Gossip-based Peer Sampling

Maarten van Steen

vrije Universiteit amsterdam

Introduction

- Epidemic-based protocols are popular for communication in large-scale distributed systems:
  - reliable in the presence of high churn and network failures
  - efficient when it comes to management
  - often local-only solutions
- Applications:
  - Information dissemination
  - Topology/overlay construction
  - Resource management (node allocation, replica mgt.)
  - Decentralized computations (aggregation, data fusion)

Basics

Assume there are no write–write conflicts:

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 $val(O, N)$
- $T(O, N)$ denotes 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: $T(O,S^*) < T(O,N) \Rightarrow val(O,S^*) \leftarrow val(O,N)$

Pull: $T(O,S^*) > T(O,N) \Rightarrow val(O,N) \leftarrow val(O,S^*)$

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.

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.

Gossiping

Basic model: A node P with an update contacts other node Q. If Q already knows the update, P stops contacting other nodes with probability $1/k$.

Fraction of ignorant nodes:
$s = e^{-(k+1)(1-s)}$

k	s
1	0.203188
2	0.059520
3	0.019827
4	0.006977
5	0.002516

Observation: If we have to ensure that all nodes are eventually updated, gossiping alone is not enough.

Observation

So far: Models assume a peer P selects node Q uniformly at random $\Rightarrow$ generally not realistic for large distributed systems:

- Systems can easily consist of 10,000+ nodes
- Nodes join and leave regularly: churn can easily be $>$ 1%
- special case: nodes fail and recover

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

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

A bit of history...

Note: Nodes will maintain a (changing) list of neighbors, inducing a (directed) communication graph.

- Márk Jelasity and I developed the Newscast protocol (2002)
- Patrick Eugster, Rachid Guerraoui and Anne-Marie Kermarrec developed lpbcast (2001–2003)
- Eugster et al. assumed communication graph to be random, developed a nice theoretical framework and got results published in ACM TOCS

A bit of history...

- We had already discovered that the lpbcast (as well as the Newscast) graph was far from being random
- We got a bit frustrated (having only our tech report)...

Issue: If you can't beat 'em, join 'em...

Collaborators

- Márk Jelasity, University of Szeged (Hungary)
- Spyros Voulgaris, ETH, Zürich
- Rachid Guerraoui, EPFL, Lausanne
- Anne-Marie Kermarrec, INRIA, Rennes
- Maarten van Steen, VU, Amsterdam

BTW: By now, we finally understand why assuming a random graph is never obvious, and should be explicitly validated.

Talk - outline

- Present framework for peer sampling that captures many different protocols
- Evaluation of local randomness
  (or: why assuming uniformity is correct)
- Evaluation of global randomness
  (or: why assuming uniformity is not correct)
- Conclusions

Framework - overview

Active thread

```
selectPeer(&Q);
selectToSend(&refs_s);
sendTo(Q,{me,refs_s});

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

Passive thread

```
receiveFromAny(&P, &refs_r);
selectToSend(&refs_s);
sendTo(P,{me,refs_s});
selectToKeep(p_view,refs_r);
```

selectPeer	Select a current neighbor (from partial view).
selectToSend	Select $c/2$ entries from partial view.
selectToKeep	Add received entries to partial view. Remove repeated items. Then keep c entries.

Note: We can also exchange data items, or combination of data and references

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 ← view.selectPeer() // Sample a live peer from the current view
  if push then // Take initiative in exchanging partial views
      buffer ← (⟨ MyAddress,0 ⟩) // 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 ← (⟨ MyAddress,0 ⟩) // 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 $\Rightarrow$ 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 $0 \le H \le c/2$ and $0 \le S \le c/2 - H$

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

- Essence: each node is allocated a unique ID from $[0, N-1]$
- Consider the series of selected IDs by a specific peer
- 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

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 $\Rightarrow$ 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.

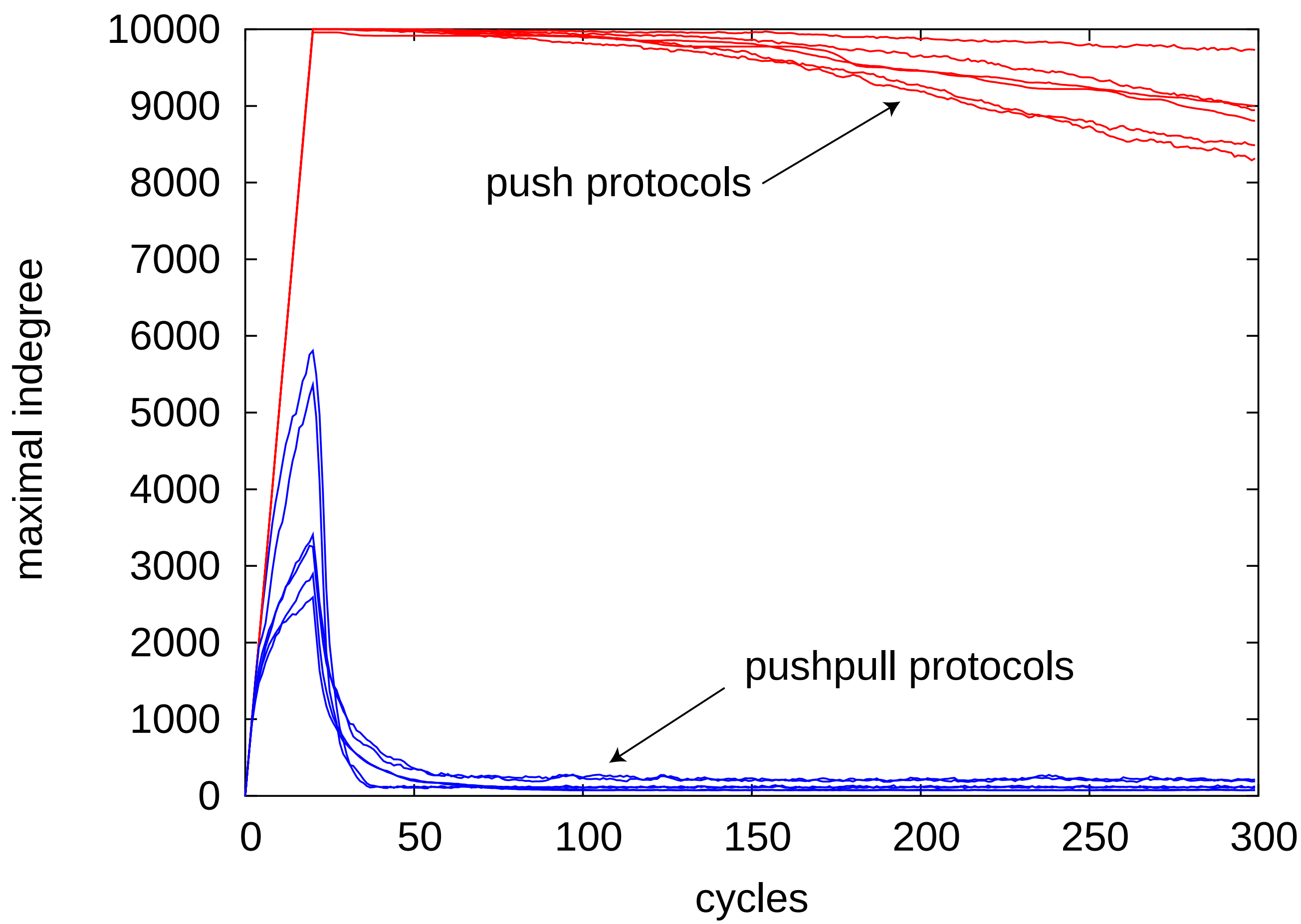
Convergence behavior

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.

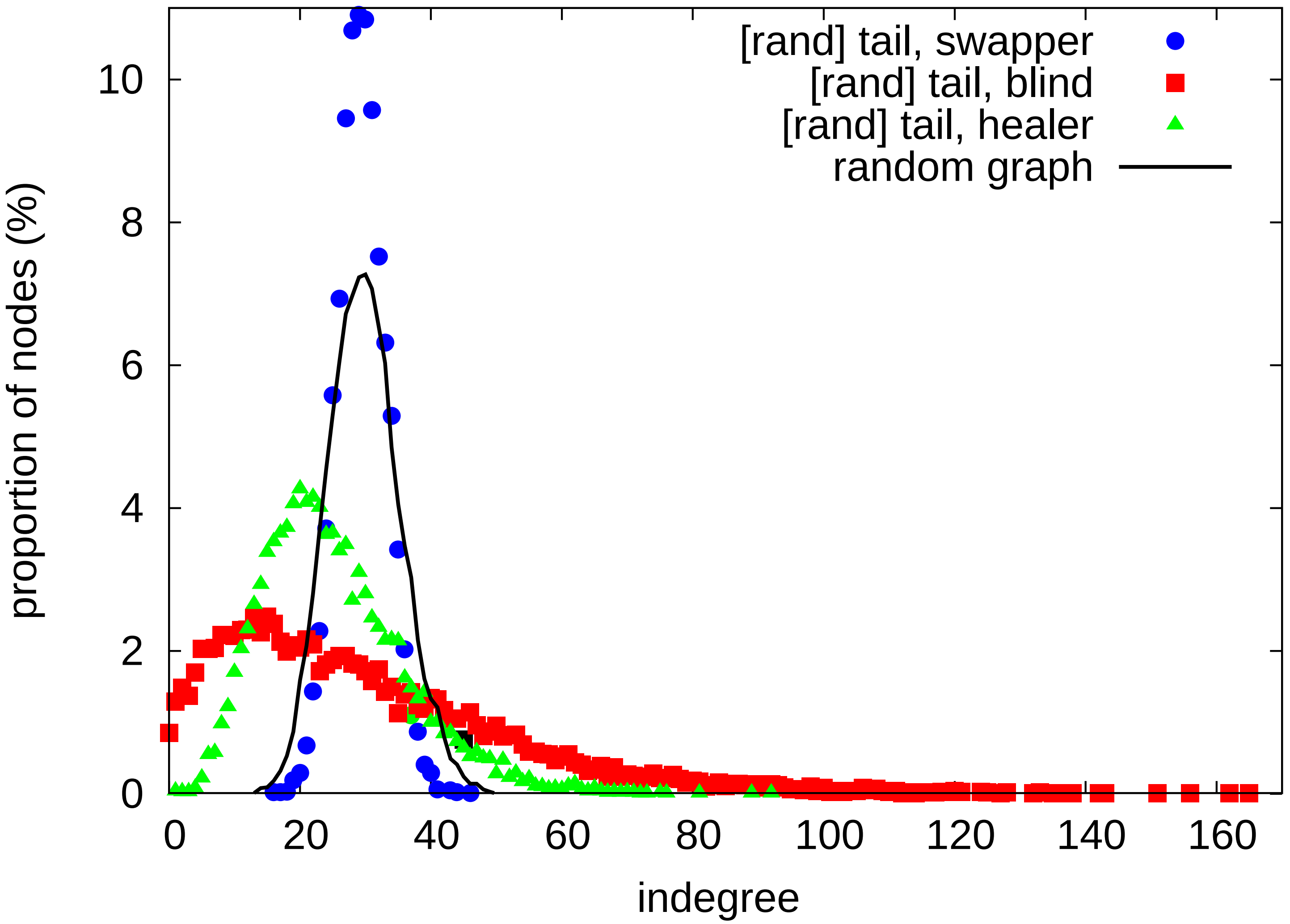
Observation: Pure pushing converges poorly and often leads to partitioned overlays in growing scenario.

Maximal indegree growing scenario



Note: From now on consider only pushpull protocols

Converged indegree distribution



Fluctuation of degree distribution (1/2)

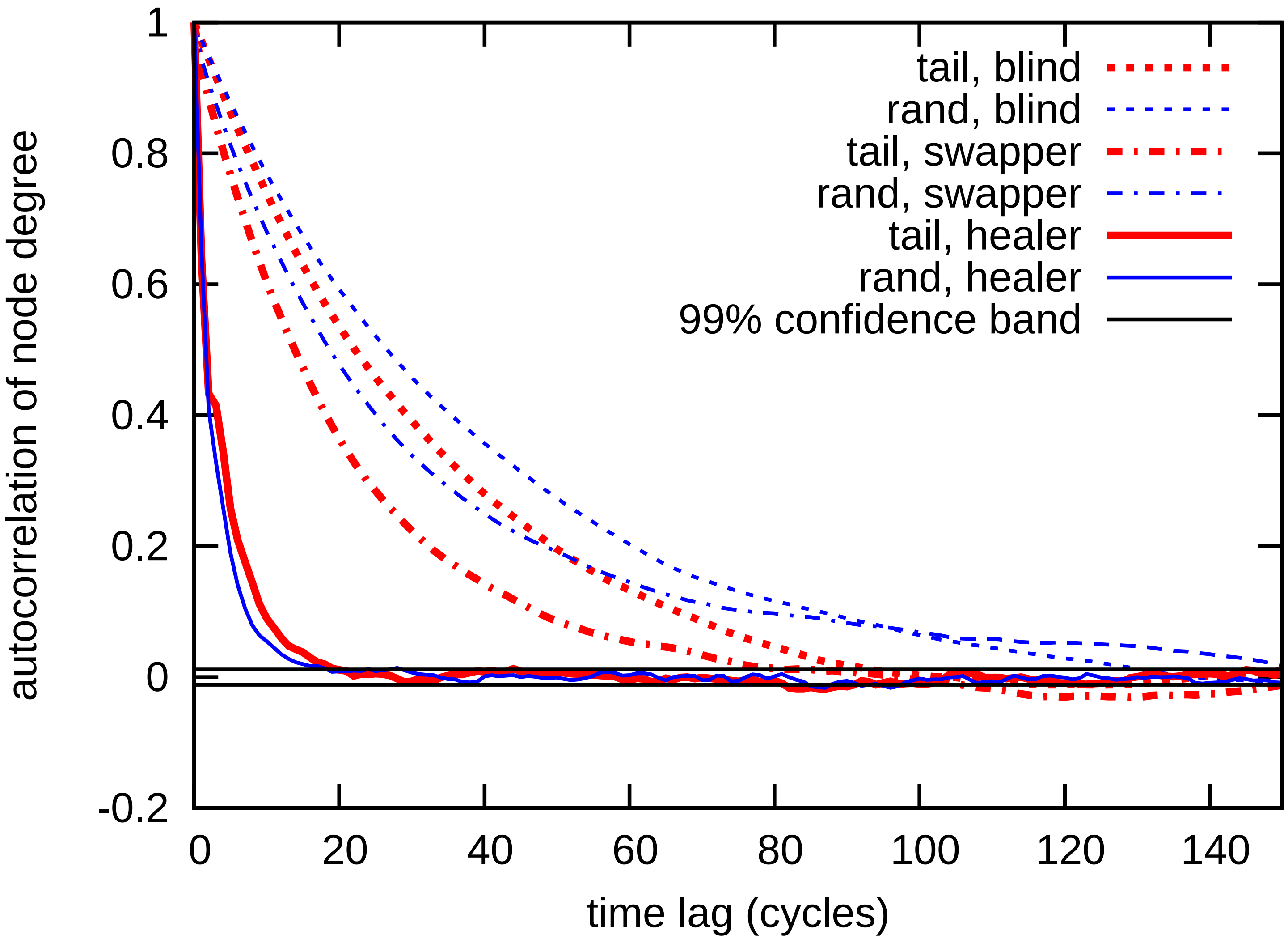
Observation: it turns out that the in-degree for each node changes over time. The question is how quickly.

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)

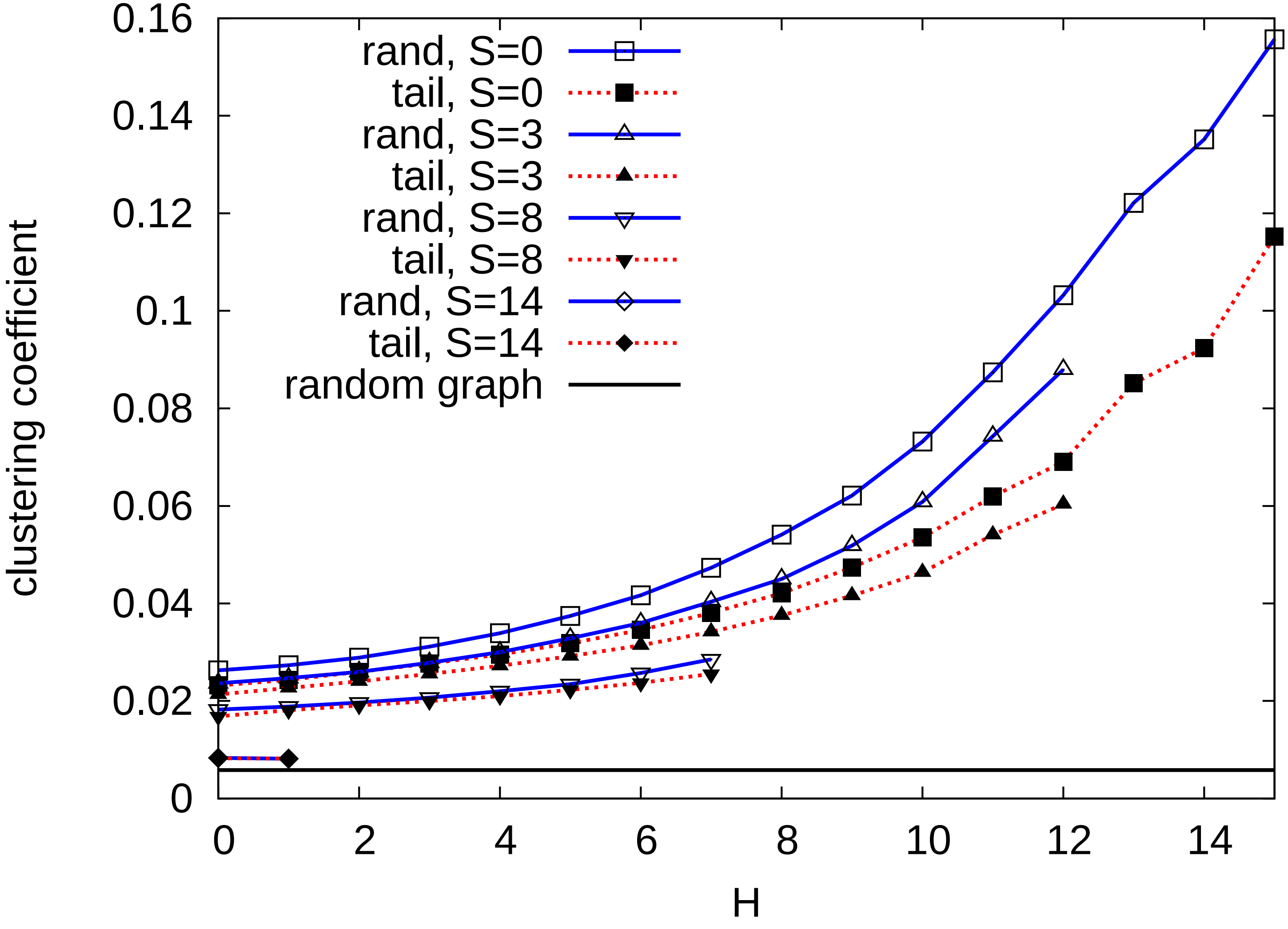
Note: Consider the undirected graph by dropping the direction.

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.

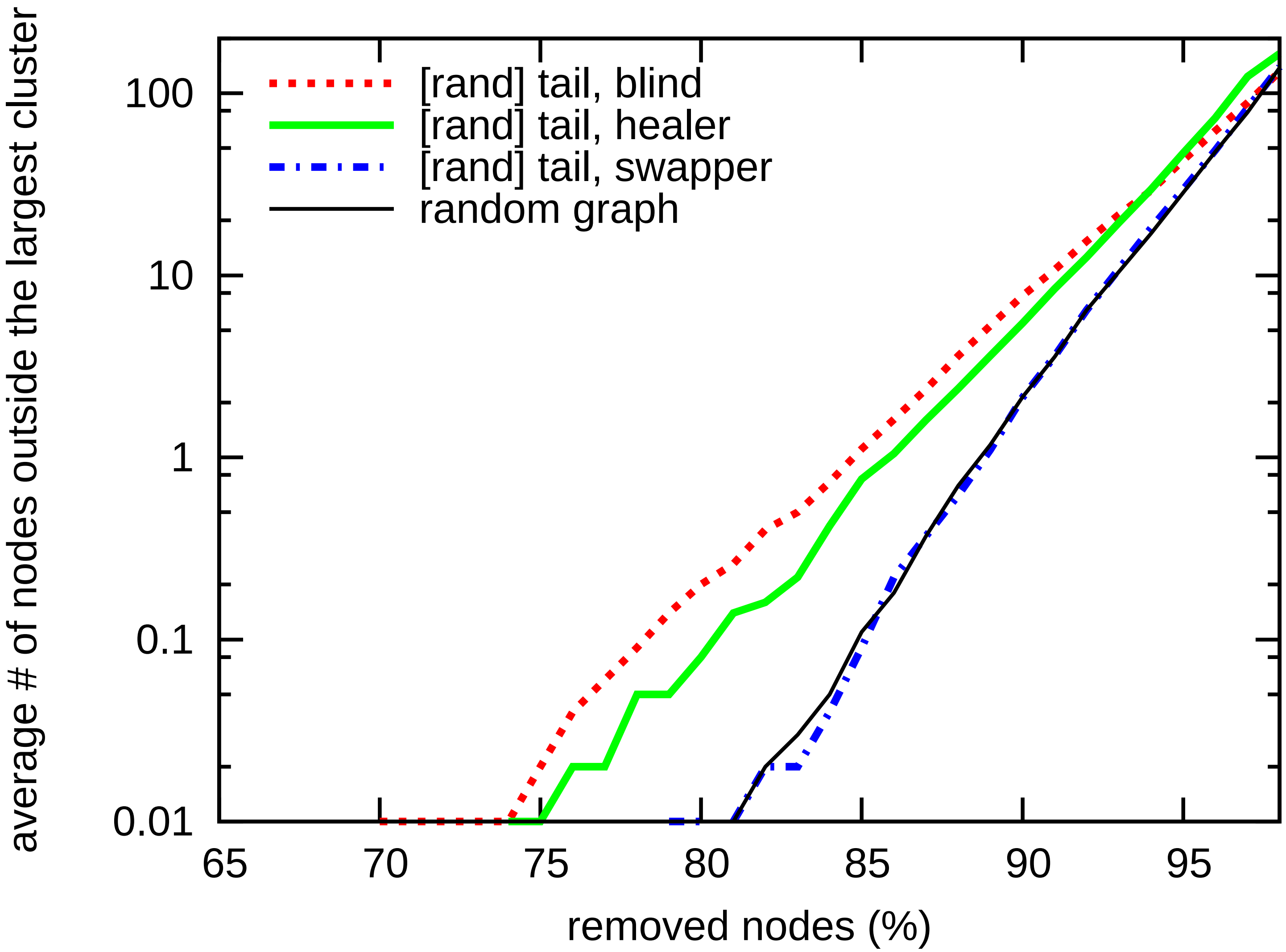
$$\gamma(X) = \frac{|E(\Gamma_X)|}{\binom{|V(\Gamma_X)|}{2}}$$

For a graph: take the average over all nodes.

Clustering coefficient (2/2)

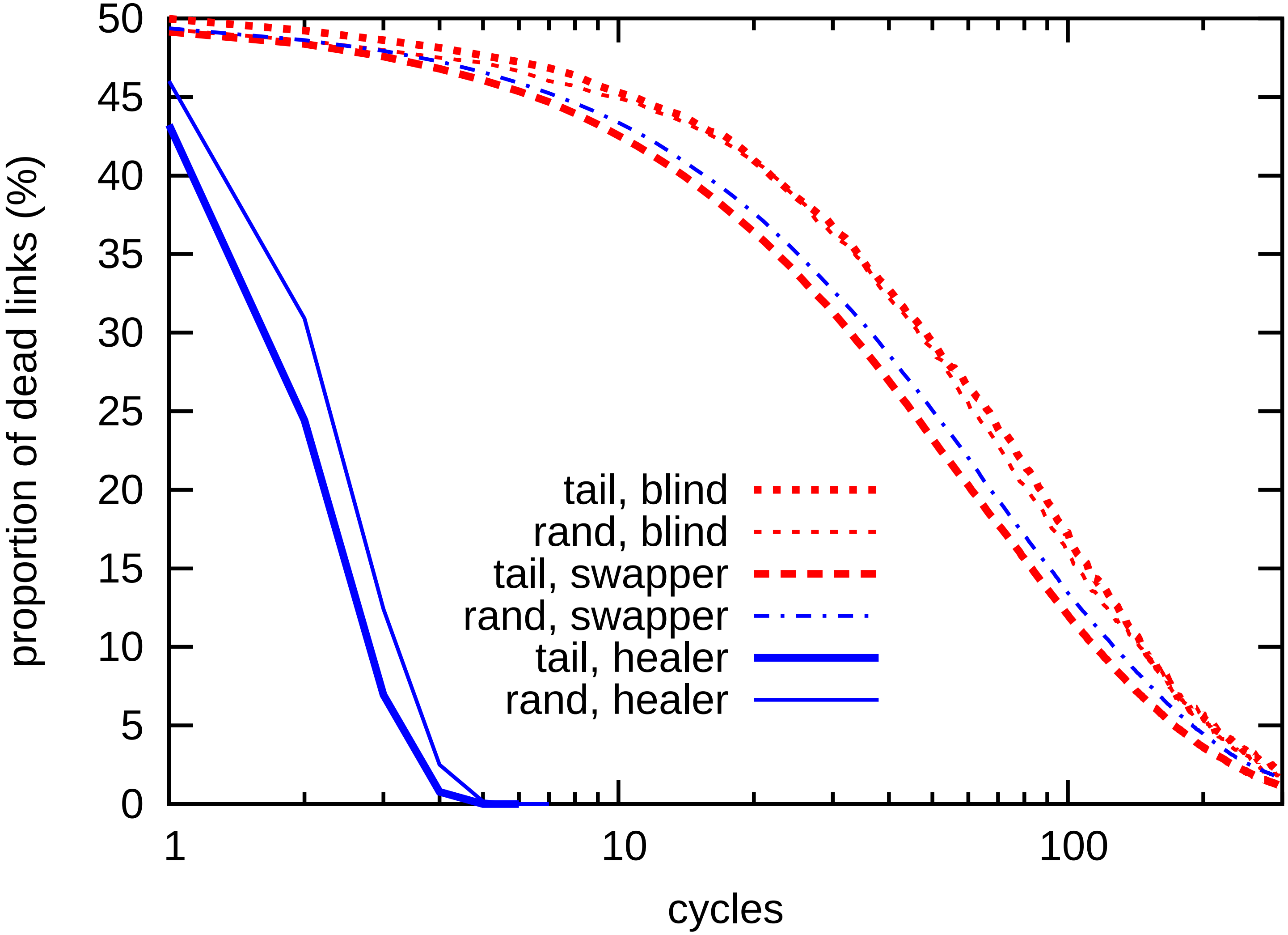


Catastrophic failure



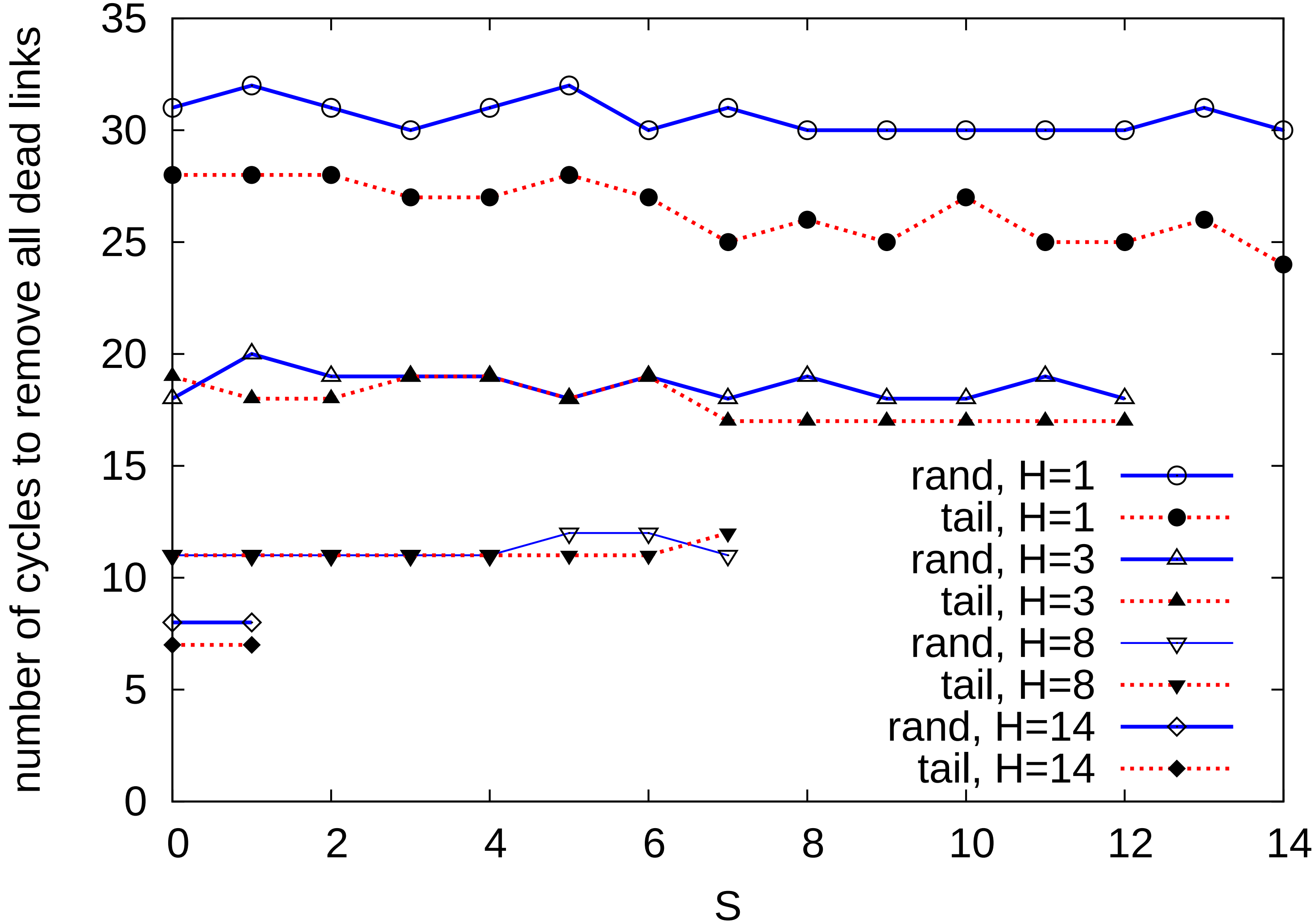
Scenario: After 300 cycles, remove large fraction of nodes.

Dead links (1/2)

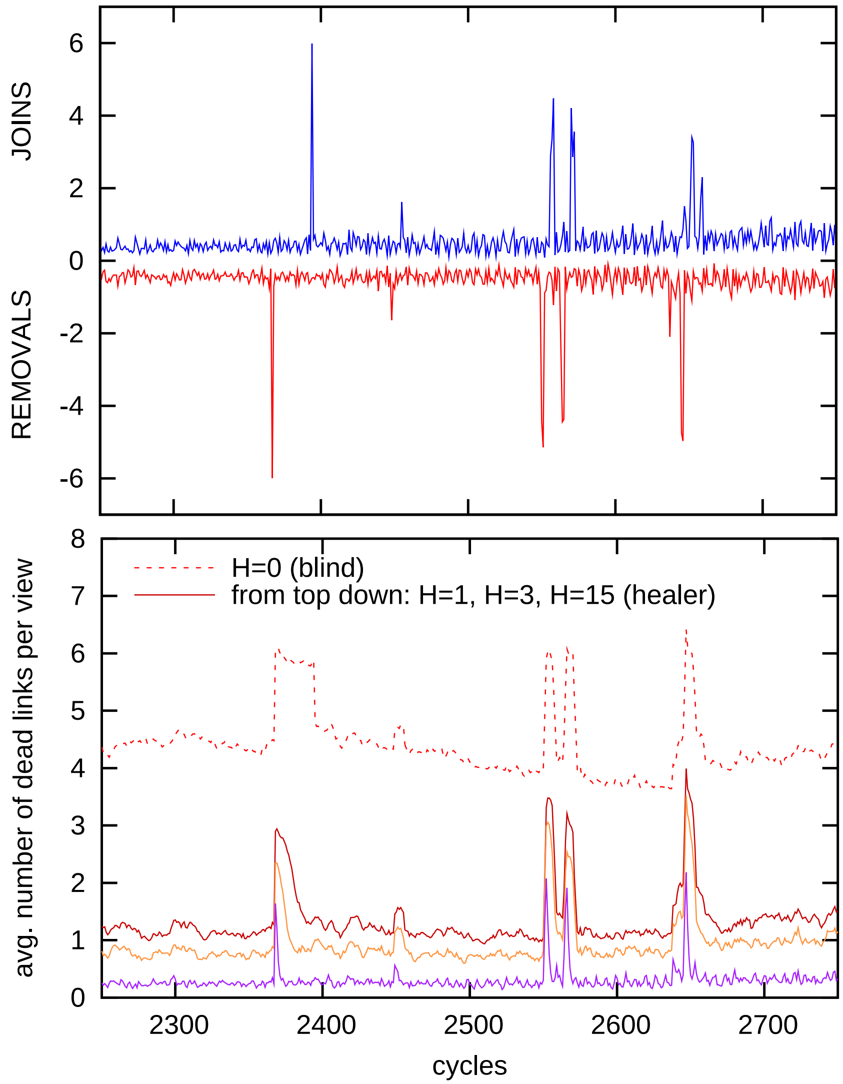


Scenario: After 300 cycles, remove 50% of nodes.

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 $\approx$ out-degree)
- Differences between protocols mainly affect the nonfunctional properties of applications

Challenge: Can we develop models that capture these nonfunctional properties?