GoCast: Gossip-Enhanced Overlay Multicast for Fast and Dependable Group Communication

Chunqiang Tang, Rong N. Chang, and Christopher Ward

IBM T.J. Watson Research Center
{ctang,rong,cwl}@us.ibm.com

Abstract

We study dependable group communication for large-scale and delay-sensitive mission critical applications. The goal is to design a protocol that imposes low loads on bottleneck network links and provides both stable throughput and fast delivery of multicast messages even in the presence of frequent node and link failures. To this end, we propose our GoCast protocol. GoCast builds a resilient overlay network that is proximity aware and has balanced node degrees. Multicast messages propagate rapidly through an efficient tree embedded in the overlay. In the background, nodes exchange message summaries (gossips) with their overlay neighbors and pick up missing messages due to disruptions in the tree-based multicast. Our simulation based on real Internet data shows that, compared with a traditional gossip-based multicast protocol, GoCast can reduce the delivery delay of multicast messages by a factor of 8.9 when no node fails or a factor of 2.3 when 20% nodes fail.

1. Introduction

With the rapid growth of the Internet, more and more applications are being developed for (or ported to) wide-area networks in order to take advantage of resources available at geographically disparate locations, e.g., Grids, peer-to-peer data sharing, and computer-supported collaborative work. As part of IBM's Advanced Operating Environment (AOE) initiative, we are developing an intelligent infrastructure to support these large-scale distributed applications. The design goals for this infrastructure include *self-organizing* to minimize manual configurations, *self-healing* to cope with failures, *self-tuning* to improve service quality, and *self-learning* through machine learning techniques to extract knowledge from past experiences to advise the self-organizing, self-healing, and self-tuning components.

In this paper, we focus on the dependable group communication protocol in our infrastructure. Group communication efficiently delivers messages to a large number of receivers. It is a basic utility for writing distributed applications and can be used for various purposes, for instance, disseminating system monitoring events to facilitate the management of distributed systems [8], and propagating updates of shared state to maintain cache consistency. A dependable group communication protocol for large-scale and delay-sensitive mission critical applications should meet at least the following basic requirements.

- **Reliable message delivery**. The system should sustain stable throughput even in the face of frequent packet losses and node failures. Systems that solely optimize for friendly environments are unacceptable.
- **Fast message delivery**. Many mission critical applications have soft real-time constraints, e.g., airline control and system monitoring. When a deadline is missed, the message becomes useless. Even within the deadline, the value of the message depreciates over time.
- **Scalable performance**. The system should be self-adaptive to handle dynamic node joins and leaves. As the system grows, the degradation in efficiency, reliability, and message delay should be graceful.
- **Efficient network resource consumption**. When multicasting a message to a large number of receivers at the application level, the underlying network links carry different traffic. It is important not to impose an extremely high load on any physical link.

Two categories of existing protocols—*reliable multicast* and *gossip multicast*—have the potential to meet some but not all the requirements above. Reliable multicast (e.g., SRM [7]) sends messages through a tree and relies on retransmissions to handle failures. In a friendly environment, it propagates messages rapidly. Previous study [14], however, has shown that a small number of disturbed nodes can lead to dramatically reduced throughput of the entire system. Reliable multicast, therefore, is not a scalable solution for dependable group communication.

In gossip multicast (e.g., Bimodal Multicast [2]), nodes periodically choose some random nodes to propagate summaries of message IDs (so-called "gossips") and pick up missing messages heard from gossips. The redundancy in gossip paths addresses both node and link failures. Gossip multicast delivers stable throughput even in an adverse environment but the propagation of multicast messages can

be slower than that in reliable multicast, as the delay is proportional to the gossip period and always exchanging gossips ahead of actual messages incurs extra delay. Moreover, our evaluation shows that, due to the obliviousness to network topology, random gossips in a large system impose extremely high loads on some underlying network links.

We propose *GoCast* (gossip-enhanced overlay multicast) to address the limitations of existing protocols. It combines the benefits of reliable multicast (topology awareness and fast message propagation) and the benefits of gossip multicast (stable throughput and scalability), while avoiding their limitations. GoCast builds a resilient, proximity-aware overlay network that has balanced node degrees. As in reliable multicast, messages propagate rapidly through an efficient tree embedded in the overlay. In the background, nodes exchange message summaries (gossips) with their overlay neighbors (as opposed to random nodes) and pick up missing messages due to disruptions in the tree-based multicast. The number of times (i.e., redundancy) that a node receives the gossip containing the ID of a message is controlled by the number of the node's overlay neighbors.

GoCast vs. Gossip Multicast

Gossip multicast protocols such as Bimodal Multicast have two key elements: *redundancy* and *randomness* in gossip paths. These are the fundamental reasons why gossip multicast can provide stable throughput and reliable message delivery. GoCast retains the spirits of redundancy and randomness—redundancy through multiple disjoint paths in the overlay and *controlled* randomness through some random overlay links. In addition, GoCast has several advantages over gossip multicast protocols.

Due to the *complete* randomness in gossip multicast (as opposed to the *controlled* randomness in GoCast), the number of times that nodes receive the gossip containing the ID of a given message varies dramatically. In a push-based gossip protocol (e.g., Bimodal Multicast), upon receiving a new message, a node gossips the ID of the message to F random nodes. F is called the *fanout* of gossips. Our simulation shows that, with fanout 5, about 0.7% of nodes in a 1,024-node system never hear about a given message while some nodes hear about the message as many as 19 times.

In an n-node system using the push-based gossip protocol with fanout F, the probability that all nodes hear about a given message through gossips is $e^{-e^{\log(n)-F}}$ [6]. Hence the probability that all nodes hear about 1,000 messages is $e^{-1000 \cdot e^{\log(n)-F}}$. Figure 1 plots these probabilities for a 1,024-node system. Even without any fault in the system, the probability that all nodes receive 1,000 messages is lower than 0.5 when the fanout is smaller than 15.[1] Note

[1]This situation can be improved by combining both push and pull in gossip disseminations [9]. The challenge, however, is to avoid the overheads of unnecessary pulls when there is no multicast message.

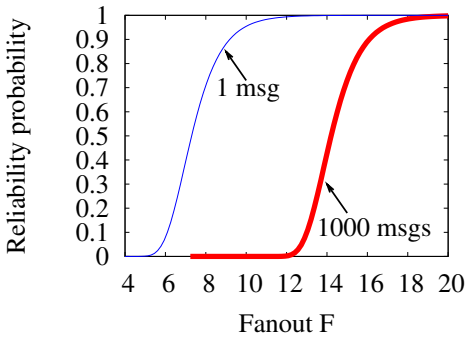


Figure 1. In a push-based gossip protocol with fanout F, the probability that all nodes in a n=1024 node system receive 1 or 1,000 multicast messages. The curves correspond to functions $e^{-e^{\log(n)-F}}$ and $e^{-1000 \cdot e^{\log(n)-F}}$, respectively.

that a reliability of 0.5 is not sufficiently high for mission critical applications. Moreover, using fanout 15 introduces three times more gossip traffic than GoCast (Section 2), and the propagation of multicast messages in gossip multicast is several times slower than that in GoCast (Section 3).

By contrast, GoCast does not completely count on randomness. It has total control of the number of times that a node receives the gossip regarding a message by tuning the node degree. So long as the overlay remains connected, nodes receive the gossip regarding a message at least once. Our evaluation shows that, for a system with thousands of nodes and six neighbors per node, the overlay in GoCast remains connected even after 25% of nodes fail concurrently.

The remainder of the paper is organized as follows. Section 2 presents the GoCast protocol. Section 3 compares several dependable group communication protocols through extensive simulations. Related work is discussed in Section 4 and Section 5 concludes the paper.

2. The GoCast Protocol

GoCast provides dependable group communication for large-scale mission critical applications that are delay sensitive. Regardless of the size of the system, it incurs a constant low overhead on each node. A node join or leave affects only a small number of other nodes and those nodes handle the change locally. GoCast is self-tuning. Its efficiency and message delay improves quickly as more is learned about the underlying network.

GoCast organizes nodes into an overlay and disseminates multicast messages both through an efficient tree embedded in the overlay and through gossips exchanged between overlay neighbors. Below we first describe the message dissemination protocol assuming the overlay and the tree are already in place, and then describe the protocols to build the overlay and the tree.

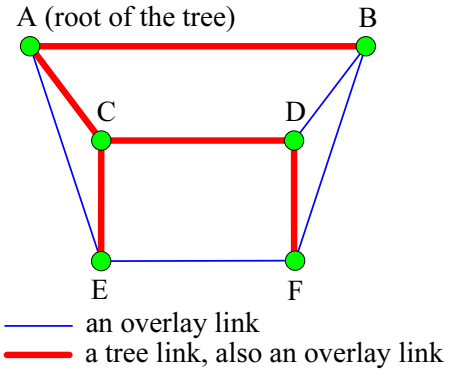


Figure 2. An example of GoCast.

2.1. Fast and Reliable Message Dissemination

Under normal operations, multicast messages propagate rapidly through an efficient tree embedded in the overlay (Figure 2). A tree link is also an overlay link. Both overlay links and tree links are undirected and can propagate messages in either direction. We refer to two nodes directly connected by an overlay link or a tree link as overlay neighbors or tree neighbors, respectively. Two tree neighbors are also overlay neighbors. Solely for the purpose of maintenance, the tree conceptually has a root, but any node can start a multicast without first sending the message to the root.

A multicast message propagates away from the message source along the tree links. We illustrate this through an example in Figure 2. Suppose node D wants to multicast a message to all nodes in the system. Node D sends the message to its tree neighbors C and F. Node D does not send the message to node B because B and D are not tree neighbors although they are overlay neighbors. Each node that receives the message immediately forwards the message to its tree neighbors except the node from which the message arrived. Each node remembers the IDs of the messages it forwarded lately to avoid forwarding a message repeatedly. Without faults, the message propagates rapidly to all nodes (under half a second for a 1,024-node system) and each node receives the message exactly once.

Faults, however, are unavoidable. In the background, nodes exchange message summaries with their overlay neighbors and pick up missing messages due to disruptions in the tree-based multicast. Every t seconds, a node X choose one of its overlay neighbors Y in a round robin fashion to send a message summary (also called "gossip"). The gossip period t is dynamically tunable according to the message rate, t = 0.1 seconds in our current design, which is suggested by Bimodal Multicast [2]. Each message injected into the system has a unique identifier. The identifier of a message injected by node P is a concatenation of P's IP address and a monotonically increasing sequence number locally assigned by P. The gossip that node X sends to node Y includes the IDs of messages that X received or started since the last gossip X sent to Y, but excludes the IDs of messages that X heard from Y. After receiving the gossip, if node Y finds out that the gossip contains the IDs of some messages that Y has not received, Y requests those messages from node X. In most cases, however, a message disseminates rapidly through the tree such that nodes receive the message from the tree before they receive gossips regarding the message from their overlay neighbors.

If node X has s overlay neighbors, it sends a gossip to its neighbor Y every $s \cdot t$ seconds, where t is the gossip period. Because t is short and s is small (typically s=6, see Section 2.2), usually a gossip is exchanged between two overlay neighbors less than every one second. A gossip can be saved if there is no multicast message during that period. The gossips are small, containing only the IDs of messages received in less than one second. Node X gossips the ID of a message to each of its neighbors only once. After gossiping the message ID to the last neighbor, node X waits for a period b and then reclaims the memory for the message if X receives no further request for the message. The waiting period b should be at least a multiple of the maximum round trip time (RTT) between node X and its neighbors to allow sufficient time for the neighbors to request the message. We set the waiting period b to two minutes.

During some transient periods, the tree may be broken into fragments due to node or link failures. Among the tree fragments, messages propagate through gossips exchanged between overlay neighbors; inside a tree fragment, messages propagate without stop through the remaining tree links that connect the fragment. In Figure 2, suppose node B starts a multicast. It sends the message to its tree neighbor A. If node A fails before forwarding the message to its tree neighbor C, the tree-based multicast would not deliver the message to nodes C, D, E, and F. At some point, node B sends a gossip to node D. Node D discovers that it has not received the message and consequently requests the message from node B. Upon receiving the message, node D immediately forwards the message to its tree neighbors C and F. A receiver of the message further immediately forwards the message without stop along the remaining tree links that connect the tree fragment. The broken tree will be repaired quickly (Sections 2.2 and 2.3), and messages will again purely propagate along the tree.

Multicast messages propagate both unconditionally through the tree and conditionally through gossips exchanged between overlay neighbors at the same time. There is a small chance that a node may receive a message through both channels, redundantly. Suppose a node receives a gossip containing the ID of a message and finds that it has not received the message. It obtains the message from the sender of the gossip but later on the message arrives again from a tree link. In other words, the message propagates faster through gossips than through the tree. The chance for this to happen is very low, because messages typically prop-

agate much faster through the efficient tree. Our simulation shows that, in a 1,024-node system with typical Internet latencies and a gossip period of 0.1 seconds, the probability for the scenario above to happen is only 0.02, i.e., on average each node receives a message 1.02 times.

This 2% overhead can be further reduced in several ways. (1) If node X has already received or is receiving a multicast message discovered through a gossip while another node Y is trying to send X the same message through a tree link, X aborts the transmission with Y immediately. This 2% overhead, therefore, is not in terms of full multicast messages when the messages are large. (2) When a node receives a gossip containing the ID of a message, it delays requesting the message from the sender of the gossip until the message was injected into the system by the source at least f seconds ago. The threshold f is chosen to allow sufficient time for the message to first propagate through the tree. We recommend setting f to the 90th percentile delay for multicast messages to reach nodes through the tree. For the 1,024-node system, setting f=0.3 seconds has almost no impact on the delivery delay of multicast messages while decreasing the probability that a node receives redundant multicast messages to 0.0005. This optimization requires multicast messages and gossips to carry the elapse time since the message was injected into the system, which can be estimated by piggybacking and adding up the propagation delays and waiting times as the message travels away from the source.

In this section, we have described the message dissemination protocol assuming the overlay and the tree are already in place. Next, we proceed to describe the protocols that build the overlay and the tree in a decentralized fashion.

2.2. Protocol for Building the Overlay

Our goal is to build an overlay that is degree constrained and richly connected, and consists of mostly low latency links. The overlay built by our protocol has several salient features. (1) In overlays built by existing protocols, the node degrees are not tightly controlled. By contrast, each node in GoCast has roughly the same number of overlay neighbors in order to spread out the maintenance overhead and the gossip overhead. (2) In existing protocols, a node either has no random neighbor or chooses at least than half of its neighbors at random. By contrast, most nodes in GoCast have exactly one random neighbor, while all other neighbors are chosen based on network proximity. This method produces overlays that are robust as well as efficient.

We choose system parameters carefully to strike a good balance between the conflicting goals of resilience and efficiency. The connectivity of the overlay (i.e., the number of disjoint paths between two nodes) directly affects the dependability of GoCast in the face of node or link failures. Higher node degrees lead to better connectivity but introduce higher protocol overhead because nodes need to maintain more neighbors and gossips are sent to more neighbors. Assuming nodes have similar capacities, we also want node degrees to be as uniform as possible such that the protocol overhead imposed on each node is roughly equal. (Tuning node degree according to node capacity can be accommodated in our protocol but is beyond the scope of this paper.) The overlay is unstructured; it mandates no particular topology. Regardless of the initial structure of the overlay, it adapts automatically such that almost all nodes converge to a target node degree chosen at design time.

Besides the target node degree, another important design choice is the way of selecting node neighbors. It affects the connectivity of the overlay, message delay, and stress on the underlying network links. On the one hand, according to the random graph theory, adding links between random nodes improves the connectivity of the overlay. On the other hand, adding low latency links between nodes that are close in the network lowers message delay, consumes less network resources, and reduces stresses on bottleneck network links. GoCast achieves a good balance by devoting a small number of overlay links to connect random nodes and selecting all other overlay links based on network proximity. Our evaluation shows that this approach results in an overlay that has both low latency and high connectivity.

We define some notations before delving into our protocol. A *component* consists of a group of nodes that are connected directly or indirectly by overlay links. We refer to overlay links that connect randomly chosen neighbors as *random links* and overlay links chosen based on network proximity as *nearby links*. Two nodes directly connected by a random link are *random neighbors* and two nodes directly connected by a nearby link are *nearby neighbors*. Let *random degree* $D_{rand}(X)$ and *nearby degree* $D_{near}(X)$ denote the number of node X's random neighbors and nearby neighbors, respectively. Let C_{degree}, C_{rand}, and C_{near} denote the target node degree, target random degree, and target nearby degree, respectively, $C_{degree} = C_{rand} + C_{near}$. C_{degree}, C_{rand}, and C_{near} are constants chosen at design time. Ideally, every node X has the same degree, $D_{rand}(X) = C_{rand}$ and $D_{near}(X) = C_{near}$.

One major contribution of this paper is the finding of a good setting for these parameters: C_{rand} = 1, and C_{near} = 5. We find that, without any random neighbor (C_{rand} = 0), the overlay is partitioned even without any node or link failure. This is because nearby links do not connect remote components. With just one random neighbor per node (C_{rand} = 1), the connectivity of the overlay is almost as good as that of overlays using multiple random neighbors per node. Intuitively, nearby links connect a group of nodes that are close and random links connect remote groups of nodes. For instance, suppose a system consists of 500 nodes in America and 500 nodes in Asia. With nearby links only, the system is decomposed into two components corresponding to the two geographical areas. Internally, each compo-

nent is richly connected. By adding just one random link to each node (500 random links in total as one link connects two neighbors), we expect an average of 250 random links to connect the America component and the Asia component, which greatly enhances the connectivity of the entire system. Moreover, we find that six neighbors per node provide sufficient connectivity. For instance, with this configuration, systems with thousands of nodes remain connected even after 25% of the nodes fail concurrently.

The target node degrees are chosen at design time. Below, we describe our protocols that enforce the node degrees at run-time and select high-quality links for the overlay.

2.2.1. Node Join

Each node knows a random subset of nodes in the system. This knowledge is maintained by piggybacking the IP addresses of some random nodes on gossips exchanged between overlay neighbors. Due to space limitations, we omit the details of this partial membership protocol. Interested readers are referred to [5, 16]. It has been shown [5] that, for gossip protocols, a "uniformly" random partial member list is almost as good as a complete member list.

When a new node N joins, it knows at least one node P already in the overlay through some out-of-band method. Node N contacts node P to obtain P's member list $\mathcal{S}$. For the time being, node N accepts $\mathcal{S}$ as its member list. Later on, node N may add nodes into or delete nodes from $\mathcal{S}$. Node N randomly selects C_{rand} nodes in $\mathcal{S}$ as its random neighbors and establish a TCP connection with each of them. All communications between overlay neighbors go through these pre-established TCP connections. (On the other hand, communications between nodes that are not overlay neighbors use UDP, e.g., RTT measurements between non-neighbor nodes.)

Among nodes in $\mathcal{S}$, ideally, node N should select those that have the lowest latencies to N as N's nearby neighbors. However, $\mathcal{S}$ can be large, including hundreds of nodes. It would introduce a large amount of traffic and long waiting time for N to measure RTTs to every node in $\mathcal{S}$. Instead, node N uses an algorithm to estimate network distance and chooses C_{near} nodes in $\mathcal{S}$ that have the smallest estimated latencies to N as its initial set of nearby neighbors. Later on, node N gradually measures RTTs to nodes in $\mathcal{S}$ and switches from long latency links to low latency links, thereby improving the efficiency of the overlay over time. We use the triangular heuristic [13] to estimate latencies. Details are omitted due to space limitations.

If the new node N chooses a node X as its neighbor, N sends a request to X. X accepts this request only if its node degrees are not too high—for adding a random link, $D_{rand}(X) < C_{rand} + 5$; for adding a nearby link $D_{near}(X) < C_{near} + 5$. If the constraint is not met, node N has to try another node. For node X to accept node N as its nearby neighbor, it must also satisfy condition C2 to be described in Section 2.2.3. Intuitively, this condition stipulates that, if accepted, the link between nodes N and X must not be the worst overlay link that X maintains.

This node join protocol guarantees that nodes do not have an excessive number of neighbors but it cannot ensure that node degrees $D_{rand}(X)$ and $D_{near}(X)$ eventually converge to the target degrees C_{rand} and C_{near}. The overlay adaptation protocols described in the following sections achieve this goal. They also automatically handle node departures and failures. After a node leaves, its previous neighbors will choose other nodes as their new neighbors.

2.2.2. Maintaining Random Neighbors

Periodically every r seconds, each node X executes a protocol to maintain its random neighbors and a protocol to maintain its nearby neighbors, respectively. The period r is dynamically tunable according to the stability of the overlay (i.e., the need for maintenance), r = 0.1 seconds in our current design. Although the period r is short, most of the time no operation is needed during a maintenance. Also note that the maintenance cost and gossip overhead at a node is independent of the size of the system.

If node X's random degree $D_{rand}(X)$ is equal to the target random degree C_{rand}, no action is needed. If $D_{rand}(X) < C_{rand}$ (which may occur due to, for instance, the failures of X's random neighbors), X randomly selects a node from its member list and establishes a random link to the node. If $D_{rand}(X) > C_{rand}$, node X tries to drop some random neighbors through one of the operations below.

1. If $D_{rand}(X) \geq C_{rand} + 2$, node X randomly chooses two of its random neighbors Y and Z and asks Y to establish a random link to Z. Node X then drops its random links to nodes Y and Z. By doing this, node X's random degree is reduced by two, while the random degrees of nodes Y and Z are not changed.
2. If one of node X's random neighbors W has more than C_{rand} random neighbors, node X drops the random link between X and W. This reduces the random degrees of both X and W by one while still keeping their random degrees equal to or larger than C_{rand}.

If neither of the conditions above is met, node X's random degree must be $C_{rand} + 1$ and all X's random neighbors must have random degrees equal to or smaller than C_{rand}. In this case, no action is taken and node X's random degree remains at $C_{rand}+1$. It can be proved that, when the overlay stabilizes, each node eventually has either C_{rand} or $C_{rand} + 1$ random neighbors. Our evaluation shows that approximately 88% of nodes have C_{rand} random neighbors and 12% of nodes have $C_{rand} + 1$ random neighbors.

2.2.3. Maintaining Proximity Aware Neighbors

In addition to maintaining its random neighbors, every r seconds, a node X also executes a protocol to maintain its

nearby neighbors. This protocol differs from the protocol for maintaining random neighbors in that it considers network proximity when adding or dropping links. It tries to confine node X's nearby degree to either C_{near} or C_{near}+1, while choosing nodes that are close to X as X's nearby neighbors. Node X runs three sub-protocols: one to replace X's long latency nearby links with low latency links; one to add more nearby links when $D_{near}(X) < C_{near}$; and one to drop long latency nearby links when $D_{near}(X) \geq C_{near}$+2.

Replacing Nearby Neighbors

Node X sorts nodes in its member list $\mathcal{S}$ in increasing estimated latency. Starting from the node with the lowest estimated latency, node X measures real latencies to nodes in $\mathcal{S}$ one by one. During each maintenance cycle (every r seconds), node X measures RTT to only one node in $\mathcal{S}$. As the overlay stabilizes, the opportunity for improvement diminishes. The maintenance cycle r can be increased accordingly to reduce maintenance overheads. We leave the dynamic tuning of r as a subject of future work.

Suppose node X measures RTT to node Q in the current maintenance cycle. Node X will add node Q as its new nearby neighbor and drop its existing nearby neighbor U if all the conditions below are met.

C1. Node X has at least one nearby neighbor U whose current nearby degree is not too low: $D_{near}(U) \geq C_{near} - 1$. Otherwise, the degrees of node X's all nearby neighbors are dangerously low. Dropping a link to one of them would endanger the connectivity of the overlay. Among node X's nearby neighbors that satisfy this condition, the node U to be replaced is chosen as the neighbor that has the longest latency to node X.

C2. $D_{near}(Q) < C_{near} + 5$. This requires that the nearby degree of the new neighbor candidate Q is not too high.

C3. If $D_{near}(Q) \geq C_{near}$, then $\text{RTT}(X, Q) < \text{max\_nearby\_RTT}(Q)$ must hold. Here $\text{RTT}(X, Q)$ is the RTT between node X and the new neighbor candidate Q, and $\text{max\_nearby\_RTT}(Q)$ is the maximum RTT between node Q and Q's nearby neighbors. If this condition is not met, node Q already has enough nearby neighbors and the link between nodes Q and X is even worse than the worst nearby link that Q currently has. Even if node X adds the link to node Q now, Q is likely to drop the link soon. Hence node X does not add this link.

C4. $\text{RTT}(X, Q) \leq \frac{1}{2} \cdot \text{RTT}(X, U)$. Here node Q is the new neighbor candidate and node U is the neighbor to be replaced (selected by condition C1). Intended to avoid futile minor adaptations, this condition stipulates that node X adopts new neighbor Q only if Q is significantly better than the current neighbor U.

Among many heuristics we tested, the conditions above are particularly effective in resolving many conflicting goals—upholding the connectivity of the overlay during adaptation, minimizing the total number of link changes without global information, and converging to a stable state quickly. For instance, condition C1 is a good example of the tradeoff we made. Because of condition C1, node U's nearby degree can be as low as $C_{near} - 2$ in a transient period after node X drops the link to U and before U adds more nearby links in the next maintenance cycle. This lower degree bound can be increased to $C_{near} - 1$ if we change condition C1 from $D_{near}(U) \geq C_{near} - 1$ to $D_{near}(U) \geq C_{near}$. However, our evaluation shows that this change would produce an overlay whose link latencies are dramatically higher than that produced by our current solution, because fewer neighbors satisfy this new condition to qualify as a candidate to be replaced. With our recommended setting, C_{rand} = 1 and C_{near} = 5, the lower bound of a node's degree during adaptation is 4, which is sufficiently high to uphold the connectivity of the overlay during short transient periods.

Originally, node X sorts nodes in its member list $\mathcal{S}$ in increasing estimated latency and measures RTTs to them one by one. Once all nodes in $\mathcal{S}$ have been measured, the estimated latencies are no longer used. But node X still continuously tries to replace its current nearby neighbors by considering candidate nodes in $\mathcal{S}$ in a round robin fashion. The hope is that some nodes that previously do not satisfy some of the conditions C1-C4 now can meet all of them and hence can be used as new nearby neighbors.

Adding Nearby Neighbors

If node X has less than C_{near} nearby neighbors, X needs to add more nearby neighbors in order to uphold the connectivity of the overlay. To spread out the load, during each maintenance cycle, node X adds at most one new nearby neighbor. Similar to the process to replace nearby neighbors, node X selects a node Q from its member list $\mathcal{S}$ in a round robin fashion and adds Q as its new nearby neighbor if both conditions C1 and C2 are met. These conditions stipulate that Q does not have an excessive number of neighbors and the link between nodes X and Q is no worse than the worst nearby link that Q currently has.

Dropping Nearby Neighbors

If node X has an excessive number of nearby neighbors (e.g., some new nodes have added links to X), X starts to drop some nearby neighbors to reduce unnecessary protocol overheads. Although the target nearby degree is C_{near}, node X starts to drop nearby neighbors only if $D_{near}(X) \geq C_{near} + 2$. This allows a node's nearby degree to stabilize at either C_{near} or $C_{near}+1$. Our evaluation shows that, under this strategy, eventually about 70% of nodes have C_{near} nearby neighbors and about 30% of nodes have $C_{near}+1$ nearby neighbors. One alternative is to drop one more nearby neighbor when $D_{near}(X) = C_{near} + 1$. Our evaluation shows that, compared with our current solution, this

aggressive strategy increases the number of link changes by almost one third and it takes longer to stabilize the overlay.

When $D_{near}(X) \geq C_{near} + 2$, node X tries to drop $D_{near}(X) - C_{near}$ nearby neighbors. The candidate neighbors to drop are those that satisfy condition C1 above, i.e., nodes U whose nearby degree is not dangerously low, $D_{near}(U) \geq C_{near} - 1$. Again, avoiding dropping links to low degree nodes helps uphold the connectivity of the overlay during adaptation. Node X sorts its nearby neighbors that satisfy this condition and drops those that have the longest latencies to X until X's nearby degree is reduced to C_{near} or no nearby neighbor satisfies condition C1.

2.3. Protocol for Building the Tree

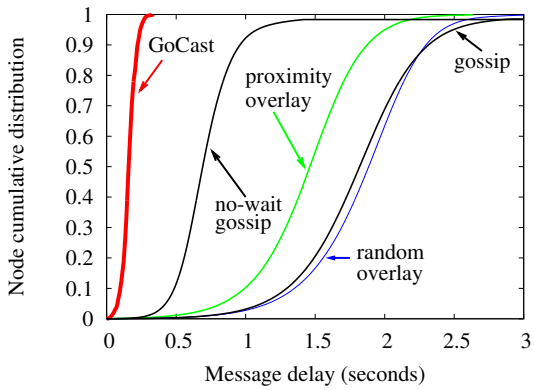
GoCast selects overlay links in a decentralized fashion to construct an efficient tree embedded in the overlay. The tree spans over all nodes and propagates messages rapidly. The algorithm to build the tree is in spirit similar to the classical Distance Vector Multicast Routing Protocol (DVMRP) [15], but note that GoCast only needs a single tree. The tree conceptually has a root and the tree links are overlay links on the shortest paths (in terms of latency) between the root and all other nodes. If the root fails, one of its neighbors will take over its role. Originally, the first node in the overlay acts as the root. Periodically every 15 seconds, the root floods a heartbeat message throughout every link in the overlay to help detect failures (e.g., partitioning) of the overlay and the tree. Due to space limitations, we omit the details of the tree protocol as it is pretty standard.

3. Experimental Results

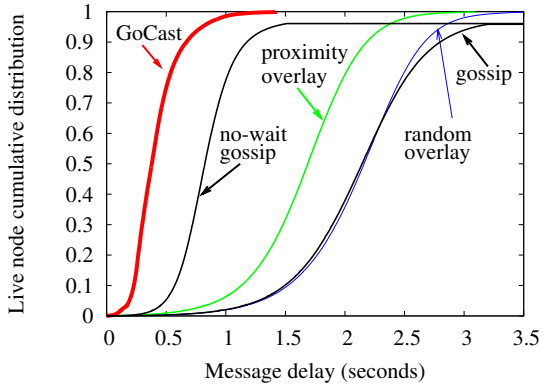
We built an event-driven simulator to evaluate GoCast. The simulator consists of 6,100 lines of C++ code and runs on Linux. It simulates a complete system, including message propagation, node and link failure, network topology, and link latency. We do not simulate the network-level packet details in order to scale to thousands of nodes. Simulating one run of an 8,192-node system on a 2.4GHz machine takes about three hours.

Our simulator uses real Internet latencies from the King dataset [4], which is extracted from real measurements of RTTs between 1,740 DNS servers in the Internet. (The original dataset contains more than 1,740 servers but we exclude those servers with empty measurements). We divide the RTTs by two to obtain one-way latencies. The average and maximum one-way latency is 91ms and 399ms, respectively. When the number of simulated nodes is larger than the number of measured DNS servers, we simulate multiple nodes at a single DNS server site.

Unless otherwise noted, we report results on a 1,024-node system and the simulation works as follows. Initially, all nodes start at the same time and one random node is designated as the root of the tree. For a target node degree



(a) No node fails.



(b) 20% of nodes fail.

Figure 3. The propagation delay of multicast messages under different protocols (1,024 nodes).

C_{degree}, each node initiates connections to $\frac{C_{degree}}{2}$ random nodes. After the initialization, the average node degree is C_{degree} and all neighbors are chosen at random. The overlay and the tree then adapt under GoCast's maintenance protocols for 500 seconds in simulated time. After 500 seconds, multicast messages are injected into the overlay from random source nodes at a rate of 100 messages per second.

The parameters of GoCast are as follows. A node sends out one gossip every t = 0.1 seconds (suggested by Bimodal Multicast [2]). Every r = 0.1 seconds, a node wakes up to maintain its neighbors. The default target node degrees are $C_{rand} = 1$ and $C_{near} = 5$.

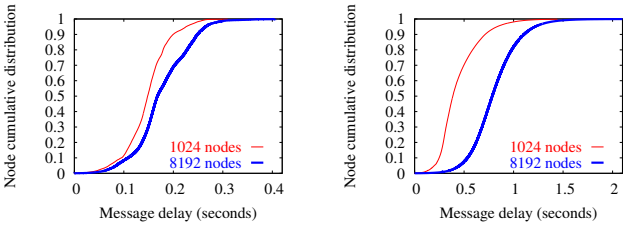
We first present results that compare the message propagation delay under different multicast protocols. The delay is averaged over 1,000 multicast messages injected from random source nodes. Figure 3(a) shows results for the ideal case when no node fails. The "GoCast" curve represents the complete GoCast protocol, in which messages propagate both through the tree and through gossips exchanged between overlay neighbors. The "proximity overlay" and "random overlay" curves represent simplified versions of

the GoCast protocol that only propagate messages through gossips exchanged between overlay neighbors; the system neither maintains nor uses the tree. "Proximity overlay" and "random overlay" differ in that each node in "proximity overlay" maintains 5 nearby neighbors and 1 random neighbor, while each node in "random overlay" maintains 6 random neighbors only.

The "gossip" curve in Figure 3(a) represents a push-based gossip protocol similar to that used in Bimodal Multicast. Every t = 0.1 seconds, each node sends a gossip to a random node. The gossip fanout is 5, i.e., a node gossips the ID of a received multicast message to 5 random nodes (one node per gossip period). The receiver of a gossip requests some multicast messages from the sender if the contents of the gossip suggest that those multicast messages are unknown to the receiver. The system represented by the "no-wait gossip" curve is different. In "no-wait gossip", upon receiving a multicast message, a node immediately gossips the message to 5 other nodes without waiting for the next gossip period (in other words, the gossip period t = 0). When message rate is high, "no-wait gossip" introduces higher gossip traffic than "gossip". We use it here to reveal the fundamental performance limits of gossip multicast.

The complete GoCast protocol (the "GoCast" curve) disseminates messages significantly faster than all other protocols, because multicast messages mainly propagate without stop through the efficient tree that consists of low latency links. Averaged over 1,000 runs, multicast messages reach every node under 0.33 seconds. Gossip multicast (the "gossip" curve) is the slowest in message propagation. Moreover, with fanout 5, some nodes never receive some of the 1,000 multicast messages due to the complete randomness of gossips. A shorter gossip period can reduce message delay at the expense of increased gossip traffic. However, even "no-wait gossip" is not as fast as "GoCast" for two reasons. (1) Tree links in GoCast have low latencies. (2) To avoid sending potentially large multicast messages redundantly, "no-wait gossip" always sends gossips first and then sends the actual multicast messages upon requests, which incurs extra delay. As in "gossip", some nodes in "no-wait gossip" never receive some multicast messages. The message delay in "random overlay" is similar to that in "gossip" but every node in "random overlay" receives every message, which is guaranteed by the connectivity of the overlay. Owing to its low latency links, "proximity overlay" propagates messages noticeably faster than "random overlay" and "gossip".

Figure 3(b) puts the different protocols under a stress test when 20% of nodes fail concurrently at simulated time 500 seconds. The failed nodes are selected uniformly at random. After node failures, multicast messages are injected into the system but the system does not execute any of GoCast's maintenance protocols to repair the overlay or the tree. The Y axis represents the cumulative distribution of live nodes. With 20% nodes down, the overlay still remains connected.



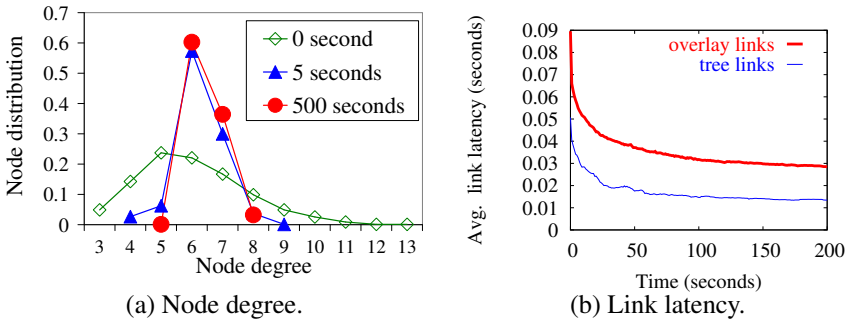
(a) No node fails. (b) 20% of nodes fail.

Figure 4. Comparing message delays in GoCast between a 1,024-node system and an 8,192-node system.

"GoCast", "proximity overlay", and "random overlay" still deliver every multicast message to every live node. Comparing the "gossip" curves in Figure 3(a) and 3(b), we see that, for gossip multicast, a higher fraction of live nodes do not receive some multicast messages in the face of node failures. In terms of message delay, "GoCast" is more sensitive to node failures than the other four protocols. This is because, without any repair, the tree in GoCast is broken into fragments and a message has to propagate among the fragments through gossips. Once the message hits a node in a tree fragment, it propagates without stop to other nodes in the fragment through remaining tree links that connect the fragment. This is the major reason why "GoCast" still significantly outperforms "proximity overlay".

Figures 4(a) and 4(b) compare the delay of multicast messages in the complete GoCast protocol between a 1,024-node system and an 8,192-node system. When no node fails, the difference between the two systems is small. For the 8,192-node system, averaged over 1,000 multicast messages, a message reaches all nodes under 0.42 seconds. When 20% of nodes fail, the difference between the two systems becomes larger. The curve for the 8,192-node system has a longer tail. The message delay for a small number of nodes in the 8,192-system is about 60% longer than the longest delay in the 1,024-node system. When a significant fraction of nodes fail concurrently, the tree in the larger system is broken into more fragments than that in the smaller system. Multicast messages propagate among the tree fragments through slow gossips, which is the reason why the difference in Figure 4(b) is more significant than that in Figure 4(a). Overall, the increase in message delay is moderate as the system size increases by eight fold, indicating that GoCast is scalable.

We next evaluate GoCast's ability to adapt the overlay and the tree. This experiment simulates a 1,024-node system. Initially, all nodes start at the same time and each node initiates three random links. After initialization, the average node degree is six. GoCast's maintenance protocols then adapt the overlay and the tree over time. The target random degree is one and the target nearby degree is five.



(a) Node degree. (b) Link latency.

Figure 5. Adaptation of the overlay and the tree (1,024 nodes).

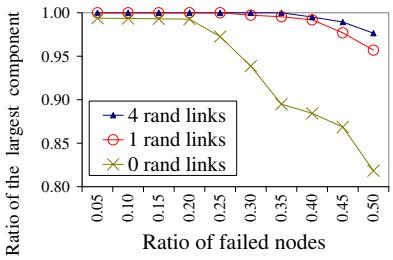


Figure 6. The largest component after node failures.

Figure 5(a) shows the distribution of node degrees (i.e., the sum of random degree and nearby degree) over time. Initially, the node degrees vary dramatically (see the "0 second" curve); only 22% of nodes have the target degree six. Under the GoCast maintenance protocols, node degrees converge quickly. After 5 seconds, 57% of nodes have degree six; after 500 seconds, 60% of nodes have degree six and the average node degree is 6.4.

Figure 5(b) shows the average latency of overlay links and tree links over time. This figure only plots the first 200 seconds to make the interesting beginning phase readable. The latency of overlay links is averaged over both random links (C_{rand} = 1) and nearby links (C_{near} = 5). Initially, all overlay links are chosen at random. In the beginning phase, the quality of the links improves quickly as many of those long latency links are replaced with low latency links. This improvement is the major reason why "proximity overlay" propagates messages faster than "random overlay" in Figure 3. After about 60 seconds, the improvement slows down as it gets harder and harder for nodes to find better neighbors. The latency of tree links is lower than that of overlay links because GoCast tends to select low latency overly links to construct the tree. After 100 seconds, the average latency of tree links is only 15.5ms in contrast to the 91ms average latency between random nodes. This good link quality is part of the reason why "GoCast" dramatically outperforms all other protocols in Figure 3.

We now turn our attention to the impact of the number of random links on the resilience of the overlay. In Figure 6, we vary the ratio of failed nodes from 0.05 to 0.5 (i.e., 5% to 50%) and report the ratio q of live nodes that are in the largest connected component (q = 1 if the overlay remains connected). Without any random link, the overlay is already partitioned even without any failure because remote components are not connected. With just one random link, the overlay remains connected even after 25% nodes fail concurrently. The difference in resilience between one random link per node and four random links per node is small, which justifies our use of one random link per node.

Due to space limitations, we briefly summarize other simulation results as follows. The detailed data will be presented in a longer version of this paper. (1) Starting with a random structure with random links only, the overlay converges quickly to a stable state under our adaptation protocols. The number of changed links per second drops exponentially over time. (2) The average latency of the overlay links grows almost linearly with the number of random links, which again justifies our use of only one random link per node. (3) The overlay is scalable; the diameter of the overlay grows from 6 hops to 10 hops when the system size increases from 256 nodes to 8,192 nodes. (4) Compared with a push-based gossip protocol using fanout 5, GoCast reduces the traffic imposed on bottleneck network links by a factor of 4-7. The network topologies used in this experiment are large-scale snapshots of the Internet Autonomous Systems. (5) The message delay in the push-based gossip protocol cannot be reduced significantly by simply increasing the gossip fanout. When the fanout is increased from 5 to 9, the message delay is reduced by only about 5%; further increasing the fanout to 15 has virtually no impact on the message delay.

4. Related Work

GoCast finds context in overlay networks, multicast protocols, and gossip protocols, among which Araneola [12] and Bimodal Multicast [2] are most relevant to GoCast.

In parallel to our development in GoCast, Melamed and Keidar recently proposed Araneola [12]. It organizes nodes into an overlay network and runs gossip protocols between overlay neighbors to implement dependable multicast. Araneola and GoCast use similar protocols to adjust a node's random neighbors. There are several major differences between Araneola and GoCast. (1) Araneola only propagates multicast messages through gossips exchanged between overlay neighbors. By contrast, most of the time GoCast propagates messages without stop through the efficient tree. (2) Araneola has no tight control on the number of nearby neighbors that a node keeps. (3) In Araneola, at least half of the overlay links are chosen at random.

Bimodal Multicast [2] disseminates messages in two phases, first through best-effort unreliable multicast and then through gossips exchanged between random nodes. Both Bimodal Multicast and GoCast intend to combine the best of tree-based multicast and gossip multicast, but their approaches and resulted performances are quite different. (1) Bimodal Multicast gossips message summaries between random nodes, which has the shortcomings described in Section 1, for instance, requiring a high fanout to achieve high reliability and imposing high loads on bottleneck network links due to its obliviousness to the underlying network topology. (2) Bimodal Multicast uses a manually configured hierarchy for unreliable multicast while GoCast automatically creates and adapts the tree. (3) In Bimodal Multicast, nodes that miss a message in the best-effort multicast phase will only receive the message through gossips, which can be slow. By contrast, even if the tree is broken into fragments, GoCast still tries to exploit the remaining tree links to rapidly propagate messages within the tree fragments.

Directional gossip [11] intends to address gossip protocols' obliviousness to network topology. It assumes nodes are connected into a topology-aware graph and prefers to propagate gossips along low-cost links. How to construct and maintain this graph is not specified. By contrast, GoCast builds the topology-aware overlay automatically.

Multicast protocols that use message retransmissions to improve reliability include SRM [7] and RMTP [10]. Narada [3] builds a mesh topology of all multicast members, and then compute a multicast spanning tree for each source. On the other hand, NICE [1] explicitly forms the multicast tree without building a mesh.

5. Conclusions

We proposed GoCast for fast and dependable group communication. GoCast builds a proximity-aware overlay network that has tightly controlled node degrees. Multicast messages propagate rapidly through an efficient tree embedded in the overlay. In the background, nodes gossip message summaries with their overlay neighbors and pick up missing messages due to disruptions in the tree-based multicast. We made the following contributions.

- We are among the first to enhance tree-based overlay multicast with gossips exchanged between overlay neighbors to achieve both stable throughput and fast delivery of multicast messages.
- We are among the first to suggest that using one random link per node is almost as robust as using multiple random links per node.
- Our overlay adaptation protocol is among the first that produces a proximity-aware overlay that has tightly controlled node degrees.
- We are among the first to evaluate gossip protocols using large-scale real Internet topology and latency data, rather than just reporting message delays in terms of abstract gossip rounds. We systematically compared several protocols through extensive simulations.

GoCast delivers multicast messages fast and reliably, and imposes low loads on network links. These features make GoCast attractive for many applications. In our scalable and intelligent infrastructure project at IBM Research, we are using GoCast to facilitate the management of large systems.

References

[1] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable Application Layer Multicast. In *SIGCOMM*, 2002.

[2] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal Multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, 1999.

[3] Y. Chu, S. G. Rao, and H. Zhang. A Case for End System Multicast. In *SIGMETRICS*, 2000.

[4] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a DHT for Low Latency and High Throughput. In *NSDI*, 2004. The network latency data are available at http://www.pdos.lcs.mit.edu/p2psim/kingdata.

[5] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight Probabilistic Broadcast. *ACM Trans. Comput. Syst.*, 21(4):341–374, 2003.

[6] P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulie. From Epidemics to Distributed Computing. To appear in IEEE Computer Magazine, 2004. http://www.irisa.fr/paris/Biblio/Papers/Kermarrec/EugGueKerMas04IEEEComp.pdf.

[7] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, December 1997.

[8] X. Gu, K. Nahrstedt, R. N. Chang, and C. Ward. QoS-Assured Service Composition in Managed Service Overlay Networks. In *ICDCS*, 2003.

[9] R. M. Karp, C. Schindelhauer, S. Shenker, and B. Vocking. Randomized Rumor Spreading. In *FOCS*, 2000.

[10] J. C. Lin and S. Paul. RMTP: A Reliable Multicast Transport Protocol. *INFOCOM*, 1996.

[11] M.-J. Lin and K. Marzullo. Directional Gossip: Gossip in a Wide Area Network. In *European Dependable Computing Conference*, 1999.

[12] R. Melamed and I. Keidar. Araneola: A Scalable Reliable Multicast System for Dynamic Environments. In *the 3rd IEEE International Symposium on Network Computing and Applications (IEEE NCA)*, 2004.

[13] T. S. Eugene Ng and H. Zhang. Predicting Internet Network Distance with Coordinates-Based Approaches. In *INFOCOM*, 2002.

[14] O. Ozkasap, Z. Xiao, and K. P. Birman. Scalability of Two Reliable Multicast Protocols. Technical Report TR99-1748, Cornell University, 1999.

[15] L. L. Peterson and B. S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann Publishers, 2000.

[16] K. Shen. Structure management for scalable overlay service construction. In *NSDI*, 2004.