GO: Platform Support for Gossip Applications

YMIR VIGFUSSON
IBM Haifa Research Lab and Cornell University
QI HUANG
Huazhong University of Science & Technology and Cornell University
KEN BIRMAN
Cornell University
and
DEEPAK P. NATARAJ
Cornell University

Gossip-based protocols are increasingly popular in large-scale distributed applications that disseminate updates to replicated or cached content. **GO** (Gossip Objects) is a per-node gossip platform that we developed in support of this class of protocols. In addition to making it easy to develop new gossip protocols and applications, **GO** allows nodes to join multiple gossip groups without losing the appealing fixed bandwidth guarantee of gossip protocols. Our heuristic is based on the observations that multiple rumors can often be squeezed into a single IP packet, and that indirect routing of rumors can speed up delivery. We formalize these observations and develop a heuristic that optimizes rumor delivery latency in a principled manner. We have implemented **GO**, and study the effectiveness of the heuristic by comparing it to the more standard random dissemination gossip strategy via simulation. We also evaluate **GO** on a trace from a popular distributed application.

Categories and Subject Descriptors: C.2.4 [**Computer Communication**]: Distributed Systems

General Terms: Design, Experimentation, Algorithms

Additional Key Words and Phrases: Gossip, Epidemic broadcast, Multicast

1. INTRODUCTION

Gossip-based communication is commonly used in distributed systems to disseminate information and updates in a scalable and robust manner [Demers et al. 1987; Kempe et al. 2001; Birman et al. 1998]. The idea is simple: At some fixed frequency, each node sends or exchanges information (known as *rumors*) with a randomly chosen peer in the system, allowing rumors to propagate to everybody in an "epidemic fashion".

The basic gossip exchange can be used for more than just sharing updates. Gossip protocols have been proposed for scalable aggregation [Jelasity et al. 2005], monitoring and distributed querying [van Renesse et al. 2003], constructing distributed hash tables (DHTs) [Gupta et al. 2003] and other kinds of overlay structures [Wong et al. 2005], orchestrating self-repair in complex networks and even for such prosaic purposes as to support shopping carts for large data centers [Decandia et al. 2007]. By using gossip to track group membership, one can implement gossip-based group multicast protocols.

Notice that the DHT we cited, Kelips, is less well known than the most widely popular distributed hash tables, such as Chord and Pastry [Stoica et al. 2001;

Rowstron and Druschel 2001]. This was deliberate: Kelips is purely based on gossip, whereas the others are peer-to-peer.

When considered in isolation, gossip protocols have a number of appealing properties.

P1. **Robustness**. They can sustain high rates of message loss and crash failures without reducing reliability or throughput [Birman et al. 1998], as long as several assumptions about the implementation and the node environment are satisfied [Alvisi et al. 2007].

P2. **Constant, balanced load**. Each node initiates exactly one message exchange per round, unlike leader-based schemes in which a central node is responsible for collecting and dispersing information. Since message exchange happens at fixed intervals, network traffic overhead is bounded [van Renesse et al. 1998].

P3. **Simplicity**. Gossip protocols are simple to write and debug. This simplicity can be contrasted with non-gossip styles of protocols, which can be notoriously complex to design and reason about, and may depend upon special communication technologies, such as IP multicast [Deering 1989], or embody restrictive assumptions, such as the common assumption that any node can communicate directly with any other node in the application.

P4. **Scalability**. All of these properties are preserved when the size of the system increases, provided that the capacity limits of the network are not reached and the information contained in gossip messages is bounded.

However, gossip protocols also have drawbacks. The most commonly acknowledged are the following. **(i)** The basic gossip protocol is probabilistic meaning that some rumors may be delivered late, although this occurs with low probability. **(ii)** The expected number of rounds required for delivery in gossip protocols is logarithmic in the number of nodes. Consequently, the latency of gossip protocols is on average higher than can that provided by systems using hardware accelerated solutions like IP multicast [Birman et al. 1998]. **(iii)** Moreover, gossip protocols support only the weak guarantee of *eventual consistency* — updates may arrive in any order and the system will converge to a consistent state only if updates cease for a period of time. Applications that need stronger consistency guarantees must employ more involved and expensive message passing schemes [Pease et al. 1980]. Weak consistency is not *always* a bad thing, of course. Relaxing consistency guarantees has become increasingly popular in large-scale industrial applications such as Amazon's Dynamo [Decandia et al. 2007] and Yahoo!'s PNUTS [Cooper et al. 2008].

Gossip also has a less-commonly recognized drawback. An assumption frequently seen in the gossip literature is that all nodes belong to a single gossip group. Since such a group will often exist to support an application component, we will also call these *gossip objects*. While sufficient in individual applications, such as when replicating a database [Demers et al. 1987], an object-oriented style of programming would encourage applications to use multiple objects and hence the nodes hosting those applications will belong to multiple gossip groups. The trends seen in other object oriented platforms (*e.g.*, Jini and .NET) could carry over to gossip objects, yielding systems in which each node in a data center hosts large numbers of gossip objects. These objects would then contend for network resources and could interfere with one-another. The gossip-imposed load on each node in the network now depends on the number of gossip objects hosted on that node, which violates property P2.

We believe that this situation argues for a new kind of operating system extension focused on nodes that belong to multiple gossip objects. Such a platform can play multiple roles. First, it potentially simplifies the developer's task by standardizing common operations, such as tracking the neighbor set for each node or sending a rumor, much as a conventional operating system simplifies the design of client-server applications by standardizing remote method invocation. The platform combines the various styles of gossip, such as aggregation and neighbor-set management, into a single API focused on rumor-mongering. The application can generate and interpret rumors as it wishes. Second, the platform can implement fair-sharing policies, ensuring that when multiple gossip applications are hosted on a single node, they each get a fair share of that node's communication and memory resources. Finally, the platform will have opportunities to optimize work across independently developed applications – the main focus of the present paper. For example, if applications A and B are each replicated onto the same sets of nodes, any gossip objects used by A will co-reside on those nodes with ones used by B. To the extent that the platform can sense this and combine their communication patterns, overheads will be reduced and performance increased.

With these goals in mind, we built a per-node service called the Gossip Objects platform (**GO**) which allows applications to join large numbers of gossip groups in a simple fashion. The initial implementation of **GO** provides a multicast-like interface: local applications can join or leave gossip objects, and send or receive rumors via callback handlers that are executed at particular rates. Down the road, the **GO** interface will be extended to support other styles of gossip protocols, such as the ones listed earlier. In the spirit of property P2, the platform enforces a configurable per-node bandwidth limit for gossip communication, and will reject a join request if the added gossip traffic would cause the limit to be exceeded. The maximum memory space used by **GO** is also limited and customizable.

GO incorporates optimizations aimed at satisfying the gossip properties while maximizing performance. Our first observation is that gossip messages are frequently short: perhaps just a few tens of bytes. Some gossip systems push only rumor version numbers to minimize waste [van Renesse et al. 1998; Balakrishnan et al. 2007], so if the destination node does not have the latest version of the rumor, it can request a copy from the exchange node. An individual rumor header and its version number can be represented in as little as 12-16 bytes. The second observation is that there is negligible difference in operating system and network overhead between a UDP datagram packet containing 10 bytes or 1000 bytes, as long as the datagram is not fragmented [von Eicken et al. 1995]. It follows from these observations that *stacking* multiple rumors in a single datagram packet from node s to d is possible and imposes practically no additional cost. The question then becomes: *Which rumors should be stacked in a packet?* The obvious answer is to include rumors from all the gossip objects of which both s and d are members. **GO** takes this a step further: s will sometimes include rumors for gossip objects that d is not interested in, and when this occurs, d will attempt to forward those rumors to nodes that will benefit from them. We formalize rumor stacking and *message indirection* by defining the *utility* of a rumor in Section 3.

We envision a number of uses for **GO**. Within our own work, **GO** will be the WAN communication layer for Live Distributed Objects (LDO), a framework for abstract components running distributed protocols that can be composed easily to create custom and flexible live applications or web pages [Ostrowski et al. 2008; Birman et al. 2007]. This application is a particularly good fit for **GO**: Live Objects is itself an object-oriented infrastructure, and hence it makes sense to talk about objects that use gossip for replication. We describe the LDO platform in Section 2.1. The **GO** interface can also be extended to resemble a gossip-based publish/subscribe system [Eugster et al. 2003]. Finally, **GO** could be used as a kind of IP tunnel, with end-to-end network traffic encapsulated, routed through **GO**, and then de-encapsulated for delivery. Such a configuration would convert a conventional distributed protocol or application into one that shares the same gossip properties enumerated earlier, and hence might be appealing in settings where unrestricted direct communication would be perceived as potentially disruptive.

Our paper focuses on the initial implementation of **GO**, and makes the following contributions:

— A natural extension of gossip protocols in which multiple gossip objects can be hosted on each node.

— A novel heuristic and a mathematical framework to exploit the similarity of gossip groups to improve propagation speed and scalability.

— An evaluation of the **GO** platform on a real-world trace.

2. PLATFORM ARCHITECTURE

We will review the basic architecture of the Gossip Objects (**GO**) platform, and the Live Distributed Objects (LDO) platform that was used when developing it.

2.1 Live Distributed Objects

Figure 1 illustrates the basic functions of the LDO system. It displays two applications constructed as graphs (mashups) of event-oriented components that interact over typed event channels. The individual object instances often encapsulate some form of replicated data or functionality abstraction, such as a coordination protocol, a fault-tolerant state machine, or a data object that uses atomic multicast for updates. The objects on any given machine should thus be thought of as *proxies* – endpoints of a distributed abstraction. Object proxies for any single object can peer with one another and exchange messages using standard Internet protocols; the LDO platform will help them find one-another, performs type checking, assists in loading the correct object instances, and provides other basic functionality.

We developed the **GO** platform using the LDO system because we wanted to leverage the convenience and simplicity of its component-composition architecture. LDO applications can often be constructed in a "drag-and-drop" manner, pulling content (represented by LDO proxies) from various sources, which could include cloud repositories, peer-to-peer protocols, sensors, media streams, and so forth [Ostrowski et al. 2008]. Once assembled and type checked, an LDO application

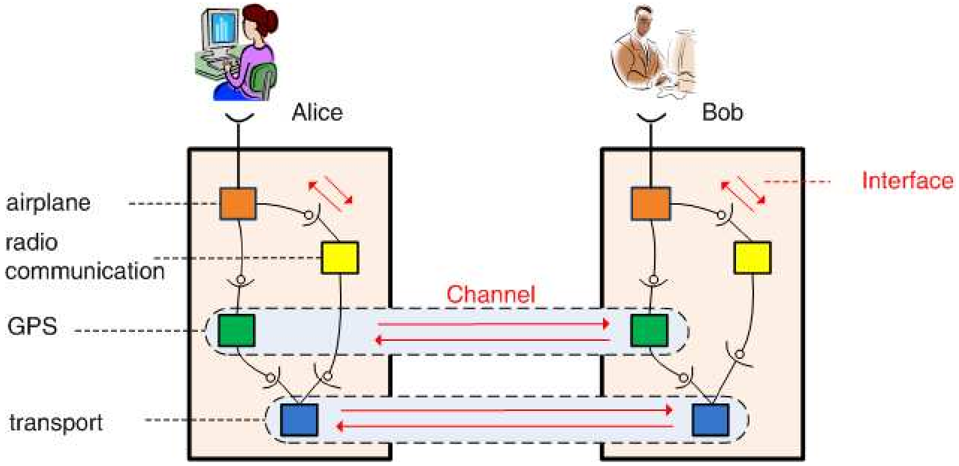


Fig. 1. Two users running the same application in the LDO system. An application is a set of components that interact over types channels. Each component can be implemented as a **GO** object.

may be saved in an XML representation, then shared perhaps via e-mail or a web page. When other users open the application, the necessary proxies are created, connected together, and assisted in contacting their peers. All of these properties carry over to our gossip objects applications.

2.1.1 *Example.* Suppose the LDO application in Figure 1 is tracking the location of an aircraft. The object at the top of the graph might be responsible for rendering the airplane against a backdrop such as a map. The two stacks of objects below it could be tracking relevant information: perhaps, GPS data in the stack on the left and ground-to-pilot communication in the stack on the right. The object at the bottom of the graph might be a secure atomic multicast protocol used by these stacks to replicate GPS coordinate data and interactions between ground and plane. The multicast objects at the bottom of the stack would interact with their peers on other machines using, for example, UDP over the Internet.

2.2 The **GO** Platform

The **GO** platform is seen in Figure 2.2, which focuses on the key elements of the system as used on a single host machine. We see an end-user, who interacts with gossip objects (three of them in this case) through whatever interface is employed by the object designer. The objects themselves do not send gossip directly: at whatever frequency was selected by the designer, they initiate rumors for transmission to their peers. The management of the list of peers and of the rumor buffer is standardized by **GO**: each object can use its own gossip-based algorithm for deciding which peers will be its neighbors, but **GO** holds the resulting neighbor

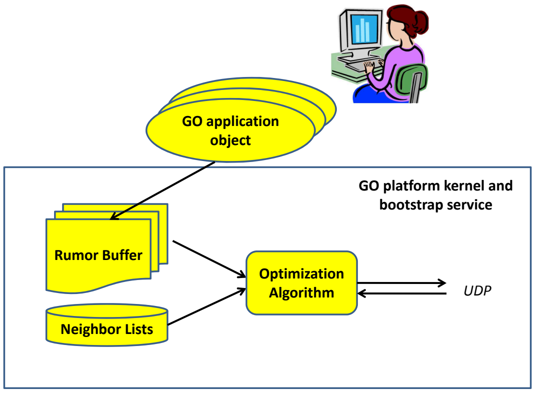


Fig. 2. The **GO** platform runs on each host and interacts with applications and provides optimized gossip dissemination by tracking membership and storing rumors.

lists. The gossip application can select the peer with which it will gossip from this neighbor set, and can generate its own rumors, but those rumors are not sent immediately. Instead, they are stored temporarily in a common **GO**-managed buffer. The **GO** optimization algorithm described in Section 3 of this paper decides which neighbor to gossip with at each time interval, and which rumors to include in the UDP packet that will be sent to that neighbor.

3. GOSSIP ALGORITHMS

3.1 Model

As noted, our model focuses on push-style gossip (rumor-mongering) but can support a wider range of gossip algorithms.

Consider a system with a set N of n nodes and a set M of m gossip objects denoted by $\{1, 2, \ldots, m\}$. Each node i belongs to some subset A_i of gossip objects. Let O_j denote *member set* of gossip object j, defined as $O_j := \{i \in N : j \in A_i\}$. The set N_i denotes the *neighbors* of i, defined as $\bigcup_{j \in A_i} O_j$. For notation, we use the character i (and i') to denote users, and j (j' and j'') to denote group identifiers and O_j ($O_{j'}$ and $O_{j''}$) to refer to member sets henceforth.

A subset of nodes in a gossip object generate *rumors*. Each rumor r consists of a payload and two attributes: (i) $r.dst \in M$: the destination gossip object for which rumor r is relevant, and (ii) $r.ts \in \mathbb{N}$: the timestamp when the rumor was created. A gossip *message* between a pair of nodes contains a collection of at most L stacked rumors, where L reflects the maximum transfer unit (MTU) for IP packets before fragmentation kicks in. For example, if each rumor has average length of 100 bytes and the MTU is 1500 bytes, L is 15.

We will assume throughout this paper that each node i knows the full membership of all of its neighbors N_i. This assumption is for theoretical clarity, and can be relaxed using peer sampling techniques [Kermarrec et al. 2003; Jelasity et al.

2004] or remote representatives [van Renesse et al. 2003]. However, the types of applications for which **GO** is appropriate, such as pub-sub systems or Live Objects, will neither produce immensely large groups nor sustain extreme rates of churn.

3.2 Random Dissemination

A gossip algorithm has two stages: a *recipient selection* stage and a *content selection* stage [Kempe et al. 2001]. The content is then sent to the recipient. For baseline comparison, we will consider the following straw-man gossip algorithm Random-Stacking running on each node i.

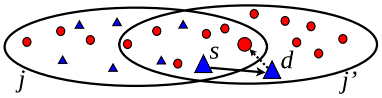
—**Recipient selection:** Pick a recipient d from N_i uniformly at random.

—**Content selection:** Pick a set of L unexpired rumors uniformly at random.

If there are fewer than L unexpired rumors, Random-Stacking will pick all of them. We will also evaluate the effects of rumor stacking; Random is a heuristic that packs only *one* random rumor per gossip message, as would occur in a traditional gossip application that sends rumors directly in individual UDP packets.

3.3 Optimized Dissemination

As mentioned earlier, the selection strategy in Random-Stacking can be improved by sending rumors indirectly via other gossip objects. In the following diagram, nodes infected by a rumor specific to gossip object j are drawn as triangles. The rumor is now sent from node s to a node d which is only in j'. Node d in turn infects a node that belongs to both gossip objects.



We will define the *utility* of including a rumor in a gossip message, which informally measures the "freshness" of the rumor once it reaches the destination gossip object, such that a "fresh" rumor has higher probability of infecting an uninfected node. If rumor r needs to travel via many hops before reaching a node in $r.dst$, by which time r might be known to most members of $r.dst$, the utility of including r in a message is limited. Ideally, rumors that are "young" or "close" should have higher utility.

3.3.1 *Hitting Time.* We make use of results on gossip within a single object. Define an *epidemic on s hosts* to be the following process: One host in a fully-connected network of s nodes starts out as infected. Every round, each infected node picks another node uniformly at random and infects it unless it is already infected.

Definition 3.1. Let $S(s,t)$ denote the number of nodes that are *susceptible* (uninfected) after t rounds of an epidemic on s hosts.

To the best of our knowledge, the probability distribution function for $S(s,t)$ has no closed form. It is conjectured by [Karp et al. 2000] that $\mathbb{E}[S(s,t)] = s\exp(-t/s)$ for push-based gossip and large n using mean-field equations, and that $\mathbb{E}[S(s,t)] =$

$s\exp(-2^t)$ for push-pull gossip. Here, we will assume that $S(s,t)$ is sharply concentrated around this mean, so $S(s,t) = s\exp(-t/s)$ henceforth. Improved approximations, such as using look-up tables for simulated values of $S(s,t)$, can easily be plugged into the heuristic code. We use a look-up table for low values of s and t.

Definition 3.2. Suppose a subset of k nodes out of s are *special*. The *expected hitting time* $H(s,k)$ is the expected number of rounds in an epidemic on s hosts until we infect some node in the subset of special nodes, assuming $S(s,t)$ nodes are susceptible in round t.

If a gossip rumor r destined for some gossip object j ends up in a different gossip object j' which overlaps with j, then the expected hitting time roughly approximates how many rounds elapse before r infects a node in the intersection of O_j and $O_{j'}$, the set of special nodes.

Two simplifying assumptions are at work here. First, that each node in O_j contacts only other nodes in O_j during each round. Second, that r has high enough utility to be included in all gossip messages exchanged within the group.

Let $p(s,k,t) = 1 - \left(1 - \frac{k}{s}\right)^{s - S(s,t)}$ denote the the probability of infecting at least one of k special nodes at time t when $S(s,t)$ are susceptible. We derive an expression for $H(s,k)$ akin to the expectation of a geometrically distributed random variable.

$$H(s,k) = \sum_{t=1}^{\infty} tp(s,k,t) \prod_{\ell=1}^{t-1} (1 - p(s,k,\ell)),$$

which can be approximated by summing a constant number *max-depth* of terms from the infinite series, and by plugging in $S(s,t)$ from above, as shown in Algorithm 1.

Algorithm 1 $H(s,k,t)$: approximate the expected hitting time of k special nodes out of s at time t.

```
if $k = 0$ then
  return $\infty$
end if
if $t \geq$ max-depth then
  return 1.0 {Prevent infinite recursion.}
end if
$p \leftarrow \exp(\log(1.0 - k/s)(s - S(s,t)))$
return $t \cdot (1.0 - p) + H(s,k,t+1) \cdot p$
```

3.3.2 *Utility.* Recall that each node i only tracks the membership of its neighbors. What happens if i receives gossip message containing a rumor r from an unknown gossip object j? To be able to compute the utility of including r in a message to a given neighbor, nodes track the size and the connectivity between every pair of gossip objects. Define an *overlap graph* for propagation of rumors across gossip objects as follows:

Definition 3.3. An *overlap graph* $G = (M,E)$ is an undirected graph on the set of gossip objects, and $E = \{\{j,j'\} \in M \times M : O_j \cap O_{j'} \neq \emptyset\}$. Define the *weight* function $w : M \times M \rightarrow \mathbb{R}$ as $w(j, j') = |O_j \cap O_{j'}|$ for all $j, j' \in M$. Let $\mathcal{P}_{j,j'}$ be the set of simple paths between gossip objects j and j' in the overlap graph G.

We can now estimate the propagation time of a rumor by computing the expected hitting time on a path in the overlap graph G. A rumor may be diffused via different paths in G; we will estimate the time taken by the *shortest* path.

Definition 3.4. Let $P \in \mathcal{P}_{j,j'}$ be a path where $P = (j = p_1, \ldots, p_s = j')$. The *expected delivery time on* P is

$$D(P) = \sum_{k=1}^{s-1} H\left(|O_{p_k}|, w\left(p_k, p_{k+1}\right)\right).$$

The *expected delivery time* from when a node $i \in N$ includes a rumor r in an outgoing message until it reaches another node in $j' = r.dst$ is

$$D(i, r) = \min_{j \in A_i} \min_{P \in \mathcal{P}_{j,j'}} D(P).$$

Algorithm 2 *Compute-graph*: determine the overlap graph, hitting times and shortest paths between every pair of nodes.

```
Require: $overlap[j][j'] = w(j, j')$ has been computed for all groups $j$ and $j'$.
  for $j \in M$ do
    for $j' \in M$ do
      $graph[j][j'] \leftarrow H(|O_j|, overlap[j][j'], 0)$
    end for
  end for
  Run an all-pairs shortest path algorithm [Floyd 1962] on $graph$ to produce $graph\text{-}distance$.
```

Algorithm 2 provides pseudo-code for computing the expected delivery time between every pair of groups.

We can now define a utility function U to estimate the benefit from including a rumor r in a gossip message.

Algorithm 3 $U_s(d, r, t)$: utility of sending rumor r from s to d at time t.

```
Require: $compute\text{-}graph$ must have been run.
  $distance \leftarrow \infty$
  $res \leftarrow 0.0$
  for $j \in d.groups$ do
    if $graph\text{-}distance[j][r.dst] < distance$ then
      $distance \leftarrow graph\text{-}distance[j][r.dst]$
      $res \leftarrow S(j.size, t - r.ts + distance)/j.size$
    end if
  end for
  return $res$
```

Algorithm 4 *Sample*(u, R, L): produce a sample $\mathbf{S}$ of L rumors without replacement from $R = \{r_1, r_2, \ldots, r_N\}$ such that $\mathbb{P}[r_i \in \mathbf{S}] \propto u(r_i)$.

```
$\mathbf{S} \leftarrow \emptyset$
for $\ell = 1$ to $N$ do
  if $\ell \leq L$, or $random(0,1) \leq Lu(r_\ell)/\sum_{t=1}^{\ell} u(r_t)$ then
    $\rho \leftarrow$ randomly picked rumor from $\mathbf{S}$
    $\mathbf{S} \leftarrow (\mathbf{S} - \{\rho\}) \cup \{r_\ell\}$ {Rumor $r_\ell$ will be included in $S$}
  end if
end for
return $\mathbf{S}$
```

Definition 3.5. The *utility* $U_s(d, r, t)$ of including rumor r in a gossip message from node s to d at time t is the expected fraction of nodes in gossip object $j = r.dst$ that are still susceptible at time $t' = t - r.ts + 1 + D(d, r)$ when we expect it to be delivered. More precisely,

$$U_s(d, r, t) = \frac{S(|O_j|, t')}{|O_j|}.$$

Pseudo-code for approximating the utility function is shown in Algorithm 3. The code is optimized by making use of the overlap graph computed by Algorithm 2.

3.3.3 *The **GO** Heuristic.*

The following code is run on node s at time t.

—**Recipient selection:** Pick a recipient d uniformly at random from N_s.

—**Content selection:** Let R denote the set of unexpired rumors. Calculate the utility $u(r) = U_s(d, r, t)$ for each $r \in R$ using Algorithm 3. Call *Sample*(u, R, L) (Algorithm 4) to pick L rumors at random from R so that the probability of including rumor $r \in R$ is proportional to its utility $u(r)$.

Algorithm 4 for sampling without replacement while respecting probabilities on the elements may be of independent interest. We prove the correctness of the algorithm after every iteration of the main loop.

THEOREM 3.6. *After m iterations of algorithm 4, element r_h for $h \leq m$ is included in the sample $\mathbf{S}$ with probability $\frac{Lu(r_h)}{q(m)}$, where $q(m) = \sum_{t=1}^{m} u(r_t)$.*

PROOF. The statement is trivially true for $m \leq L$ (round all probabilities down to 1), so assume $m > L$. We obtain

$$\begin{aligned}
\mathbb{P}[r_h \in \mathbf{S}] &= \mathbb{P}[r_h \text{ picked in iteration } h] \cdot \prod_{t=h+1}^{m} \mathbb{P}[r_h \text{ not discarded in iteration } t] \\
&= \frac{Lu(r_h)}{q(h)} \prod_{t=h+1}^{m} \left(1 - \frac{Lu(r_t)}{q(t)} \cdot \frac{1}{L}\right) \\
&= \frac{Lu(r_h)}{q(h)} \prod_{t=h+1}^{m} \frac{q(t) - u(r_t)}{q(t)} \\
&= \frac{Lu(r_h)}{q(h)} \prod_{t=h+1}^{m} \frac{q(t-1)}{q(t)} \\
&= \frac{Lu(r_h)}{q(m)}.
\end{aligned}$$

□

In order to compute the utility of a rumor, each node needs to maintain complete information about the overlap graph and the sizes of gossip objects. We describe the protocol that maintains this state in Section 4.3.

The cost of storing and maintaining such a graph may become prohibitive for very large networks. We intend to remedy this potential scalability issue by maintaining only a *local view* of the transition graph, based on the observation that if a rumor belongs to distant gossip object with respect to the overlap graph, then its utility is automatically low and the rumor could be discarded. Evaluating the trade-off between the view size and the benefit that can be achieved by the above optimizations is a work in progress.

Consider the content selection policies for the RANDOM-STACKING and the **GO** heuristic. A random policy will often include rumors in packets that have no chance of being useful because the recipient of the packet has no "route" to the group for which the rumor was destined. **GO** will not make this error: if it includes a rumor in a packet, the rumor has at least some chance of being useful. We evaluate the importance of this effect in Section 5.

3.4 Traffic Adaptivity and Memory Use

The above model can be generalized to allow gossip objects to gossip at different *rates*. Let λ_j be the rate at which new messages are generated by nodes in gossip object j, and R_i the rate at which the **GO** platform gossips at node i.

For simplicity, we have implicitly assumed that all platforms gossip at the same fixed rate R, and that this rate is "fast enough" to keep up with all the rumors that are generated in the different gossip objects. Viewing a gossip object as a queue of rumors that arrive according to a Poisson process, it follows from Little's law [Little 1961] that the average rate at which node i sends and receives rumors, R_i, cannot be less than the rate λ_j of message production in j if rumors are to be diffused to all interested parties in finite time with finite memory. In the worst case there is no exploitable overlap between gossip objects, in which case we require R to be at least $\max_{i \in N} \sum_{j \in A_i} \lambda_j$. Furthermore, the amount of memory required is at least

$\max_{i \in N} \sum_{j \in A_i} \mathcal{O}(\log |O_j|) \lambda_j$ since rumors take logarithmic time on average to be disseminated within a given gossip object.

The **GO** platform dynamically adjusts its gossip rate based on an exponential average of the rate of incoming messages per group. The platform speed is set to match that of the group with the highest incoming rate, a feature we call *traffic adaptivity*. Furthermore, **GO** enforces customizable upper bounds on both the memory use and gossip rate (and hence bandwidth), rejecting applications from joining gossip objects that would cause either of these limits to be violated.

Rumors are stored in a priority queue based on their maximum possible utility over all neighbor choices; if the rumors in the queue exceed the memory bound then the least beneficial rumors are discarded.

4. PLATFORM IMPLEMENTATION

As noted earlier, **GO** was implemented using Cornell's Live Distributed Objects technology, and inherits many features from the Live Objects system. **GO** runs on all nodes in the target system, and currently supports applications via an interface focused on group membership and multicast operations. The platform consists of three major parts: the membership component, the rumor queue and the gossip mechanism, as illustrated in Figure 2.2.

GO exports a simple interface to applications. Applications first contact the platform via a client library or an IPC connection. An application can then `join` (or `leave`) gossip objects by providing the name of the group, and a *poll rate* R. Note that a `join` request might be rejected. An application can start a rumor by adding it to an outgoing rumors queue which is polled at rate R (or the declared system-wide poll rate for this gossip object) using the `send` primitive. Rumors are received via a `recv` callback handler which is called by **GO** when data is available.

Rumors are garbage collected when they expire, or when they cannot fit in memory and have comparatively low utility to other rumors as discussed in Section 3.4.

4.1 Bootstrapping

We bootstrap gossip objects using a rendezvous mechanism that depends upon a directory service (**DS**), similar to DNS or LDAP. The **DS** tracks a random subset of members in each group, the size of which is customizable. When a **GO** node i receives a request by one of its applications to join gossip object j, i sends the identifier for j (a string) to the **DS** which in turn returns a random node $i' \in O_j$ (if any). Node i then contacts i' to get the current state of gossip object j: (i) the set O_j, (ii) full membership of nodes in O_j, and (iii) the subgraph spanned by j and its neighbors in the overlap graph G along with weights. If node i is booting from scratch, it gets the full overlap graph from i'.

4.2 Gossip Mechanism

GO's main loop runs periodically, receiving gossip messages from other nodes and performing periodic upcalls to applications, which may react by adding rumors to the *rumor queue*. Each activity period ends when the platform runs the **GO** heuristic (from Section 3.3.3) to send a gossip message to a randomly chosen neighbor. The platform then discards old rumors.

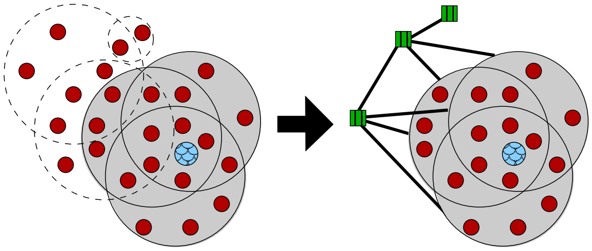


Fig. 3. Membership information maintained by **GO** nodes. The topology of the whole system on the left is modeled by the node in center as (i) the set of groups to which it belongs and neighbor membership information (local state), and (ii) the overlap graph for other groups, whose nodes are depicted as squares and edges are represented by thick lines (remote state).

4.3 Membership Component

Each **GO** node i maintains the membership information for all of its neighbors, N_i (*local* state). It also tracks the overlap graph G and gossip group sizes (*remote* state), as discussed in Section 3. Figure 4.3 illustrates an example of system-wide group membership (left) and the local and remote state maintained by the center node (right). The initial implementation of **GO** maintains both pieces of state via gossip.

4.3.1 *Remote state.* After bootstrapping, all nodes join a dedicated gossip object j^* on which nodes exchange updates for the overlap graph. Let P be a global parameter that controls the rate of system-wide updates, that should reflect both the anticipated level of churn and membership changes in the system, and the $\mathcal{O}(\log n)$ gossip dissemination latency constant [van Renesse et al. 1998]. Every $P \log |O_j|$ rounds, some node i in O_j starts a rumor r^* in j^* that contains the current size of O_j and overlap sizes of O_j and j's neighboring gossip objects. The algorithm is leaderless and symmetrical: each node in O_j starts their version of rumor r^* with probability $1/|O_j|$. In expectation, only one node will start a rumor in j^* for each gossip object. If multiple rumors are started for the same gossip object, information from the initiator with the lowest IP-address is kept.

4.3.2 *Local state.* **GO** tracks the time at which each neighboring node was last heard from; a node that fails will eventually be removed from the membership list of any groups to which it belongs. When node i joins or changes its membership, an upcall is issued to each gossip object in A_i as a special system rumor. We rate-limit the frequency of membership changes by allowing nodes to only make special system announcements every P rounds.

In ongoing work, we are changing the **GO** membership algorithm to bias it in favor of accurate *proximal* information at the expense of decreased accuracy about membership of remote groups. The rationale for this reflects the value of having accurate information in the utility computation. As observed earlier, rumors have diminishing freshness with time, which also implies that the expected utility of indirect routing via a long path is low. In effect, a rumor sent indirectly still needs to reach a destination quickly if it is to be useful. We conjecture that the **GO** heuristic is insensitive to information about remote groups and membership — *i.e.*,

Mechanism	**Platform support**	**Traffic adaptivity**	**Content selection**
GO	Yes	Yes	**GO** heuristic
Platform with utility	Yes	No	**GO** heuristic
Platform with traffic	Yes	Yes	Random-Stacking
Platform skeleton	Yes	No	Random-Stacking
Random-Stacking	No	No	Random-Stacking
Random	No	No	Random (1 rumor/message)

Table I. The dissemination mechanisms we evaluate. Mechanisms without platform support gossip independently for each group joined at the native gossip rate. In the **Random-Stacking** heuristic, each group runs at its own native gossip rate, sending its own rumors but also including additional randomly selected rumors up to the message MTU size, without regard for the expected value of those rumors at the destination node.

several hops from a sender node — but highly sensitive to what might be called proximal topology information. It would follow that proximal topology suffices for efficient dissemination.

4.4 Rumor Queue

As mentioned in Section 3.4, **GO** tracks a bounded set of rumors in a priority queue. The queue is populated by rumors received by the gossip mechanism (remote rumors), or by application requests (local rumors). The priority of rumor r in the rumor queue for node s at time t is $\max_{d \in N_i} U_s(d, r, t)$, since rumors with lowest maximum utility are least likely to be included in any gossip messages. As previously discussed, priorities change with time so we speed up the recomputation by storing the value of $\text{argmax}_{d \in N_i} D(d, r)$.

5. EVALUATION

GO is implemented as a Windows Remoting service using the .NET framework. The focus of our experiments is on quantifying the effectiveness of **GO** in comparison to implementations in which each gossip object runs independently without any platform support at all. We evaluate the dissemination mechanisms listed in Table I.

Our first experiment highlights the usefulness of the various components of the **GO** dissemination mechanism. We run **GO** on a synthetic scenario to evaluate the utility-based **GO** heuristic and traffic adaptivity.

We then evaluate **GO** on a trace of a widely deployed web-management application, IBM WebSphere Virtual Enterprise (WVE). This trace shows WVE's patterns of group membership changes and group communication in connection with a whiteboard abstraction used heavily by the product, and thus is a good match with the kinds of applications for which **GO** is intended.

5.1 Synthetic Scenario

We evaluated the benefits of the two components of the **GO** dissemination mechanism (traffic adaptivity and the utility-based heuristic) on the topology shown in Figure 4. The scenario constitutes a group j that contains nodes s and d in which s sends frequent updates for d. Both nodes also belong to a number of other gossip objects that overlap so that they share common neighbors, in this case four. The

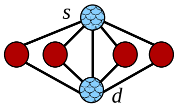
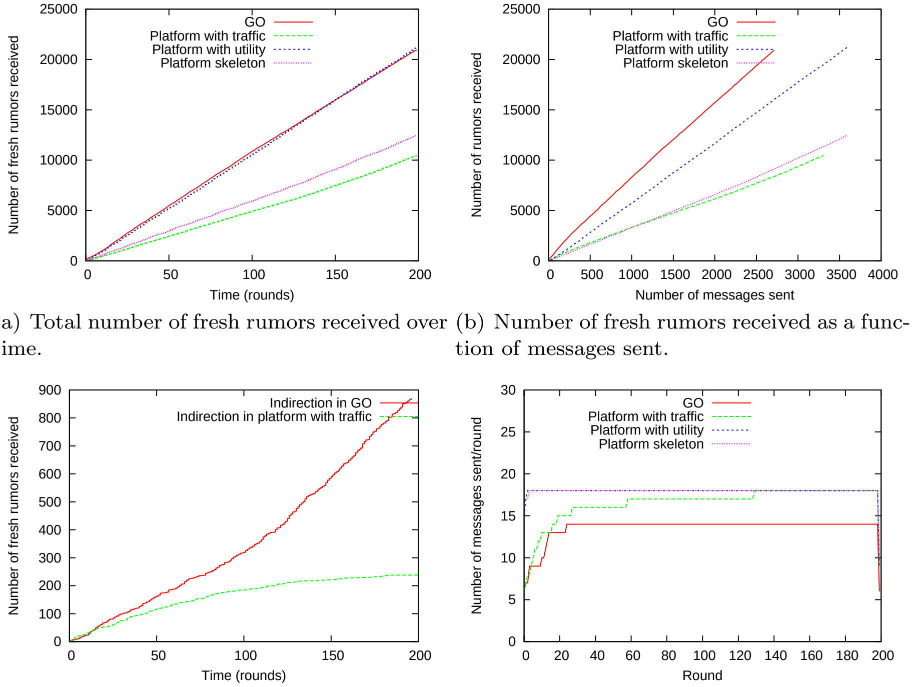


Fig. 4. The topology used in first experiment. Each edge denotes a group of two members.



(a) Total number of fresh rumors received over time.

(b) Number of fresh rumors received as a function of messages sent.

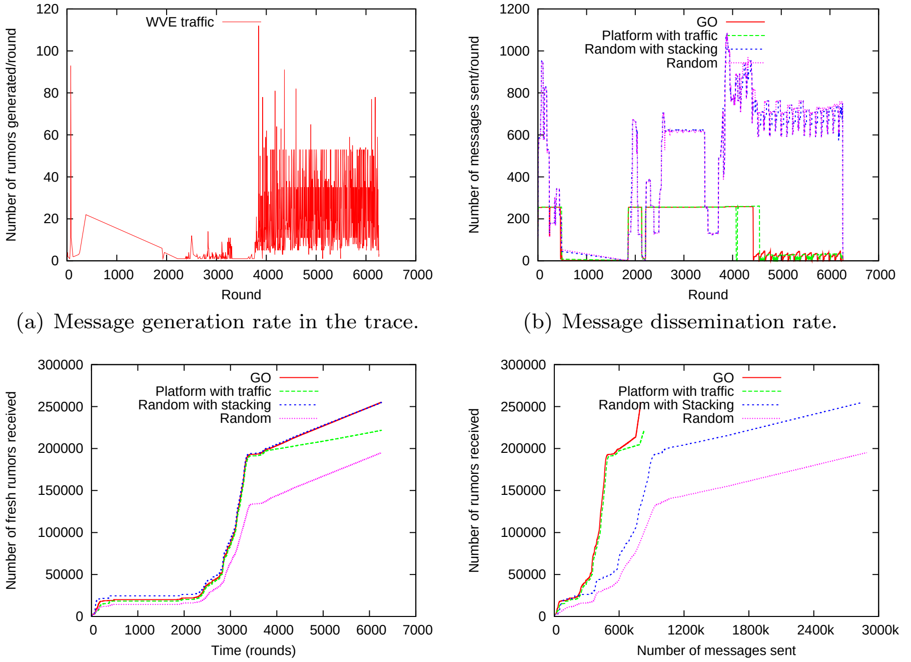
(c) Number of rumors received indirectly.

(d) Message rate over time.

Fig. 5. Synthetic scenario. The **GO** heuristic with traffic adaptivity and utility-based dissemination successfully delivers all rumors (top-left), yet using fewer messages than when traffic adaptivity or utility-based dissemination are disabled (top-right). The utility function takes advantage of indirect paths (bottom-left). **GO**'s message rate is bounded (bottom-right).

scenario is constructed such that the neighbors of s and d are in a position to propagate messages intended for other gossip objects. At every time step, nodes generate 10 rumors for each of the gossip objects to which they belong, and s and d generate 10 additional rumors intended for one another. We assume a platform rate of 1 gossip message per round unless traffic adaptivity is enabled, that 15 rumors can be stacked in every packet and that nodes can fit at most 100 rumors in memory.

We count the total number of "fresh" rumors that were received by interested nodes, by which we mean rumors that nodes had not previously seen. We plot the total number of fresh rumors over time and as a function of the number of messages that were used to deliver them.



(a) Message generation rate in the trace.

(b) Message dissemination rate.

(c) Total number of fresh rumors received over time.

(d) Number of fresh rumors received as a function of messages sent.

Fig. 6. IBM WebSphere Virtual Enterprise (WVE) trace. **GO** disseminates rumors efficiently compared to other approaches while sending substantially fewer messages at a controllable rate. The nodes using the random heuristics gossip once per-group every round, whereas **GO** sends a single gossip message per round.

5.2 Real-World Scenario

5.2.1 *Trace Details.* IBM WebSphere Virtual Enterprise (WVE) is a widely deployed commercial application for running and managing web applications [IBM 2008]. A WebSphere cell consists of a (possibly large) number of servers, on top of which application clusters are deployed. Cell management, which entails workload balancing, dynamic configuration, inter-cluster messaging and performance measurements, is implemented by a form of built-in whiteboard, which in turn interfaces to the underlying communication layer via a pub-sub interface [Bortnikov et al. 2009; Eugster et al. 2003]. To obtain a trace, IBM deployed 127 WVE nodes constituting 30 application clusters for a period of 52 minutes, and recorded group subscriptions as well as the messages sent by every node via the whiteboard abstraction. The rate of message generation in the trace is shown in Figure 6(a). An average process subscribed to 474 groups and posted to 280 groups, and there were a total of 1364 groups with at least one subscriber and one publisher. The group membership is strongly correlated. On the one hand, 26 groups contain at least 121 of the 127 nodes. On the other hand, none of the remaining groups contained more than 10 nodes. The details of the trace are discussed in [Vigfusson 2009].

5.2.2 *Experimental set-up.* We deployed **GO** on 64 nodes, and ran two parallel instances of the platform on each node. We used the WVE trace to drive our simulation by assigning a gossip group to each group. Each gossip round corresponds to one second of the trace. All publishers and subscribers or a group are members of the corresponding gossip group. Each rumor is assumed to be 100 bytes, meaning that 15 rumors may be stacked in a single message. Rumors expire 100 rounds after they were first sent to limit memory and bandwidth use.

Our experiment simulates a "port" of WVE to run over each of the dissemination mechanisms listed in Table I.

5.3 Experimental results

We first discuss the results from the synthetic scenario. The importance of the utility-based heuristic in **GO** is shown in Figure 5(a). After 200 rounds, both **GO** and **Platform with utility** were keeping up with the message load in the system whereas the heuristics deploying random content selection delivered at most 55% of the messages. Figure 5(c) plots the number of rumors that were delivered indirectly over time, clearly showing that utility-based **GO** outperforms myopic random delivery.

Now consider the case for traffic adaptivity. After 200 rounds in Figure 5(a), the utility-based heuristics **GO** and **Platform with utility** have both delivered roughly 21,000 rumors with no loss. However, we learn from Figure 5(b) that the traffic-aware **GO** delivered these rumors in 25% fewer messages than the non-adaptive mechanism. The message rates are shown in Figure 5(d), where regular **GO** converges to a lower value than the alternative policies.

Next we discuss the results of the WVE trace experiment. Figure 6(c) shows the total number of rumors received by nodes in the system over time as the trace is played. A surge in messages of the WVE trace beginning at round 3000 causes the different mechanisms to diverge from one-another. We observe that both **Random-Stacking** and **GO** are able to successfully disseminate all the messages sent in the trace, whereas **Random** and **Platform with traffic** fall behind. The message rates between **Random** and **Random-Stacking** are identical, as shown in Figure 6(b), allowing us to conclude that stacking is effective for rumor dissemination in the WVE trace.

Now consider the discrepancy in performance between the **GO** platform with and without using the utility based **GO** heuristic. The surge in the trace starting at round 3000 consists primarily of messages being sent on groups of size two (*unicast* traffic). Without carefully handling such traffic patterns, unicast rumors pile up while the platform gossips with nodes that have marginal benefit from the rumors exchanged, and gradually time out. We see that the **GO** heuristic avoids this problem as it packs relevant rumors into the messages, whereas randomly selecting rumors for inclusion in those same messages is insufficient.

An important benefit of **GO** can be seen in Figure 6(b), which shows that the **GO** platform limits message rates — sending at most 250 messages/round in total whereas the random approaches send on average roughly 600 messages/round with spikes up to 1100 messages/round. This corresponds to the goal set out in the introduction of bounding platform load despite the number of groups scaling up.

An even bigger win for **GO** can be seen in Figure 6(d), which shows the number of new rumors delivered versus the number of messages exchanged. The **GO** platform sends 3.9 times fewer messages than the greedy per-group **Random-Stacking** dissemination strategy, while delivering rumors just as rapidly.

5.4 Discussion

There are two take-away messages from the first experiment. First, rumor stacking is inherently useful even when using **Random-Stacking** without a utility-driven rumor selection scheme. Second, we see a substantial gain when using the **GO** utility-based heuristic to guide the platform's stacking choices. When processes exhibit correlated but not identical group membership there may often be indirect paths that can be exploited using message indirection. **GO** learns these paths by exploring membership of nearby groups, and can then ricochet rumors through those indirectly accessible groups. The non-utility policies lack the information needed to do this.

Focusing on the WVE experiment, the benefits of coupling utility based dissemination with traffic adaptivity is apparent from Figure 6(c), where the **GO** heuristic successfully accommodates the load surge in the trace at time 3,000, as shown in Figure 6(a). The results support our belief that the **GO** platform is able to cope with real-world message dissemination at a rate close to that of a naïve implementation without losing the fixed bandwidth guarantee discussed in the introduction while using substantially fewer messages than a non-platform approach.

6. RELATED WORK

We are not aware of any prior work on creating a platform or operating system for gossip. The pioneering work by Demers *et al.* [Demers et al. 1987] used gossip protocols to enable a replicated database to converge to a consistent state despite node failures or network partitions. The repertoire of systems that have since employed gossip protocols is impressive [Balakrishnan et al. 2007; van Renesse et al. 1998; van Renesse et al. 2003; Eugster et al. 2003; Decandia et al. 2007; Rodrigues et al. 2003], although this work is focused on application-specific use of gossip instead of providing gossip communication as a fundamental service.

Optimized gossip dissemination has been considered in sensor networks, for instance for gossip aggregation [Dimakis et al. 2006], although the focus is on single gossip objects. The authors of Ricochet [Balakrishnan et al. 2007] consider multiple groups in the context of reliable message dissemination, and propose a scheme that is based on partitioning overlapping groups.

7. CONCLUSION

The **GO** platform generalizes gossip protocols to allow them to join multiple groups without losing the appealing fixed bandwidth guarantee of gossip protocols, and simultaneously optimizing latency in a principled way. Our heuristic is based on the observations that a single IP packet can contain multiple rumors, and that indirect routing of rumors can accelerate delivery. The platform has been implemented, but remains a work in progress. Experimental evaluation shows that indirect message delivery using our utility based scheme as well as adaptivity to different traffic rates outperforms several other natural platform and non-platform based approaches.

Our vision is that **GO** can become an infrastructure component in various group-heavy distributed services, such as a robust multicast or publish-subscribe layer, and an integral layer of the Live Distributed Objects framework.

ACKNOWLEDGMENTS

Krzys Ostrowski and Danny Dolev were extremely helpful in the design of the basic **GO** platform. We acknowledge Mike Spreitzer for collecting the IBM WebSphere Virtual Enterprise trace. We thank Anne-Marie Kermarrec, Davide Frey and Martin Bertier for contributions at an earlier stage of this project [Birman et al. 2007]. **GO** was supported in part by the Chinese National Research Foundation (grant #6073116063), AFOSR, AFRL, NSF, Intel Corporation and Yahoo!.

REFERENCES

Alvisi, L., Doumen, J., Guerraoui, R., Koldehofe, B., Li, H. C., van Renesse, R., and Trédan, G. 2007. How robust are gossip-based communication protocols? *Operating Systems Review 41,* 5, 14–18.

Balakrishnan, M., Birman, K. P., Phanishayee, A., and Pleisch, S. 2007. Ricochet: Lateral error correction for time-critical multicast. In *NSDI.* USENIX.

Birman, K., Kermarrec, A.-M., Ostrowski, K., Bertier, M., Dolev, D., and van Renesse, R. 2007. Exploiting gossip for self-management in scalable event notification systems. Distributed Event Processing Systems and Architecture Workshop (DEPSA).

Birman, K. P., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., and Minsky, Y. 1998. Bimodal multicast. *ACM Transactions on Computer Systems 17*, 41–88.

Bortnikov, V., Chockler, G., Roytman, A., and Spreitzer, M. 2009. Bulletin board: A scalable and robust eventually consistent shared memory over a peer-to-peer overlay. In *LADIS '09: Proceedings of the 3rd Large-Scale Distributed Systems and Middleware Workshop.*

Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., and Yerneni, R. 2008. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow. 1,* 2, 1277–1288.

Decandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. 2007. Dynamo: Amazon's highly available key-value store. In *SOSP.* ACM Press, New York, NY, USA, 205–220.

Deering, S. 1989. Host Extensions for IP Multicasting. *RFC 1112*.

Demers, A. J., Greene, D. H., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H. E., Swinehart, D. C., and Terry, D. B. 1987. Epidemic algorithms for replicated database maintenance. In *PODC.* 1–12.

Dimakis, A. G., Sarwate, A. D., and Wainwright, M. J. 2006. Geographic gossip: efficient aggregation for sensor networks. In *IPSN '06: Proceedings of the 5th international conference on Information processing in sensor networks.* ACM, New York, NY, USA, 69–76.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. 2003. The many faces of publish/subscribe. *ACM Comput. Surv. 35,* 2, 114–131.

Floyd, R. W. 1962. Algorithm 97: Shortest path. *Commun. ACM 5,* 6, 345.

Gupta, I., Birman, K., Linga, P., Demers, A., and van Renesse, R. 2003. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03).*

IBM. 2008. WebSphere. `http://www-01.ibm.com/software/webservers/appserv/was/`.

Jelasity, M., Guerraoui, R., Kermarrec, A.-M., and van Steen, M. 2004. The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In *Middleware.* Toronto, Canada.

Jelasity, M., Montresor, A., and Babaoglu, O. 2005. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst. 23,* 3, 219–252.

Karp, R., Schindelhauer, C., Shenker, S., and Vocking, B. 2000. Randomized rumor spreading. In *FOCS*. 565–574.

Kempe, D., Kleinberg, J. M., and Demers, A. J. 2001. Spatial gossip and resource location protocols. In *STOC*. 163–172.

Kermarrec, A.-M., Massoulié, L., and Ganesh, A. J. 2003. Probabilistic reliable dissemination in large-scale systems. *IEEE Trans. Parallel Distrib. Syst. 14,* 3, 248–258.

Little, J. D. C. 1961. A proof for the queuing formula: $L = \lambda W$. *Operations Research 9,* 3, 383–387.

Ostrowski, K., Birman, K., Dolev, D., and Ahnn, J. H. 2008. Programming with live distributed objects. In *ECOOP*, J. Vitek, Ed. Lecture Notes in Computer Science, vol. 5142. Springer, 463–489.

Pease, M., Shostak, R., and Lamport, L. 1980. Reaching agreement in the presence of faults. *J. ACM 27,* 2, 228–234.

Rodrigues, L., Lisboa, U. D., Handurukande, S., Pereira, J., do Minho, J. P. U., Guerraoui, R., and Kermarrec, A.-M. 2003. Adaptive gossip-based broadcast. In *DSN*. 47–56.

Rowstron, A. I. T. and Druschel, P. 2001. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*. Springer-Verlag, London, UK, 329–350.

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. 2001. Chord: A scalable peer-to-peer lookup service for Internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, New York, NY, USA, 149–160.

van Renesse, R., Birman, K. P., and Vogels, W. 2003. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst. 21,* 2 (May), 164–206.

van Renesse, R., Minsky, Y., and Hayden, M. 1998. A gossip-style failure detection service. Tech. Rep. TR98-1687. August,.

Vigfusson, Y. 2009. Affinity in distributed systems. Ph.D. thesis, Cornell University.

von Eicken, T., Basu, A., Buch, V., and Vogels, W. 1995. U-net: A user-level network interface for parallel and distributed computing. In *SOSP*. 40–53.

Wong, B., Slivkins, A., and Sirer, E. G. 2005. Meridian: a lightweight network location service without virtual coordinates. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, New York, NY, USA, 85–96.