Directional Gossip: Gossip in a Wide Area Network

Meng-Jang Lin
University of Texas at Austin
Department of Electrical Engineering
Austin, TX

Keith Marzullo
University of California, San Diego
Department of Computer Science and Engineering
La Jolla, CA

March 4, 1999

1 Introduction

A *reliable multicast* protocol ensures that all of the intended recipients of a message m that do not fail eventually deliver m. For example, consider the reliable multicast protocol of [10], and consider a message m, sent by process p_1, that is intended to be delivered by p_1, p_2, and p_3. We impose a directed spanning tree on these processes that is rooted at the message source. For example, for m we could have the directed spanning tree $p_1 \rightarrow p_2 \rightarrow p_3$. The message m propagates down this spanning tree and acknowledgements of the receipt of m propagate back up the tree. A leaf process in this tree delivers m when it receives m, and a non-leaf process delivers m when it gets the acknowledgement for m from all of its children. If a non-leaf process (say, p_1) does not get an acknowedgement for m from one of its children (here, p_2), then it removes the child from the tree and "adopts" that child's children (here, p_3). The process sends m to the newly-adopted children and continues the broadcast. A similar monitoring and adoption approach is used to recover from the failure of the root of the tree.

Reliable multicast protocols are intended for local area networks. Unfortunately, most implementations of reliable multicast do not scale well to large numbers of processes even when all are in the same local area network [3]. For example, with the protocol given above, the sender cannot deliver its own message m until it knows that all non-failed processes have already delivered m. The latency can be reduced by using a bushy directed spanning tree, but doing so increases the overhead of some processes, where by overhead we mean the number of messages a process sends and receives in the reliable multicast of a single m. As the number of processes increases, either the latency or the overhead at some processes increases. Hence, when a multicast is to be sent to a large number of processes or processes located on a wide area network, a protocol like IP Multicast [4] that has been specifically designed for these cases is preferable even though it is not as reliable as reliable multicast.

More recently, *gossip-based protocols* have been developed to address scalability while still providing high reliability of message delivery. These protocols, which were first developed for replicated database consistency management in the Xerox Corporate Internet [5], have been built to implement not only reliable multicast [3, 7] but also failure detection [11] and garbage collection [12].

Gossip protocols are scalable because they don't require as much synchronization as traditional reliable multicast protocols. A generic gossip protocol running at process p has a structure something like the following:

```
when (p receives a new message m)
while (p believes that not enough of its neigbors have received m) {
   q = a neighbor process of p;
   send m to q;
}
```

Since they lack the amount of synchronization that traditional multicast protocols have, the reliability of gossip-based protocols is evaluated in a different manner. The mathematics of *epidemiology* are often applied, since the spread of a message with a gossip protocol is much like the spread of a disease in a susceptible population. When the mathematics become intractable, simulation is often used.

If one wished to implement gossip-based reliable multicast with as high reliability as possible, then one would use a *flooding protocol* [2] like the following

```
when (p receives a new message m from neighbor q)
for each (r : r neighbor of p)
   if (r != q) send m to r;
```

Flooding can be thought of as a degenerate gossip protocol in which a process chooses all the neighbors that it doesn't know already have the message. Flooding, however, can have a high overhead. Consider the undirected graph $G = (V, E)$ in which the nodes V are processes and edges E connect processes that are neighbors. The total number of messages sent in flooding a single message in G is between $|E|$ and $2|E|$. If the processes are all on a single local area network, then one can consider G to be a clique (that is, all processes can directly communicate with each other), and so the number of messages is quadratic in $|V|$. Gossip protocols are attractive when G is a clique because they provide negligably less reliability than flooding with a much lower overhead.

If G is not a clique, then the reliability of gossip protocols is less. This is not hard to see, and has already been observed in the context of the spreading of computer viruses [8, 9]. Consider a process p_1 that is in a clique of n processes $p_1, p_2, ...p_n$ and that has a pendant neighbor q: that is, the only neighbor of q is p_1. Suppose that these processes are running a gossip protocol in which p_1 continues to forward a new message m as long as it believes that less than B of its neighbors have received m. If p_1 receives a new message m from p_2 and p_1 selects its neighbors uniformly, then the probability that q will receive m is $1 - \binom{n-2}{B-1}/\binom{n-1}{f-1} = (f-1)/(n-1)$. Thus, f must be large (and the corresponding overhead high) for the reliability of this protocol to be high. A more intelligent protocol would have p_1 always forward new messages to q and use gossip to communicate with the rest of its neighbors.

We present a protocol that behaves like this more intelligent protocol. Each process determines a *weight* for each of its neighbors. This weight is measured dynamically and is the minimum number of edges that must be removed for the process to become disconnected from its neighbor. For example, assuming no links are down, p_1 would assign a weight of 1 to q and weights of $n-1$ to each of its remaining $n-1$ neighbors. A process floods to neighbors that have small weights and gossips to neighbors that have large weights.

2 System Model

We consider a wide area network of the size that a large corporation might have: 10^3 local area networks with, on average, 10^2 processors per local area network. We model the structure of such a network using the techniques presented in [1].

Consider two processors on different local area networks that are connected by a single router r. We assume that a process p_1 on one of the processors can send a message to a process p_2 on the other processor via that router. In particular, if r is down then p_1's message will not reach p_2 even if there is another route connecting these two processors. This can be implemented in IP using either hop counts or source routing.

3 Architecture

It has already been observed [13] that the overhead of gossip protocols in a wide area network can be reduced by taking the network topology into account. For example, consider two local area networks, each with the same number of processors and that are connected by a single router. If one ignores the network topology, then on average a processor will have half of its neighbors in one local area network and half of its neighbors in the other. Hence, on average half of the gossip messages will traverse the router, which is an unnecessarily high load. The work in [13] addresses this problem by having each processor aware of which local area network each of its neighbors is in. A processor then only rarely decides to send a gossip message to a processor in another local area network. This approach is attractive because it attenuates the traffic across a router without adding any additional changes to the gossip protocol. Its drawback is that it doesn't differentiate between wide area traffic and local area traffic. The performance characteristics and the link failure probabilities are different for wide area networks and local area networks. Hence, we adopt a two-level gossip hierarchy: one level for gossip within a local area network and another level for gossip among local area networks.

Each local area network runs a *gossip server* that directs gossip to the local area networks that are one hop away. Two gossip servers are neighbors if the local area networks with which they are associated are connected by an internetwork router. For example, Figure 1 shows three local area networks connected by routers A, B and C. Each gossip server is labeled with the routers that are connected to its local area network. Two gossip servers are neighbors if they both have the same internetwork router listed in their label. Hence, the neighbors relation of these three gossip servers in this figure is a three-clique. As will be discussed in the next section, the state that a gossip server maintains is small, and so a gossip server could easily be replicated if the reliability of a single server is not adequately high.

Messages are disseminated to the processes in a local area network, including the gossip servers, using a traditional gossip protocol. When a gossip server receives a message m for the first time via the local area network gossip protocol, it initates an *wide area network* gossip protocol with message m. When a gossip server receives for the first time a message m via the wide area network gossip protocol, it injects m into its local area network using the local area network gossip protocol.

The protocol that we develop in this paper is the wide area network gossip protocol; we do not address local area network issues further. In Section 1 we argued that to have a high reliability of message delivery, a wide area network gossip protocol needs to have some information about the network topology. Wide area networks can be large and their topology may change frequently, and so we decided not to require each gossip server to have *a priori* knowledge about about the entire network topology. Instead, all a gossip server needs to know is its neighbors, which is equivalent

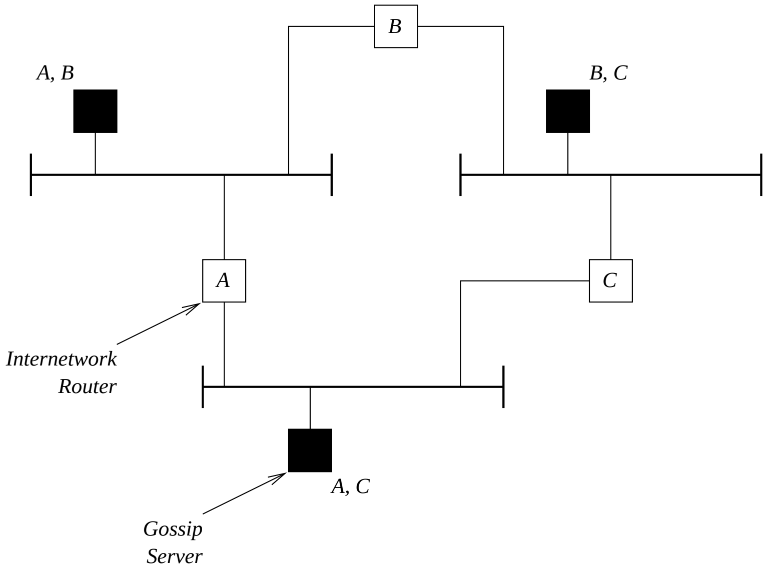

Figure 1: Gossip Server Architecture

to knowing the identity of all local area networks that are one hop away. This is the kind of information that a network administrator will know about a local area network, and so a gossip server can obtain this information from an administrator-generated configuration file.

We believe that the wide area gossip protocol should run on top of IP. Since the gossip protocol determines information about the internetwork connectivity on the fly, it needs to circumvent to some degree the internet routing protocol. As will be described in the next section, a gossip server records the trajectory a gossip message follows to determine the number of link-disjoint paths between itself and a neighbor. Internet routing, on the other hand, abstracts away the notion of a path; routing can change the trajectory of a message as routers fail or become overloaded. Hence, wide area network gossip runs at the OSI transport level, but it thwarts many features of the underlying network level.

4 Protocol

In this section we develop a wide area gossip protocol that we call *directional gossip*. We first review some ideas from graph theory and then describe how we use them to measure weights. We then describe the directional gossip protocol in terms of these weights.

4.1 Weights

A *link cut set* of a connected graph G is a set of edges that, if removed from G, will disconnect G. A link cut set with respect to a pair of nodes p and q is a set of edges that, if removed from G, will disconnect p and q. Clearly, the link cut set with respect to a pair of nodes is also a link cut set of the graph.

A gossip server p assigns as a weight to a neighor gossip server q the size of the smallest link cut set with respect to p and q. If this weight is low, then p will always send new messages to q; else

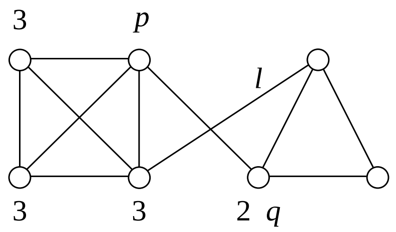

Figure 2: Weights

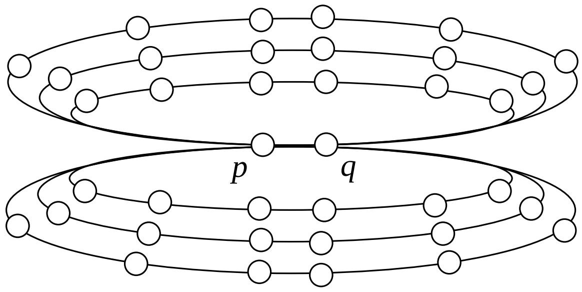

Figure 3: Pathological Graph

it will send them to q only if p selects q as a neighbor with whom to gossip. The intuition behind this strategy is similar to what was illustrated in Section 1. For example, if this weight is 2, then there are two links, at least one of which must be up and selected when gossiping, for a message to propogate from p to q. As the weight of a neighbor increases, the liklihood of at least one link in the link cut set being up and selected becomes sufficiently large that p and q can exchange information using gossip. Otherwise, p always forwards each new message to q.

Figure 2 gives an example of the weights of a gossip server p. All of the neighbors of p in the four-clique have a weight three, since three edges must be deleted to isolate p from any of these neighbors. The neighbor of p in the three-clique, however, has a weight of two since only the two links connecting the four-clique and the three-clique need be deleted to isolate q from p.

One can imagine other weights that might be interesting. For example, consider the graph in Figure 3 that consists of many long cycles, each distinct except for the (p, q) edge. The weight that p would assign to q is large (in this graph, seven) since there are many link-disjoint paths that connnect p and q. Thus, our strategy would most likely have p only probabilistically choose q. If links fail frequently enough, however, then the probability that a message will make it along one of the long cycles from p to q may be low. Hence, under these conditions p should always forward to q. The benefit of the strategy that we have is that the weights are easy to compute dynamically and the strategy works well for common internetwork interconnection topologies. In addition, our protocol measures the dynamic connectivity between two neighboring nodes. Under the assumptions that the long links are often broken, the weight that p would assign q would in fact be low.

4.2 Measuring Weights

We use the following version of Menger's Theorem, due to Ford and Fulkerson [6], in a method for a gossip server to measure the weights of its neighbors.

> For any two nodes of a graph, the maximum number of link-disjoint paths equals the minimum number of links that separate them.

Thus, a gossip server can maintain for each of its neighbors a list of link-disjoint paths between itself and that neighbor. The size of this set is the weight of the neighbor. A gossip server collects these paths by observing the trajectories that gossip messages traverse, and it ensures through randomization that all such paths are found.

Each gossip message m carries $m.path$ which is the trajectory that m has traversed. Each element in this trajectory identifies an internetwork router that has forwarded m. The internetwork router is implicity identified by the pair of gossip servers that communicate via that router. Before a gossip server s forwards m to another gossip server r, s adds an identifier for r to the end of $m.path$ if $m.path$ is not empty; otherwise, it sets $m.path$ to the list $\langle s; r\rangle$. Thus, given a trajectory $m.path$ of $g > 1$ gossip servers, we can construct a path $INR(m.path)$ of $g-1$ internetwork routers. Note that the length of $m.path$ is bounded by the diameter D of the wide area network.

Let $Neighbors_s$ be the set of neighbors of a gossip server s. For each neighbor $r \in Neighbors_s$, each gossip server s maintains a list $Paths_s(r)$ of link-disjoint paths that connect s and r. This list cannot contain more than $|Neighbors_s|$ paths. When a gossip server s receives a gossip message m, for every $r \in Neighbors_s$ such that r is in $m.path$, if for every path $p \in Paths_s(r)$, p and $INR(m.path)$ do not have any common elements, then $INR(m.path)$ is added to $Paths_s(r)$. A simple implementation of this algorithm has $O(D(\log(D) + |Neighbors_s|)$ running time for each gossip message that a gossip server receives. The weight a gossip server s computes for its neighbor r is then simply $|Paths_s(r)|$.

The weights that a gossip server computes for its neighbors should by dynamic. For example, consider Figure 2. If the link ℓ fails, then the weight that p assigns to its neighbor q should drop from two to one. It is not hard to modify the above algorithm to dynamically maintain $Paths_s(r)$ so that failures and recoveries are taken into account. Each element in $m.path$ includes, as well as the identity of a gossip server, the time that the gossip server first received m. Such a time is interpreted, for each element in $INR(m.path)$, as the time that m traversed that internetwork router. Then, when $INR(m.path)$ is compared with a path $p \in Paths_s(r)$, when an element of p is equal to an element of $INR(m.path)$, then the time associated with the link in p is set to the maximum of its current time and the time associated with the same link in $INR(m.path)$. We can then associate a time $Time(p)$ with each element $p \in Paths_s(r)$ as the oldest time of any link in p. If $Time(p)$ is too far in the past, then s can remove p from $Paths_s(r)$.

This simple method of aging link-disjoint paths can result in a temporarily low weight. For example, consider the two gossip servers s and r in Figure 4. Assume that $Paths_s(r)$ contains three paths: the direct path connecting s and r, the path indicated by dashed lines, and the path indicated by dotted lines. Hence, s computes a weight of three for r. Now assume that the link ℓ fails. Eventually, the time associated with the dotted path will become old enough that this path is removed from $Paths_s(r)$, at which point s computes a weight of two for r. This weight is too low: three links must be removed for these two nodes to become disconnected. Eventually, though, s will receive a message following the remaining link-disjoint path, and thus will again compute a weight of three for r. And, as discussed in the next section, computing a too-low weight does not hurt the reliability of the gossip protocol, but only increases the overhead.

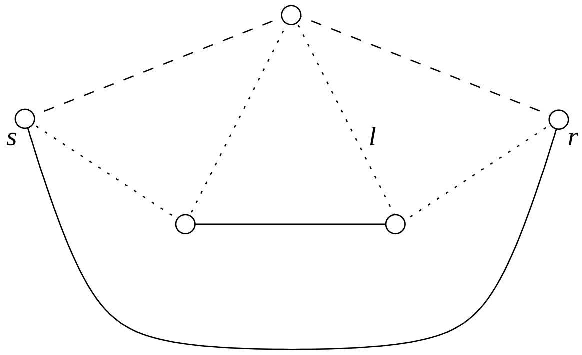

Figure 4: Dynamic Weight Computation

4.3 Directional Gossip

The protocol that a gossip server s executes is the following. We first give the initialization. A gossip server only knows about the direct path connecting itself to a neighbor. Thus, s will assign an initial weight of one to each of its neighbors. This weight may be low, and will have s forward new messages to all of its neighbors. As s learns of more paths, it will compute more accurate weights for its neighbors, and the overhead will correspondingly reduce.

init

for each $r \in \mathit{Neighbors}_s : \mathit{Paths}_s(r) = \{\mathit{INR}(\langle s, r\rangle)\}$;

Note that, in order to simplify the exposition, we haven't given a time for the last traversal of this initial path. We assume that whenever a gossip server is added to a trajectory, the current time is also added to the trajectory.

The next code block is executed when s receives a new gossip message m. It first updates $s.path$r for each neighbor r that is in $m.path$. It then sends m to all neighbors that s believes may not have m and that have a weight less than K. Gossip server s then chooses enough of the remaining neighbors that may not have m so that at least B neighbors are sent m and at least one neighbor is chosen at random to be sent m.

when s receives gossip message m for the first time: {
    int sent = 0; for each $r \in \mathit{Neighbors}_s$
        if ($r \in m.path$) UpdatePaths($\mathit{Paths}_s(r)$, $\mathit{INR}(\mathit{Trim}(m.path, r))$);
    for each $r \in \mathit{Neighbors}_s$ AgePaths($\mathit{Paths}_s(r)$);
    for each $r \in \mathit{Neighbors}_s$
        if ($r \notin m.path$ &&$|\mathit{Paths}_s(r)| < K$){
            $m' = m$;
            append r to m';
            send m' to r;
            sent = sent + 1;
        }
    for each $r \in$ Choose(min(B, sent) of $\mathit{Neighbors}_s - \{q : q \in m.path\}$){
        $m' = m$;
        append $(r, \mathit{Now}())$ to m';

```
        send m' to r;
    }
}
```

The following procedure updates the set of link-disjoint paths between itself and a neigbor based on the trajectory that m has followed. It also updates the times that the links were last traversed. The test for common links can be efficiently implemented by having each path be a sorted list of links, and sorting the trajectory T.

void UpdatePaths(**ref** set of paths P, trajectory T) {
    if (all elements of P have no links in common with T) add T to P;
    else for each p in P:
        for each link $\ell_1 \in p$ and link $\ell_2 \in$ T:
            if (ℓ_1 and ℓ_2) name the same internetwork router)
                set the time ℓ_1 was last traversed to
                    max(time ℓ_1 was last traversed, time ℓ_2 was last traversed);
    }

The following procedure determines if a path is to be removed because too much time has passed since a link in the path has been traversed.

void Age(**ref** set of paths P) {
    for each p in P:
        if (there is a link ℓ in p: Now() $-$ the last time ℓ was traversed $>$ Timeout)
            remove p from P;
    }

Finally, the following function removes a prefix from the sequence of gossip servers a message has traversed.

server sequence Trim(server sequence S, gossip server s) {
    return (the sequence S with all servers visisted before s removed)
}

5 Simulation

We built a simple discrete event simulator to measure the performance of directional gossip. Ths simulator takes as input a graph with nodes representing gossip servers and links representing internetwork routers. Messages are reliably sent between gossip servers and are delivered with a time chosen from a uniform distribution. We do not model link failures or gossip server failures.

We simulated three protocols: flooding, gossip with a fanout B, and directional gossip with a fanout B and a critical weight K. We compared the message overheads of these three different protocols, and when interesting compared their reliability. We also measured the ability of directional gossip to accurately measure weights.

We considered four different network topologies: a ring of 16 gossip servers, a clique of 16 gossip servers, two cliques of eight gossip servers, connected by a single link, and a topology meant to resemble a wide area network.

Ring

Clique

Two Cliques

Wide Area Network

6 Discussion

References

[1] That paper that talks about how to model an internet.

[2] Gregory Andrews. Flooding protocols.

[3] Kenneth Birman and his friends. Bimodal multicast.

[4] Steve Deering. Some paper on IP Multicast.

[5] Alan Demers. Paper on epidemiological protocols.

[6] L. R. Ford and D. R. Fulkerson. Maximum flow through a network. *Canadial Journal of Mathematics* 8(1956):399-404.

[7] Richard Golding. Paper on gossip.

[8] Jeffery Kephart. Paper on how hard it is to spread a virus with floppies.

[9] Meng-Jang Lin, Aleta Ricciardi, and Keith Marzullo. Security New Paradigms paper.

[10] F. B. Schneider, D. Gries, and R. D. Schlichting. Fault-tolerant broadcasts. *Science of Computer Programming* 4(1):1–15, April 1984.

[11] Robbert van Rennesse. Paper on gossip-based failure detection.

[12] Robbert van Rennesse. Paper on gossip-based garbage collection.

[13] Robbert van Rennesse. Paper in which he talks about excess load on routers.