Gossip-based protocols

Where we were

- Programmers face problems in building distributed applications
- Fundamental problems
  - Consensus
  - Atomic Broadcast / Multicast
  - Group membership
- Isis Toolkit [Birman, van Renesse et al.]

Where we are

Scalability

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 *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, BC, August 1987, pp. 1-12.

Setup

- Database replicated at thousands of sites
- Network is slightly unreliable
- Point-to-Point communication abstraction
- Crash failure model

Setup

- Database replicated at thousands of sites
- Network is slightly unreliable
- Point-to-Point communication abstraction
- Crash failure model
- Updates injected at a single site
- Updates must propagate to all other sites*
- Want contents of all replicas to be identical if updates stop and system left alone

Notation

- S is a set of n sites (replicas)
- K is a set of keys
- V is a set of values
- T is a set of timestamps (totally ordered)
- For any site s and key k,

  s.ValueOf : $K \rightarrow (V \times T)$

More notation

- Pretend there is only one key

  s.ValueOf $\in (V \times T)$
- Consistency definition

  $\forall$ *s*, *s'* $\in S$: *s*.ValueOf = *s'*.ValueOf
- To update the database with value *v* at time *t*

  s.ValueOf := (v, t)

Direct mail

Idea: If an update is injected at site *s*, then *s* mails the update to every other site in *S*

```
Upon an update at site s:
        for each s' ∈ S \ {s}   do
           send (Update, s.ValueOf) to s'
        endloop

Upon receiving (Update, (v,t)):
        if  s.ValueOf.t < t  then
           s.ValueOf := (v,t)
        endif
```

Weakness: send is not reliable
what if site crashes?

Anti-entropy

Idea: Every site regularly chooses another site at random and exchanges database contents with it to resolve differences.

```
Each server s periodically executes:
    for some s' ∈ S \ {s} do
        ResolveDifference(s,s')
    endloop
```

```
Push:
  ResolveDifference(s,s') {
    if  s.ValueOf.t  > s'.ValueOf.t   then
          s'.ValueOf := s.ValueOf
    endif
```

```
Pull:
  ResolveDifference(s,s') {
    if  s.ValueOf.t  < s'.ValueOf.t   then
          s.ValueOf := s'.ValueOf
    endif
```

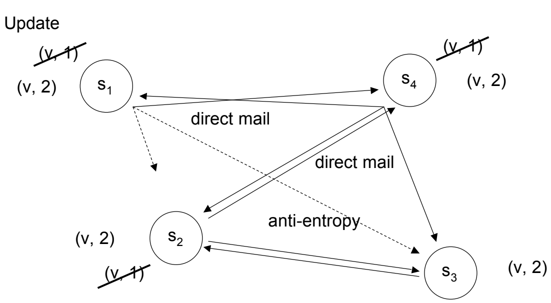
Push vs. pull analysis

Let p_i be the probability that a site still has not been updated by the i^{th} try at anti-entropy

For large values of n:

- Push: $p_{i+1} = p_i e^{-1}$
- Pull: $p_{i+1} = (p_i)^2$ Converges much faster for small p_i

Example using pull mechanism



Anti-entropy facts

- Guaranteed to eventually propagate update to everyone with probability 1
- Anti-entropy infects everyone in O(log n) for uniformly chosen sites
- Backup mechanism for direct mail
- Weakness: must go through entire database

Epidemic terminology

- Resilient to unreliable communication
- Anti-entropy is a simple epidemic
- Complex epidemics
  - Sites can become "cured"
  - Terminology: susceptible, infective, removed
  - Strengths: sites do not mail everyone and do not have to enumerate entire database
  - Weakness: some may be left susceptible

Rumor mongering (informal)

- All sites start out susceptible
- When a site *s* receives a new update, it becomes infective
- *s* periodically chooses another site *s'*

Rumor mongering (informal)

- All sites start out susceptible
- When a site *s* receives a new update, it becomes infective
- *s* periodically chooses another site *s'*
- If *s'* does not know the rumor, then it receives the update and also becomes infective
- If *s'* already knows the rumor, then *s* becomes removed with some probability

Rumor mongering protocol

```
For a site s:
   let L be a list of (initially empty) infective updates

   periodically:
        for some s' ∈ S \ {s}  do
            for each update u ∈ L
                send u to s'
                if s' already knows about u then
                   remove u  from  L  with probability 1/k
            endloop

   upon receiving new update u:
        insert u into L
```

Analysis of rumor mongering

i = fraction of infective sites
s = fraction of susceptible sites
r = fraction of removed sites

$$\frac{ds}{dt} = -si$$

$$\frac{di}{dt} = +si - \frac{1}{k}(1-s)i$$

$$\frac{di}{ds} = -\frac{k+1}{k} + \frac{1}{ks}$$

$$\int di = \int \left(-\frac{k+1}{k} + \frac{1}{ks} \right) ds$$

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

$$c = \frac{k+1}{k}$$

$$s = e^{-(k+1)(1-s)}$$

Rumor mongering facts

- Expected fraction of susceptible sites
  $s = e^{-(k+1)(1-s)}$
- Back up mongering with anti-entropy
- Mongering vs. direct mail
  - Redistribution
  - Consider case when half of sites receive update
  - Old rumors die fast

Death and its consequences

- Replace deleted item with a death certificate = (NIL, t_{now})
- Provided no further updates, a death certificate eventually “deletes” all copies of an item…but when?
- Problem: what if a single site is down?

Death certificates

- Death certificate contains two values
  - t – time of deletion
  - t_1 – threshold value, all servers discard death certificate after time $t + t_1$

Dormant death certificates

- Death certificate contains four values
  - R – set of sites that keep a dormant death certificate after $t + t_1$
  - t – time of deletion
  - t_1 – threshold value, all servers not in R discard death certificate after time $t + t_1$
  - t_2 – all servers discard the certificate after $t + t_2$

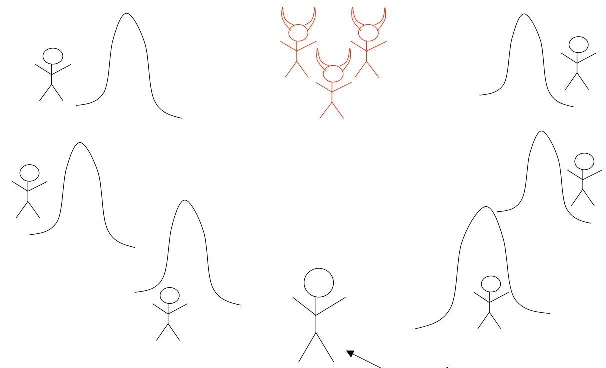
Dormant death certificates

- Death certificate contains five values
  - R – set of sites that keep a dormant death certificate after $t_a + t_1$
  - t – time of deletion
  - t_a – time of activation
  - t_1 – all servers not in R discard certificate after $t_a + t_1$
  - t_2 – all servers discard the certificate after $t_a + t_2$

Bimodal multicast

K.P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal Multicast. *ACM Transactions on Computer Systems*. 17(2): 41-88. May 1999

War games



Class I – Strong reliability

- Properties: Agreement, validity, termination, integrity
- Costly protocols
- Limited scalability
- Unpredictable performance under congestion
- Degraded throughput under transient failures (full buffers and flow control)

Class II – Best effort reliability

- "If a participating process discovers a failure, a reasonable effort is made to overcome it."
- Better scalability than Class I protocols
- Difficult to reason about systems without concrete guarantees

Bimodal multicast claims

- Scales well
- Provides predictable reliability and steady throughput under highly perturbed conditions
- Very small probability a few processes deliver
- High probability almost everyone delivers
- “Vanishingly small probability” in between

A problem to our solution

- Applications that need high throughput (frequent updates) and can tolerate small inconsistencies
- Examples: health care, stock trading, streaming data

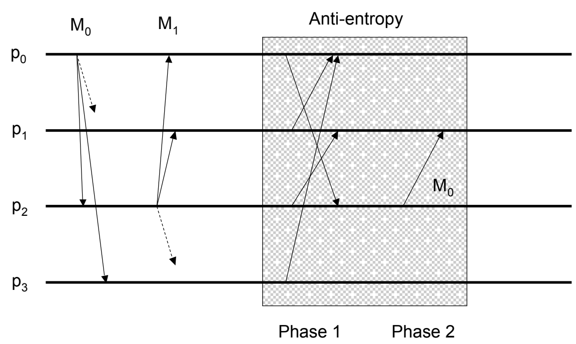
System assumptions

- At least 75% of healthy processes will respond to incoming messages within a known bound
- 75% of messages will get through the network
- Crash failures

Protocol details

- Consists of two subprotocols
- Unreliable multicast (i.e. – IP multicast)
- Anti-entropy that operates in rounds
  - Each round contains two phases
  - Phase 1: randomly choose another process and send message history to it
  - Phase 2: upon receiving a message history, solicit any messages you may be missing

Bimodal multicast example



Optimizations

- Reducing unnecessary communication
  - Service only recent solicitations
  - Retransmission limit
  - Most recent first transmission
- Random graphs for scalability
- Multicast some retransmissions

What's new about this?

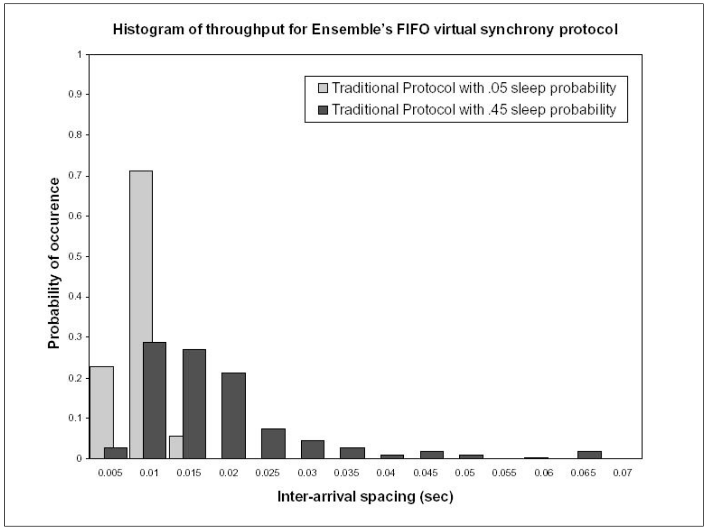
- To save space, keep a message for anti-entropy only for a fixed number of rounds
- Processes try to achieve a common ~~prefix~~ suffix
- If a process cannot recover a message, it gives up and notifies application

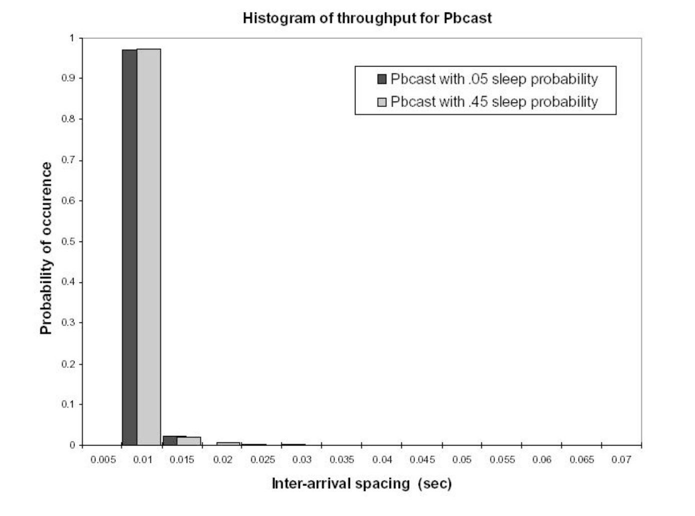
Recovery from delivery failures

- In previous protocols, a lagging process could drag the system down
- In bimodal multicast, a lagging process is effectively partitioned from the rest of the system
  - Do nothing
  - Maintain a few very large buffers
  - Employ a state transfer technique

Throughput results

- Eight processes running on an SP2
- Data rate = 75 7 KB multicasts per second
- Two cases
  - Sleep a process for 100 ms with .05 probability
  - Sleep a process for 100 ms with .45 probability





Lightweight Probabilistic Broadcast

P. Th. Eugster, R. Guerraoui, S.B. Handurukande, P. Kouznetsov, A.-M. Kermarrec. Lightweight Probabilistic Broadcast. ACM Transactions on Computer Systems (TOCS) 21(4):341-374, November 2003

Bimodal Multicast

- Scalability addressed with respect to reliability and throughput
- Processes knew entire membership set

Probabilistic Membership

- Each process has a *view* of ℓ processes it believes are members
- Each buffer *b* has at most $|b|_m$ elements
  i.e. - $|view|_m = \ell$
- Piggyback membership updates on each gossip message

Setup

- Set of processes $\{p_1, p_2, \ldots\}$ with distinct identifiers
- Unreliable point-to-point network
- Processes join and leave dynamically
- Two kinds of messages
  - Broadcast messages (events)
  - Gossip messages
    (events, membership updates)

Broadcast message

- id = uniquely identifies each message as well as the sender
- event

Gossip message

1. *events* =
   Set of all events received for the first time since the last outgoing gossip message
2. *eventIDs* =
   Set of all eventIDs for messages received by this process
3. *subs* = Set of processes "currently" joining
4. *unsubs* = Set of processes "currently" leaving

Process variables

Each process maintains 6 variables

1. *events* : set of events received for first time since last gossip
2. *eventIDs* : set of eventIDs received
3. *subs* : set of processes "currently" joining
4. *unsubs* : set of processes "currently" leaving
5. *view* : set of ℓ "current" members
6. *retrieveBuf* : set of eventIDs to retrieve

Broadcast reception

Upon receipt of broadcast (id, event)
events := events ∪ {event}
eventIDs := eventIDs ∪ {id}

Gossip transmission

```
periodically
    let gossip be a new gossip message
    gossip.events := events
    gossip.eventIDs := eventIDs
    gossip.subs := subs ∪ {p_i}
    gossip.unsubs := unsubs
    choose F random members t_1, t_2, … , t_F ∈ view
    for all j ∈ [1..F]  do
        send  gossip  to t_j
    events := ∅
```

upon reception of gossip

```
{phase 1: update unsubscriptions}
for all unsub ∈ gossip.unsubs  do
    view := view \ {unsub}
    subs := subs \ {unsub}
    unsubs := unsubs ∪ unsub
while |unsubs| > |unsubs|_m do
    remove random element from unsubs
{phase 2: update subscriptions}
for all newsub ∈ gossip.subs \ {p_i} do
    if newsub ∉ view   then
        view := view ∪ newsub
        subs := subs ∪ newsub
while |view| > ℓ   do
    target := random element in view
    view := view \ {target}
    subs := subs \ {target}
while |subs| > |subs|_m do
    remove random element from subs
```

```
{phase 3: update events}
for all e ∈ gossip.events  do
    if e.id ∉ eventIDs   then
        events := events ∪ {e}
        DELIVER(e)
        eventIDs := eventIDs ∪ {e.id}
for all id ∈ gossip.eventIDs  do
    if  id ∉ eventIDs  then
        retrieveBuf := retrieveBuf ∪ {id}
while |eventIDs| > |eventIDs|_m  do
    remove oldest element from eventIDs
while |events| > |events|_m  do
    remove random element from events
```

Subscribing & Unsubscribing

- To subscribe, a process p_i must know a process p_j already in the membership set and send (∅, ∅, ∅, { p_i }) to p_j
- To unsubscribe, a process p_i can inject its own unsubscription with a timestamp -or- just leave

Analytical evaluation

Assumptions

- *n* processes $\{p_1, p_2, \ldots, p_n\}$
- Gossip protocol runs in synchronized rounds
- Independent uniformly distributed views

Probability that a given process belongs to p_i's view

\# of possible views for p_i containing our given process

$$\frac{\binom{n-2}{len-1}}{\binom{n-1}{len}} = \frac{\frac{(n-2)!}{(n-len-1)!\,(len-1)!}}{\frac{(n-1)!}{(n-len-1)!\,(len)!}} = \frac{len}{n-1}$$

\# of possible views for p_i

$len = |view|_m = \ell$

Event propagation analysis

Consider an event *e*
Let s_r be the number of processes infected with *e* at round *r*
$s_0 = 1$

Goal: define a lower bound on probability that a given susceptible process p_1 is infected by a given gossip message from a given process p_2

$$(\text{prob. that } p_1 \text{ is in } p_2\text{'s view}) \times \begin{pmatrix}\text{prob. that } p_2 \text{ chooses} \\ p_1 \text{ to gossip with}\end{pmatrix} \times \begin{pmatrix} p_1 \text{ does not crash and} \\ p_2 \text{ does not crash and} \\ \text{message is not lost}\end{pmatrix}$$

$$p = \left(\frac{len}{n-1}\right) \times \left(\frac{F}{len}\right) \times k \quad = \frac{F}{n-1}k$$

Event propagation analysis

Let s_r be the number of processes infected with *e* at round *r*

$$p = \frac{F}{n-1}k$$

Probability that a given susceptible process is infected by a given gossip message

$$q = 1 - p$$

Probability that a given susceptible process is not infected by a given gossip message

\# of combinations of susceptible processes to infect

prob. that j-i susceptible processes will be infected in this round

prob. that n-j processes will remain susceptible

$$P(s_{r+1} = j \mid s_r = i) = \begin{cases} \binom{n-i}{j-i}\left(1-q^i\right)^{j-i} q^{i(n-j)} & , \ j \geq i \\ 0 & , \ j < i \end{cases}$$

Calculating distribution for s_r

Let s_r be the number of processes infected with *e* at round *r*

$$P(s_{r+1} = j \mid s_r = i) = \begin{cases} \binom{n-i}{j-i}\left(1-q^i\right)^{j-i} q^{i(n-j)} & , \ j \geq i \\ 0 & , \ j < i \end{cases} \qquad q = 1 - \frac{F}{n-1}k$$

$$P(s_0 = j) = \begin{cases} 1 & , j = 1 \\ 0 & , j = 0 \end{cases}$$

$$P(s_{r+1} = j) = \left\{ \sum_{i \leq j} P(s_r = i) P(s_{r+1} = j \mid s_r = i) \right)$$

Analysis?

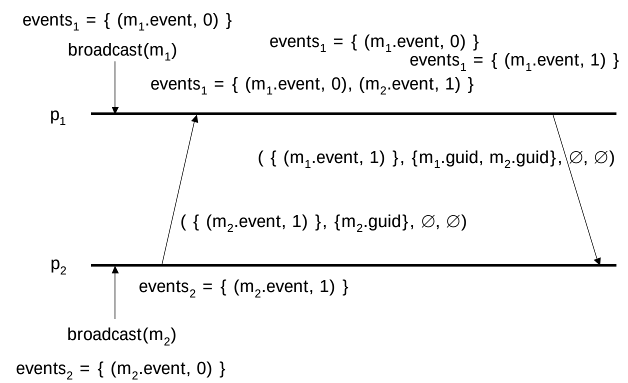
- $|\text{view}|_m$ has no expected effect on latency or reliability
- Mathematical guarantees only hold for independent uniformly distributed views
- Results show that their algorithm is “close” to perfect views, but also show that their reliability <u>does</u> depend on ℓ

Events buffer optimization

- Age-based message purging
- Idea
  - Estimate number of rounds event has been in system
  - If necessary, purge buffer of “older” events

Why can’t you use age-based for subs buffer, too?

Age-based purging example

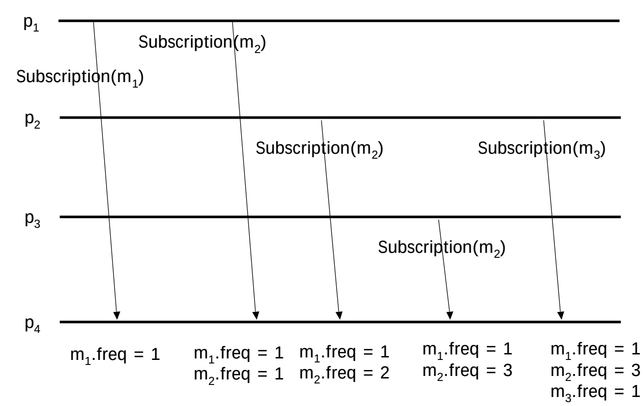


Subs buffer optimization

- Frequency-based message purging
- Idea
  - Tag each subscription with number of times it has been gossiped
  - If necessary, purge subscriptions that have been gossiped more

Why does this not work for events buffer?

Frequency-based purging



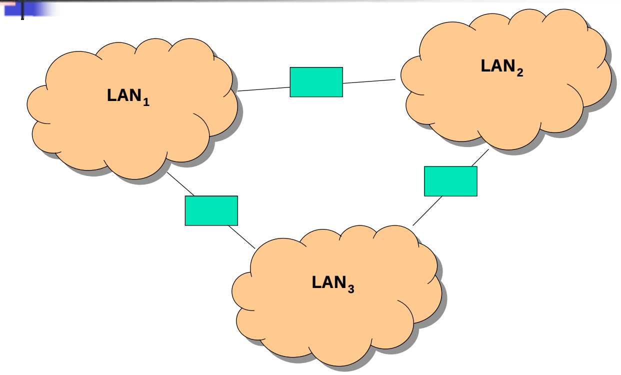
Directional Gossip

M.-J. Lin and K. Marzullo. Directional gossip: Gossip in a wide area network. In *European Dependable Computing Conference (EDCC)*, pp. 364-379, 1999.

Probabilistic broadcasts

- Initial unreliable multicast followed by subsequent gossip rounds
- Achieves high reliability
- Assumes an underlying point-to-point communication mechanism

Example WAN



Flooding

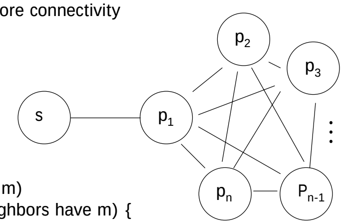
- Upon receiving a new message, a process forwards it to all neighbors the process believes have not received it yet
- Easy to implement
- High overhead in LAN

Marrying gossip and flooding

- Uniform gossip:
  - Less overhead in LAN than flooding
  - Only sends to random subset of processes
- Flooding
  - Less overhead on inter-network routers than uniform gossip
  - Only sends to neighbors

First marriage

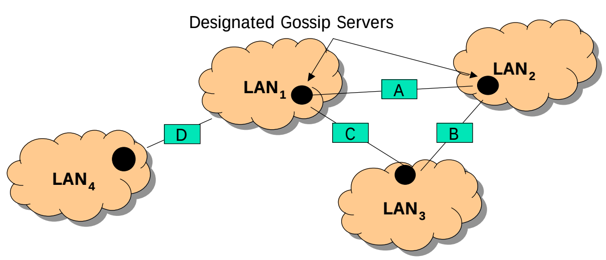
Idea: Forward messages to processes with less connectivity.
Gossip to processes with more connectivity




```
when (p receives a new message m)
while  (p believes not enough neighbors have m) {
        q = a neighbor process of p
        send m to q
}
```

Setting

Each gossip server maintains a set of adjacent routers
Two servers are neighbors if their adjacent routers sets have a common element
How to measure connectivity?



Link cut sets

- Given a connected graph G = ⟨V, E⟩, the *link cut set* is a set of edges E_{lcs} , such that G' = ⟨V, E \ E_{lcs}⟩ is disconnected
- The link cut set with respect to nodes *p* and *q* is a set of edges E_{pq}, such that removing all edges in E_{pq} will disconnect *p* and *q*

Weights and link cut sets

- *p* assigns a weight to *q* equal to the size of the smallest link cut set for *p* and *q*
- Menger's Theorem:

For any two nodes of a graph, the maximum number of link-disjoint paths between them equals the size of the minimum link cut set between them.

Inter-network router notation

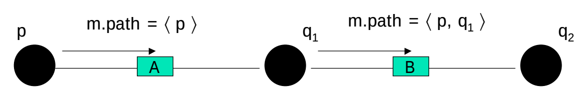
- A pair of servers (in different LANs) that are neighbors identifies an inter-network router
- A path of k servers $\langle p_1, p_2, \ldots p_k \rangle$ identifies a trajectory of k-1 inter-network routers
  - $INR(\langle p_1, p_2, \ldots p_k \rangle) = \langle r_1, r_2, \ldots r_{k-1} \rangle$

Paths in a gossip message *m*

- m.path is a path of servers $\langle p_1, p_2, \ldots p_k \rangle$ such that p_1 originated the message and sent it to neighbor p_2, who then sent it to p_3, etc.
- A path of servers $\langle p_1, p_2, \ldots p_k \rangle$ implicitly contains *n-1* timestamps
  - p_2*.timestamp* is the time that process p_2 received the message
  - Given $INR(\langle p_1, p_2, \ldots p_k \rangle) = \langle r_1, r_2, \ldots r_{k-1} \rangle$, r_i.timestamp = p_{i+1}.timestamp

Initiating a gossip message

for process *p*:
  let *m* be a new gossip message
  m.path = ⟨ *p* ⟩
  for each $q \in Neighbors_p$
    send *m* **to** *q*



```
Initially: ∀ q ∈ Neighbors_p : Trajectories(q) = { INR( ⟨ p, q ⟩ ) }

when p receives a gossip message m for the first time {
    int sent := 0
    for each q ∈ Neighbors_p
        if (q ∈ m.path) then
            UpdateTrajectories( Trajectories(q), INR (Trim(m.path,p) ) )
```

```
void UpdateTrajectories( reference to set of trajectories T, trajectory R ) {
    if  (all trajectories in T are disjoint with R)  then
        T := T ∪ { R }
    else  for each t ∈ T
            for each router r_1 ∈ t and router r_2 ∈ R
                if r_1 = r_2 then
                    r_1.timestamp := max(r_1.timestamp, r_2.timestamp)
}
```

```
Initially: ∀ q ∈ Neighbors_p : Trajectories(q) = { INR( ⟨ p, q ⟩ ) }

when p receives a gossip message m for the first time {
    int sent := 0
    for each q ∈ Neighbors_p
        if (q ∈ m.path) then
                UpdateTrajectories( Trajectories(q), INR (Trim(m.path,p) ) )
    for each q ∈ Neighbors_p
        remove old trajectories from Trajectories(q)
    for each q ∈ Neighbors_p
        if ( (q ∉ m.path) ∧ ( | Trajectories(q) | < k ) )  then
                m' := m
                append p to m'.path
                send m' to q
                sent := sent + 1
    let S be a random subset of Neighbors_p \ { q : q ∈ m.path },
        s.t. |S| = min( F – sent, | Neighbors_p \ { q : q ∈ m.path } | )
    for each  q ∈ S
        m' := m
        append p to m'.path
        send m' to q
}
```

Spatial Gossip [KKD01]

- Embed processes in D dimensional space
- let $d_{u,v}$ be distance between u and v
- u sends a gossip message to v with probability proportional to $\frac{1}{d_{u,v}^{pD}}$, $p \in [1,2]$

Expected time for message to reach nodes at distance d = $O(\log^{1+\varepsilon} d)$

DoS attacks [BKS04]?

System model

- Network is fully connected
- Asynchronous communication
- Insecure channels
- Loss rate on communication links is bounded and uniform
- Adversary can generate and insert messages into channels and snoop on messages

Drum protocol (informal)

- Use public key cryptography
- Pull mechanism
  - p sends ⟨history, $port_{rand}$⟩ to q on well-known port
  - q sends ⟨msg_{missed}⟩ to p on $port_{rand}$
- Push mechanism
  - p sends ⟨push-offer, $port_{rand}$⟩ to q on well-known port
  - q sends ⟨history, $port'_{rand}$⟩ to p on $port_{rand}$
  - p sends ⟨msg_{missed}⟩ to q on $port'_{rand}$
- Bound number of messages processed per port
- Discard all messages in buffers after a round