Efficient Reconciliation and Flow Control for Anti-Entropy Protocols

Robbert van Renesse* Dan Dumitriu† Valient Gough Chris Thomas

Amazon.com, Seattle

ABSTRACT

The paper shows that anti-entropy protocols can process only a limited rate of updates, and proposes and evaluates a new state reconciliation mechanism as well as a flow control scheme for anti-entropy protocols.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design – *network communications*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems – *distributed applications*; D.1.3 [**Programming Techniques**]: Concurrent Programming – *distributed programming*; D.4.4 [**Operating Systems**]: Communications Management – *network communication*; D.4.5 [**Operating Systems**]: Reliability – *fault tolerance*;

General Terms: Algorithms, Reliability.

Additional Key Words and Phrases: Epidemics, Anti-Entropy, Gossip, Flow Control

1. INTRODUCTION

Anti-entropy, or *gossip*, is an attractive way of replicating state that does not have strong consistency requirements [3]. With few limitations, updates spread in expected time that grows logarithmic in the number of participating hosts, even in the face of host failures and message loss. The behavior of update propagation is easily modeled with well-known epidemic analysis techniques. As a result, many distributed applications use gossip to contain various inconsistencies.

In spite of its popularity, little study has been done into how gossip protocols behave under high update load. Gossip protocols purport to deliver messages within a certain configurable number of rounds with high probability, and thus provide synchronous guarantees. Like any other synchronous communication channel, gossip has capacity that is limited by available bandwidth for transporting gossip data and CPU cycles for generating and processing the gossip messages. Under high update load, a gossip protocol may not be able to send all updates required to reconcile differences between peers. Updates would take arbitrary time to propagate as the gossip channel gets backed up.

Gossip protocols are designed to be non-invasive and have predictable performance, and for this a designer has to fix not only the gossip rate per participant but also the maximum size of gossip messages (*e.g.*, maximum UDP packet size). While this avoids network and CPU overload, it also limits the capacity of the gossip channel.

This paper makes two contributions. First, it presents a new state reconciliation mechanism that is designed both for minimal CPU overhead and for situations in which only limited bandwidth is available (Section 3). Second, it proposes and analyzes a flow control scheme for gossip (Section 4). Related work is discussed in Section 5.

2. GOSSIP BASICS

There are two classes of gossip: *anti-entropy* and *rumor-mongering* protocols. Anti-entropy protocols gossip information until it is made obsolete by newer information, and are useful for reliably sharing information among a group of participants. Rumor-mongering has participants gossip information for some amount of time chosen sufficiently high so that with high likelihood all participants receive the information. In this paper, we shall focus on anti-entropy—reconciliation and flow control for rumor-mongering have received considerably attention already (see Section 5).

Let $\mathcal{P} = \{p, q, ...\}$ be a set of participants. Each participant maintains state, which we model as a mapping $\sigma \in \mathcal{S} = \mathcal{K} \rightarrow (\mathcal{V} \times \mathcal{N})$. Here $\mathcal{K}$ is a set of keys, $\mathcal{V}$ a set of values, and $\mathcal{N}$ an infinite ordered set of version numbers. $\sigma(k) = (v, n)$ means that key k is mapped to value v and version n. A more recent mapping for the same key contains a larger version number. Both value and version number spaces contain a $\perp$ element, and in case of $\mathcal{N}$, $\perp$ is the lowest element. Initially all keys on all participants are mapped to $(\perp, \perp)$.

A participant's state is mutable and is replicated onto all participants. We model this as a mutable mapping $\mu_p : \mathcal{P} \rightarrow \mathcal{S}$ maintained by each participant p. A participant p is only

*Contact author. Current address: Dept. of Comp. Sc., Cornell University. Email: rvr@cs.cornell.edu

†Current address: Ballista Securities, New York.

allowed to update its own state $\mu_p(p)$ directly. $\mu_p(q), p \neq q$, can only be updated indirectly through gossip.

Anti-entropy protocols use a *merge* or *reconciliation* operator $\oplus$ that operates on two states in $\mathcal{S}$ and returns a new state. The semantics of $\sigma = \sigma_1 \oplus \sigma_2$ is as follows. Let $\sigma_1(k) = (v_1, n_1)$, and $\sigma_2(k) = (v_2, n_2)$. Then $\sigma(k) = (v_1, n_1)$ if $n_1 \geq n_2$, and (v_2, n_2) otherwise.

Each participant p has a set of peers $\mathcal{F}_p \subseteq \mathcal{P} - \{p\}$ that it gossips with. In many protocols, $\mathcal{F}_p = \mathcal{P} - \{p\}$. Periodically, p chooses a participant q from $\mathcal{F}_p$ at random, and initiates gossip with q. p and q then apply the reconciliation operator to their copies of the participants' states.

There are three styles of gossip that are used. In the most basic form, p simply sends μ_p to q, and q applies $\oplus$ to all mappings in μ_q. That is, $\forall r : \mu'_q(r) = \mu_q(r) \oplus \mu_p(r)$. This is often referred to as *push-gossip*. In *pull-gossip*, p sends to q a *digest* of its state, which is essentially μ_p with the values removed, leaving only the keys and version numbers. q then returns to p only the necessary updates to μ_p, avoiding sending values that are not needed. *Push-pull-gossip* is like pull-gossip, except that q also sends a list of participant-key pairs for which it has outdated entries compared to p. Upon receipt, p sends the corresponding entries of μ_p back to q. Push-pull is the most efficient style, and it is the one we focus on in this paper.

If a particular key on a particular participant is no longer updated, its value will propagate to all other participants in expected time that is logarithmic in the number of participants, and is guaranteed to get to all other participants eventually with probability 1 (assuming $\mathcal{F}_p$ is chosen sufficiently large [5]). Message loss and individual participant failures, within reason, do little to slow gossip down.

While seemingly simple, tricky issues arise when gossip is used under high update load while available network bandwidth or CPU cycles are limited. Gossip exchanges use CPU and network resources, but in most published works these have been considered negligible. If the amount of data in updates is significant, then a backlog can build up.

Say that each participant introduces updates at the same rate. While expected latencies continue to grow logarithmic in N, the size of gossip messages grows linearly in N. In practice, the maximum size of gossip messages is limited by the maximum amount of network bandwidth that the gossip protocol may use or by other considerations such as CPU time used for generating and handling messages. This means that it is important that gossip messages are used efficiently, and that the rate of updates is carefully controlled. If the rate of updates were too high, latency may grow without bounds.

In practice, we expect that limiting the rate of updates is less of a problem for applications than increasing the latency of updates. Information that is gossiped is sampled at certain intervals. Applications are typically more interested in recency of information than in how often it is sampled.

For this paper we assume that a gossip message contains at most *mtu* (Maximum Transmission Unit) tuples, each tuple consisting of a participant identifier p, a key k, a value v, and a version number n, signifying that $\mu_p(p)(k) = (v, n)$ at some time in the past. We call such tuples *deltas*. (In practice, different deltas may have different sizes—but this is not a major complication.)

3. RECONCILIATION

When two participants p and q gossip using reconciliation mechanism π, p ends up sending to q a set of deltas $\Delta_\pi^{p \to q} \subseteq \{(r, k, v, n) \mid \mu_p(r)(k) = (v, n)\}$ while q sends to p a similar set $\Delta_\pi^{q \to p}$. Should $|\Delta_\pi^{p \to q}| > mtu$, then only a subset of size *mtu* may be sent, and the same holds for $\Delta_\pi^{q \to p}$. To decide which deltas to send, deltas are totally ordered by $<_\pi$, so that for any two different elements δ_1 and δ_2 in Δ_π, either $\delta_1 <_\pi \delta_2$ or $\delta_2 <_\pi \delta_1$. Only the highest *mtu* deltas are included in a gossip message. The possible orderings for a reconciliation mechanism may be constrained by correctness requirements. Within those constraints, different orderings may lead to different performance as we will demonstrate later.

We now present two different reconciliation mechanisms. The first, *precise reconciliation*, sends only necessary updates, and is used for baseline measurements. Next we present our own mechanism, *Scuttlebutt*.

3.1 Precise Reconciliation

In precise reconciliation, the two participants in a gossip exchange send exactly those mappings that are more recent than those of the peer. Thus, if the participants are p and q, p sends to q the set of deltas

$$\Delta_{precise}^{p \to q} = \{(r, k, v, n) \in \mathcal{P} \times \mathcal{K} \times \mathcal{V} \times \mathcal{N} \mid \mu_p(r)(k) = (v, n) \wedge \mu_q(r)(k) = (\cdot, n') \ \wedge \ n > n'\}$$

while q sends to p a similar set $\Delta_{precise}^{q \to p}$.

On receipt, q uses $\Delta_{precise}^{p \to q}$ to update $\mu_q(r)$. Due to concurrent gossip activity, not all entries may be updates, so it only applies an update if the version number is larger than what is in its current $\mu_q(r)$. The same holds for p and $\Delta_{precise}^{q \to p}$.

In practice, precise reconciliation is quite difficult. To determine the differences between the participants, *digests* of the participants' states have to be exchanged. As described before, and as is clear from the definitions of $\Delta_{precise}^{p \to q}$ and $\Delta_{precise}^{q \to p}$, this can consist of sending the state without the values. This could still amount to a considerable amount of data, while compressions schemes (*e.g.*, using hashes) may consume significant CPU cycles.

How to order deltas in these sets for best results in case an *mtu* is enforced is not clear. Ordering may be done randomly or deterministically. In this paper, we use two deterministic approaches for baseline purposes. For the "precise-oldest" approach, the *mtu* most out-of-date deltas are included in the gossip message. For the "precise-newest" approach, the most recent updates are used instead. The latter may result

	$t=1$	$t=2$
p	$\mu_p(r)(a)=(\cdot,1)$ $\mu_p(r)(b)=(\cdot,2)$ $\mu_p(r)(c)=(\cdot,3)$	$\mu_p(r)(a)=(\cdot,21)$ $\mu_p(r)(b)=(\cdot,2)$ $\mu_p(r)(c)=(\cdot,3)$
q	$\mu_q(r)(a)=(\cdot,11)$ $\mu_q(r)(b)=(\cdot,12)$ $\mu_q(r)(c)=(\cdot,13)$	$\mu_q(r)(a)=(\cdot,21)$ $\mu_q(r)(b)=(\cdot,12)$ $\mu_q(r)(c)=(\cdot,13)$
r	$\mu_r(r)(a)=(\cdot,21)$ $\mu_r(r)(b)=(\cdot,22)$ $\mu_r(r)(c)=(\cdot,23)$	$\mu_r(r)(a)=(\cdot,21)$ $\mu_r(r)(b)=(\cdot,22)$ $\mu_r(r)(c)=(\cdot,23)$

Figure 1: The state of the system at two times. Values are omitted.

in starvation, where some older updates never get a chance to propagate.

Note that, if implemented, both these orderings would require a synchronized clock among the members and that all updates be timestamped with this clock.

3.2 Scuttlebutt Reconciliation

Next we present a mechanism that can be more efficient than precise reconciliation, both in terms of network bandwidth used and CPU cycles spent. Let $\max(\sigma)$ be the maximum version number used in $\sigma \in S$. A participant p is allowed to update its own state $\sigma_p = \mu_p(p)$ one key at a time. Say that the participant wishes to update key k_0 to value v_0, creating a new state σ'_p. Scuttlebutt requires that the participant uses a version number higher than any used before, that is, $\sigma'_p(k_0) = (v_0, n_0)$ where $n_0 > \max(\sigma_p)$, and $\sigma'_p(k) = \sigma_p(k)$ for all $k \neq k_0$. As a result, it is not possible for two different keys in a state to map to the same version number.

When p and q start gossiping, they first exchange digests $\{(r, \max(\mu_p(r))) \mid r \in \mathcal{P}\}$ and $\{(r, \max(\mu_q(r))) \mid r \in \mathcal{P}\}$ resp.[1] On receipt, p sends to q

$$\Delta^{p\to q}_{scuttle} = \{(r,k,v,n) \mid \mu_p(r)(k) = (v,n) \ \wedge \ n > \max(\mu_q(r))\}$$

while q sends to p a similar set $\Delta^{q\to p}_{scuttle}$.

A gossiper never transmits updates that were already known at the receiver. If gossip messages were unlimited in size, then the sets contains the exact differences, just like with precise reconciliation. If a set does not fit in the gossip message, then it is not allowed to use an arbitrary subset as in precise reconciliation. Scuttlebutt requires that if a certain delta (r,k,v,n) is omitted, then all the deltas with higher version numbers for the same r should be omitted as well. We enforce this using a constraint on the ordering between deltas:

$$n > n' \Rightarrow (r,k,v,n) <_{scuttle} (r,k',v',n') \qquad (1)$$

[1]In practice, the gossiper sends the first message, and the gossipee responds on receipt. For clarity, we simplify the protocol here.

Scuttlebutt satisfies the global invariant $\mathcal{C}(p,q)$ for any two processes p and q:

$$\mathcal{C}(p,q) \equiv \forall k \in \mathcal{K} : \begin{cases} \mu_p(p)(k) = \mu_q(p)(k) \ \vee \\ \mu_p(p)(k).n > \max(\mu_q(p)) \end{cases} \qquad (2)$$

In other words, each process q either has the current mapping for a key k at process p, or the current version of k is larger than the maximum that q has for p. Because initially all keys at all processes are mapped to $(\perp,\perp)$, and $\perp$ is the lowest possible version number, this invariant holds in the initial state of the system. It is obvious too that at all times $\mathcal{C}(p,p)$ holds, and thus local updates to a mapping preserve the invariant. Finally, it is easy to verify that a gossip exchange of the mapping of r between processes p and q maintains the invariant, that is, the new version of $\mu_p(r)$ satisfies $\mathcal{C}(r,p)$ while the new version of $\mu_q(r)$ satisfies $\mathcal{C}(r,q)$. (In fact, at most one of the two tables, the one with the smaller maximum version number, would be updated.) This is why it is important that deltas are communicated in the order of version number; if not, the invariant could be violated.

A remarkable property of Scuttlebutt is that the gossip exchange between two participants is not designed to eliminate all differences between the two participants, even if there is sufficient room in the gossip messages. See Figure 1 for an example. The table shows part of the states of participant r at participants p, q, and r for keys a, b, and c. Between times $t=1$ and $t=2$, p and r have been able to exchange one delta, as have q and r. At time $t=2$, $\max(\mu_p(r))$ and $\max(\mu_q(r))$ are the same, and should p and q gossip, q will not send its more recent entries to p.

This may appear to be a shortcoming—precise reconciliation can eliminate all differences and therefore potentially converge faster. However, the differences that would be eliminated involve keys that have already been updated at r, and thus would have to be reconciled again. Not sending the unnecessary updates leaves room in the gossip message for deltas of other participants.

Given invariant $\mathcal{C}(p,q)$, the following property holds because no key of p can have a version number higher than $\max(\mu_p(p))$:

$$\max(\mu_q(p)) = \max(\mu_p(p)) \Rightarrow \mu_q(p) = \mu_p(p) \qquad (3)$$

It is then clear that Scuttlebutt converges to consistency in spite of not necessarily updating all keys in every exchange.

More than one ordering satisfies the constraint in Equation 1, but these may result in different performance characteristics. We present two possible orderings. The "scuttlebreadth" ordering function tries to be fair to all participants by including in each gossip message deltas from as many different participants as possible. It uses a ranking on deltas for the same participant. The delta with the lowest version number has rank 0, the next lowest rank 1, and so on. The deltas are first ordered by rank so that deltas with lower

ranks are included before deltas with higher ranks. For two different deltas with the same rank, and thus necessarily for different participants, the ordering is based on an ordering among the participants. This ordering should be different for each gossip exchange to eliminate long-term bias, and our experiments use a different pseudo-random ordering for each gossip message. Note that this ordering easily satisfies the constraint on $<_{scuttle}$.

The other ordering function we present, "scuttle-depth," is the best we encountered from various trials. Instead of being fair to all participants, it prioritizes updates for those who are most left behind. That is, scuttle-depth prefers deltas of participants for which more deltas are available over deltas of participants with few available deltas. For participants with the same number of available deltas, random ordering among participants is used to remove bias (as in scuttlebreadth). Finally, deltas from the same participant are inversely ordered by their version number, consistent with the ordering constraint.

So far we have assumed that a Scuttlebutt digest, consisting of a version number for each participant, fits in a network message, but in a large system this may not be the case. Large systems that use gossip are often organized hierarchically (*e.g.*, [13]) such that individual gossip populations are limited in size. If need be, multiple messages may be used to exchange the digests. or the set of participant identifiers and corresponding version numbers itself could be reconciled using some reconciliation mechanism. For example, instead of including all participants in a digest, it is possible to include a pseudo-random participants subset of the required maximum size, which still works but increases propagation time.

3.3 Evaluation

To evaluate the reconciliation strategies discussed, we ran various simulated experiments in overload scenarios. There are 128 participants, each having 64 key/value pairs. Each participant gossips once per second. The values are updated at a certain rate, the same for each participant. Values are updated uniformly at random.

The experiment starts at time $t = 0$. The initial update rate per participant, ρ, is 1 per second. In the first 15 seconds of the simulation, no message size limit is enforced, allowing the system to warm up to remove bias. After this, a message limit is enforced. At $t = 25$, we double the rate. At $t = 75$, we restore the rate. Finally, at $t = 120$, updates are terminated, allowing the system to converge on a single set of values.

In Figure 2(a) we report maximum staleness as a function of time for a typical experiment with $mtu = 100$. A key-value mapping $\mu_q(p)(k)$ is stale if $\mu_q(p)(k) \neq \mu_p(p)(k)$. The staleness of such a mapping $\mu_q(p)(k)$ is the amount of time that has lapsed since $\mu_q(p)(k)$ was last updated.

The aggregate update rate is 128 updates per second, and in a system without message size limitations one would expect the average gossip message to contain half this number of updates (as there is a gossip message in each direction in a gossip exchange). Thus we would expect each of the protocols to work well, and the experiments bear this out. Latency of dissemination under these conditions is about 5 or 6 rounds of gossip, as can be derived from epidemic analysis.

At $t = 25$ the aggregate rate doubles to 256 updates per second, creating overload, and we see that the different approaches diverge. Precise-newest results in the highest maximum staleness due to starvation of values that did not get a chance to be disseminated early on, but recovers fastest after the update stream stops. To see why this is, see Figure 2(b), which reports the number of stale mappings as a function of time: precise-newest has the fewest stale mappings of all protocols. This explains its fast recovery: it has the fewest values to disseminate after $t = 75$. Precise-oldest has much better maximum staleness, but recovers slowly exactly because it generates the most stale mappings.

There appears to be a trade-off between maximum staleness and the number of stale mappings, although as we can see, scuttle-depth scores well on both metrics. The scuttlebreadth protocol has disappointing results, strongly indicating the importance of a good ordering. After updates cease at $t = 120$, the states of the participants converges for all protocols.

We did various other experiments, using different numbers of participants, different numbers of gossip neighbors, different update rates and message sizes, and using a Zipf distribution for updates, but in all cases the trends are much the same.

Scuttle-depth performs best, almost able to keep up in some of these heavy load cases and recovering quickly. But no matter how good the reconciliation protocol, overload remains a problem.

4. FLOW CONTROL

The objective of a flow control mechanism for gossip is to determine, adaptively, the maximum rate at which a participant can submit updates without creating a backlog of updates. A flow control mechanism should be fair, and under high load afford each participant that wants to submit updates the same update rate. As there is no global oversight, the flow control mechanism has to be decentralized, where the desired behavior emerges from participants responding to local events.

Local events that may be monitored are overflows of gossip messages. Occasional overflow is not problematic—all our reconciliation protocols can deal with this. But if there is a trend in which the overflow becomes increasingly worse, then a participant should back off generating updates.

In isolation, such a strategy by itself results in some participants backing off, while other participants increase their update rate. To compensate for such unfairness, we can use the gossip mechanism itself. When two participants gossip, they exchange their current maximum update rates. Assuming both participants wish to update at the maximum rate possible, they split the difference, each ending up with the same maximum update rate, which is the average of their old update rates. The aggregate update rate will not have changed, but the system as a whole has increased its fairness.

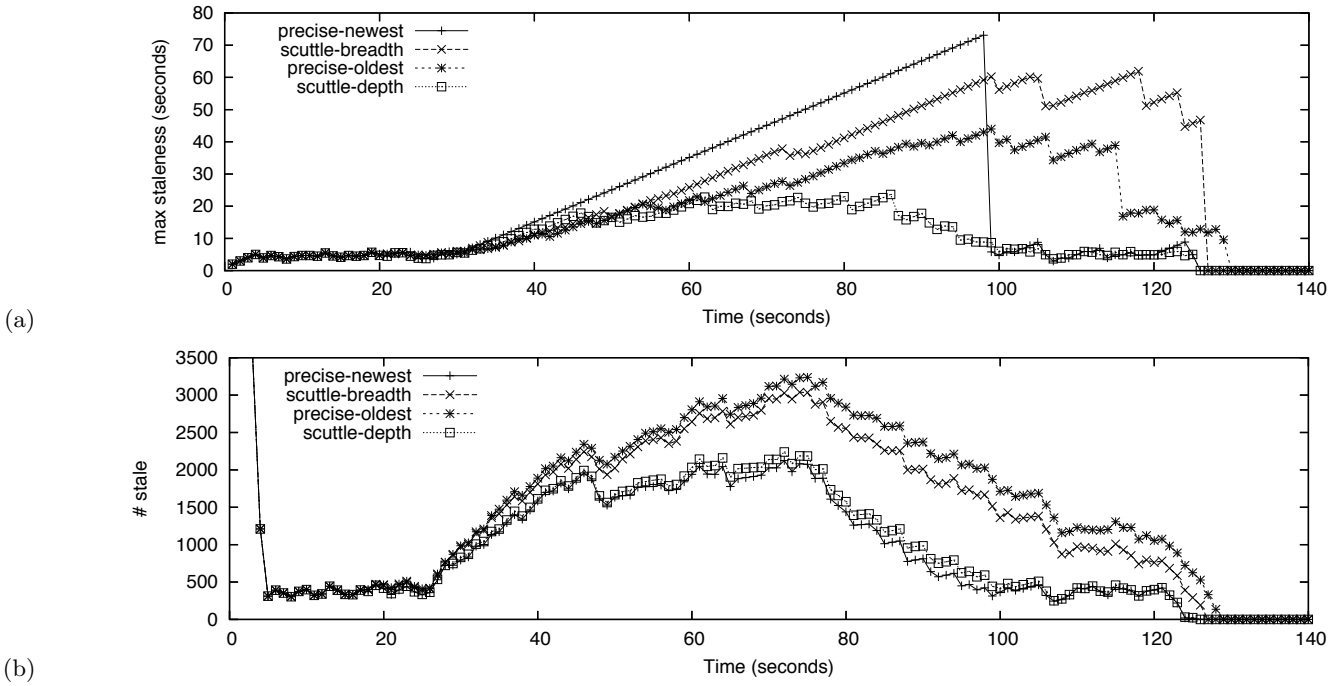

Figure 2: (a) Maximum staleness and (b) total number of values that are stale in a system with $mtu = 100$ and $\rho = 1$.

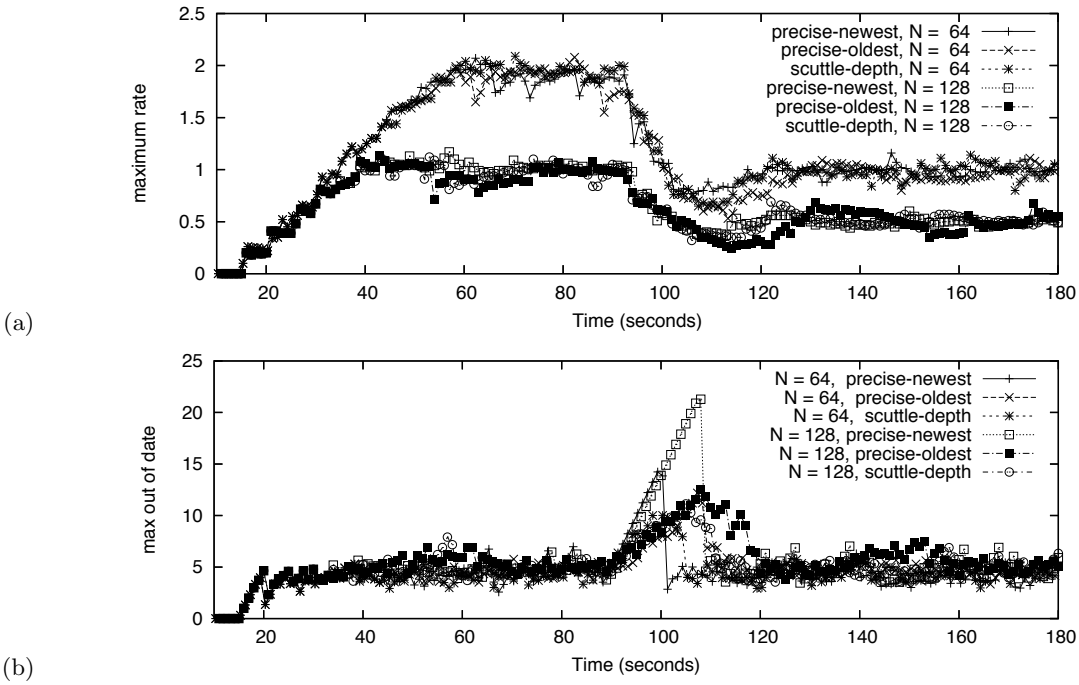

Figure 3: (a) Update rate and (b) Maximum staleness at one participant for $mtu = 100$.

If one of the participants is not updating at the maximum rate available to it, the participant can reduce its maximum update rate and transfer the remainder to its peer.

4.1 Spreading Capacity Fairly

Each participant p has a desired update rate ρ_p,[2] but will not submit updates faster than a maximum update rate, τ_p, which it adjusts as necessary. When two participants p and q gossip, they exchange their values of these variables. If $\rho_p + \rho_q \leq \tau_p + \tau_q$, both will be able to send at their desired rates, and the remainder may be split evenly between them:

- $\tau'_p = \rho_p + (\tau_p + \tau_q - \rho_p - \rho_q)/2$
- $\tau'_q = \rho_q + (\tau_p + \tau_q - \rho_p - \rho_q)/2$

Should, however, $\rho_p + \rho_q > \tau_p + \tau_q$, at least one of the two participants will not be able to send at their desired rate. If $\rho_p \geq (\tau_p + \tau_q)/2 \ \wedge \ \rho_q \geq (\tau_p + \tau_q)/2$, then the participants both get the same share:

- $\tau'_p = (\tau_p + \tau_q)/2$, $\tau'_q = (\tau_p + \tau_q)/2$

If either p or q wants to send less than the average maximum rate, that participant can be accommodated while the other receives the remainder. Without loss of generality, say $\rho_p < (\tau_p + \tau_q)/2$. Then

- $\tau'_p = \rho_p$, $\tau'_q = \tau_p + \tau_q - \rho_p$

Note that in all three cases, $\tau'_p + \tau'_q = \tau_p + \tau_q$, so no capacity is lost or gained in the process.

4.2 Local Adaptation

For local adaptation, we use an approach inspired by TCP flow control [4]. In TCP, the send window adapts according to a strategy called Additive Increase Multiplicative Decrease (AIMD). In this strategy, window size grows linearly with each successful transmission, but is decreased by a certain factor whenever overflow occurs. In the case of TCP, the overflow signal is the absence of an acknowledgment.

We apply a similar strategy to adapt τ_p, the maximum number of updates that p can send per gossip interval. The overflow signal is generated whenever a certain number of consecutive gossip messages are completely filled. We found that a single such occurrence is not a good indicator. Our AIMD rules are as follows:

- if for ϕ_1 gossip exchanges in a row one or both (sent and received) of the delta sets is larger than mtu, then $\tau'_p = \alpha \cdot \tau_p$, $0 < \alpha < 1$;
- if for ϕ_2 gossip exchanges in a row both delta sets are smaller than mtu, then $\tau'_p = \min(\tau_p + \beta, mtu), \beta > 0$.

For our experiments, $\phi_1 = \phi_2 = 3$, $\alpha = 0.75$, and $\beta = .2$.

[2] To determine the desired update rate, each participant would monitor the rate at which updates become available.

4.3 Evaluation

To demonstrate the efficacy of the adaptation, we ran more experiments. Each participant submits as many updates as its maximum update rate allows, starting at $t = 15$ seconds. At this time $mtu = 100$. At time $t = 90$, we synthetically created a sudden network load increase by reducing the mtu to 50.

Figures 3(a) shows the update rate at a randomly chosen participant for a selection of reconciliation mechanisms and orderings. (We now exclude scuttle-breadth for readability.) As can be seen, starting at $t = 15$ the rate creeps up and the participant is sending increasingly faster until a balance is achieved. At $t = 90$, when the gossip message size is halved, the rate quickly drops and settles at a rate that can be supported with the new message size limit. The adaptation appears mostly independent of the protocol used, although precise-oldest has more trouble adjusting at $t = 90$ and drops temporarily to well below a rate that it can handle.

Figures 3(b) shows the maximum staleness as seen across all participants. At $t = 90$, when the mtu is halved, the protocols are having some difficulty adjusting. All protocols recover, but scuttle-depth provides the best trade-off between recovery time and maximum staleness.

5. RELATED WORK

In the seminal Clearinghouse paper [3], the authors propose an iterative precise reconciliation technique, exchanging most recent updates first until hashes show that the states have been reconciled. Our experiments with this approach have been disappointing, and we found that two-thirds or more of updates are sent unnecessarily. Trachtenberg, Minsky, and Zippel [8] as well as Byers, Considine, and Mitzenmacher [1] propose techniques that work well for rumor-mongering, but cannot support anti-entropy.

For global adaptation, a variety of papers have addressed flow control for multicast protocols (*e.g.*, [2]). Typically these protocols consider a single sender trying to send messages as quickly as possible without overloading the receivers, and use end-to-end feedback from the receivers to the sender. Anti-entropy protocols, on the other hand, have all participants contributing traffic, and are mostly interested in reducing staleness of information.

Previous work on flow control for gossip has focused on rumor-mongering, where the objective is dissemination of message streams rather than fixing inconsistencies between participants. Several projects have proposed strategies for dealing with limited message buffering capacity at participants [9, 7, 6]. Pereira et al. [10] present NEEM, an epidemic multicast protocol that uses TCP connections between participants to provide flow control. To deal with backlog on TCP connections, the authors propose various message purging strategies.

Rodrigues et al. [12] use a strategy similar to ours, piggybacking resource information on existing gossip traffic and adjusting senders' rates accordingly. Their objective is to reduce the number of messages dropped. Pereira et al. [11]

look at reducing latency in rumor-mongering protocols by strategically picking gossip partners.

6. CONCLUSION

Anti-entropy has a limited capacity given a certain budget of network bandwidth and CPU cycles. We have demonstrated that if too much data is gossiped, anti-entropy protocols lose their objective of predictable performance. The paper presents two complementary techniques.

The first technique is a new reconciliation mechanism that, in the face of overload, aggressively selects updates that have not been made obsolete by later updates, but without starving updates that are not yet obsolete. The second technique is a flow control mechanism for anti-entropy protocols. In this mechanism, each participant locally adapts its rate of updates. The protocol assures fairness by dividing the available network capacity among the participants that are actively gossiping new updates.

While each technique is useful by itself, the combination appears particularly effective. We believe that both techniques are amenable to further improvements, such as giving differentiated performance for classes of updates, and accommodating heterogeneous participants, for example in environments where not all participants have equal network access.

Acknowledgments

We thank the anonymous LADIS 2008 reviewers for their helpful suggestions; they have significantly improved the quality of this paper.

7. REFERENCES

[1] J. Byers, J. Considine, and M. Mitzenmacher. Fast approximate reconciliation of set differences. Technical Report 2002-019, CS Dept., Boston University, July 2002.

[2] P. Danzig. Flow control for limited buffer multicast. *IEEE Trans. on Software Engineering*, 20(1), Jan. 1994.

[3] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proc. of the 6th ACM Symp. on Principles of Distributed Computing*, pages 1–12, Vancouver, BC, Aug. 1987.

[4] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM'88*, Stanford, CA, Aug. 1988.

[5] A.-M. Kermarrec, L. Massoulié, and A. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Trans. on Par. and Distr. Systems*, 14(3), Mar. 2003.

[6] B. Koldehofe. Buffer management in probabilistic peer-to-peer communication protocols. In *Proc. of the 22nd Symp. on Reliable Distributed Systems (SRDS '03)*, Oct. 2003.

[7] P. Kouznetsov, R. Guerraoui, S. Handurukande, and A.-M. Kermarrec. Reducing noise in gossip-based reliable broadcast. In *Proc. of the 20th IEEE Symp. on Reliable Distributed Systems (SRDS '01)*. IEEE, Oct. 2001.

[8] Y. Minsky, A. Trachtenberg, and R. Zippel. Set reconciliation with nearly optimal communication complexity. *IEEE Trans. on Information Theory*, 49(9), Sept. 2003.

[9] O. Ozkasap, R. van Renesse, K. Birman, and Z. Xiao. Efficient buffering in reliable multicast protocols. In *Proceedings of the First International Workshop on Networked Group Communication (NGC)*, Pisa, Italy, Nov. 1999.

[10] J. Pereira, L. Rodrigues, M. Monteiro, R. Oliveira, and A.-M. Kermarrec. NEEM: Network-friendly epidemic multicast. In *Proc. of the 22nd Symp. on Reliable Distributed Systems (SRDS '03)*, Oct. 2003.

[11] J. Pereira, L. Rodrigues, A. Pinto, and R. Oliveira. Low latency probabilistic broadcast in wide area networks. In *Proc. of the 23rd Int. Symp. on Reliable Distributed Systems*, Florianópolis, Brazil, Oct. 2004.

[12] L. Rodrigues, S. Handurukande, J. Pereira, R. Guerraoui, and A.-M. Kermarrec. Adaptive gossip-based broadcast. In *Proc. of the International Conference on Distributed Systems and Networks (DSN)*, San Francisco, CA, June 2003. IEEE.

[13] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed systems monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(3), May 2003.