GO: Platform Support For Gossip Applications

(Invited Paper)

Ymir Vigfusson, Ken Birman, Qi Huang, Deepak P. Nataraj
Department of Computer Science
Cornell University
Ithaca, New York
Email: {ymir,ken,qhuang}@cs.cornell.edu, deepak.pn@gmail.com

***Abstract*—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, and the platform optimizes rumor delivery latency in a principled manner. 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 theoretical analysis of this heuristic. We have also 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.**

***Keywords*-gossip; epidemic broadcast; multicast**

I. INTRODUCTION

Gossip-based communication is commonly used in distributed systems to disseminate information and updates in a scalable and robust manner [1], [2], [3]. 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, monitoring and distributed querying, constructing distributed hash tables and other kinds of overlay structures, orchestrating self-repair in complex networks and even for such prosaic purposes as to support shopping carts for large data centers [4]. By using gossip to track group membership, one can implement gossip-based group multicast protocols.

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 [3], as long as several assumptions about the implementation and the node environment are satisfied [5].

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 [6].

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 [7], 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. The basic gossip protocol is probabilistic meaning that some rumors may be delivered late, although this occurs with low probability. 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. Finally, 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 [3]. We note that weak consistency is not *always* a bad thing. Indeed, relaxing consistency guarantees has become increasingly popular in large-scale industrial applications such as Amazon's Dynamo [4] and Yahoo!'s PNUTS [8].

Gossip also has a less-commonly recognized drawback. An assumption commonly 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 [1], 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. 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** interfaces 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 [6], [9], 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 [10]. 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 II.

We envision a number of uses for **GO**. Within our own work, **GO** will be the WAN communication layer for Live Distributed Objects, a framework for abstract components running distributed protocols that can be composed easily to create custom and flexible live applications or web pages [11], [12]. 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. The **GO** interface can also be extended to resemble a gossip-based publish/subscribe system [13]. 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 to exploit the similarity of gossip groups to improve propagation speed and scalability.
- An evaluation of the **GO** platform on a real-world trace by simulation.

II. GOSSIP ALGORITHMS

A. Model

Our model focuses on push-style gossip, but can easily be extended to the push-pull or pull-only cases.

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 $O_j := \{i \in N :$

$j \in A_i\}$. We let N_i denote the set of *neighbors* of i, defined as $\bigcup_{j \in A_i} O_j$.

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 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 [14] or remote representatives [15]. Furthermore, large groups can likely be fragmented at a cost of higher latency, although we leave this avenue of research to future work. 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.

B. Random Dissemination

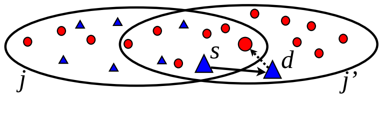
A gossip algorithm has two stages: a *recipient selection* stage and a *content selection* stage [2]. The content is then sent to the recipient. For baseline comparison, we will consider the following straw-man gossip algorithm RANDOM-STACKING running on each node i.

- **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.

C. Optimized Dissemination

As mentioned earlier, the selection strategy in RANDOM can be improved by sending rumors indirectly via other gossip objects. In the following diagram, a triangle representing a rumor specific to gossip object j is sent from node s to a node d only in j'. Node d in turn infects a node in the overlap of the two 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.

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

Definition 1: Let $S(n,t)$ denote the number of nodes that are *susceptible* (uninfected) after t rounds of an epidemic on n hosts.

To the best of our knowledge, the probability distribution function for $S(n,t)$ has no closed form. It is conjectured in [1], [16] that $\mathbb{E}[S(n,t)] = n\exp(-t/n)$ for push-based gossip and large n using mean-field equations, and that $\mathbb{E}[S(n,t)] = n\exp(-2^t)$ for push-pull. Here, we will assume that $S(n,t)$ is sharply concentrated around this mean, so $S(n,t) = n\exp(-t/n)$ henceforth. Improved approximations, such as using look-up tables for simulated values of $S(n,t)$, can easily be plugged into the heuristic code.

Definition 2: The *expected hitting time* $H(n,k)$ is the expected number of rounds in an epidemic on n hosts until we infect some node in a given subset of k special nodes assuming $S(n,t)$ 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' that 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 $O_{j'}$. Two simplifying assumptions are at work here, first that each node in j contacts only nodes within j in each round, and second that r has high enough utility to be included in all gossip messages exchanged within the group.

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

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

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

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, we will have nodes track

Algorithm 1 $H(n, k, t)$: approximate the expected hitting time of k of n at time t.

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

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 $j \in groups$ do
    for $j' \in groups$ do
        if $overlap(j, j') > 0$ then
            $graph[j][j'] \leftarrow H(overlap(j, j'), j.size, 0)$
        else
            $graph[j][j'] \leftarrow \infty$
        end if
    end for
end for
Run an all-pairs shortest path algorithm [17] on graph to produce graph-distance.
```

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: 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 $w : M \times M \to \mathbb{R}$ 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 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 $r.dst$ is

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

Algorithm 2 shows 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-graph must have been run.
$distance \leftarrow \infty$
for $j \in d.groups$ do
    $distance \leftarrow \min\{distance, graph\text{-}distance[j][r.dst]\}$
end for
if $distance = \infty$ then
    return 0.0
end if
return $S(j.size, t - r.ts + dist)/j.size$
```

Algorithm 4 *Sample*(u, R, L): sample L rumors without replacement from R with probability proportional to u.

```
$\mathbf{S} \leftarrow \emptyset$ {The set of rumors in the sample}
$sum \leftarrow \sum_{r \in R} u(r)$
Let $r_1, r_2, \ldots, r_k$ be a random permutation of $R$.
$z \leftarrow random(0, 1)$ {Uniformly random number in $[0, 1)$}
$\zeta \leftarrow 0$
for $\ell = 1$ to $k$ do
    $\zeta \leftarrow \zeta + u(r_\ell) \cdot L/sum$
    if $\zeta \geq z$ then
        $\mathbf{S} \leftarrow \mathbf{S} \cup \{r_\ell\}$ and $\zeta \leftarrow \zeta - 1.0$
    end if
end for
return $\mathbf{S}$
```

Definition 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 $t' = t - r.ts + D(s, r)$ 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) The **GO** Heuristic:* The following code is run by client 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 include it here without proof for the curious reader.

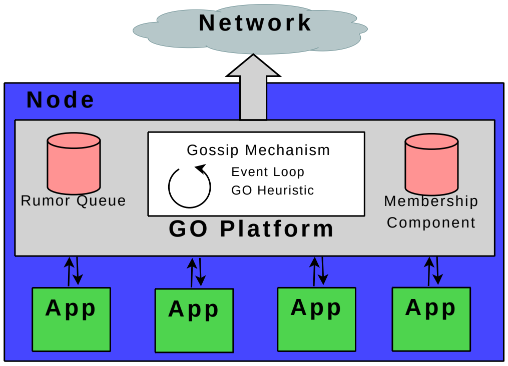


Figure 1. The **GO** Platform.

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 III-C.

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 IV.

D. Traffic Rates 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 [18] 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 $\max_{i \in N} \sum_{j \in A_i} \mathcal{O}\left(\log |O_j|\right) \lambda_j$ since rumors take logarithmic time on average to be disseminated within a given gossip object.

The **GO** platform 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; if the rumors in the queue exceed the memory bound then the least beneficial rumors are discarded.

III. PLATFORM IMPLEMENTATION

As noted earlier, **GO** was implemented using Cornell's Live Distributed Objects technology, and inherits many features from the Live Objects system. For reasons of brevity, we limit ourselves to a short summary. Each **GO** application runs as a small component, coded in any of the 40 or so languages supported by Microsoft .NET, and implements a standard interface defined by the Live Objects framework. At runtime, an "end user" application can link to **GO** applications through simple library interfaces. Moreover, gossip objects can be composed into graphs, with one object talking to another through typed endpoints over which events are passed. The resulting architecture is rich, flexible, and quite easy to extend.

The **GO** platform 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 1.

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 poll rate in the 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 II-D.

A. 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,

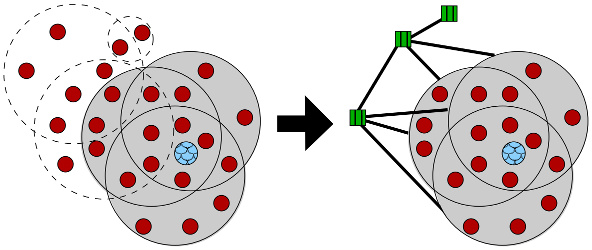


Figure 2. Membership information maintained by **GO** nodes. The topology of the whole system on the left is modeled by the node in center as (i) the set of groups to which it belongs and neighbor membership information (local state), and (ii) the overlap graph for other groups, whose nodes are depicted as squares and edges are represented by thick lines (remote state).

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'.

B. *Gossip Mechanism*

GO's main loop runs periodically, receives gossip messages from other messages 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 II-C3) to send a gossip message to a randomly chosen neighbor. The platform then discards old rumors.

C. *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 II. Figure 2 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.

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. Every $P \log |O_j|$ rounds, some node i in 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 symmetric: 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.

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 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 routing a rumor very indirectly 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 can be proved to be insensitive to information about groups and membership very remote (i.e., several hops from a sender node), but highly sensitive to what might be called proximal topology information. It would follow that proximal topology suffices.

D. *Rumor Queue*

As mentioned in Section II-D, **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 $\text{argmax}_{d \in N_i} D(s, r)$.

IV. EVALUATION

We evaluate the **GO** platform using a discrete time-based simulator[1]. 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.

Our first experiment explores the usefulness of rumor stacking, and evaluates the benefits of computing utility for rumors. We compare the three different gossip algorithms (the **GO** heuristic, RANDOM and RANDOM-STACKING) running in a simple topology.

We then evaluate **GO** on a trace of a widely deployed web-management application, IBM WebSphere. This trace shows WebSphere'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.

---

[1] Although the **GO** platform has been fully implemented, length constraints forced us to choose between simulation and real world experimental findings in this paper. A future extended paper will discuss our experimental findings.

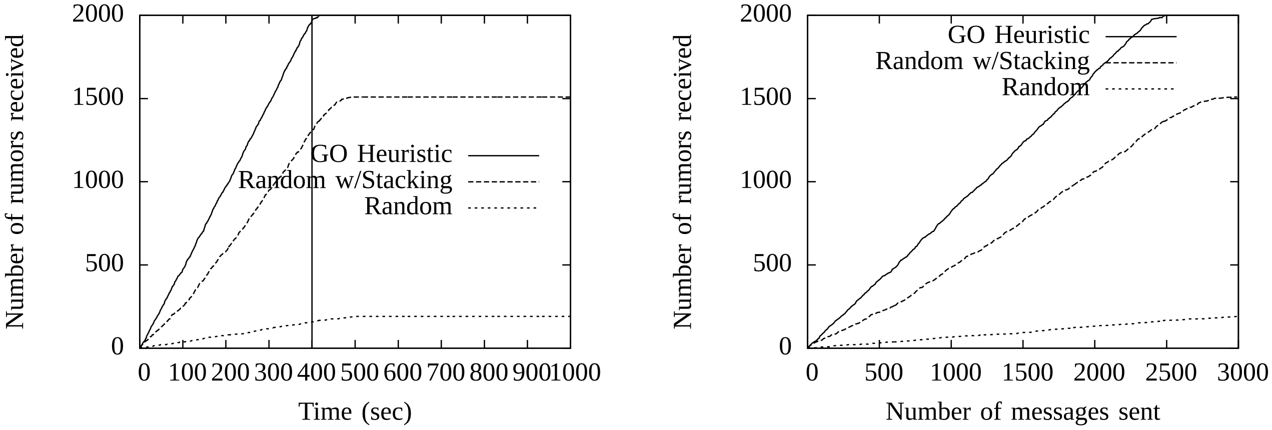


Figure 3. **Rumor Stacking and Indirection.** Different heuristics running on the **GO** platform over the topology from Figure 4. The plots show the number of new rumors received by nodes in the system over time (left) and as a function of messages sent (right). The vertical line shows the time when all 2,000 rumors have been generated.

A. *Rumor Stacking and Message Indirection*

We evaluated the benefits of message indirection used by the **GO** heuristic using 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 some set of common neighbors, in this case four. Assuming the **GO** platform at s only sends one gossip message per round, the shared neighbors are in a position to propagate messages intended for other gossip objects.

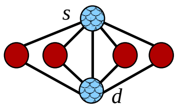


Figure 4. The topology used in first experiment. Each edge corresponds to a gossip group, the members of which are the two endpoints.

We measured the speed of propagation of messages in group j using our simulator. All nodes simulate the **GO** platform with a message rate of 1 message per round, using one of the three gossip algorithms discussed earlier. During each time step until time 400 (vertical line), node s generates a new rumor for each group in A_s, after which rumor generation stops. We assume that 15 rumors can be stacked in each packet, and that nodes can fit at most 100 rumors in memory.

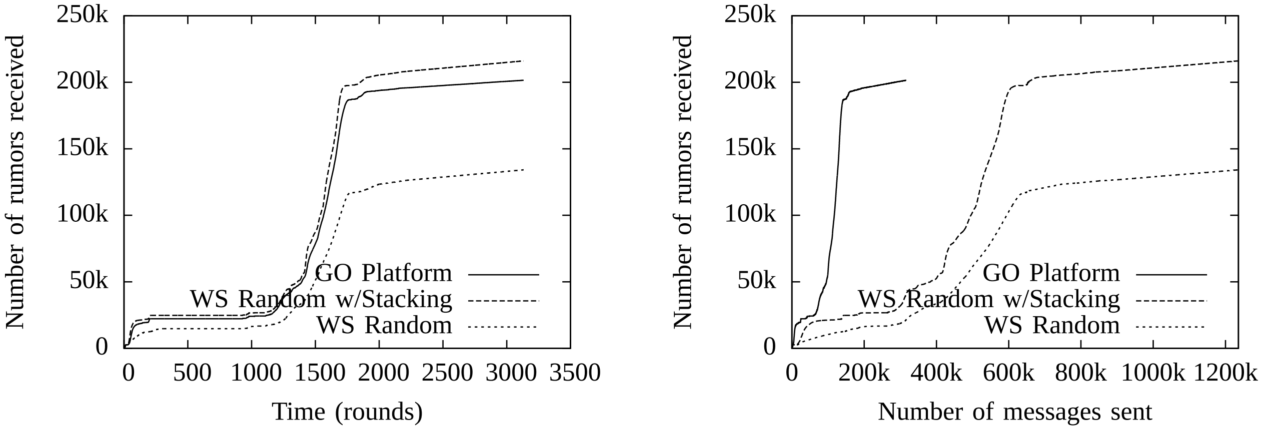
Figure 3 shows the total number of distinct rumors node d has received for group j. The benefits of rumor stacking are evident when one compares the results of the Random-Stacking algorithm to the Random one. Random-Stacking diffuses rumors more than 5 times faster than the single-message Random.

Next, compare the **GO** heuristic results to those of the Random-Stacking algorithm. The **GO** heuristic delivers rumors efficiently: nodes are on average only 11.5 rumors behind an optimal delivery, compared to 460 for Random-Stacking and 1,460 for Random.

B. *Real-World Scenarios*

As noted earlier, IBM WebSphere [19] is a widely deployed commercial application for running and managing web applications. A WebSphere cell may contain hundreds 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 [13] interface. To obtain a trace, IBM deployed 127 WebSphere nodes constituting 30 application clusters for a period of 52 minutes, and recorded topic subscriptions as well as the messages sent by every node. An average process subscribed to 474 topics and posted to 280 topics, and there were a total of 1,364 topics with at least two subscribers and at least one publisher. The topic membership is strongly correlated, in fact 26 topics contain at least 121 of the 127 nodes. On the other hand, none of the remaining topics contained more than 10 nodes.

We used the WebSphere trace to drive our simulation by assigning a gossip group to each topic. All publishers and subscribers for the topic are members of the corresponding gossip group. We limited the memory and bandwidth requirements by expiring rumors 100 rumors after they were first generated. Again, we compare the **GO** heuristic with Random and Random-Stacking. However, in contrast to the experiment of Section IV-A, in which the **GO** platform itself used the specified stacking policy, this WebSphere experiment is slightly different: it compares a simulated "port" of WebSphere to run over **GO** with a simulation of WebSphere running over independent gossip groups that exhibit the same membership and communication patterns,



(a) WebSphere, new rumors vs. number of messages.

(b) WebSphere, ratio of new rumors per message vs. time.

Figure 5. **IBM WebSphere Trace.** The number of new rumors received by nodes in the system and the number of messages sent (left), also plotted as a ratio of new rumors per message over time (right). The nodes using the random heuristics gossip per-group every round, whereas **GO** sends a single gossip message.

but do not benefit from any form of platform support. To emphasize that these group policies are not identical to Random-Stacking and Random, as used internally by the **GO** platform itself in the first experiment, we designate the policies as WS-Random-Stacking and WS-Random in what follows.

We expect the naïve approaches to disseminate rumors faster than **GO** because each WebSphere group is operated independently and in a ”greedy” manner. As a consequence, each node sends one gossip message per group per round, as opposed to only one message per round as the **GO** platform does. As can be seen in Figure 5(a), the delivery speed of the **GO** platform is 6.7% percent lower on average than the naïve WS-Random-Stacking approach. **GO**, however, beats WS-Random by a factor of 2. An even bigger win for **GO** can be seen in Figure 5(b), 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 naïve approaches, thus keeping bandwidth bounded, while disseminating rumors almost as fast.

At the end of the trace, the total number of rumors received by all nodes was 8% lower when using **GO** than WS-Random-Stacking, meaning that some rumors had not reached all intended recipients. We traced this loss to a specific point in the execution at which WebSphere generates a burst of communication, exceeding the **GO**-imposed bandwidth limit. One reasonable inference is that such loss is an unavoidable consequence of our approach, in which a single platform handles communication on behalf of all gossip groups. However, it is interesting to realize that the WebSphere traffic burst was brief and that averaged over even a short window, need not have overwhelmed **GO**. This observation is motivating us to explore dynamically adjusting the platform gossip rate to cope with bursty senders, but in ways that would still respect operator-imposed policies over longer time periods.

C. 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. Nonetheless, we see a substantial gain when using the **GO** heuristic to guide the platform’s stacking choices. Although not reported here, we have conducted additional experiments that confirm this finding under a wide range of conditions. Second, if processes exhibit correlated but not identical group membership, then 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 Random-Stacking policy lacks the information needed to do this. While the topology in the first experiment is deliberately adversarial, it is also extremely simple. For this reason, we believe that patterns of this sort may be common in the wild, where correlated group membership is known to be a pervasive phenomenon.

The WebSphere experiment supports 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, and in fact using substantially fewer messages than a non-platform approach.

We believe that the scenarios we evaluated illustrate the potential benefit of the **GO** methodology in a reasonably general way. If a large number of groups overlap at a single node, conditions could arise that would favor the **GO** heuristic to an even greater degree than in our examples. For example, this would be the case if a large number of groups overlap, generating high volumes of gossip traffic, and yet the pattern of membership is such that relatively few rumors are legitimate candidates for stacking in any particular gossip message. **GO** has the information to optimize for such cases, including only high-value rumors; random stacking would tend to fill packets with useless content, missing the opportunity.

V. FUTURE DIRECTIONS

At present, **GO** rejects gossip join requests if the resulting additional gossip load would overflow its rumor buffers. One might imagine a more flexible scheme that would allocate rumor buffer space among applications in an optimized manner, so as to accommodate applications with varied data production rates. If we then think about information flow rates within individual groups, and compare this with those achievable using the **GO** (where groups carry traffic for one-another), it would be possible to demonstrate an increase in the peak data rates when using **GO** relative to systems that lack this cooperative behavior.

A second direction for future investigation concerns other potential uses for **GO**. As noted earlier, our near term plan is to extend **GO** so that it can support a wider range of gossip styles. Beyond this, we are considering hosting non-gossip protocols "over" **GO**, tunneling their communication traffic through **GO** so as to gain the properties of those protocols (such as consistency, tolerance of application-level Byzantine faults, *etc.*) while also benefiting from **GO**'s simple worst-case communication loads.

Yet a third open topic concerns security. The **GO** rumor stacking scheme does not currently provide true performance isolation: an aggressive application may be able to dominate a less aggressive one, seizing an unfair share of stacking space. A thorough exploration of this form of fairness, and of other security issues raised by **GO**, would represent an appealing subject for further study.

In summary, **GO** is a work in progress. While gossip protocols for individual applications are a relatively mature field, it is interesting to realize that by building a platform – an operating system – to support multiple gossip applications, one encounters such a wide range of challenging problems. We conjecture that practitioners who use gossip aggressively will encounter these problems too, and that in the absence of good solutions, might conclude that gossip is not as effective a technology as generally believed. Yet there seems to be every reason to expect that these problems can be solved. By doing so we advance the theory, while also enlarging the practical utility of gossip in large data centers and WAN peer-to-peer settings, where gossip seems to be a good fit to the need.

VI. RELATED WORK

The pioneering work by Demers *et al.* [1] 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 [9], [6], [15], [13], [4], [20], although most work is focused on application-specific use of gossip instead of providing gossip communication as a fundamental service.

VII. 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. 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 trace. We also thank Anne-Marie Kermarrec, Davide Frey and Martin Bertier for contributions at an earlier stage of this project [12]. **GO** was supported in part by the Chinese National Research Foundation (grant #6073116063), AFOSR, AFRL, NSF, Intel Corporation and Yahoo!.

REFERENCES

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

[2] D. Kempe, J. M. Kleinberg, and A. J. Demers, "Spatial gossip and resource location protocols," in *STOC*, 2001, pp. 163–172.

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

[4] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *SOSP*. New York, NY, USA: ACM Press, 2007, pp. 205–220. [Online]. Available: http://dx.doi.org/10.1145/1294261.1294281

[5] L. Alvisi, J. Doumen, R. Guerraoui, B. Koldehofe, H. C. Li, R. van Renesse, and G. Trédan, "How robust are gossip-based communication protocols?" *Operating Systems Review*, vol. 41, no. 5, pp. 14–18, 2007.

[6] R. Van Renesse, Y. Minsky, and M. Hayden, “A gossip-style failure detection service,” Tech. Rep. TR98-1687, FebruaryAugust, 1998. [Online]. Available: http://citeseer.ist.psu.edu/vanrenesse98gossipstyle.html

[7] S. Deering, “Host Extensions for IP Multicasting.” *RFC 1112*, August 1989.

[8] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, “PNUTS: Yahoo!’s hosted data serving platform,” *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1277–1288, 2008.

[9] M. Balakrishnan, K. P. Birman, A. Phanishayee, and S. Pleisch, “Ricochet: Lateral error correction for time-critical multicast,” in *NSDI*. USENIX, 2007.

[10] T. von Eicken, A. Basu, V. Buch, and W. Vogels, “U-net: A user-level network interface for parallel and distributed computing,” in *SOSP*, 1995, pp. 40–53.

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

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

[13] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, “The many faces of publish/subscribe,” *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.

[14] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, “The peer sampling service: Experimental evaluation of unstructured gossip-based implementations,” in *Middleware*, Toronto, Canada, October 2004. [Online]. Available: http://www.irisa.fr/paris/Biblio/Papers/Kermarrec/JelGueKerSte04MIDDLEWARE.pdf

[15] R. Van Renesse, K. P. Birman, and W. Vogels, “Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining,” *ACM Trans. Comput. Syst.*, vol. 21, no. 2, pp. 164–206, May 2003. [Online]. Available: http://dx.doi.org/10.1145/762483.762485

[16] R. Karp, C. Schindelhauer, S. Shenker, and B. Vocking, “Randomized rumor spreading,” in *FOCS*, 2000, pp. 565–574.

[17] R. W. Floyd, “Algorithm 97: Shortest path,” *Commun. ACM*, vol. 5, no. 6, p. 345, 1962.

[18] J. D. C. Little, “A proof for the queuing formula: $L = \lambda W$,” *Operations Research*, vol. 9, no. 3, pp. 383–387, 1961. [Online]. Available: http://dx.doi.org/10.2307/167570

[19] “IBM WebSphere,” http://www-01.ibm.com/software/webservers/appserv/was/, 2008.

[20] L. Rodrigues, U. D. Lisboa, S. Handurukande, J. Pereira, J. P. U. do Minho, R. Guerraoui, and A.-M. Kermarrec, “Adaptive gossip-based broadcast,” in *DSN*, 2003, pp. 47–56.