Distributed Algorithms
Epidemic Protocols: Beyond dissemination

Alberto Montresor
University of Trento, Italy

2014/09/15

Acknowledgments: S. Voulgaris
System model

- A dynamic collection of distributed nodes that want to participate in a common epidemic protocol
  - Node may join / leave
  - Node may crash at any time

- Communication:
  - To communicate with node $q$, node $p$ must know its address
  - Potentially high levels of message omissions
Motivation

In the original paper:
- Nodes have full view of the network
- Each node periodically “gossips” with a random node, out of the whole set

Why it was reasonable:
- (Almost) static network
- Relatively small size

Currently:
- Nodes have a partial view of the network
- The partial view is dynamic, reflecting nodes joining/leaving
- Each node periodically gossips with a random node, out of its partial view

What you get:
- Unlimited scalability
- Capable to deal with churn
Peer Sampling

Service specification

Peer sampling service

- **Input**: the distributed collection of nodes
- **Output**: a method `getPeer()` that returns a peer as a result of an independent uniform random sampling among the collection

The idea:

- Nodes gossip with their neighbors about... other neighbors!
- Old nodes are removed / new nodes are inserted
- Random **shuffling** of views
Service architecture

- Each node has a view containing $C$ neighbors
- Each node periodically contacts a neighbor
- They exchange their views

The neighbor descriptor of node $p$ contains
- The address needed to communicate with $p$
- Timestamp information about the age of the descriptor
- Additional information that may be needed by upper layers
A generic algorithm

Generic protocol executed by $p$:

upon initialization do

\[ \text{view} \leftarrow \text{descriptor(s) of nodes already in the system} \]

repeat every $\Delta$ time units

\[ \text{PROCESS } q \leftarrow \text{selectNeighbor(view)} \% \text{ Select a random neighbor} \]
\[ m \leftarrow \text{prepareRequest(view, } q) \]
\[ \text{send } \langle \text{REQUEST, } m, p \rangle \text{ to } q \]

upon receive $\langle \text{REQUEST, } m, q \rangle$ do

\[ m' \leftarrow \text{prepareReply(view, } q) \]
\[ \text{send } \langle \text{REPLY, } m', p \rangle \text{ to } q \]
\[ \text{view} \leftarrow \text{merge(view, } m, q) \]

upon receive $\langle \text{REPLY, } m, q \rangle$ do

\[ \text{view} \leftarrow \text{merge(view, } m, q) \]
NEWSCAST

- **selectNeighbor()**
  - Select one node at random from the local view

- **prepareRequest**(view, q), **prepareReply**(view, q)
  - Returns the entire view + a local descriptor with a fresh timestamp

- **merge**(view, m, q):
  - Returns the C freshest descriptors (w.r.t. timestamp) from the union of local view and message

### Relevant bibliography


Peer Sampling

NEWSCAST
NEWSCAST
1. Pick random peer from my view
1. Pick random peer from my view
1. Pick random peer from my view
2. Send each other view + own fresh link
1. Pick random peer from my view
2. Send each other view + own fresh link
1. Pick random peer from my view
2. Send each other view + own fresh link
NEWSCAST

1. Pick random peer from my view
2. Send each other view + own fresh link
3. Keep $c$ freshest links (remove own info, duplicates)
1. Pick random peer from my view
2. Send each other view + own fresh link
3. Keep $c$ freshest links (remove own info, duplicates)
1. Pick random peer from my view
2. Send each other view + own fresh link
3. Keep $c$ freshest link (remove own info, duplicates)
Cyclon

- **selectNeighbor()**:  
  - Select the oldest descriptor in the view
- **prepareRequest(view, q)**:  
  - Remove \( t - 1 \) random neighbors from the local view  
  - Return the \( t - 1 \) descriptors plus a fresh local one
- **prepareReply(view, q)**:  
  - Remove and return \( t \) fresh neighbors from the local view
- **merge(view, m, q)**:  
  - Merge the local view and the message  
  - Remove \( p \), keep freshes in case of duplicates  
  - Re-insert entries sent to \( q \) if space permits

Relevant bibliography


Cyclon

<table>
<thead>
<tr>
<th>ID &amp; Address</th>
<th>Time stamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue</td>
<td>9</td>
</tr>
<tr>
<td>Blue</td>
<td>12</td>
</tr>
<tr>
<td>Red</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID &amp; Address</th>
<th>Time stamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orange</td>
<td>7</td>
</tr>
<tr>
<td>Yellow</td>
<td>10</td>
</tr>
<tr>
<td>Orange</td>
<td>14</td>
</tr>
</tbody>
</table>
1. Pick **oldest** neighbor from my view
1. Pick **oldest** neighbor from my view
**Cyclon**

1. Pick **oldest** neighbor from my view
2. Exchange **some** neighbors
Cyclon

1. Pick **oldest** neighbor from my view
2. Exchange **some** neighbors

<table>
<thead>
<tr>
<th>ID &amp; Address</th>
<th>Time stamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>blue</td>
<td>9</td>
</tr>
<tr>
<td>red</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID &amp; Address</th>
<th>Time stamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>orange</td>
<td>7</td>
</tr>
<tr>
<td>yellow</td>
<td>10</td>
</tr>
<tr>
<td>orange</td>
<td>14</td>
</tr>
<tr>
<td>blue</td>
<td>12</td>
</tr>
</tbody>
</table>
1. Pick **oldest** neighbor from my view
2. Exchange **some** neighbors
1. Pick **oldest** neighbor from my view
2. Exchange **some** neighbors
1. Pick **oldest** neighbor from my view
2. Exchange **some** neighbors
1. Pick **oldest** neighbor from my view
2. Exchange **some** neighbors
Experimental evaluation

Simulation parameters

- $N = 100,000$ nodes
- $C = 20$ neighbors

We are measuring:

- Clustering coefficient
- Average path length
- Degree distribution
- Robustness to catastrophic failures
- Self-cleaning
Definition (Clustering coefficient)

The clustering coefficient of graph $G$ is the average over the local clustering coefficient of all nodes in the graph

$$CC(G) = \frac{1}{N} \sum_{i \in V} \frac{2|E_i|}{|V_i|(|V_i| - 1)}$$

For epidemic diffusion:
- high clustering coefficient means more redundant messages
Clustering coefficient

High clustering is bad for:
- Flooding: It results in many redundant messages
- Self-healing: Strongly connected cluster weakly connected to the rest of the network
**Brief recap**

**Definition (Average path length)**

The average path length of a connected graph $G = (V, E)$ is defined as:

$$\ell(G) = \frac{1}{|V|(|V| - 1)} \sum_{i,j \in V, i \neq j} d(i, j)$$

For epidemic diffusion:

- high average path length means a longer number of rounds to reach everybody else
Average path length

![Graph showing the average path length over cycles for different network configurations: Newscast lattice (green), Newscast star (green with crosses), Cyclon lattice (red), Cyclon star (red with crosses), and random graph (gray).]
Degree distribution

![Degree distribution graph]

- Affects:
  - Robustness (shows weakly connected nodes)
  - Load balancing
  - Way epidemics spread
Robustness

![Graph showing the number of clusters against the percentage of nodes removed for different models: Newscast, Cyclon, and random graph. The y-axis represents the number of clusters, and the x-axis represents the percentage of nodes removed. The graph illustrates how robust different models are to node removal.]
Self-cleaning behavior

The graph shows the number of dead nodes still remembered over cycles since 50,000 nodes disconnected. The two lines represent different systems: Newscast and Cyclon. The graph illustrates how quickly each system is able to maintain the integrity of the network by removing dead nodes as they are disconnected.
Hub attack

- **Input**: a set of colluding, malicious nodes
- **How**: they only gossip their IDs
- **Output**:
  - all views become “polluted”
  - non-malicious nodes are cut-off from each other
  - malicious nodes may leave the network, leaving the network disconnected with no way to recover
Secure peer sampling

**Algorithm**

- Maintain multiple independent views in each node
- During a gossip exchange measure similarity of exchanged views
- With probability equal to proportion of identical nodes in two views, reject the gossip and blacklist the node
- Apply an aging policy to the black list
- When supplying a random peer, select the current “best” view

**Relevant bibliography**

Secure peer sampling

- 1000 nodes
- 20 malicious nodes
Introduction

Topology bootstrap

Informal definition: building an overlay topology from the ground up as quickly and efficiently as possible

- Do not confuse with node bootstrap
  - Placing a single node in the right place in the topology

- Do not confuse with topology maintenance
  - Stabilization (routing table cleanup)
  - Can be used later to optimize the topology
State of the art

- Current DHT protocols do not support bootstrapping:
  - They assume an already formed network
  - Even supposing the network exists, they work “unless a tremendous number of nodes joins the system” [Chord]
  - We could envision a “join orchestration approach”, but it would require a linear time

Relevant bibliography

T-Man

- T-man is a generic protocol for topology formation
- Topologies are expressed through ranking functions:
  - “which are my preferred neighbors?”

Examples

- Rings, tori, trees, DHTs, etc.
- Distributed sorting
- Semantic proximity
- Latency proximity
- Etc.
The T-Man algorithm

Protocol executed by $p$:

**upon initialization do**

- $\text{view} \leftarrow \text{a collection of random descriptors}$

**repeat** every $\Delta$ time units

- $\text{PROCESS } q \leftarrow \text{selectNeighbor}(\text{view})$ % Select a neighbor
- $m \leftarrow \text{prepareRequest}(\text{view}, q)$
- $\text{send } \langle \text{REQUEST}, m, p \rangle \text{ to } q$

**upon receive** $\langle \text{REQUEST}, m, q \rangle$ **do**

- $m' \leftarrow \text{prepareReply}(\text{view}, q)$
- $\text{send } \langle \text{REPLY}, m', p \rangle \text{ to } q$
- $\text{view} \leftarrow \text{merge}(\text{view}, m, q)$

**upon receive** $\langle \text{REPLY}, m, q \rangle$ **do**

- $\text{view} \leftarrow \text{merge}(\text{view}, m, q)$
Node descriptors and ranking functions

Node descriptor

- IP address + timestamp, as in NEWSCAST/CYCLON
- One or more attributes of the node, used for ranking:
  - The ID of a node in a DHT; or
  - A semantic description of the node; or
  - The geographic location of the node

Ranking function

- A ranking function $\text{rank}(U, p)$ ranks a collection $U$ of nodes based on $p$’s order of preference.
- Alternatively, a ranking function $\text{rank}(U, p, r)$ returns $r$ nodes from $U$ that are the most preferred by $p$. 
Ranking functions

- The ranking function may be based on a distance over a space
  - **Space**: set of possible attribute values
  - **Distance**: a metric $d(x, y)$ over the space

- Given a collection $U$ of nodes
  - rank them in increasing distance order, or
  - return the $r$ closest node based on the notion of distance

**Example (Line/ring)**
- **Space**: $[0, 1)$
- **Distance** over the line/ring:
  \[
  d(a, b) = |a - b| \\
  d(a, b) = \min\{ |a - b|, 1 - |a - b| \}
  \]

**Example (Grid or torus – Manhattan Distance)**
- **Space**: $[0, 1) \cdot [0, 1)$
- **Distance**:
  \[
  d(a, b) = |a_x - b_x| + |a_y - b_y|
  \]
How to express the ranking function

**Generic line/ring**: returns
- the \( r \) nodes closest to \( p \) w.r.t. the distance function

**Sorted line/ring**: returns
- the \( r/2 \) nodes “\( \geq p \)” closest to \( p \) w.r.t. the distance function
- the \( r/2 \) nodes “\( \leq p \)” closest to \( p \) w.r.t. the distance function
The T-Man algorithm

- **selectNeighbor()**
  - Select a random node from $\text{rank}(\text{view}_p, p, r)$, i.e. from the top-$r$ nodes in $p$’s local view as ranked by $p$

- **prepareRequest(view, q), prepareReply(view, q):**
  - Returns $\text{rank}(\text{view}_p, q, r)$, i.e. the top-$r$ nodes in $p$’s local view as ranked by $q$

- **merge(view, m, q):**
  - merge the local view and the message
T-Man: How it works

Diagram showing the process of constructing a topology with labels p and q.
T-Man: How it works
T-Man: The movie

http://www.disi.unitn.it/~montreso/ds/handouts/ring-fingers-20.gif
How to start and stop

In the previous animation:

- Nodes starts simultaneously
- Convergence is measured globally

In reality:

- We must start the protocol at all nodes
  - Broadcasting, using the random topology obtained through a peer sampling services
- We must stop the protocol
  - Local condition: when a node does not observe any view change for a predefined period, it stops
  - Idle: number of cycles without variations
Scalability, with different starting protocols

![Graph showing convergence time vs network size for Anti-Entropy (Push), Anti-Entropy (Push-Pull), Flooding, and Synchronous start protocols.](image)

- **Convergence Time (s)**
- **Network Size**
- **Anti-Entropy (Push)**
- **Anti-Entropy (Push-Pull)**
- **Flooding**
- **Synchronous start**
Bootstrap protocol: Starting and stopping
Stopping condition: different values for idle

![Graph showing time versus idle time for different network sizes](image)
Communication costs

![Graph showing communication costs](image)

- size = $2^{16}$
- size = $2^{13}$
- size = $2^{10}$

Messages vs. Idle cycles
Parameter tuning: message size

![Graph showing the relationship between termination time and message size]
Robustness to failures

![Graph showing robustness to failures](image)
Robustness to message losses
T-Chord

- The general idea:
  - Node descriptor contains node ID in a $[0..2^t)$ space
  - Nodes are sorted over the ring defined by IDs
  - Final output is the Chord ring
  - As by-product, many other nodes are discovered

- Example:
  - $t = 32, N = 2^{14}, r = 20$
T-Chord

- Run T-Man until the ring is formed
- Use the closest neighbors to select successors
- Use the entire view to fill the finger table
  - For each node $p$:
  - For each $i = 0 \ldots m - 1$ ($m$ number of bits)
  - search in the local view the closest node $q$ s.t.:

$$q \in [(p + 2^i) \mod 2^m, (p + 2^{i+1} - 1) \mod 2^m]$$
T-Chord-Prox

- Run T-Man until the ring is formed
- Use the closest neighbors to select successors
- Use the entire view to fill the finger table
  - For each node $p$
  - For each $i = 0 \ldots m - 1$ ($m$ number of bits)
  - search in the local view the closest node (in terms of latency) $q$ s.t.:

\[
q \in [(p + 2^i) \mod 2^m, (p + 2^{i+1} - 1) \mod 2^m]
\]
Routing through T-Chord

![Graph showing the relationship between network size and hops for T-Chord and Chord. The graph indicates an upward trend with increasing network size.]
Robustness

The graph illustrates the performance of different overlay networks under robustness testing. The x-axis represents the percentage of crashed nodes, while the y-axis shows the percentage of routing failures. The overlay networks compared are Chord (crash), T-Chord (crash), T-Chord (churn), and T-Chord-Prox (churn).

The data points show a clear trend where the percentage of routing failures increases as the percentage of crashed nodes increases. This trend is consistent across all four overlay networks, indicating that their robustness decreases with a higher number of crashed nodes.
Aggregation

Definition (Aggregation)

The collective name of a set of functions that provide statistical information about a system

- Useful in large-scale distributed systems:
  - The average load of nodes in a computing grid
  - The sum of free space in a distributed storage
  - The total number of nodes in a P2P system

- Wanted: a decentralized, robust, proactive solution

Relevant bibliography


A generic aggregation algorithm

Protocol executed by $p$:

upon initialization do
\[ state \leftarrow \text{some initial state} \]

repeat every $\Delta$ time units
\[
\text{PROCESS } q \leftarrow \text{getPeer()} \]
\[ s_p \leftarrow \text{prepareRequest}(state, q) \]
\[ \text{send } \langle \text{REQUEST}, s_p, p \rangle \text{ to } q \]

upon receive $\langle \text{REQUEST}, s_q, q \rangle$ do
\[
\text{send } \langle \text{REPLY}, s_p, p \rangle \text{ to } q \]
\[ state \leftarrow \text{mergeRequest}(state, s_q, q) \]

upon receive $\langle \text{REPLY}, s_q, q \rangle$ do
\[ state \leftarrow \text{mergeReply}(state, s_q, q) \]
Avgaggregation

- Local state maintained by nodes:
  - initialized with a real number representing the quantity to be averaged
  - contains the current estimate of the average
- Function `getPeer()`:
  - Based on peer sampling service
  - Returns a random peer from the entire collection of nodes
- Function `prepareRequest(state, q)`, `prepareReply(state, s_q, q)`:
  - Returns `state`
- Function `mergeRequest(state, s_q, q)`, `mergeReply(state, s_q, q)`:
  - Returns $\frac{state + s_q}{2}$
Example
Example

\[(10+2)/2 = 6\]
Example
Example

\[(16+4)/2=10\]
Example
Example

\[(6+8)/2=7\]
Average aggregation

The exchange of messages is not atomic, so pairwise averaging is better described in this way:

- Function `prepareRequest(state, q)`
  - Returns `state`

- Function `prepareReply(state, s_q, q)`:
  - Returns `state - \frac{state + s_q}{2}`

- Function `mergeRequest(state, s_q, q)`:
  - Returns `state - \left( state - \frac{state + s_q}{2} \right) = \frac{state + s_q}{2}`

- Function `mergeReply(state, d, q)`:
  - Returns `state + d`
Convergence

- If the graph is connected, each node converges to the average of the original values.

- After each exchange:
  - Average does not change
  - Variance is reduced

- Different from load balancing due to lack of constraints
Convergence

![Graph showing convergence of estimated average cycles over cycles]
Other problems

- **Average**
  
  \[ \text{merge}(a, b) \text{ returns } (a + b)/2 \]

- **Geometric mean**
  
  \[ \text{merge}(a, b) \text{ returns } \sqrt{ab} \]

- **Min/max**
  
  \[ \text{merge}(a, b) \text{ returns } \min(a, b) \text{ or } \max(a, b) \]

- **Sum**
  
  \[ \text{Average} \cdot \text{Count} \]

- **Product**
  
  \[ \text{Geometric} \cdot \text{Count} \]

- **Variance**
  
  \[ \text{compute } \overline{a^2} - \overline{a}^2 \]
Counting

Counting protocol
- Init: one node starts with 1, all the others with 0
- Expected average: $1/N$

- Problem: how to select that “one node”? 
- Concurrent instances of the counting protocol
- Each instance is lead by a different node
- Messages are tagged with a unique identifier
- Nodes participate in all instances
- Each node acts as leader with probability $P = c/N_E$
Questions

- How fast is convergence on different topologies?
  - Analytical results for fully connected graphs: exponential convergence
  - Random, NEWSCAST topologies: similar results

- Can we solve other problems
  - Counting, sum, variance estimation

- What are the effects of failures
  - Link failures: not critical
  - Crashes/message omissions can ruin convergence
  - But we have a solution for that
Analytical results

- Consider a “centralized translation” of the aggregation protocol
- $N$ values to be aggregated are stored in vector $A$
- An infinite number of rounds is executed
- At each round, the average operation is run on $N$ pairs of values

---

Aggregation

repeat forever

```plaintext
for i ← 1 to N do
    (p, q) ← getPair()
```
Some definitions

- We measure the speed of convergence by measuring population mean and variance at round \( i \):

\[
\sigma_i^2 = \frac{1}{N} \sum_{k=1}^{N} (A_i[k] - \mu_i)^2 \quad \text{where} \quad \mu_i = \frac{1}{N} \sum_{k=1}^{N} A_i[k]
\]

- Elementary variance reduction step:

\[
\rho_i = \frac{\sigma_{i+1}^2}{\sigma_i^2}
\]

- Variable \( \Phi_k \): the number of times that node \( k \) has been selected from \( \text{getPair()} \)
Analytical results

Theorem (Convergence)

If

1. The indexes returned by each call to getPair() are uncorrelated;
2. the random variables $\Phi_k$ are identically distributed;
3. $\Phi$ is a random variable with this common distribution;
4. after $(p, q)$ is returned by getPair(), the number of times $p$ and $q$ will be selected by the remaining calls to getPair() has identical distribution

then:

$$E[\sigma^2_{i+1}] = E[2^{-\Phi}]\sigma^2_i$$
Analytical results

- **Optimal case:** $E(2^{-\Phi}) = E(2^{-2}) = 1/4$
  - `getPair()` implements perfect matching
  - no corresponding distributed protocol

- **Random case:** $E(2^{-\Phi}) = 1/e$
  - `getPair()` implements random global sampling
  - A local corresponding protocol exists

- **Aggregation protocol:** $E(2^{-\Phi}) = 1/(2\sqrt{e}) \approx 0.3032\ldots$
  - Scalability: results independent of $N$
  - Efficiency: convergence is very fast
Convergence factor

![Graph showing convergence factor over cycles for different pair selection methods.](graph.png)

- **getPair_rand, complete**
- **getPair_rand, 20-reg. random**
- **getPair_seq, complete**
- **getPair_seq, 20-reg. random**
Convergence factor

![Convergence factor graph]

- **getPair_rand, complete**
- **getPair_rand, 20-reg. random**
- **getPair_seq, complete**
- **getPair_seq, 20-reg. random**

The graph shows the variance reduction in relation to network size for different methods of pairing within a network.
Different topologies

- Theoretical results are based on the assumption that the underlying overlay is sufficiently random

- What about other topologies?
  - Our aggregation scheme can be applied to generic connected topologies
  - Small-world, scale-free, newscast, random, complete
  - Convergence factor depends on randomness
Different topologies
Different topologies

![Graph showing variance reduction over cycles for different topologies: W-S (β=0.00), W-S (β=0.25), W-S (β=0.50), W-S (β=0.75), Newscast, Scale-free, Random, Complete. The graph illustrates the rate of variance reduction across cycles for each topology, with distinct line styles and markers for each category.]
Adaptivity

- The generic protocol is not adaptive
  - Dynamism of the network
  - Variability of values

- Periodical restarting mechanism
  - At each node:
    ★ The protocol is terminated
    ★ The current estimate is returned as the aggregation output
    ★ The current values are used to re-initialize the estimates
    ★ Aggregation starts again with fresh initial values
Dynamic membership

- **Termination**
  - Run protocol for a predefined number of cycles $\lambda$
  - $\lambda$ depends on:
    - required accuracy of the output
    - the convergence factor that can be achieved

- **Restarting**
  - Divide run in consecutive epochs of length $\Delta$
  - Start a new instance of the protocol in each epoch

- **Node joining**
  - Receives from a node already in the network:
    - Next epoch identifier
    - The time until the start of the next epoch
  - To guarantee convergence:
    - Joining node is not allowed to participate in the current epoch
Synchronization

- The protocol described so far:
  - Assumes synchronized epochs and cycles
  - Requires synchronized clocks / communication

- This is not realistic:
  - Clocks may drift
  - Communication incurs unpredictable delays

- Complete synchronization is not needed
  - It is sufficient that the time between the first/last node starting to participate in an epoch is bounded
Cost analysis

- If the overlay is sufficiently random:
  - exchanges per round $= 1 + \Phi$, where $\Phi$ has Poisson distribution with average 1

- Round length $\delta$ defines trade-off between overhead and convergence
  - $\delta = 1s$
  - Overhead $= 2 \cdot 2 \cdot (28 + 4) = 128B/S$

- N. of rounds $\lambda$ defines trade-off between accuracy and convergence:
  - $E(\sigma^2_\lambda)/E(\sigma^2_0) = \rho^\lambda$
  - If $\epsilon$ the desired accuracy, then $\lambda \geq \log_\rho \epsilon$
Evaluation

- Underlying topology is based on Newscast
- The count protocol is used
  - More sensitive to failures
  - Some parameters:
- Some parameters:
  - Network size is 100,000
  - Partial view size in Newscast is 30
  - Epoch length is 30 cycles
  - Number of experiments is 50
Sudden death

![Graph showing estimated size vs cycle for experiments. The x-axis represents cycle, ranging from 0 to 20, and the y-axis represents estimated size per 10^5, ranging from 0.5 to 4.5. The graph includes multiple data points indicating the trend of estimated size increasing as cycle progresses.]
Churn

Experiments

Estimated Size (\times 10^5)

Nodes Substituted per Cycle

Alberto Montresor (UniTN)
Gnutella trace

[Graph showing network size and epoch with estimated and actual size lines]

Alberto Montresor (UniTN)
Link failure probability
Message losses

![Graph showing estimated size vs fraction of messages lost](image-url)
Multiple executions

To improve accuracy in the case of failures:
- Multiple concurrent instances of the protocol may be run
- Median value taken as result

Simulations
- Variable number of instances
- With node failures: 1000 nodes substituted per cycle
- With message omissions 20% of messages lost
Multiple executions - churn
Multiple executions - churn
Planet-lab

- PlanetLab is a global research network that supports the development of new network services
- PlanetLab currently consists of 1075 nodes at 525 sites
Planet-lab

![Graph showing network size vs epoch for estimated and actual sizes.](image-url)

- Estimated size
- Actual size

Network size vs Epoch plot showing fluctuations in network size over epochs.
Most frequent items

System model

- A collection \( \text{PROCESS} \) of nodes of size \( N = |\text{PROCESS}| \)
- Each node \( p \in \text{PROCESS} \) stores a local multiset \( I_p \) of items taken from a universe \( \mathcal{I} \) of size \( m = |\mathcal{I}| \)

Problem statement

Identify the \( k \) most frequent items, i.e. the \( k \) distinct items that appear more frequently in the distributed multiset given by \( I = \bigcup_{p \in \text{PROCESS}} I_p \)

Relevant bibliography

To appear
**Formal model**

**Global frequency**

Global absolute frequency: \( F(i) = \sum_{p \in \text{PROCESS}} F_p(i) \)

Global relative frequency: \( \hat{F}(i) = F(i)/M \)

Note: \( M = |I| \)

**Most frequent items**

Let \( j \) be the \( k \)-th item in the sequence of all items ordered by decreasing global frequency. We want to identify the top-\( k \) most frequent items:

\[ MF = \{ i : F(i) \geq F(j) \} \]
Formal model

Absolutely frequent items
To identify absolutely frequent items ($AF$) whose global absolute frequency is larger than the absolute threshold $f$:

$$AF = \{ i : F(i) \geq f \}$$

Relatively frequent items
to identify relatively frequent items ($RF$) whose global relative frequency is larger than the relative threshold $\phi$:

$$RF = \{ i : \hat{F}(i) \geq \phi \}$$
Solution

**Naive protocol**

- Compute the average global frequency $F(i)/N$ of all items
- Sort items by decreasing frequency
- Identify the item $j$ ranked $k$-th in such ordered sequence
- Return the items whose average frequency is larger or equal than $F(j)/N$
**FREQMF protocol (1)**

FREQMF algorithm executed by node $p$

upon initialization do

| MAP $est_p$ ← new MAP() |
| foreach $i \in I_p$ do |
| $est_p[i]$ ← $F_p(i)$ |

function getTop ($k$)

| return extract($est_p, k$) |

repeat every $\Delta$ time units

| NODE $q$ ← random(PROCESS) |
| send ⟨REQUEST, extract($est_p, s$)⟩ to $q$ |
\section*{FREQMF protocol (2)}

\textbf{FREQMF algorithm executed by node $p$}

\begin{verbatim}
upon receive ⟨REQUEST, $req_q$⟩ from $q$ do
  MAP $rep_p$ ← new MAP()
  ∀ $i \in req_q$
    $real \ \delta_i \leftarrow \frac{1}{2}(req_q[i] - est_p[i])$
    $est_p[i] \leftarrow est_p[i] + \delta_i$
    $rep_p[i] \leftarrow \delta_i$
  send ⟨REPLY, $rep_p$⟩ to $q$

upon receive ⟨REPLY, $rep_q$⟩ do
  ∀ $i \in rep_q$
    $est_p[i] \leftarrow est_p[i] - rep_q[i]$
\end{verbatim}
FREQAF and FREQRF protocols

- How can you implement the FREQAF protocol?
- How can you implement the FREQRF protocol?
Aggregation

Most frequent items

**FREQMF evaluation**

![Graph showing convergence time for different combinations of N and M](image)

- N = 10^2, M = 10^3
- N = 10^2, M = 10^4
- N = 10^2, M = 10^5
- N = 10^2, M = 10^6
- N = 10^3, M = 10^3
- N = 10^3, M = 10^4
- N = 10^3, M = 10^5
- N = 10^3, M = 10^6
- N = 10^4, M = 10^3
- N = 10^4, M = 10^4
- N = 10^4, M = 10^5
- N = 10^4, M = 10^6

Alberto Montresor (UniTN)
FREQMF evaluation
**FREQMF evaluation**

![Graph showing FREQMF evaluation](image-url)
**FreqMF evaluation**

![Graph showing FreqMF evaluation](image)

Legend:
- **M=10^3**
- **M=10^4**
- **M=10^5**
- **M=10^6**

Graph shows the comparison of MFreq Conv. time over Naive Conv. time for different values of M (M=10^3 to M=10^6) as K increases from 0 to 30.
Reading material

- M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen.
  
  Gossip-based peer sampling.
  