at the Digital Systems Research Center. He recently completed his PhD at Cornell University, where he took part in the Ensemble project. His research interests include distributed systems and the application of formal methods and programming languages in distributed systems.