Gossip protocols for large-scale distributed systems

Alberto Montresor
In a recent workshop on the future of gossip

- many attempts to formally define gossip
- we failed!
  - either too broad
  - or too strict

Gossip best described with a prototypical gossip scheme

- “I cannot define gossip, but I can recognize it when I see it”
A generic gossip protocol - executed by process $p$

Init: initialize my local $state$

Active thread

Do once every $\delta$ time units

$q = \text{getPeer}(state)$

$s_p = \text{prepareMsg}(state, q)$

send (REQ, $s_p$, $p$) to $q$

A "cycle" of length $\delta$

Passive thread

Do forever

receive ($t$, $s_q$, $q$) from *

if ($t = \text{REQ}$) then

$s_p = \text{prepareMsg}(state, q)$

send (REP, $s_p$, $p$) to $q$

$state = \text{update}(state, s_q)$
During a cycle of length $\delta$, every node has the possibility of contacting one random node.
A generic gossip protocol

- **Generic scheme is... too generic!**

- **Gossip “rules of thumb”**
  - peer selection must be random, or at least guarantee enough *peer diversity*
  - only *local* information is available at all nodes
  - communication is *round-based* (periodic)
  - transmission and processing capacity per round is *limited*
  - all nodes run the *same* protocol
A bit of history

✦ 1987
  ✦ Demers et al. introduced the first gossip protocol, for information dissemination

✦ '90s
  ✦ Gossip applied to solve communication problems

✦ '00s
  ✦ Gossip revival: beyond dissemination

✦ 2006
  ✦ First workshop on the future of gossip, Leiden (NL)
What is going on?

* In the last decade, we have seen dramatic changes in the distributed system area

* **Shift in the scale of distributed systems**
  * larger
  * geographically more dispersed

* **Traditional failure model do not hold any more**
  * “let $p_1 \ldots p_n$ be a set of processes…”
  * $f < 3n+1, f < n/2$ anyone?
  * dynamic membership: “churn”
What is going on?

- **We need to re-think our solutions**
  - Focus on behavior under continuous change
  - Focus on large-scale
  - Focus on convergence, maintenance

- **The laid-back approach of gossip is the right answer**
  - Gossip protocols are indifferent to changes in the group of communicating nodes, single nodes are not important
  - Nodes act based on local knowledge, they are only aware of a small (constant/logarithmic size) portion of the global state
  - Convergence is quick (often logarithmic in size)
The plan

✦ Introduce gossip ✓

✦ Let’s start from the beginning
  ✦ Information dissemination

✦ Beyond dissemination
  ✦ Peer sampling
  ✦ Aggregation
    ✦ Average computation
    ✦ Size estimation
  ✦ Topology management
  ✦ Slicing
  ✦ ....
Gossip Lego

- Gossip solves a diverse collection of problems
- Solutions can be combined to solve more complex problems
- Toward Gossip Lego?

Peer sampling

Swarm intelligence  Dist. optimization  Heuristics

Aggregation  Slicing  ....

Load balancing  Topology bootstrap  ...

© Alberto Montresor
Bibliography

Setting the stage

- **XEROX Clearinghouse Servers**
  - Database replicated at *thousands of nodes*
  - Heterogeneous, *unreliable* network
  - *Independent* updates to single elements of the DB are injected at multiple nodes
  - Updates must propagate to all other nodes or be supplanted by a later updates of that same element
  - Replicas become consistent after no more new updates
  - Assuming a reasonable update rate, most information at any given replica is “current”
**Epidemics** study the spread of a disease or infection in terms of populations of infected/uninfected individuals and their rates of change.

Following the epidemiology literature, we will name a node $p$ as:

- **Susceptible** if $p$ has not yet received an update.
- **Infective** if $p$ holds an update it is willing to share.
- **Removed** if $p$ has the update but is no longer willing to share it.
Model of epidemics

✦ **How does it work?**
   ✦ Initially, a single individual is infective
   ✦ Individuals get in touch with each other, spreading the update

✦ **Rumor spreading, or gossiping, is based on the same principles**

✦ **Can we apply the same ideas to distributed systems?**
   ✦ Our goal is to spread the “infection” (update) as fast as possible!
System model

• A database that is replicated at a set of $n$ nodes $S = \{ s_1, \ldots, s_n \}$

• The copy of the database at node $s$ can be represented by a time-varying partial function:
  - $s.value: K \rightarrow V \times T$
    - $K$ set of keys
    - $V$ set of values
    - $T$ set of timestamps

• In the following slides
  - We will omit the key and we will consider only a single key,value pair
System model

- For simplicity we will assume a database that stores value and timestamp of a single entry at each node $s$
  - $s.value = (v, t)$

- To indicate a deletion at time $t$
  - $s.value = (\text{deleted}, t)$

- The update operation is formalized as
  - $s.value \leftarrow (v, \text{now}())$

- It is assumed by this work that $\text{now}()$ is a function returning a globally unique timestamp (no details)

- So, a pair with a larger timestamp is considered “newer”
The goal

When a database is replicated at many sites, maintaining consistency in the presence of updates is a significant problem.

Eventual Consistency: If no updates take place for a long time, all replicas will gradually become consistent (i.e., the same)

$$\forall r,s \in S : r.value = s.value$$
Several algorithms for distributing updates

- **Best effort**

- **Anti-entropy (simple epidemics)**
  - Push
  - Pull
  - Push-pull

- **Rumor mongering (complex epidemics)**
  - Push
  - Pull
  - Push-pull

- **Eager epidemic dissemination**
How it works?

- Notify *all* other nodes of an update soon after it occurs.
- When receiving an update, check if it is “news”

Node s executes:

- **upon** \( s.\text{value} \leftarrow (v, \text{now}()) \) **do**
  - **foreach** \( r \in S \) **do**
    - **send** \(<\text{UPDATE}, s.\text{value}>\) **to** \( r \)

- **upon receive** \(<\text{UPDATE}, (v, t)>\) **do**
  - **if** \( s.\text{value}.\text{time} < t \) **then**
    - \( s.\text{value} \leftarrow (v, t) \)

- **Not an epidemic algorithm: just the simplest**
  - **What happens if the sender fail “in between”?**
  - **What happens if messages are lost?**
  - **What is the load of the sender?**
A generic gossip protocol - executed by process \( p \)

**Init:** initialize my local \( state \)

**Active thread**

do once every \( \delta \) time units

\[
q = \text{getPeer}(state) \\
sp = \text{prepareMsg}(state, q) \\
\text{send (REQ, } sp, p) \text{ to } q
\]

**Passive thread**

do forever

receive \( (t, sq, q) \) from *

if \( t = \text{REQ} \) then

\[
sp = \text{prepareMsg}(state, q) \\
\text{send (REP, } sp, p) \text{ to } q \\
state = \text{update}(state, sq)
\]

A "cycle" of length \( \delta \)
Anti-entropy - Simple epidemics

* With respect to a single update (identified by its timestamp), all nodes are either
  * susceptible (they don’t know the update), or
  * infective (they know the update)

* Every node regularly chooses another node at random and exchanges database contents, resolving differences
  * Method `getPeer()`
    * Select a random member from $S-\{p\}$
  * Method `prepareMsg()`
    * Simple version: `return s.value`
    * In most cases: prepare a digest of new updates
  * Method `update()`
    * Simple version: see next page
    * In most cases: ask for other data
Implementation of \textit{update()}\newline

\begin{itemize}
  \item \textbf{Push}
    \begin{itemize}
      \item \textbf{if} \texttt{p.value.time > r.value.time} \textbf{then}
        \begin{itemize}
          \item \texttt{r.value} ← \texttt{p.value}
        \end{itemize}
    \end{itemize}
  \item \textbf{Pull}
    \begin{itemize}
      \item \textbf{if} \texttt{p.value.time < r.value.time} \textbf{then}
        \begin{itemize}
          \item \texttt{p.value} ← \texttt{r.value}
        \end{itemize}
    \end{itemize}
  \item \textbf{Push-pull}
    \begin{itemize}
      \item \textbf{if} \texttt{p.value.time > r.value.time} \textbf{then}
        \begin{itemize}
          \item \texttt{r.value} ← \texttt{p.value}
        \end{itemize}
      \item else
        \begin{itemize}
          \item \texttt{p.value} ← \texttt{r.value}
        \end{itemize}
    \end{itemize}
\end{itemize}
Anti-entropy
Anty-entropy: Convergence
To analyze convergence, we must consider what happens when only a few nodes remain susceptible

Let $p(i)$ be the probability of a node being (remaining) susceptible after the $i$-th anti-entropy cycle.

Pull:
To analyze convergence, we must consider what happens when only a few nodes remain susceptible

Let $p(i)$ be the probability of a node being (remaining) susceptible after the $i$-th anti-entropy cycle.

- **Pull:**
  - $p(i+1) = p(i)^2$

- **Push:**
To analyze convergence, we must consider what happens when only a few nodes remain susceptible.

Let $p(i)$ be the probability of a node being (remaining) susceptible after the $i$-th anti-entropy cycle.

- **Pull:**
  - $p(i+1) = p(i)^2$

- **Push:**
  - $p(i+1) = p(i)(1 - \frac{1}{n})^{n(1-p(i))}$
  - For small $p(i)$, $p(i+1) \sim p(i)/e$

- **Push-pull:**
  - both mechanisms are used, convergence is even more rapid

- All converge to 0, but pull is more rapid than push, so in practice pull (or push-pull) is used.
Push vs pull

![Graph showing the comparison between Push only and Pull only with cycles on the x-axis and a decreasing value on the y-axis, with Push only approaching zero much faster than Pull only.]
Benefits:

- Simple epidemics eventually “infect” all the population
- For a push implementation, the expected time to infect everyone is $\log_2(n) + \ln(n)$

Drawbacks:

- Propagates updates much slower than best effort
- Requires examining contents of database even when most data agrees, so it cannot practically be used too often

Normally used as support for best effort, i.e. left running in the background
Anti-entropy: Optimizations

* To avoid expensive databases checks:
  - Maintain checksum, compare databases if checksums unequal
  - Maintain recent update lists for time $T$, exchange lists first
  - Maintain inverted index of database by timestamp; exchange information in reverse timestamp order, incrementally re-compute checksums

* Note:
  - These optimizations apply to the update problem for large DB
  - We will see how the same principle (anti-entropy) may be used for several other kind of applications
Rumor mongering - complex epidemics

- Susceptive-infective-removed (SIR)
  - Nodes initially *susceptive*
  - When a node receives a new update it becomes a “*hot rumor*” and the node *infective*
  - A node that has a rumor periodically chooses randomly another node to spread the rumor
  - Eventually, a node will “*lose interest*” in spreading the rumor and becomes *removed*
    - Spread too many times
    - Everybody knows it

- Optimizations
  - A sender can hold (and transmit) a list of infective updates rather than just one.
Rumor mongering
Rumor mongering
Rumor mongering
Rumor mongering
Rumor mongering: loss of interest

- **Counter vs. coin (random)**
  - *Coin (random):* lose interest with probability $1/k$
  - *Counter:* lose interest after $k$ contacts

- **Feedback vs blind**
  - *Feedback:* lose interest only if the recipient knows the rumor.
  - *Blind:* lose interest regardless of the recipient.
Rumor mongering

- How fast does the system converge to a state where all nodes are not infective? (inactive state)
  - Eventually, everybody will lose interest

- Once in this state, what is the fraction of nodes that know the rumor?
  - The rumor may stop before reaching all nodes
Rumor mongering: analysis

• Analysis from “real” epidemics theory

• Feedback, coin
  
  • Let $s$, $i$ and $r$ denote the fraction of susceptible, infective, and removed nodes respectively. Then:

  $s + i + r = 1$

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

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

  • Solving the equations:

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

  • Thus, increasing $k$ we can make sure that most nodes get the rumor, exponentially better
Quality measures

✦ **Residue:**
  - The nodes that remain susceptible when the epidemic ends: value of $s$ when $i = 0$
  - Residue must be as small as possible

✦ **Traffic:**
  - The average number of database updates sent between nodes
  - $m = \text{total update traffic} \div \# \text{ of nodes}$

✦ **Delay - We can define two delays:**
  - $t_{\text{avg}}$: average time it takes for the introduction of an update to reach a node.
  - $t_{\text{last}}$: time it takes for the last node to get the update.
## Simulation results

### Using feedback and counter

<table>
<thead>
<tr>
<th>Counter $k$</th>
<th>Residue $s$</th>
<th>Traffic $m$</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.176</td>
<td>1.74</td>
<td>$t_{avg}$: 11.0, $t_{last}$: 16.8</td>
</tr>
<tr>
<td>2</td>
<td>0.037</td>
<td>3.30</td>
<td>$t_{avg}$: 12.1, $t_{last}$: 16.9</td>
</tr>
<tr>
<td>3</td>
<td>0.011</td>
<td>4.53</td>
<td>$t_{avg}$: 12.5, $t_{last}$: 17.4</td>
</tr>
<tr>
<td>4</td>
<td>0.0036</td>
<td>5.64</td>
<td>$t_{avg}$: 12.7, $t_{last}$: 17.5</td>
</tr>
<tr>
<td>5</td>
<td>0.0012</td>
<td>6.68</td>
<td>$t_{avg}$: 12.8, $t_{last}$: 17.7</td>
</tr>
</tbody>
</table>

### Using blind and random

<table>
<thead>
<tr>
<th>Counter $k$</th>
<th>Residue $s$</th>
<th>Traffic $m$</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.960</td>
<td>0.04</td>
<td>$t_{avg}$: 19, $t_{last}$: 38</td>
</tr>
<tr>
<td>2</td>
<td>0.205</td>
<td>1.59</td>
<td>$t_{avg}$: 17, $t_{last}$: 33</td>
</tr>
<tr>
<td>3</td>
<td>0.060</td>
<td>2.82</td>
<td>$t_{avg}$: 15, $t_{last}$: 32</td>
</tr>
<tr>
<td>4</td>
<td>0.021</td>
<td>3.91</td>
<td>$t_{avg}$: 14.1, $t_{last}$: 32</td>
</tr>
<tr>
<td>5</td>
<td>0.008</td>
<td>4.95</td>
<td>$t_{avg}$: 13.8, $t_{last}$: 32</td>
</tr>
</tbody>
</table>
Push and pull

- **Push (what we have assumed so far)**
  - If database becomes quiescent, this scheme stops trying to introduce updates.
  - If there are many independent updates, more likely to introduce unnecessary messages.

- **Pull**
  - If many independent updates, pull is more likely to find a source with a non-empty rumor list
  - But if database quiescent, it spends time doing unnecessary update requests.
Push and pull

- **Empirically, in the database system of the authors (frequent updates)**
  - Pull has a better residue/traffic relationship than push

- **Performance of pull epidemic on 1000 nodes using feedback & counters**

<table>
<thead>
<tr>
<th>Counter</th>
<th>Residue</th>
<th>Traffic</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$k$</td>
<td>$s$</td>
<td>$m$</td>
</tr>
<tr>
<td>1</td>
<td>0.031</td>
<td>2.70</td>
<td>9.97</td>
</tr>
<tr>
<td>2</td>
<td>0.00058</td>
<td>4.49</td>
<td>10.07</td>
</tr>
<tr>
<td>3</td>
<td>0.000004</td>
<td>6.09</td>
<td>10.08</td>
</tr>
</tbody>
</table>
Mixing with anti-entropy

• **Rumor mongering**
  - spreads updates fast with low traffic
  - however, there is still a nonzero probability of nodes remaining susceptible after the epidemic

• **Anti-entropy**
  - can be run (infrequently) in the background to ensure all nodes eventually get the update with probability 1.
  - Since a single rumor that is already known by most nodes dies out quickly
Deletion and death certificates

✦ Deletion

✦ We cannot delete an entry just by removing it from a node - the absence of the entry is not propagated.

✦ If the entry has been updated recently, there may still be an update traversing the network!

✦ Death certificate

✦ Solution: replace the deleted item with a *death certificate* (DC) that has a timestamp and spreads like an ordinary update
Deletion and death certificates

✦ **Problem:**
  - we must, at some point, delete DCs or they may consume significant space

✦ **Strategy 1:**
  - retain each DC until all nodes have received it
  - requires a protocol to determine which nodes have it and to handle node failures

✦ **Strategy 2:**
  - hold DCs for some time (e.g. 30 days) and discard them
  - pragmatic approach, still have the “resurrection” problem; increasing the time requires more space
Spatial Distribution

• **In the previous exposition**
  - The network has been considered uniform (i.e. all nodes equally reachable)

• **In reality**
  - More expensive to send updates to distant nodes
  - Especially if a critical link needs to be traversed
  - Traffic can clog these links
Peer sampling

- **Bibliography**


Peer sampling

- **The first problem to be solved:**
  - Where `getPeer()` get nodes from?
  - We assumed complete knowledge of the distributed system

- **But complete knowledge is costly**
  - System is dynamic
  - Network can be extremely large

- **Solution: peer sampling**
  - Provides random samples from the participant set
  - Keeps the participants together in a connected network
Can you spot the difference?

- **Traditional gossip**
  - Each node has *full* view of the network
  - Each node periodically “gossips” with a random node, out of the *whole* set

- **Peer sampling**
  - Nodes have a *partial view* of the network (a set of “neighbors”)
  - Each node periodically “gossips” with a random node, out of its partial view
An overlay network is a logical network overimposed on a physical network

- Nodes
- Logical links between nodes

Examples

- Structured overlay network
  - DHTs, trees
- Unstructured overlay network
  - Gnutella
  - Bittorrent
  - etc.
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
  - Messages can be lost – high levels of message omissions can be tolerated
Our Overlays

* **State of each node:**
  * A partial view containing $c$ descriptors
  * ($c =$ view size)

* **Descriptors of node $p$ contains**
  * The address needed to communicate with $p$
  * Additional information that may be needed by different implementations of the *peer sampling service*
  * Additional information that may be needed by *upper layers*
A generic gossip protocol - executed by process $p$

**Init:** initialize my local $state$

**Active thread**

do once every $\delta$ time units

\[ q = \text{getPeer}(state) \]
\[ s_p = \text{prepareMsg}(state, q) \]
\[ \text{send } (REQ, s_p) \text{ to } q \]

**Passive thread**

do forever

receive $(t, s_q)$ from *

if $t = \text{REQ}$ then

\[ s_p = \text{prepareMsg}(state, q) \]
\[ \text{send } (REP, s_p) \text{ to } q \]

\[ state = \text{update}(state, s_q) \]

A "cycle" of length $\delta$
A generic algorithm

- **getPeer()**
  - select one of the neighbor contained in the view

- **prepareMsg(view, q)**
  - returns a subset of the descriptors contained in the local view
  - may add other descriptors (e.g. its own)

- **update(view, msgq)**
  - returns a subset of the descriptors contained in the union of the local view and the received view
Newscast

- **Descriptor**: address + timestamp

- **getPeer()**
  - select one node at random

- **prepareMsg(view, q)**
  - returns the entire view + a local descriptor
    with a fresh timestamp

- **update(view, msgq)**
  - returns the C freshest identifiers (w.r.t. timestamp) from the union of local view and message
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
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)
Newscast

- **Experiments**
  - 100,000 nodes
  - $C = 20$ neighbors per node
Evaluation framework

- **Average path length**
  - The average of shortest path lengths over all pairs of nodes in the graph

- **In epidemic dissemination protocols**
  - A measure of the time needed to diffuse information from a node to another
Clustering coefficient

- The clustering coefficient of a node $p$ is defined as the # of edges between the neighbors of $p$ divided by the # of all possible edges between those neighbors.
- Intuitively, indicates the extent to which the neighbors of $p$ know each other.
- The clustering coefficient of the graph is the average of the clustering coefficients of all nodes.

Examples
- for a complete graph it is 1
- for a tree it is 0

In epidemic dissemination protocols

- High clustering coefficient means several redundant messages are sent when an epidemic protocol is used.
Average path length

- Indication of the time and cost to flood the network
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

Newscast forms a SMALL WORLD

- High clustering
- Low diameter
In-Degree Distribution

- **Affects:**
  - Robustness
    (shows weakly connected nodes)
  - Load balancing
  - The way epidemics spread
Robustness

Sustains up to 68% node failures

Random sustains up to 80%
Self-healing behaviour
Cyclon

- **Descriptor: address + timestamp**

- **getPeer()**
  - select the oldest descriptor in the view
  - remove it from the view

- **prepareMsg(view, q)**
  - In active thread:
    - returns a subset of $t-1$ random nodes, plus a fresh local identifier
  - In passive thread:
    - returns a subset of $t$ random nodes

- **update(view, msg_q)**
  - discard entries in $msg_q$: $p$, nodes already know
  - add $msg_q$, removing entries sent to $q$
Cyclon
1. Pick **oldest** peer from my view
1. Pick **oldest** peer from my view
1. Pick **oldest** peer from my view
2. Exchange some neighbors (the pointers)
1. Pick **oldest** peer from my view
2. Exchange some neighbors (the pointers)
1. Pick **oldest** peer from my view
2. Exchange some neighbors (the pointers)
1. Pick **oldest** peer from my view
2. Exchange some neighbors (the pointers)
1. Pick **oldest** peer from my view
2. Exchange some neighbors (the pointers)
1. Pick **oldest** peer from my view
2. Exchange some neighbors (the pointers)
1. Pick **oldest** peer from my view
2. Exchange some neighbors (the pointers)
Obvious advantages of Cyclon

✦ Connectivity is guaranteed

✦ Uses less bandwidth
  ✔ Only small part of the view is sent
Average path length

- Indication of the time and cost to flood the network
**Clustering coefficient**

- **High clustering is bad for:**
  - Flooding: It results in many redundant messages
  - Self-healing: Strongly connected cluster $\rightarrow$ weakly connected to the rest of the network
Clustering coefficient

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

- Low clustering
- Low diameter

Cyclon approx. a RANDOM GRAPH
In-Degree Distribution

- **Affects:**
  - Robustness (shows weakly connected nodes)
  - Load balancing
  - The way epidemics spread
Robustness

Sustains up to 80% node failures
Self-healing behaviour
Self-healing behaviour

Killed 50,000 nodes at cycle 19
Non-symmetric overlays

- **Non-uniform period** → **Non symmetric topologies**
  - A node’s in-degree is proportional to its gossiping frequency
  - Can be used to create topologies with “super-nodes”
Secure peer sampling

- This approach is vulnerable to certain kinds of malicious attacks

- Hub attack
  - Hub attack involves some set of colluding nodes always gossiping their own ID’s only
  - This causes a rapid spread of only those nodes to all nodes - we say their views become “polluted”
  - At this point all non-malicious nodes are cut-off from each other
  - The malicious nodes may then leave the network leaving it totally disconnected with no way to recover
  - Hence the hub attack hijacks the speed of the Gossip approach to defeat the network
Secure peer sampling
Peer sampling - solution

- **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
  - Otherwise, whitelist the node and accept the exchange
  - Apply an aging policy to both white and black lists
  - When supplying a random peer to API select the current “best” view
Secure peer sampling

- 1000 nodes
- 20 malicious nodes
How to compose peer sampling

Aggregation
prepareMsg()
update()

Information dissemination
prepareMsg()
update()

Peer sampling
prepareMsg()
update()

Peer sampling
prepareMsg()
update()
Bibliography


Additional bibliography

Aggregation

• **Definition**
  - *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: solutions that are**
  - completely decentralized, robust
A generic gossip protocol - executed by process \( p \)

Init: initialize my local \( state \)

**Active thread**

\[
\text{do once every } \delta \text{ time units}
\]

\[
q = \text{getPeer}(state)
\]

\[
s_p = \text{prepareMsg}(state, q)
\]

\[
\text{send (REQ, } s_p \text{) to } q
\]

**Passive thread**

\[
\text{do forever}
\]

\[
\text{receive } (t, s_q) \text{ from } *
\]

\[
\text{if } (t = \text{REQ}) \text{ then}
\]

\[
s_p = \text{prepareMsg}(state, q)
\]

\[
\text{send (REP, } s_p \text{) to } q
\]

\[
state = \text{update}(state, s_q)
\]

A "cycle" of length \( \delta \)
Average Aggregation

- Using the gossip schema presented above to compute the average
  - Local state maintained by nodes:
    - a real number representing the value to be averaged
  - Method `getPeer()`
    - invokes `getPeer()` on the underlying peer sampling layer
  - Method `prepareMessage()`
    - `return state_p`
  - Function `update(state_p, state_q)`
    - `return (state_p+state_q)/2`
The idea
Basic operation

\[(10+2)/2=6\]
Basic operation
Basic operation

\[
\frac{16+4}{2} = 10
\]
• **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**
A run of the protocol
Questions

- Which topology is optimal?
- How fast is convergence on different topologies?
- What are the effects of node/link failures, message omissions?
- Fully connected topol.: exponential convergence
- Random topology: practically exponential.
- Link failures: not critical
- Crashes/msg omissions can destroy convergence
- but we have a solution for that
Theoretical framework

- From the “distributed” version to a centralized one

```plaintext
do N times
  (p, q) = getPair()
  // perform elementary aggregation step
  a[p] = a[q] = (a[p] + a[q])/2
```

- Notes:
  - Vector $a[1 ... N]$
  - $N$ number of nodes
  - The code corresponds to the execution of single cycle
Some definitions

- We measure the speed of convergence of empirical variance at cycle $i$

\[
\mu_i = \frac{1}{n} \sum_{k=1}^{n} a_i[k]
\]

\[
\sigma_i^2 = \frac{1}{n} \sum_{k=1}^{n} (\mu_i - a_i[k])^2
\]

- Additional definitions
  - Elementary *variance reduction step*: $\sigma_{i+1}^2 / \sigma_i^2$
  - *Variable $\varphi_k$*: the number of times that node $k$ has been selected from $getPair()$
The base theorem

- **If**
  - Each pair of values selected by each call to `getPair()` are uncorrelated;
  - the random variables $\varphi_k$ are identically distributed;
    - let $\varphi$ denote a random variable with this common distribution
  - 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:**

$$E(\sigma_{i+1}^2) = E(2^{-\varphi}) \sigma_i^2$$
Results

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

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

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

The graph illustrates the convergence factor of a network size against the network size. The convergence factor is a measure of how closely the network converges to a certain state. The graph compares two types of network structures: fully connected and 20-regular random. The fully connected network shows a slight increase in the convergence factor as the network size increases, while the 20-regular random network maintains a relatively stable convergence factor across different network sizes.
Convergence factor

![Diagram showing convergence factor over cycles for fully connected and 20-regular random networks.](image)
Other functions

- **Average:** \( \text{update}(a,b) := (a+b)/2 \)
- **Geometric:** \( \text{update}(a,b) := (a \cdot b)^{1/2} \)
- **Min/max:** \( \text{update}(a,b) := \min/\max(a,b) \)
- **Sum:** Average \( \cdot \) Count
- **Product:** Geometric \( ^\text{Count} \)
- **Variance:** compute \( \overline{a^2} - a^2 \)

Means

How?

Obtained from means
Counting

- **The counting protocol**
  - Init: one node starts with 1, 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 $p=c/NE$
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
Adaptivity

- **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
  - Concurrent epochs depending on the ratio $\frac{\lambda\delta}{\Delta}$
Dynamic Membership

- When a node joins the network
  - Discovers a node $n$ already in the network
  - Membership: initialization of the local neighbors
  - Receives from $n$:
    - 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
Dynamic Membership

 renown change, message omissions

- In the active thread:
  - A timeout is set to detect the failure of the contacted node
  - If the timeout expires before the message is received → the exchange step is skipped

- What are the consequences?
  - In general: convergence will slow down
  - In some cases: estimate may converge to the wrong value
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 = $1 + \phi$, where $\phi$ has Poisson distribution with average 1

- **Cycle length $\delta$ defines the time complexity of convergence:**
  - Small $\delta$: fast convergence
  - Large $\delta$: small cost per unit time, may be needed to complete exchanges

- **$\lambda$ defines the accuracy of convergence:**
  - $E(\sigma_{\lambda}^2)/E(\sigma_0^2) = \rho \lambda$, $\varepsilon$ the desired accuracy $\rightarrow \lambda \geq \log_\rho \varepsilon$
The 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
Topologies

![Graph showing variance reduction over cycles for different topologies and values of B. The topologies include \(<B=0.00>\), \(<B=0.25>\), \(<B=0.50>\), \(<B=0.75>\), Newscast, Sci-free, Random, and Complete.]
Simulation scenario

- **The underlying topology is based on Newscast**
  - Realistic, Robust

- **The count protocol is used**
  - More sensitive to failures

- **Some parameters:**
  - Network size is 100,000
  - Partial view size in Newscast is 30
  - Epoch length is 30 cycles
  - Number of experiments is 50
Effects of node failures

✦ Effects depend on the value lost in a crash:
  ✦ *If lower than actual average*: estimated average will increase, estimated size will decrease
  ✦ *If higher than actual average*: estimated average will decrease, estimated size will increase

✦ The latter case is worst:
  ✦ In the initial cycles, some nodes hold relatively large values

✦ Simulations:
  ✦ Sudden death / dynamic churn of nodes
Sudden death

Experiments

Estimated Size

Cycle
Nodes joining/crashing

Experiments

Estimated Size

Nodes Substituted per Cycle
Communication failures

✦ **Link failures**
  
  ✷ The convergence is just slowed down – some of the exchanges do not happen

✦ **Partitioning:**
  
  ✷ If multiple concurrent protocols are started, the size of each partition will be evaluated separately

✦ **Message omissions:**
  
  ✷ Message carry values: losing them may influence the final estimate
Link failures
Message omissions

Experiments (Max values)
Experiments (Min values)
Multiple instances of aggregation

✦ 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
Node failures
Message omissions
All together now!
• **Consortium**
  • >500 universities, research institutes, companies
  • >1000 nodes
600 hosts, 10 nodes per hosts
Topology management

- Bibliography

- Additional bibliography
Topology bootstrap

* Informal definition:
  * building a 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
  * Much more complicated: start from scratch
**The T-Man Algorithm**

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

- **Examples**
  - Rings, tori, trees, DHTs, etc.
  - Distributed sorting
  - Semantic proximity for file-sharing
  - Latency for proximity selection.....

![Images of graphs after different cycles](after_2_cycles.png) ![Images of graphs after different cycles](after_3_cycles.png) ![Images of graphs after different cycles](after_4_cycles.png) ![Images of graphs after different cycles](after_7_cycles.png)
Node descriptors contain attributes of the nodes

- A number in a sorting application
- The id of a node in a DHT
- A semantic description of the node

Example: Sorted “Virtual” Ring

- Let the ranking function be defined based on the distance

\[ d(a,b) = \min(|a-b|, 2^t - |a-b|) \]

assuming that attributes are included in \([0, 2^t[\)
Ranking function

- **Node descriptors contain attributes of the nodes**
  - A number in a sorting application
  - The id of a node in a DHT
  - A semantic description of the node

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

- `getPeer()`, `prepareMsg()` are based on a ranking function defined over node descriptors
A generic gossip protocol - executed by process $p$

**Init:** initialize my local state

**Active thread**

_do once every $\delta$ time units_

$q = \text{getPeer}(\text{state})$

$s_p = \text{prepareMsg}(\text{state}, q)$

-send $(\text{REQ}, s_p)$ to $q$

**Passive thread**

_do forever_

_receive $(t, s_q)$ from *

if $(t = \text{REQ})$ then

$s_p = \text{prepareMsg}(\text{state}, q)$

-send $(\text{REP}, s_p)$ to $q$

$\text{state} = \text{update}(\text{state}, s_q)$

_A "cycle" of length $\delta"_
Gossip customization for topology construction

- **local state**
  - partial view, initialized randomly based on Newscast
  - the view grows whenever a message is received

- **getPeer()**:
  - randomly select a peer $q$ from the $r$ nodes in my view that are closest to $p$ in terms of distance

- **prepareMsg()**:
  - send to $q$ the $r$ nodes in local view that are closest to $q$
  - $q$ responds with the $r$ nodes in its view that are closest to $p$

- **update()**:
  - both $p$ and $q$ merge the received nodes to their view
## T-man: Topology Management

<table>
<thead>
<tr>
<th>175</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
</tr>
<tr>
<td>90</td>
</tr>
<tr>
<td>130</td>
</tr>
<tr>
<td>499</td>
</tr>
<tr>
<td>700</td>
</tr>
<tr>
<td>993</td>
</tr>
</tbody>
</table>
T-man: Topology Management

getPeer
T-man: Topology Management

Exchange of partial views
T-man: Topology Management

Both sides apply update thereby redefining topology
Distance functions

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

- **Example: Grid or torus (Manhattan Distance)**
  - Space: $[0, 1[ \times [0, 1[$
  - Distance: $d(a,b) = | a_x - b_x | + | a_y - b_y |$
Example: Line
Sorted Line / Ring

* Directional ranking function over the ring defined as follows:
  * Distance function, line: \( d(a,b) = |a-b| \)
  * Distance function, ring: \( d(a,b) = \min(|a-b|, 1-|a-b|) \)

* Given a collection (view) of nodes and a node \( x \), return
  * the \( r/2 \) nodes “smaller” than \( x \) that are closest to \( x \)
  * the \( r/2 \) nodes “larger” than \( x \) that are closest to \( x \)
Sorted Ring
Sorted Ring

\[ p, q \]

\[ 0 \rightarrow 2^{m-1} \]

\[ 0 \rightarrow 2^{m-1} \]
Nodes: 1000
Showing 1 successor,
1 predecessor

Cycles 00.000 Nodes
Start / 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 the 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
In the worst of our experiments, we observed that no more than 0.1% of the target links were missing at termination. This may be sufficient for most applications, especially considering that the target graphs will never be constructed perfectly in a dynamic scenario, where nodes are added and removed continuously. Nonetheless, from now on, we discard the parameter combinations that do not always converge.

Apart from longer executions, an additional consequence of choosing large values of $d_{idle}$ is a higher communication cost. However, since not all nodes are active during the execution, the overall number of messages sent per node on average is less than one quarter of the number of cycles until global termination. To understand this better, Fig. 13 shows how many nodes are active during the construction of SORTED RING and TREE, respectively. The curves show both an exponential increase in the number of active nodes when starting, and an exponential decrease when stopping. The period of time in which all nodes are active is relatively short.

These considerations suggest the use of higher values for $d_{idle}$, at the cost of a larger termination time and a larger number of exchanged messages. The chosen value of $d_{idle}$ (4 s) represents a good tradeoff between the desire of obtaining a perfect target graph and the consequently larger cost in time and communication.

6.7. Parameter tuning

6.7.1. Cycle length

If a faster execution is desired, one can always decrease the cycle length. However, after some point, decreasing cycle length does not pay off because message delay becomes longer than the cycle length and eventually the network will be congested by T-MAN messages. Fig. 14 shows the behavior.

- **Fig. 12.** Quality of the target TREE graph at termination time as a function of $d_{idle}$.
- **Fig. 13.** Proportion of active nodes during execution.
- **Fig. 14.** Termination time as a function of cycle length.
Communication cost. The number of messages exchanged. Note that all messages ever exchanged are of the same size.

The unit of time will be cycles or seconds, depending on which is more convenient (note that cycle length defaults to 1 s). We also note that convergence time is not defined if the protocol terminates before converging. In this case, we use the number of identified target links as a measure.

6.5. Evaluating the starting mechanism

Fig. 10 shows the convergence time for SORTED RING and TREE, using the starting protocols described in Section 6.1.2. The cycle length of the anti-entropy versions was the same as that of T-MAN, and the flooding protocol used 20 random neighbors at all nodes. The case of synchronous start is also shown for comparison. Note that these figures do not represent a direct measure of the performance of well-known starting protocols; rather, convergence time plotted here represents the overall time needed to both start the protocol and reach convergence, with T-MAN and the broadcast protocol running concurrently.

In the case of flooding, "wake up" messages quickly reach all nodes and activate the protocol; almost no delay is observed compared to the synchronous case. Anti-entropy mechanisms result in a few seconds of delay. In the experiments that follow, we adopt the anti-entropy, push–pull approach, as it represents a good tradeoff between communication costs and delay. Note however that (unlike the push approach) the push–pull approach assumes that at least the starting service was started at all nodes already.

6.6. Evaluating the termination mechanism

We experimented with various settings for $d_{idle}$ ranging from 2 s to 12 s. Fig. 11 shows both convergence time (bottom three curves) and termination time (top three curves) for different values of $d_{idle}$, for SORTED RING and TREE, respectively. In both cases, termination time increases linearly with $d_{idle}$. This is because, assuming the protocol has converged, each additional cycle to wait simply adds to the termination time.

For small values convergence was not always reached, especially for TREE. For SORTED RING, all runs converged except the case when $d_{idle} = 2$ and $N = 2^16$, when 76% of the runs converged. For TREE, all runs converged with $d_{idle} > 5$ and no runs converged for $(d_{idle} = 2; N = 2^{13})$; $(d_{idle} = 2; N = 2^16)$, and $(d_{idle} = 3; N = 2^16)$. Even in these cases, the quality of the target graph at termination time was almost perfect.
6.7.2. Message size

In Section 5, we have examined the effect of the message size parameter ($m$) in detail. Here we are interested in the effect of message size on termination time.

Fig. 15 shows that by increasing the size of messages exchanged by SORTED RING, the termination time slightly increases after around $m = 20$. The reason is that a node becomes suspended only after the local view remains unchanged for a fixed number of cycles, but increasing the message size has the effect of increasing the number of cycles in which view changes might occur, thus delaying termination. The results for TREE have more variance, which might have to do with the unbalanced nature of the topology, as discussed in Section 5.3.

6.8. Failures

The results discussed so far were obtained in static networks, without considering any form of failure. Here, we consider two sources of failure: message losses and node crashes. Since in this paper we consider only the overlay construction problem, and not maintenance, we do not explicitly consider scenarios involving node churn. Instead, we model churn through nodes leaving, and do not allowing joining nodes to participate in an ongoing construction. Furthermore, since we do not have a leave protocol, leaving nodes are identical to crashing nodes from our point of view.

6.8.1. Message loss

While a simple solution could be to adopt a reliable, connection-oriented transport protocol like TCP, it is more attractive to rely on a lightweight but perhaps unreliable transport. In this case, we need to demonstrate that T-MAN can cope well with message loss.

Fig. 16 shows that T-MAN is highly resilient to message loss and so a datagram-oriented protocol like UDP is a perfectly suitable choice, as message losses only slow down the protocol slightly. Many message exchanges are either never started or never completed, thus requiring more cycles to terminate the protocol execution. The quality does not suffer much either. In both SORTED RING and TREE, around 1% of the target links may be missing, as shown by Fig. 17. Note that the mean message loss ratio for geographic networks like the Internet is around 2% [30], an order of magnitude smaller than the maximum message loss ratio tested in our experiments.
T-Man: Robustness to crashes

Node failures per node per second

Target Links Found (%)
T-Man: Robustness to message losses

![Graph showing the relationship between message loss and target links found]
T-Chord

- **How it works?**
  - 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:**
  - $t=32$, size=$2^{14}$, msg size=20
Robustness to failures
Robustness to failures
Conclusions

- This mechanism to build Chord is tightly tailored on the particular structure of Chord

- A more generic approach:
  - Define a ranking function where nodes have a preference for successors and predecessors AND fingers
  - Approx. same results, only slightly more complex to explain

- Can be used for Pastry, for example:
  - Define a ranking function where nodes have a preference for successors and predecessors AND nodes in the prefix-based routing table
Bibliography

Network size estimation at runtime

✦ **Why**

- $f(n)$ routing pointers
  - to bound the hop count
  - to provide churn resilience
- build group of size $f(n)$
  - Slicing
- $f(n)$ messages
  - to reduce overhead in gossip protocols

✦ **How**

- Combine and improve existing protocols
- Compared to existing systems:
  - More precise, more robust, slightly more overhead
- Simple idea $\rightarrow$ short paper
A brief explanation

- Assign random numbers in \([0, d]\)
  - Locally; here, \(d=127\)

- Build a ring topology
  - Gossip topology construction (T-Man)

- Compute the distance to the successor
  - Locally

- Compute the average distance \(a\)
  - Gossip aggregation

- Compute size
  - \(d / a = n\)
Scalability

The graphs show the convergence time (s) and overhead per node (kB) for different network sizes. The convergence time is measured in seconds, and the overhead is measured in kilobytes per node. The graphs are divided into two parts: one for T-Size and another for Average. The T-Size graph shows a steady increase in convergence time and overhead with increasing network size, while the Average graph shows a more linear increase. The x-axis represents the network size, which is indicated by the base-2 exponent, and the y-axes show the respective metrics.
Accuracy – w.r.t. parameter Precision
**Bibliography**

Introduction

✦ **System model**
  ✷ An huge collection of networked nodes (resource pool)
  ✷ Potentially owned/controlled by a single organization that deploys massive services on them

✦ **Examples**
  ✷ ISPs that place smart modems / set-top boxes at their customers' homes
  ✷ BT, France Telecom

✦ **Note: similar to current P2P systems, but with some peculiar differences**
Introduction: possible scenarios

✦ **Multiple-services architecture**
  - Nodes must be able to host a large number of services, potentially executed by third-party entities

✦ **On-demand services**
  - A subset of the nodes can be leased temporally and quickly organized into an overlay network

✦ **Adaptive resource management**
  - Resource assignments of long-lived services could be adapted based on QoS requirements and changes in the environment
Overview

✦ **What we need to realize those scenarios?**
  ✦ Maintain a dynamic membership of the network (peer sampling)
  ✦ Dynamically allocate subset of nodes to applications (*slicing*)
  ✦ Start overlays from scratch (bootstrap)
  ✦ Deploy applications on overlays (broadcast)
  ✦ Monitor applications (aggregation)

✦ **This while dealing with massive dynamism**
  ✦ Catastrophic failures
  ✦ Variations in QoS requirements (flash crowds)
Architecture: a decentralized OS

Applications

Other middleware services (DHTs, indexing, publish-subscribe, etc.)

Slicing Service
Topology Bootstrap
Monitoring Service
Broadcast

Peer sampling service
The problem

- **Distributed Slicing**
  - Given a distributed collection of nodes, we want to allocate a subset (“slice”) of them to a specific application, by selecting those that satisfy a given condition over group or node attributes

- **Ordered Slicing (Fernandez et al., 2007)**
  - Return top $k\%$ nodes based on some attribute ranking

- **Absolute slicing**
  - Return $k$ nodes and maintain such allocation in spite of churn

- **Cumulative slicing**
  - Return nodes whose attribute total sum correspond to a target value
Problem definition

- We consider a dynamic collection $N$ of nodes

- Each node $n_i \in N$ is provided with an attribute function
  - $f_i: A \rightarrow V$

- Slice $S(c,s)$: a dynamic subset of $N$ such that
  - $c$ is a first-order-logic condition defined over attribute names and values, identifying the potential member of the slice
  - $s$ is the desired slice size

- Slice quality:
  \[
  \frac{|S(c, s)| - s}{s}
  \]
Problem definition

Slice nodes
(total slice size \( \sim s \))

Potential nodes
(each node satisfying \( c \))

Total nodes
Issues

✦ What we mean with “return a slice”?  
  ✦ We cannot provide each node with a complete view of large scale subset  
  ✦ Slice composition may continuously change due to churn

✦ How we compute the slice size?  
  ✦ without a central service that does the counting?

✦ How do we inform nodes about the current slice definition?  
  ✦ Multiple slices, over different conditions, with potentially changing slice sizes
Gossip to the rescue

• **Turns out that all services listed so far can be implemented using a gossip approach**
  
  • *Peer sampling*: continuously provides uniform random samples over a dynamic large collection of nodes
    
    • random samples can be used to build other gossip protocols
    
    • side-effect: strongly connected random graph
  
  • *Aggregation*: compute aggregate information (average, sum, total size, etc.) over large collection of nodes
    
    • we are interested in size estimation
  
  • *Broadcast*: disseminate information

• **Gossip beyond dissemination**

  • Information is not simply exchanged, but also manipulated
Architecture of the absolute slicing protocol

Application Protocol

Aggregation

Peer Sampling

Aggregation

Peer Sampling

Broadcast

Peer Sampling

Peer Sampling

Peer Sampling
The slicing algorithm

- **Total group**
  - All nodes participate in the peer sampling protocol to maintain the total group

- **Potential group**
  - Nodes that satisfy the condition $c$ join the potential group peer sampling
    - Means: inject their identifier into message exchanged at the 2nd peer sampling layer

- **Aggregation**
  - Estimate the size of the potential group $size(P)$
The slicing algorithm

✦ **Slice group**
  ✦ Nodes that “believe to belong” to the slice join the slice peer sampling
    ✦ Means: inject their identifier into messages exchanged at the 3\textsuperscript{rd} peer sampling layer

✦ **Aggregation**
  ✦ Estimate the size of the slice $size(S)$

✦ **Nodes “believe to belong” or not to the slice**
  ✦ join the slice with prob. $(s-size(S)) / (size(P)-size(S))$
  ✦ leave the slice with prob. $(size(S)-s) / size(S)$
Experimental results: actual slice size
Experimental results: churn $10^{-4}$ nodes/s
Experimental results: variable churn
Experimental results: message losses
Slicing - conclusion

✦ Absolute slicing protocol
  ✦ Extremely robust (high level of churn, message losses)
  ✦ Low cost (several layers, but each requiring few bytes per sec)
  ✦ Large precision

✦ The message
  ✦ Gossip can solve many problems in a robust way
  ✦ Customizable to many needs

✦ What's next?
  ✦ Cumulative slicing:
    ✦ very similar, but it's a knapsack problem
**Bibliography**


**Additional bibliography**


Particle swarm optimization

**Input:**
- A multi-dimensional function
- A multi-dimensional space to be inspected

**Output:**
- The point where the minimum is located, together with its value

**Approximation problem**
• A solver is a swarm of particles spread in the domain of the objective function

• Particles evaluate the objective function in a point $p$, looking for the minimum

• Each particle knows the best points
  • found by itself ($b_p$)
  • found by someone in the swarm ($b_g$)

• Each particle updates its position $p$ as follows:
  • $v = v + c_1 \times \text{rand()} \times (b_p - p) + c_2 \times \text{rand()} \times (b_g - p)$
  • $p = p + v$
Particle swarm optimization

- Modular architecture for distributed optimization:
  - The topology service (NEWSCAST)
    - creates and maintains an overlay topology
  - The function optimization service (D-PSO)
    - evaluates the function over a set of points
    - Local/remote history driven choices
  - The coordination service (gossip algorithm)
    - determines the selection of points to be evaluated
    - spread information about the global optimum
• **Communication failures are harmless**
  • Losses of messages just slow down the spreading of (correct) information

• **Churning is inoffensive**
  • Nodes can join and leave arbitrarily and this does not affect the consistency of the overall computation
Scalability
Gossip Lego, reloaded

✦ The baseplate
  ✷ Peer sampling

✦ The bricks
  ✷ Slicing (group management)
  ✷ Topology bootstrap
  ✷ Aggregation (monitoring)
  ✷ Load balancing (based on aggregation)

✦ Applications
  ✷ Function optimization
  ✷ P2P video streaming
  ✷ Social networking
Conclusions

✦ We only started to discover the power of gossip
  ✦ Many other problems can be solved

✦ The right tool for
  ✦ large-scale
  ✦ dynamic systems

✦ Caveat emptor: security
  ✦ We only started to consider the problems related to security in open systems
PeerSim: A Peer-to-Peer Simulator

Introduction

Peer-to-peer systems can be of a very large scale such as millions of nodes, which typically join and leave continuously. These properties are very challenging to deal with. Evaluating a new protocol in a real environment, especially in its early stages of development, is not feasible.

PeerSim has been developed with extreme scalability and support for dynamicity in mind. We use it in our everyday research and chose to release it to the public under the GPL open source license. It is written in Java and it is composed of two simulation engines, a simplified (cycle-based) one and an event driven one. The engines are supported by many simple, extendable, and pluggable components, with a flexible configuration mechanism.

The cycle-based engine, to allow for scalability, uses some simplifying assumptions, such as ignoring the details of the transport layer in the communication protocol stack. The event-based engine is less efficient but more realistic. Among other things, it supports transport layer simulation as well. In addition, cycle-based protocols can be run by the event-based engine too.

PeerSim started under EU projects BISON and DELIS DELIS. The PeerSim development in Trento (Alberto Montresor, Gian Paolo Jesi) is now partially supported by the Napa-Wine project.

People

The main developers of PeerSim are:

- Mark Jelasity, Alberto Montresor,
- Gian Paolo Jesi, Spyros Voulgaris

Other contributors and testers (in alphabetical order):

- Stefano Arteconi
- David Hales
- Andrea Marcozzi
- Fabio Picconi
Thanks

• **My co-authors:**
  • Mark, Ozalp, Gian Paolo, Marco, Roberto, Ali, Maarten, Mauro, Balazs

• **For some slides:**
  • Spyros

• **Those who invited me:**
  • Marinho, Luciano and Lisandro

• **And finally:**
  • You for listening!