Optimizing Information Flow in the Gossip Objects Platform

Ymir Vigfusson*
IBM Haifa Research Lab
Mount Carmel
Haifa, Israel
ymirv@il.ibm.com

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

ABSTRACT

Gossip-based protocols are commonly used for diffusing information in large-scale distributed applications. **GO** (Gossip Objects) is a per-node gossip platform that we developed in support of this class of protocols. **GO** allows nodes to join multiple gossip groups without losing the appealing fixed bandwidth guarantee of gossip protocols, and the platform also optimizes latency in a principled manner. Our algorithm 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 algorithm. We have also implemented **GO**, and studied the effectiveness of the algorithm by comparing it to the more standard random dissemination gossip strategy.

Categories and Subject Descriptors

C.2.4 [**Computer Communication**]: Distributed Systems

Keywords

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 [10, 15, 5]. The idea is simple: Each node sends or exchanges information (known as *rumors*) with a randomly chosen node in the system, allowing messages to propagate to everybody in an "epidemic fashion". For this reason, gossip-based group communication is also known as *epidemic broadcast*.

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 [5], as long as several assumptions about the implementation and the node environment are satisfied [1].

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 [20].

P3. **Simplicity**. There is no need for careful synchronization across nodes, specific hardware support like for IP Multicast [9], or to maintain the health of a structured overlay network. Gossip protocols are simple to write and debug.

P4. **Scalability**. All of these properties are preserved when the size of the system increases.

However, gossip protocols also have drawbacks. The most commonly acknowledged are these: the basic protocol is probabilistic meaning that some rumors may be delivered late, although with low probability. The expected number of rounds required for delivery in gossip protocols is logarithmic in the number of nodes. The latency of gossip protocols is thus on average higher than can that provided by systems using hardware accelerated solutions like IP Multicast [9]. 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 if updates cease for a period of time. Applications that need stronger consistency guarantees need to use more involved and expensive message passing schemes [4], although relaxing consistency guarantees has become increasingly popular in large-scale industrial applications such as Amazon's Dynamo [8] and Yahoo!'s PNUTS [7].

But gossip also has a less-commonly recognized drawback. A common assumption in the gossip literature is that all nodes belong to a single gossip *group*, which we will also call a *gossip object*. While sufficient in individual applications, such as when replicating a database [10], our object-oriented perspective suggests that nodes might use multiple objects and hence belong to multiple gossip groups. The solution is more involved than running multiple independent gossip processes on the same node, because load now depends on the number of concurrent processes on a node, which violates property P2. Similarly, the trivial scheme of broadcasting

*Work done while the author was at Cornell University.

LADIS '09 Big Sky, MT

each message to all nodes in the system forces nodes to filter out unwanted messages — an expensive operation if there are many groups and typical nodes belong to just a few, particularly if there are high rates of gossip [6].

We have built a per-node service called the Gossip Objects platform (**GO**) which allows applications to join a multitude of gossip groups in a simple fashion. **GO** provides a multicast-like interface [9]: local applications can join or leave gossip objects, and send or receive rumors via callback handlers that are executed at particular rates. 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 customizable.

To satisfy these goals and maximize performance, **GO** incorporates some novel optimizations. Our first observation is that gossip messages are frequently short, perhaps containing just a few tens of bytes. This is not surprising: many gossip systems push only rumor version numbers to minimize waste [20, 2], 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 as short as 12-16 bytes total. The second observation is that there is negligible difference in network overhead between a UDP datagram packet containing 15 bytes or 1500 bytes, as long as the datagram is not fragmented [21].

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 network overhead. 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: If s includes rumors for gossip objects that d is not interested in, d might in turn propagate these rumors to nodes that will benefit from them. We formalize rumor stacking and *message indirection* by defining the *utility* of a rumor in section 2.

The **GO** platform can be used as a generic transport layer for group-heavy distributed systems, supporting one-to-many and many-to-many communication patterns out-of-the-box. For instance, the **GO** interface can be easily extended to be a gossip-based publish/subscribe system [11]. **GO** can also complement the low-reliability but low-latency IP Multicast with a robust high-latency UDP multicast primitive for improved group communication with applications in multi-player gaming, online lecture feeds, and so forth [9]. **GO** was developed to become the transport 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 [17, 3].

Our paper makes the following contributions.

- A natural extension of gossip protocols by supporting multiple groups per node.
- A novel algorithm to exploit the similarity of gossip groups to improve propagation speed and scalability.
- An evaluation of an implementation of the **GO** platform on a real-world trace.

2. 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 $O_j := \{i \in N : j \in A_i\}$. 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 its neighbors N_i. This assumption is for theoretical clarity, and can be relaxed using peer sampling techniques [13] or remote representatives [19]. 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 [17], will neither produce immensely large groups nor sustain extreme rates of churn.

2.1 Random Dissemination

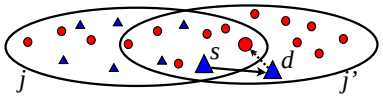
A gossip algorithm has two stages: a *recipient selection* stage and a *content selection* stage. [15] For baseline comparison, we will consider the following straw man gossip algorithm RANDOM-STACKING running on each node i.

- **Recipient selection:** Pick a recipient d from N_i uniformly at random.
- **Content selection:** Pick a set of at most L unexpired rumors uniformly at random.

We will also evaluate the effects of rumor stacking; RANDOM is a heuristic that packs only *one* random rumor per gossip message.

2.2 Optimized Dissemination

As mentioned earlier, the selection strategy in RANDOM can be improved by sending rumors indirectly via other gossip objects. Here, a triangular 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 intersection 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 $r.dst$, r might be known to most members of $r.dst$ by the time it reaches the destination object, and thus the utility of including it in a message is limited. Ideally, rumors that are "young" or "close" should have higher utility.

2.2.1 Hitting Time

We make use of results on gossip within a single object. Let 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 [10, 14] that $\mathbb{E}[S(n,t)] \approx n\exp(-t/n)$ for push-based gossip and large n using mean-field equations, and that $\mathbb{E}[S(n,t)] \approx n\exp(-2^t)$ 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 $H(n,k,S)$ *is the expected number of rounds until we infect any of k special nodes in an epidemic on n hosts if $S(n,t)$ fraction of nodes are susceptible in round t.*[1]

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 $p(t) = 1 - \left(1 - \frac{k}{n}\right)^{n-S(n,t)}$ denote the the probability of infecting any 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.

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

which can be approximated by summing a constant number T of terms from the infinite series, and by plugging in $S(n,t)$ from above.

2.2.2 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 the size and the pairwise connectivity of gossip objects. Define a *transition graph* for propagation of rumors across gossip objects as follows:

DEFINITION 3. *A* transition graph $G = (M, E)$ *is an undirected graph on the set of gossip objects, and $E = \{\{j,j'\} \in V \times V : O_j \cap O_{j'} \neq \emptyset\}$. Define the* weight *function $w : E \to \mathbb{R}$ as $w(\{j,j'\}) = |O_j \cap O_{j'}|$ for all $\{j,j'\} \in E$. Let $\mathcal{P}_{j,j'}$ be the set of simple paths between gossip objects j and j' in the transition graph G.*

[1]We will write $H(n,k)$ if S is clear from context.

We can now estimate the propagation time of a rumor by computing the expected hitting time on a path in the transition 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*

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

We can now define a utility function U to estimate how beneficial it is to include a rumor r in a gossip message.

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 $t' = t + D(s,r)$ when we expect it to be delivered. More precisely,*

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

2.2.3 The GO Algorithm

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

- **Recipient selection:** Pick a recipient d uniformly at random from a random gossip object O_j from A_s.
- **Content selection:** Calculate the utility $U_s(d,r,t)$ for each unexpired rumor r. Pick L rumors at random from the set of unexpired rumors so that the probability of including rumor r is proportional to its utility $U_s(d,r,t)$.

In order to compute the utility of a rumor, each node needs to maintain complete information about the transition graph and the sizes of gossip objects. We describe the protocol that maintains this state in section 3.2. The cost of storing and maintaining such a graph may become prohibitive for very large networks. We intend **GO** 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 transition graph, then its utility is automatically low and the rumor could be discarded. Evaluating the trade-off between the view size and benefit from the above optimizations is a work in progress.

2.3 Gossip 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 the 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 [16] that the average rate at which node i receives rumors, R_i, cannot be less

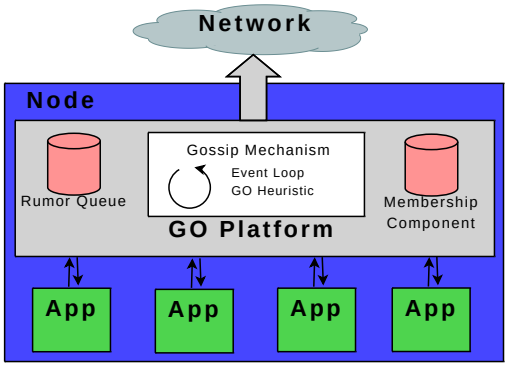


Figure 1: The GO Platform.

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. 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; if the rumors in the queue exceed the memory bound then the least beneficial rumors are discarded.

3. PLATFORM IMPLEMENTATION

The **GO** platform is a per-node service that provides gossip to applications via a multicast-like interface. The platform constitutes 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.

3.1 Bootstrapping

We assume that a directory service (**DS**), similar to DNS or LDAP, is available for **GO** users. 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 transition graph G along with weights. If node i is booting from scratch, it gets the full transition graph from i'.

3.2 Gossip Mechanism and Membership

GO's main loop runs periodically, receives gossip messages from other messages and adds rumors to the *rumor queue*, and finally runs the **GO** algorithm (from section 2.2.3) to send a gossip message to a randomly chosen neighbor.

Each **GO** node i maintains the membership information for all the nodes in A_i (*local* state). and also tracks the transition graph G and gossip group sizes (*remote* state), as discussed in section 2. **GO** maintains both pieces of state via gossip.

Remote state. After bootstrapping, all nodes join a dedicated gossip object j^* on which nodes communicate updates to the transition graph. Let P be a global parameter that controls the rate of system-wide updates, and 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, as long as this information needs updating. Instead of picking i via leader election, 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.

Local state. When node i joins or changes its membership, this information is announced to each gossip object in A_i as a special system rumor. We rate limit the frequency of these changes by allowing nodes to only make such announcements every P rounds.

3.3 Rumor Queue

As mentioned in section 2.3, **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. Because priorities change with time, we speed up the recomputation by storing the value of $\text{argmax}_{d \in N_i} D(s, r)$.

4. EVALUATION

GO is implemented as a Windows Remoting service using the .NET framework. We 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.

4.1 Trace Details

IBM WebSphere [12] 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

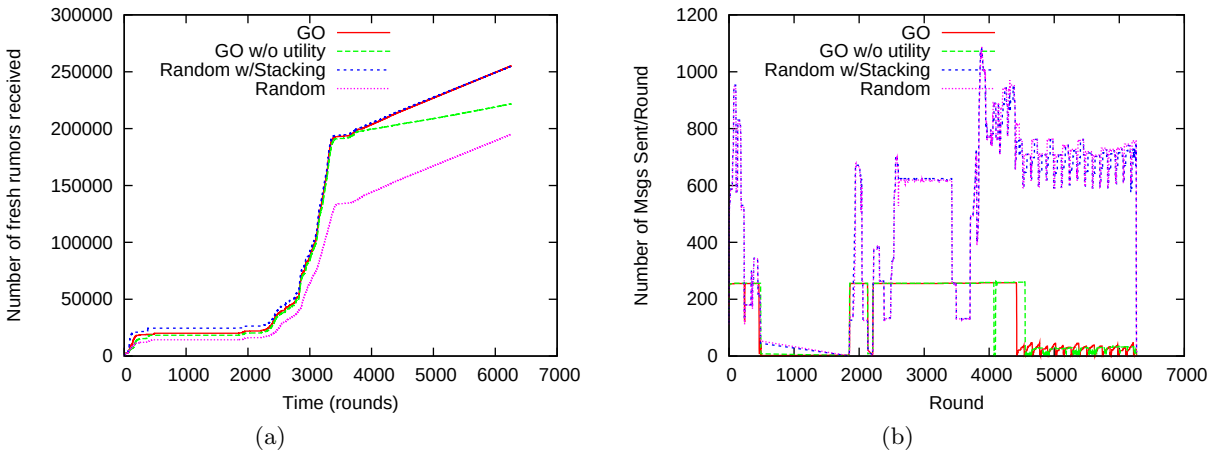


Figure 2: IBM WebSphere Trace. (a) The number of new rumors received by nodes in the system over time, and (b) the message rate over time.

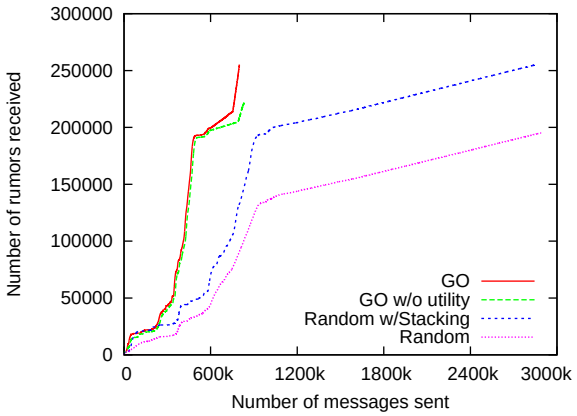


Figure 3: IBM WebSphere Trace. The number of new rumors received by nodes in the system as a function of the number of messages sent.

interfaces to the underlying communication layer via a pub-sub [11] interface. IBM produced the trace by deploying 127 WebSphere nodes constituting 30 application clusters for a period of 52 minutes, and recording 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 one subscribers and at least one publisher used to disseminate messages. 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.

4.2 Experimental set-up

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

Our experiment simulates a "port" of WebSphere to run over each of the following dissemination mechanisms.

- **GO**: The **GO** platform with traffic adaptivity which uses the **GO** algorithm for content selection.
- **GO without utility**: The **GO** platform with traffic adaptivity which uses RANDOM-STACKING heuristic for content selection.
- **Random**: Independent gossip objects using the RANDOM heuristic with no benefit from any form of platform support or traffic adaptivity.
- **Random-Stacking**: Independent gossip objects using the RANDOM-STACKING heuristic. In contrast to GO without utility, 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.

4.3 Discussion

Figure 2(a) shows the total number of rumors received by nodes in the system over time as the trace is played. Starting at round 3,000, a surge in the message rates of the WebSphere trace causes the different mechanisms to diverge. We observe that both RANDOM-STACKING and **GO** are able to successfully disseminate all the messages sent in the trace, whereas RANDOM and **GO** without utility fall behind. The message rates between RANDOM and RANDOM-STACKING are identical, as shown in figure 2(b), allowing us to conclude that stacking is effective for rumor dissemination in the WebSphere trace.

Now consider the discrepancy in performance between **GO** with and without using the utility based **GO** algorithm. The surge in the trace starting at round 3,000 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** algorithm 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 2(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 4, 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 per-group RANDOM-STACKING dissemination strategy, while delivering rumors just as rapidly.

5. RELATED WORK

The pioneering work by Demers et al. [10] 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 [2, 20, 19, 11, 8, 18], although most work is focused on application-specific use of gossip instead of providing gossip communication as a fundamental service. The **GO** platform realizes the vision of a self-managed event notification platform first presented in [3].

6. 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 algorithm is based on the observations that a single IP packet can contain multiple rumors, and that indirect routing of rumors can accelerate delivery. Experimental evaluation on a real-life trace demonstrates the competitiveness of our algorithm, matching the delivery speed of per-group random gossip dissemination yet sending in total 3.9 times fewer messages.

Our vision is that **GO** can become a key 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.

7. ACKNOWLEDGMENTS

We thank the anonymous reviewers for insightful comments. Krzys Ostrowski and Danny Dolev were extremely helpful in the design of the basic **GO** platform. We are also grateful to Anne-Marie Kermarrec, Davide Frey and Martin Bertier for contributions at an earlier stage of this project. **GO** was supported in part by grants from AFOSR, AFRL, NSF, Intel Corporation and Yahoo!.

8. REFERENCES

[1] 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*, 41(5):14–18, 2007.

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

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

[4] K. P. Birman. Replication and fault-tolerance in the isis system. In *SOSP*, pages 79–86, 1985.

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

[6] B. Carmeli, G. Gershinsky, A. Harpaz, N. Naaman, H. Nelken, J. Satran, and P. Vortman. High throughput reliable message dissemination. In *SAC*, pages 322–327, 2004.

[7] 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.*, 1(2):1277–1288, 2008.

[8] 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*, pages 205–220, New York, NY, USA, 2007. ACM Press.

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

[10] 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*, pages 1–12, 1987.

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

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

[13] 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.

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

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

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

[17] K. Ostrowski, K. Birman, D. Dolev, and J. H. Ahnn. Programming with live distributed objects. In J. Vitek, editor, *ECOOP*, volume 5142 of *Lecture Notes in Computer Science*, pages 463–489. Springer, 2008.

[18] 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*, pages 47–56, 2003.

[19] 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.*, 21(2):164–206, May 2003.

[20] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. Technical Report TR98-1687, August, 1998.

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