Gossip Midterm assignment


• Write a report that describes:
  
  1) the gossip strategy to perform aggregation with the main characteristics of the protocol. Provide a short description of the functions that is possible to implement with gossip aggregation. (Optional bonus question: how would you estimate a distribution of values?)

  2) the impact of node failures, message losses, and message delay on the correctness of aggregation. Describe one way to cope with failures.
Gossip Java Libraries

**incubator-gossip** ([https://github.com/apache/incubator-gossip](https://github.com/apache/incubator-gossip))

Seems rather simple, recently updated and improving.

**GossipLib** ([http://gossiplib.gforge.inria.fr/](http://gossiplib.gforge.inria.fr/))

No documentation apart from Javadoc. Most of the popular protocols are already implemented.

**java-gossip** ([https://github.com/jolira/java-gossip](https://github.com/jolira/java-gossip))

Last update 6 years ago. Some documentation on the old google code site ([https://code.google.com/archive/p/java-gossip/](https://code.google.com/archive/p/java-gossip/)).
Peer Sampling
select a sample of peers from the whole population
Peer sampling

A source node wants to choose another peer of the network that has certain characteristics
Node Selection

- Gossip algorithms (aggregation, dissemination, etc..) are based on the following assumption:
  - after a given interval, a node P may select a node Q chosen uniformly at random among the set of all the nodes participating to the protocol

- **Problem:** How can we select a random peer in a fully distributed P2P system?
Peer Sampling

• We need a **Peer Sampling Service**, i.e. a mechanism approximating a (random) choice on the whole set of nodes by exploiting only local information

• we don't want a centralised server doing that

• The peer sampling service is built with gossip as well

• The basic idea: the nodes gossip with their neighbours and the topics of the gossip is.. the knowledge of the neighbours!
Two-layer architecture

- We have a two-layer architecture:
  - a gossip application layer (e.g. Aggregation)
  - an underlying peer sampling service
- The application layers exploit the peer sampling when it needs another node to communicate with

```plaintext
node's layered architecture

Peer Sampling

select_peer()

Gossip application (Aggregation, etc..)
```
Ideal Peer Sampling

- In the ideal case nodes keep a full view of the network, and can sample from the entire node population.
- This cannot be true in reality (it wouldn't scale!)
  - Too many nodes
  - Churn
- There is no point in having scalable gossip protocols if the peer sampling support does not...
Actual Peer Sampling

- Instead of the whole network, each node maintains a **partial view** of fixed size of the network.

- When gossiping, nodes exchange such partial view and decide which peers to keep of the other’s view.

- The idea is to build a **dynamic unstructured overlay** through gossiping.

- In other words, the peer sampling service dynamically changes the neighbours (hence the overlay) of a node.
Peer Sampling

• Random Peer Sampling protocols (Newscast, Cyclon) provides an always changing subset of the view from where to pick
Peer Sampling

• Random Peer Sampling protocols (Newscast, Cyclon) provides an **always changing** subset of the view from where to pick

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
</table>
Peer Sampling

- Random Peer Sampling protocols (Newscast, Cyclon) provides an **always changing** subset of the view from where to pick

network

```
 A B C D E F G H I L M
```

```
 A B C E
```
Peer Sampling

• Random Peer Sampling protocols (Newscast, Cyclon) provides an *always changing* subset of the view from where to pick.
Peer Sampling

- Random Peer Sampling protocols (Newscast, Cyclon) provides an **always changing** subset of the view from where to pick.
Peer Sampling

- Random Peer Sampling protocols (Newscast, Cyclon) provides an **always changing** subset of the view from where to pick
Peer Sampling

- Random Peer Sampling protocols (Newscast, Cyclon) provides an **always changing** subset of the view from where to pick

![Network Diagram]

A B C D E F G H I L M

A B C E → gossip → A F G L
Peer Sampling

- Random Peer Sampling protocols (Newscast, Cyclon) provides an always changing subset of the view from where to pick.
Peer Sampling

- Random Peer Sampling protocols (Newscast, Cyclon) provides an **always changing** subset of the view from where to pick.
Peer Sampling

- Random Peer Sampling protocols (Newscast, Cyclon) provides an always changing subset of the view from where to pick.
Peer Sampling

- Random Peer Sampling protocols (Newscast, Cyclon) provides an **always changing** subset of the view from where to pick

```
A B C D E F G H I L M
```

- Nodes keep a minimal (fixed!) subset of the node in the view

- The view adapts to churn
General Structure

- **select_peer**: selects a peer from the local view

- **select_to_send**: selects some entries from the local view

- **select_to_keep**: merges the received information to the local view, eliminates the duplicates and selects a subset of the resulting view which defines the new local view

A peer sampling service is defined by specialised these operations

```plaintext
active behaviour()

p = select_peer()
sent = select_to_send()
recv = send(p, sent)
view = select_to_keep(view, recv)
```

```plaintext
passive behaviour()

recv = receive(q)
sent = select_to_send()
view = select_to_keep(view, recv)
```
Partial view

- The peer sampling service keeps in the partial view information about the age and the contact point.
- Age of the partial view is increased/decrease at every cycle.

<table>
<thead>
<tr>
<th>ID</th>
<th>Contact point (IP:port)</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>192.168.0.44:4403</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>127.0.0.1:3463</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>8.9.0.42:5389</td>
<td>0</td>
</tr>
</tbody>
</table>

Partial view of size 3
Random Peer Sampling

Choose a peer **at random** from the entire node population

- It’s a core peer sampling service for almost any gossip applications
- Two popular protocols
  - Newscast
  - Cyclon


Newscast

- **SelectPeer**: selects a peer at random from the local view

- **SelectToSend**: selects all the descriptors in the local view + the descriptor of the actual peer
  - more recent descriptors have higher time-stamps

- **SelectToKeep**: keeps the most recent descriptors
Newscast

B’s view

<table>
<thead>
<tr>
<th>ID</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>9</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
</tr>
<tr>
<td>E</td>
<td>16</td>
</tr>
</tbody>
</table>

Diagram:

- B connects to A, C, and E.
- E connects to D and F.
- C connects to E.
Newscast

B’s view

<table>
<thead>
<tr>
<th>ID</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>9</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
</tr>
<tr>
<td>E</td>
<td>16</td>
</tr>
</tbody>
</table>

E’s view

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>7</td>
</tr>
<tr>
<td>D</td>
<td>10</td>
</tr>
<tr>
<td>G</td>
<td>14</td>
</tr>
</tbody>
</table>

1. B picks a random node from the view
Newscast

1. B picks a random node from the view
2. B and E exchanges views + their fresh descriptors
1. B picks a random node from the view
2. B and E exchanges views + their fresh descriptors
3. B and E keep the c=3 freshest links
Newscast

Resulting graph after the iteration

<table>
<thead>
<tr>
<th>ID</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>20</td>
</tr>
<tr>
<td>G</td>
<td>14</td>
</tr>
<tr>
<td>C</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>12</td>
</tr>
<tr>
<td>B</td>
<td>20</td>
</tr>
<tr>
<td>G</td>
<td>14</td>
</tr>
</tbody>
</table>
Newscast

• Robust to node and link failure and dynamism (churn) and scalable

• Newscast approximates small-world networks, with high clustering coefficient (CC) and small average path length

• High Clustering coefficient is bad for:
  • Flooding, due to the amount of redundant messages
  • Robustness, large clusters may be weakly connected to the rest of the network

CC of a node = quantifies how close the neighbours are to being a clique
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SelectPeer</strong></td>
<td>Select a random peer ( p ) from view</td>
</tr>
<tr>
<td><strong>SelectToSend</strong></td>
<td>Select ( L ) entry of own view + peer descriptor</td>
</tr>
<tr>
<td><strong>SelectToKeep</strong></td>
<td>Replace peer’s view with the one received from ( p ) (discard duplicates)</td>
</tr>
</tbody>
</table>

- Cyclon better approximates a random graph
- Small clustering coefficient
- Small average path length
- Connectivity is always guaranteed
Cyclon Example

Fig. 1. An example of shuffling between nodes 2 and 9. Note that, among other changes, the link between 2 and 9 reverses direction.

The protocol is extremely simple: each peer knows a small, continuously changing set of other peers, called its neighbors, and occasionally contacts a random one to exchange some of their neighbors. More formally, each peer maintains a neighbor list in a small, fixed-sized cache of $c$ entries (with typical value 20, 50, or 100). A cache entry contains the network address (i.e., IP address and port) of another peer in the overlay. Each peer $P$ repeatedly initiates a neighbor exchange operation, known as shuffle, by executing the following six steps:

1. Select a random subset of $\ell$ neighbors (1 $\leq \ell$ $\leq c$) from $P$'s own cache, and a random peer, $Q$, within this subset, where $\ell$ is a system parameter, called shuffle length.
2. Replace $Q$'s address with $P$'s address.
3. Send the updated subset to $Q$.
4. Receive from $Q$ as subset from more than $\ell$ of $Q$'s neighbors.
5. Discard entries pointing to $P$, and entries that are ready in $P$'s cache.
6. Update $P$'s cache to include all remaining entries, by firstly using empty cache slots (if any), and secondly replacing entries among the ones originally sent to $Q$.

On reception of a shuffling request, peer $Q$ randomly selects a subset of its own neighbors, of size no more than $t$, sent to initiating node, and executes steps 5 and 6 to update its own cache accordingly.

Figure 1 presents a schematic example of the basic shuffling operation.

2.2. Enhanced Shuffling

CYCLON employs an enhanced version of shuffling, that, as we shall show in subsequent sections, among other things improves the quality of the overlay.
Cyclon Convergence

Starting from a 100k nodes chain graph, Cyclon converges to the average path length of a random graph in few cycles.
Cyclon Robustness

Fig. 8. Tolerance to node removal, as a function of the cache size.

Note that the graph for the experiment with cache size 100 is practically a flat line. That is, for 100,000 nodes and cache size 100, the overlay created is so robust, that no matter how many nodes are removed, the remaining ones remain connected in a single cluster.

The effect of the cache size on the overlay’s robustness is shown in Fig. 8. We carried out 100 experiments, with cache sizes 1, 2, ..., 100, and for each of them we determined the percentage of random nodes needed to be removed in order to partition the overlay. It can be seen that there is a critical value of the cache size around 11. Overlays with smaller cache sizes exhibit significantly worse behavior with respect to robustness. On the other hand, overlays with cache size over 85 or 90, are almost impossible to partition, no matter how many nodes are removed.

It is important to point out that the results presented in this and the previous section, suggest that CYCLON is capable of repairing an overlay after a serious disaster, a property often referred to as self-healing behavior. This comes as a consequence of the following two acts. First, the overlay has proven to be highly resilient to large-scale node failures. Second, once such a massive failure has occurred, the surviving nodes quickly strengthen the connectivity among themselves by replacing links to dead nodes with valid links in a timely manner.

7. BANDWIDTH CONSIDERATIONS

Due to the periodic behavior of gossiping, the price of maintaining a robust overlay may inhibit a high usage of network resources (i.e., bandwidth). In the case of a 100K nodes, cache size should be larger than 11 for good robustness.
Topology Management

- Ring
- Mesh
- Star
- Fully Connected
- Line
- Tree
- Bus
Overlay Network

- **Overlay network**: a network built on top of another underlying network
  - Nodes in the overlay network are connected by logical links
  - Each overlay connection can correspond to a path in the underlying network.
- Peer-to-peer networks are overlay networks built on top of the Internet
  - DHT
What is a Topology

• The set of all nodes and their neighbourhood represent a network with a given **topology**

• The best way to visualise a topology is to think to them as a graph
  • if node A has node B in its view, it exits an edge from A to B in the graph of the network

• The topology defines the properties of the network
  • random vs small scale
  • how information percolates in the network
Toplogy Example

With gossip protocols we can manipulate the view of the node, therefore the network topology.
A node dynamically organises its local connection, thereby affecting the topology of the whole overlay network.
Fixed and variable topologies

• Dissemination and Aggregation considered fixed overlay topology, in which the connections of the network never changes

• In random peer sampling (especially with Cyclon) we aimed to a random graph topology

It is possible to modify the topology on demand, according to one’s needs?
Pirate Illegal Movie Download
Gossip Protocol

• Let’s suppose I’m downloading Batman vs Superman and I want to connect to the nodes that are downloading the same movie

• Node reorganise the topology such that their neighbours are downloading the same movie
Pirate Illegal Movie Download
Gossip Protocol

let’s add a field to the node entry in the partial view

<table>
<thead>
<tr>
<th></th>
<th>192.168.0.44:4403</th>
<th>2</th>
<th></th>
</tr>
</thead>
</table>

SelectPeer

SelectToSend

SelectToKeep
Pirate Illegal Movie Download
Gossip Protocol

let’s add a field to the node entry in the partial view

**SelectPeer**

select a peer (Q) with my very same movie from the partial view. If none available, select random

**SelectToSend**

**SelectToKeep**
Pirate Illegal Movie Download
Gossip Protocol

<table>
<thead>
<tr>
<th>SelectPeer</th>
<th>select a peer (Q) with my very same movie from the partial view. If none available, select random</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelectToSend</td>
<td>send at least size/2 entries from my partial view to Q. Preference to the entries having Q’s movies</td>
</tr>
<tr>
<td>SelectToKeep</td>
<td>let’s add a field to the node entry in the partial view</td>
</tr>
</tbody>
</table>

| 1 | 192.168.0.44:4403 | 2 |
| **SelectPeer** | select a peer (Q) with my very same movie from the partial view. If none available, select random |
| **SelectToSend** | send at least size/2 entries from my partial view to Q. Preference to the entries having Q’s movies |
| **SelectToKeep** | keep the freshest entries that match my movie. If not enough to fill up my view, keep the freshest |

let’s add a field to the node entry in the partial view

| 1 | 192.168.0.44:4403 | 2 |
Pirate Illegal Movie Download
Gossip Protocol

- Now I’m happy and I can continue to download the movie.
Pirate Illegal Movie Download
Gossip Protocol

• Now I’m happy and I can continue to download the movie..
• .. but what if later I want to download Age of Ultron?
Now I’m happy and I can continue to download the movie.

.. but what if later I want to download Age of Ultron?

The network is disconnected and I cannot reach the other peers downloading AoU!
A Connectivity Problem

- Topology that connects nodes according to a criteria are useful, but also is maintaining connectivity

- The solution is that each node runs two or more protocols
  
  - some to connect with nodes according to an “application” criteria (in our example, download the right movie)
  
  - the other to remain connected to the network

- For its characteristics **Cyclon** is often the best protocol to keep connectivity
  
  - realises a topology that maintains a random graph
Layered Gossip

- Multiple protocols run on the same node
- Each protocol keeps its own view

Layer 1

Layer 0
Vicinity


- Layered approach

- Define a topology of **semantically close** neighbours

- Exploits Cyclon to maintain connectivity and for random peer sampling
Vicinity

let’s define close according to proximity

SelectPeer

SelectToSend

SelectToKeep
Vicinity

let’s define close according to proximity

|   | 192.168.0.44:4403 | 2 | 14,54 |

**SelectPeer**
Select a random peer (P) from the underlying Cyclon protocol

**SelectToSend**

**SelectToKeep**
Vicinity

let’s define close according to proximity

|   | 192.168.0.44:4403 | 2   | 14,54 |

**SelectPeer** Select a random peer (P) from the underlying Cyclon protocol

**SelectToSend** Order your own partial view and the cyclon view according to P position. Select the closest peers

**SelectToKeep**
Vicinity

let’s define close according to proximity

SelectPeer
Select a random peer (P) from the underlying Cyclon protocol

SelectToSend
Order your own partial view and the cyclon view according to P position.
Select the closest peers

SelectToKeep
Order the received peers according to your own position. Replace the most distant peers in the partial view
Vicinity

- Nodes are **ordered** according to their distance with respect to the target

q’s active behaviour()

\[
\begin{align*}
p &= \text{select_peer()} \\
sent &= \text{select_to_send}(p, \text{view}, q.\text{cyclon}) \\
recv &= \text{send}(p, sent) \\
view &= \text{select_to_keep}(\text{view}, recv, q.\text{cyclon})
\end{align*}
\]

p’s passive behaviour()

\[
\begin{align*}
recv &= \text{receive}(q) \\
sent &= \text{select_to_send}(q, \text{view}, p.\text{cyclon}) \\
view &= \text{select_to_keep}(\text{view}, recv, p.\text{cyclon})
\end{align*}
\]
Beyond Vicinity: T-man

- Vicinity exploits distance between items to provide a ranking of peers

- A generalisation of Vicinity would be to allow any function that provides a ranking

- Such generalisation exists and it’s called T-man
  - Let the user define its own ranking systems
  - Vicinity can be seen as a special case of T-man

T-man

- Objective: build an overlay network (or graph) from scratch filling the partial views of nodes properly
  - It can be even a structured one, like a DHT (we see an example later)
  - Initially the views of the nodes can be whatever (even empty). Only requirement is to have an underlying random peer sampling service
    - Exploits Cyclon as underlying random peer sampling service
  - T-man exploits a ranking function that is a generalisation of the ordering by distance; can do what a simple distance cannot do (DHT)
T-man

• T-man initial graph evolves to a given overlay, which is defined by means of the ranking function and a number $K$
  
  • Node[] rank(node [], internal_state)

  • Node exchanges their view and keep the first $K$ peers that are ranking higher
**T-man**

- T-man initial graph evolves to a given overlay, which is defined by means of the ranking function and a number $K$
  
  - Node[] rank(node [], internal_state)
  
  - Node exchanges their view and keep the first $K$ peers that are ranking higher

---

**Fig. 3:** Evolution of Torus Computation at different Supersteps

**3 Evaluation**

To evaluate the effectiveness of our approach we conducted a set of three experiments. In Section 3.1 we validate the framework, in Section 3.2 we evaluate the scalability, and finally Section 3.3 presents a multi-layer solution for graph partitioning.

**3.1 Torus overlay**

The aim of this experiment is to show that the layered architecture of the Telos framework can be a suitable tool to port concepts from the massively distributed systems to large graph processing. As a proof-of-concept we built an experiment that organises the vertices and the edges of a graph into a torus shaped overlay. The implementation considers a two layers approach. The bottom layer implements a random protocol, and the upper layer implements a ranking protocol. We tested the implementation on a graph made of $20^K$ vertices. The results in Figure 3 show the evolution of the graph in real time. The topology recalls the shape of a torus already at super-step 10. At super-step 20 no edges connect "distant" vertices in the torus. This experiment shows that, if properly instrumented, Telos can correctly manage multiple layers and can build the requested topology in a fewer number of super-steps.

**3.2 Scalability**

This experiment evaluates how the Telos framework manages larger input graphs when increasing the number of cores involved in the computation. For this ex-
T-man

- T-man initial graph evolves to a given overlay, which is defined by means of the ranking function and a number $K$
  - $\text{Node}[] \ \text{rank} (\text{node} [], \ \text{internal\_state})$
  - Node exchanges their view and keep the first $K$ peers that are ranking higher

Fig. 3: Evolution of Torus Computation at different Supersteps

Ranking protocols. These protocols are widely exploited in gossip frameworks to create and manage topology overlays [8] [20]. The overlays are created and maintained according to a ranking function which measures the similarities between two vertices. In Telos we implemented a generic ranking protocol able to take as input the ranking function. It dynamically keeps in the neighbourhood of each vertex the more similar vertices according to the user defined ranking function. As an example, if every vertex is represented as a point in a two dimensional space and the similarity metrics is the euclidean distance, the ranking protocol eventually keeps in the neighbourhood of each vertex the $k$ closest vertices.

3 Evaluation

To evaluate the effectiveness of our approach we conducted a set of three experiments. In Section 3.1 we validate the framework, in Section 3.2 we evaluate the scalability, and finally Section 3.3 presents a multi-layer solution for graph partitioning.

3.1 Torus overlay

The aim of this experiment is to show that the layered architecture of the Telos framework can be a suitable tool to port concepts from the massively distributed systems to large graph processing. As a proof-of-concept we built an experiment that organises the vertices and the edges of a graph into a torus shaped overlay. The implementation considers a two layers approach. The bottom layer implements a random protocol, and the upper layer implements a ranking protocol. We tested the implementation on a graph made of $20K$ vertices. The results in Figure 3 show the evolution of the graph in real time. The topology recalls the shape of a torus already at super-step 10. At super-step 20 no edges connects "distant" vertices in the torus. This experiment shows that, if properly instrumented, Telos can correctly manage multiple layers and can build the requested topology in a fewer number of super-steps.

3.2 Scalability

This experiment evaluates how the Telos framework manages larger input graphs when increasing the number of cores involved in the computation. For this ex-
T-Man

Node[] rank(node [], internal_state)

SelectPeer

SelectToSend

SelectToKeep
T-Man

Node[] rank(node [], internal_state)

SelectPeer

Select p random among the first Z
rank(view, my_state)

SelectToSend

SelectToKeep
T-Man

Node[] rank(node [], internal_state)

SelectPeer
Select p random among the first Z
rank(view, my_state)

SelectToSend
send to p the first m elements of
rank(view+myself, p.state)

SelectToKeep
T-Man

Node[] rank(node [], internal_state)

SelectPeer
Select p random among the first \( Z \) rank(view, my_state)

SelectToSend
send to p the first \( m \) elements of rank(view+myself, p.state)

SelectToKeep
Replace local view with rank(view+p.view, my_state)
T-man performance

N=2000 nodes
T-man Case Study: Coverage Peer Sampling

- Position dissemination in distributed online games.
- Players are usually interested in events that happen in their AOI
  - AOI: the area of the virtual world which is of interest for a generic participant
- Events are delivered to players according to the game architecture client/server
T-man Case Study: Coverage Peer Sampling

• Position dissemination in distributed online games.

• Players are usually interested in events that happens in their AOI
  • AOI: the area of the virtual world which is of interest for a generic participant

• Events are delivered to players according to the game architecture
  client/server  distributed

SERVER

client/server distributed

SERVER
T-man Case Study: Coverage Peer Sampling

- Position dissemination in distributed online games.
- Players are usually interested in events that happen in their AOI
  - AOI: the area of the virtual world which is of interest for a generic participant
- Events are delivered to players according to the game architecture
  - client/server
  - distributed
  - gossip protocol
The problem

- When there are many players their AOI overlaps, and they can communicate each other the events that happens in the overlapping area. Two issues:
  - I want to keep connection with a **small, fixed** amount of other player (view)
  - Players are moving, so the view must be **updated** over time
Ranking function

- Definition of a function that ranks the peers according to their overlapping of the AOI

- Two functions
  - score-based
  - greedy-based
The protocol

- Layered architecture
- CPS on the top
- Cyclon on the bottom

- Each peer is identified by a descriptor that contains the position of the player in the virtual world

- **SelectToSend**
  - Rank peers according to the coverage of the receiver position

- **SelectToKeep**
  - Rank peers according to the coverage of my position
Lesson Take Away

• Gossip can be used to emulate the selection of a random peer in a completely decentralised network

• In decentralised gossip protocols connective is a huge issue

• Two-layer gossip architectures are very convenient. Bottom layer (i.e. Cyclon) provides connectivity and (random) peer sampling, upper layer focuses on the application
Lesson Take Away

• Definition of a specific ranking function allows to build from scratch complex overlays

• It is possible to combine overlay construction and peer sampling to support complex applications
Further Readings


