Gossiping in Distributed Systems
Foundations

Maarten van Steen

vrije Universiteit amsterdam

Introduction

Observation: We continue to face hard scalability problems in distributed systems:

- Systems continue to grow in size:
  - There are many participating nodes
  - Membership changes: there is no equilibrium
- Systems continue to expand geographically:
  - Nodes lie farther apart, leading to an increase in latency
  - Diameter (expressed in time) increases
  - Nodes fall under different administrative domains

Needed: General-purpose, **decentralized solutions**

Gossiping as a partial solution

Principle: spread (meta-)information to allow for local-only decision making:

- Nodes exchange data with neighbors:
  - data is efficiently disseminated
  - set of neighbors need not be fixed
- Nodes rely only on incomplete information
- Exchanged data can be anything: from actual data to references to nodes to programs
- There is no centralized control or management

Gossip-based applications

- Raw information dissemination
- Data aggregation
- Topology construction for overlay networks
- Semantic clustering of nodes
- Realizing storage facilities in ad hoc networks

Note: Gossiping is not a universal solution

Some observations

- There's a lot of emergent behavior (i.e., behavior we don't understand).
- Theory is (partially) lacking: models are often difficult to validate.
- There are many practical issues still to solve:
  - Adaptiveness (too many design-time parameters)
  - Security (attacking a gossip-based system is easy)
  - Competitive alternative single-point solutions

Lectures: overview

- Lecture 1: Foundations
  - Basics
  - Peer selection
  - Theory versus practice
- Lecture 2: Applications
  - Data aggregation
  - Structure management:
    - ∗ topology management
    - ∗ file searching
  - Storage in wireless networks

Gossiping: principle operation

Anti-entropy: Each replica regularly chooses another replica at random, and exchanges state differences, leading to identical states at both afterwards.

Gossiping: A replica which has just been updated (i.e., has been contaminated), tells a number of other replicas about its update (contaminating them as well).

System Model

- Consider N nodes, each storing a number of objects
- Each object O has a primary node at which updates for O are always initiated.
- An update of object O at node S is always timestamped; the value of O at S is denoted $val(O,S)$
- $T(O,S)$ is the timestamp of the value of object O at node S

Anti-Entropy

Basic issue: When a node S contacts another node S^* to exchange state information, three different strategies can be followed:

Push: $T(O,S^*) < T(O,S) \Rightarrow val(O,S^*) \leftarrow val(O,S)$
Pull: $T(O,S^*) > T(O,S) \Rightarrow val(O,S) \leftarrow val(O,S^*)$
Push-Pull: S and S^* exchange their updates

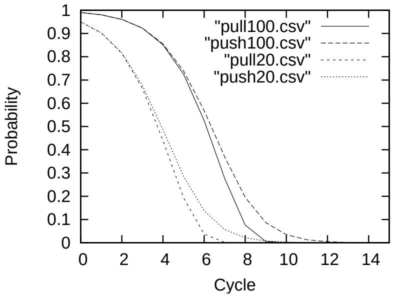
Observation: if each node periodically randomly chooses another node for exchanging updates, an update is propagated in $O(\log(N))$ cycles.

Anti-Entropy: Analysis

Consider a single source, propagating its update. Let p_i be the probability that a node has not received the update after the i-th cycle.

- With pull, $p_{i+1} = (p_i)^2$: the node was not updated during the i-th cycle and should contact another ignorant node during the next cycle.
- With push, $p_{i+1} = p_i(1 - \frac{1}{N})^{N(1-p_i)} \approx p_i e^{-1}$ (for small p_i and large N): the node was ignorant during the i-th cycle and no updated node chooses to contact it during the next cycle.

Anti-entropy: some figures



Pure gossiping: basic model

1. A node P with an update (P is infected) contacts other node Q.
2. If Q already knows the update (Q is not susceptible), P stops with probability $1/k$ (P is effectively removed).
3. Otherwise, P contacts another (randomly selected) node.

Gossiping: basic math

Notation: s is fraction of nodes not yet updated, i is fraction of active (updated) nodes, r is fraction of passive (updated) nodes: $s+i+r=1$. From epidemics:

$$\begin{array}{lrcl}
(1) & ds/dt & = & -si \\
(2) & di/dt & = & si - \frac{1}{k}(1-s)i \\
(3) & di/ds & = & -\frac{k+1}{k} + \frac{1}{ks} \\
(4) & i(s) & = & -\frac{k+1}{k}s + \frac{1}{k}\ln s + C
\end{array}$$

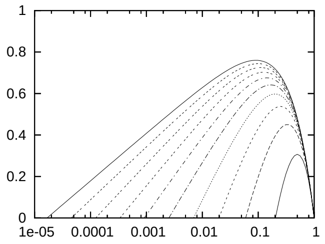
Gossiping: basic math

With $i(1) = 0$, we obtain $C = \frac{k+1}{k}$, and thus

$$i(s) = \frac{k+1}{k}(1-s) + \frac{1}{k}\ln s$$

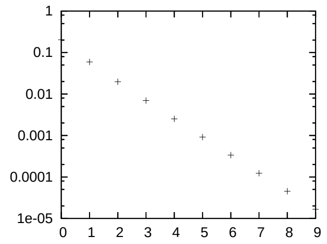
left to right

$k = 10, 9, \ldots, 1$



Gossiping: the unaffected

$i(s) = 0$ implies no more activity $\Rightarrow s = e^{-(k+1)(1-s)}$



Consider 10,000 nodes

k	s	N_s
1	0.203188	2032
2	0.059520	595
3	0.019827	198
4	0.006977	70
5	0.002516	25
6	0.000918	9
7	0.000336	3

Observation: all nodes need to be updated $\Rightarrow$ pure gossiping is not enough.

Getting a random peer

Important: Gossip-based systems rely on the following important assumption:

> A node P can select another peer Q drawn uniform at random from the current set of nodes.

Observation: This seems to imply that every node as an accurate view on the complete membership!

Getting a random peer

Question: What does it take to build a decent peer-sampling service?

- Nodes are provided a peer drawn uniform at random from the complete set of nodes
- Sampling is accurate, reflecting current set of nodes
- Draws by different nodes are independent
- The service should be scalable

Key issue: The service can be built entirely with epidemic-based techniques.

Framework - overview

Active thread

```
selectPeer(&Q);
selectToSend(&bufs);
sendTo(Q, bufs);

receiveFrom(Q, &bufr);
selectToKeep(p_view, bufr);
```

Passive thread

```


receiveFromAny(&P, &bufr);
selectToSend(&bufs);
sendTo(P, bufs);
selectToKeep(p_view, bufr);
```

selectPeer	Randomly select a neighbor
selectToSend	Select some entries from local list
selectToKeep	Add received entries to local list. Remove repeated items.

Simple? Not quite when getting into some details...

Framework - for real

- N nodes, each having an address
- Every node has a partial view: a local list of c node descriptors
- Node descriptor $= \langle$ *address, age* $\rangle$ pair
- Operations on partial view:

selectPeer()	return an item
permute()	randomly shuffle items
increaseAge()	forall items add 1 to age
append(...)	append a number of items
removeDuplicates()	remove duplicates (on same address), keep youngest
removeOldItems(n)	remove n descriptors with highest age
removeHead(n)	remove n first descriptors
removeRandom(n)	remove n random descriptors

Active thread (one per node)

```
do forever
  wait(T time units) // T is called the cycle length
  $p \leftarrow$ view.selectPeer() // Sample a live peer from the current view
  if push then // Take initiative in exchanging partial views
      buffer $\leftarrow$ ($\langle$ MyAddress,0 $\rangle$) // Construct a temporary list
      view.permute() // Shuffle the items in the view
      move oldest H items to end of view // Necessary to get rid of dead links
      buffer.append(view.head(c/2)) // Copy first half of all items to temp. list
      send buffer to $p$
  else // empty view to trigger response
      send (null) to $p$
  if pull then // Pick up the response from your peer
      receive buffer$_p$ from $p$
      view.select(c,H,S,buffer$_p$) // Core of framework – to be explained
  view.increaseAge()
```

Passive thread (one per node)

```
do forever
  receive buffer$_p$ from $p$ // Wait for any initiated exchange
  if pull then // Executed if you're supposed to react to initiatives
      buffer $\leftarrow$ ($\langle$ MyAddress,0 $\rangle$) // Construct a temporary list
      view.permute() // Shuffle the items in the view
      move oldest H items to end of view // Necessary to get rid of dead links
      buffer.append(view.head(c/2)) // Copy first half of all items to temp. list
      send buffer to $p$
  view.select(c,H,S,buffer$_p$) // Core of framework – to be explained
  view.increaseAge()
```

View selection

Parameters:

c: length of partial view
H: number of items moved to end of list (healing)
S: number of items that are swapped with a peer
buffer$_p$: received list from peer

```
method view.select( c, H, S, buffer_p )
  view.append(buffer_p) // expand the current view
  view.removeDuplicates() // Remove by duplicate address, keeping youngest
  view.removeOldItems( min(H,view.size-c) ) // Drop oldest, but keep c items
  view.removeHead( min(S,view.size-c) ) // Drop the ones you sent to peer
  view.removeAtRandom(view.size-c) // Keep c items (if still necessary)
```

Design space – peer selection

selectPeer() returns a live peer from the current view. Essentially, there are three possibilities:

head: pick the address of the youngest descriptor (i.e., with low age) – bad choice, since this is the neighbor the node most recently communicated with ⇒ offers little opportunities for selecting unknown nodes (confirmed by experiments)

rand: pick the address of a randomly selected descriptor

tail: pick the address of the oldest descriptor (i.e., with high age)

Design space – view propagation

push: Node sends descriptors to selected peer

pull: Node only pulls in descriptors from selected peer

pushpull: Node and selected peer exchange descriptors

Note: pulling alone is pretty bad: a node has no opportunity to insert information on itself. Loss of all incoming connections will throw a node out of the network (may actually happen).

Design space – view selection

Note: Critical parameters are H and S in method select(c, H, S, buffer). Assume c is even.

- $[H > c/2] \equiv [H = c/2]$, as minimum view size is always c
- Likewise, $[S > c/2 - H] \equiv [S = c/2 - H]$
- Do random removal (last step) only if $S < c/2 - H$
- Conclusion: consider only $0 \leq H \leq c/2$ and $0 \leq S \leq c/2 - H$

blind: remove($H = 0, S = 0$) — select blindly a random subset

healer: remove($H = c/2, S = 0$) — select freshest items

swapper: remove($H = 0, S = c/2$) — min. loss of descriptors

Local evaluations (1/2)

Method: Organize a network of $N = 2^n + 1$ nodes and let node N sample the network, each time providing an n-bit sample.

- With $n = 10$, node N generates 4 samples per cycle, and constructs a 32-bit integer.
- The 32-bit integers together form a stream of numbers, which should be random if peer sampling is random.
- Series is tested by the “diehard battery of randomness tests.” (see `www.stat.fsu.edu/pub/diehard`)
- Examined blind,healer,swapper, fixing to tail and pushpull

Local evaluations (2/2)

Results: All tests could be passed (!)

One exception: construction of binary matrices produced too many matrices with a high rank. This failure is caused by our tendency to maximize diversity.

“Fix”: by considering only every 8th sample in the generated series, all tests are passed.

Conclusion: it is difficult to observe nonrandom local behavior. The functional properties of peer sampling are barely affected by the choice of implementation.

Applications will often not see the difference

Global randomness

Issue: Deciding on global randomness is a bit tricky ⇒ focus on structural properties by comparing to random graph (= partial view consists of c uniform randomly chosen peers).

Indegree distribution: has a serious effect on load balancing: hot spots, bottlenecks, but also on the spreading of information.

Fault tolerance: to what extent can the service withstand catastrophic failures and high churn?

Note: concentrate on $N = 10,000$ and $c = 30$. Results are based on simulations and emulations.

Convergence behavior

Consider three starting situations:

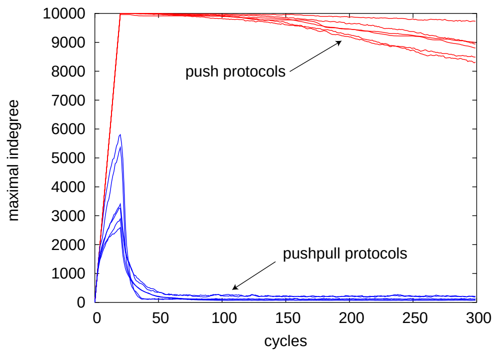
Growing: Start with one node X. Before starting a next cycle, add 500 nodes. Each new node knows only about X.

Lattice: Organize all nodes in a ring. Add descriptors of nearest nodes in the ring.

Random: Every view is filled with a uniform random sample of all nodes.

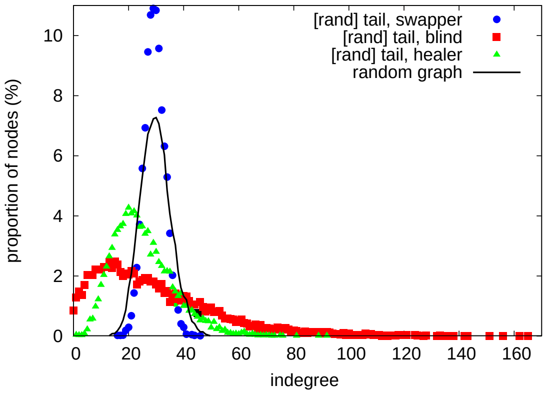
Observation: Pure pushing converges poorly and often leads to partitioned overlays in growing scenario.

Maximal indegree growing scenario



Note: From now on consider only pushpull protocols

Converged indegree distribution



Fluctuation of degree distribution (1/2)

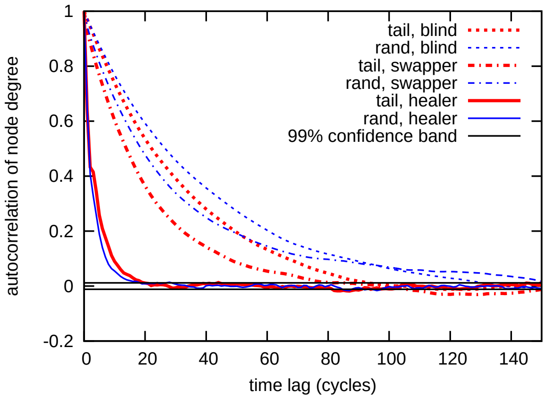
Observation: it turns out that the in-degree for each node changes over time. The question is how quickly.

Let $d_1, \ldots, d_K$ denote in-degree for a fixed node for K consecutive cycles, and $\bar{d}$ the average in-degree. Let

$$r_k = \frac{\sum_{j=1}^{K-k}(d_j - \bar{d})(d_{j+k} - \bar{d})}{\sum_{j=1}^{K}(d_j - \bar{d})^2}$$

be the correlation between pairs of in-degree separated by k cycles.

Fluctuation of degree distribution (2/2)



Clustering coefficient (1/2)

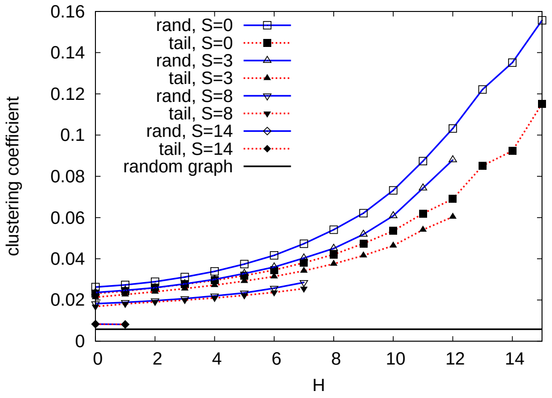
Note: Consider the undirected graph by dropping the direction.

Clustering coefficient indicates to what extent the neighbors of a node X are each other's neighbors. Let Γ_X denote the graph induced by the neighbors of node X.

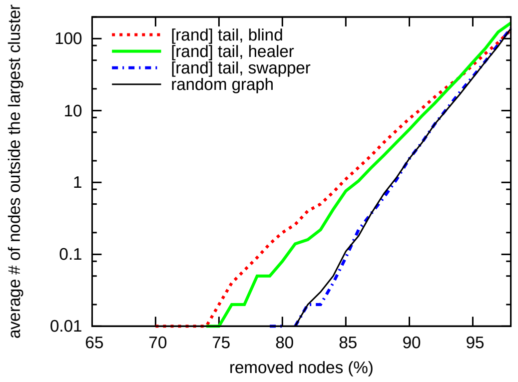
$$\gamma(X) = \frac{|E(\Gamma_X)|}{\binom{|V(\Gamma_X)|}{2}}$$

For a graph: take the average over all nodes.

Clustering coefficient (2/2)

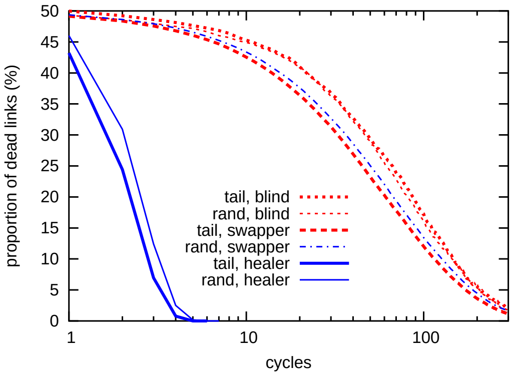


Catastrophic failure



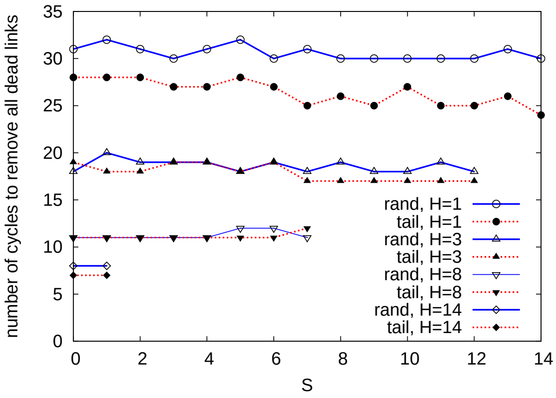
Scenario: After 300 cycles, remove large fraction of nodes.

Dead links (1/2)

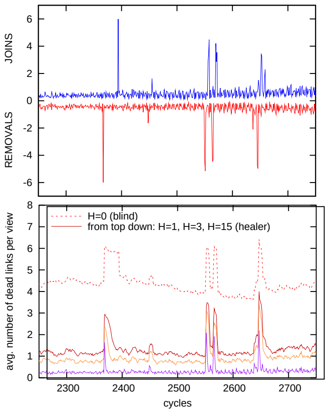


Scenario: After 300 cycles, remove 50% of nodes.

Dead links (2/2)



Handling churn: Gnutella traces



Conclusions

- Push-pull gossip protocols perform better than only push or pull
- Discarding old references is good for fault tolerance (but may also be “too” good)
- Swapping references is good for maintaining well-balanced graphs (in-degree ≈ out-degree)
- Differences between protocols mainly affect the nonfunctional properties of applications

Reading material

[1] 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. Sixth Symp. on Principles of Distributed Computing*, pp. 1–12, Aug. 1987. ACM.

[2] P. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulié. “Epidemic Information Dissemination in Distributed Systems.” *IEEE Computer*, 37(5):60–67, May 2004.

[3] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. “Gossip-based Peer Sampling.” *ACM Trans. Comp. Syst.*, 25(3), Aug. 2007.