Large-Scale Distributed Systems - LSDS 2015

Gossip-based dissemination, Peer Sampling Service

October 1, 2015

1 Introduction

We recommend that you use your notes and slides for Lecture 2 and Lecture 3 while working on this project.

There are four lab sessions allocated for this project. Your first task is to implement the gossip-based dissemination protocol explained in class and evaluate its performance in terms of delays when *anti-entropy* and *rumor mongering* are used separately or combined. It is assumed that each node knows the entire system, which is not realistic for a large-scale dynamic system. Hence, your next task is to implement the *Peer Sampling Service* and evaluate its performance, behavior, and its interaction with the gossip-based dissemination protocol previously implemented.

1.1 Report and grading

It is mandatory to submit a report of your work to the assistant; this report will be taken into account for the final grading (see slides of Lesson 1).

Your report should contain:

- *A text report* – including the graphs plotted during the lab sessions and the associated explanations. Remember that a graph without a corresponding explanation or an assertion that is not conveyed by a graph, does not have much interest. Write your report in **English**. Only **PDF files** will be accepted.
- *Your code* – formatted in a proper way, with indentations and with appropriate comments. Submit only the most advanced version of the code you wrote for each part of the assignment. Only `.lua`, `.gp` (gnuplot), `.sh`, etc. files for the code will be accepted.[1]

Deadline: Send the report to the assistant and the instructor by **October 29, 2:00 pm**. Use the following email subject: `LSDS - assignment 1 - FIRSTNAME LASTNAME`.

Grading: The teaching assistant will grade your progress, the correctness, and the clarity of both the code and the report. The content is what matters most, but the presentation also has an impact on the final grading.

[1] You may use another language for processing the logs, as long as it is an interpreted and widely-available scripting language, such as python, ruby, perl, etc.

2 First Session: Gossip-Based Dissemination

Implement the *anti-entropy* and *rumor mongering* mechanisms presented in class. For the sake of simplicity, assume that:

- Each node propagates a single message.
- A node can be in one of the following two states: "infected" or "not infected"; in your implementation, we recommend that you represent the state of a node using a string (e.g.,"yes" for infected, and "no" for not infected).
- We are interested in the following metrics:
  - The speed of message propagation, represented by the cumulative number of infected nodes as a function of time.
  - The completeness of message propagation: do all nodes eventually get the message?
  - The number of duplicates: how many messages are redundant for each node? (This only applies to *rumor mongering*.)
- Each node knows the complete network and this network is not dynamic. Each node chooses a random node from `job.nodes` to gossip with; `job.nodes` is of type `HEAD` with full network size (i.e., set SplayWeb - Advanced Options - Number of splayds in list to 0).
- For anti-entropy, consider push-pull updates: if any of the two nodes is infected, then both nodes get infected after the update.
- For rumor mongering, consider the "Hops To Live (HTL)" stopping condition.

2.1 Anti-entropy

Algorithm 1: Anti-entropy (push-pull)

Constants
| *period*: integer, set to 5 seconds
| *job.nodes*: set of all nodes in the system (provided by `"require splay.base"`)

Variables
| *infected*: boolean, initially `false`
| *current cycle*: integer, initially 0 (used for logging purposes)

selectPartner()
| `return` a random peer from *job.nodes*

selectToSend()
| `return` *infected*

selectToKeep(received)
| `return` *infected* $\vee$ received

Task 2.1.1: Implement the anti-entropy mechanism (1) and test the dissemination using a gossiping period of 5 seconds and a population of 40 peers. As a starting point for your implementation, use the code skeleton provided on ILIAS. This code skeleton contains:

- The code for also running the job locally, in the provided virtual machine;
- A `terminator` function that kills the nodes after a predefined running time;
- A main function that "desynchronizes" the nodes such that the cycles do not start at the same time on all nodes, as would happen in a real-world deployment.

Task 2.1.2: For each node, when receiving a message, print the current time (in the skeleton, the source already prints this message as it is the first infected). Ignore duplicates (i.e., nodes that are contacted while they are already *infected*).

Task 2.1.3: Write a script that parses the log of your application and prints the total number of infected peers as a function of time. Consider the following log:

```
16:52:30.02 (4) i_am_infected
16:52:31.33 (1) i_am_infected
16:53:03.21 (3) i_am_infected
16:53:44.58 (2) i_am_infected
16:54:34.41 (5) i_am_infected
```

The script should produce (first column = time in seconds, second column = number of notified peers, third column = the proportion of notified peers):

```
0    1   0.2
1    2   0.4
33   3   0.6
74   4   0.8
124  5   1
```

Task 2.1.4: Write a Gnuplot script and produce a plot that presents on the abscissa (Ox axis) the time and on the ordinate (Oy axis) the proportion of peers that have been notified. An example of such a plot is presented in Figure 1. Use the Gnuplot script skeleton provided on ILIAS; use the same Gnuplot script to produce all plots required in the remainder of this section, for easier comparison with the initial anti-entropy plot.

2.2 Rumor mongering

When comparing the performance of rumor mongering and anti-entropy, we face the following problem: anti-entropy uses cyclic exchanges, and during each cycle, each node contacts another node in order to update its information. In rumor mongering, a node *immediately* sends a message it receives for the first time to f of its neighbors, up to a point where the stopping condition becomes true (e.g., reaching a *HTL*).

It is therefore difficult to compare the two protocols with respect to the time required to propagate some message.

To be able to compare both mechanisms, we will synchronize the rumor mongering to also use cycles. Note that cycles do not need to be synchronized across nodes, nor they need to be synchronized with the cycles of the anti-entropy mechanism when we will combine them.

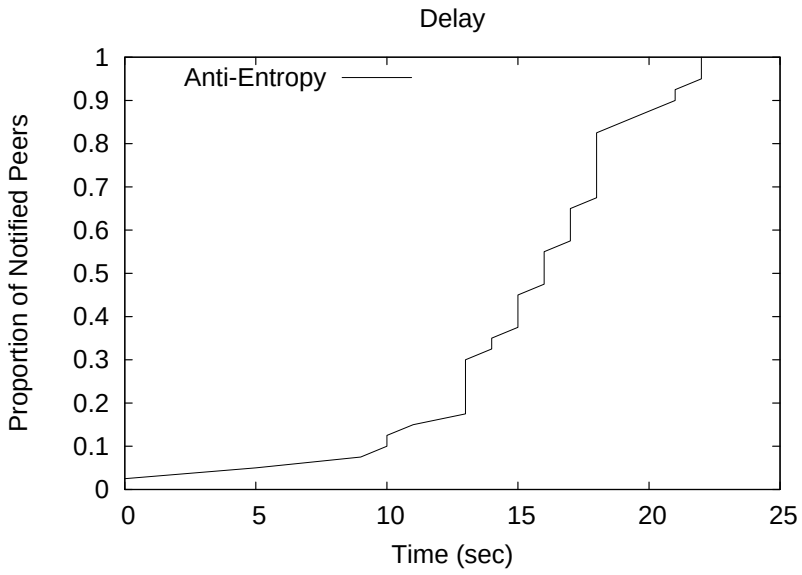

Figure 1: An example of the plot displaying the proportion of notified peers vs. time for anti-entropy.

Every time a node receives a message that it did not know about (it was not already *infected*), instead of sending directly the message to f randomly chosen other nodes, it will delay this sending by *buffering* the message until the next periodic forwarding. The process is illustrated by Figure 2. Every time a new message is received, it is buffered for being sent at the next rumor-mongering-propagation period. A process, running at the same frequency as the anti-entropy (e.g., every 5 seconds), is responsible for periodically sending this buffer to other nodes.

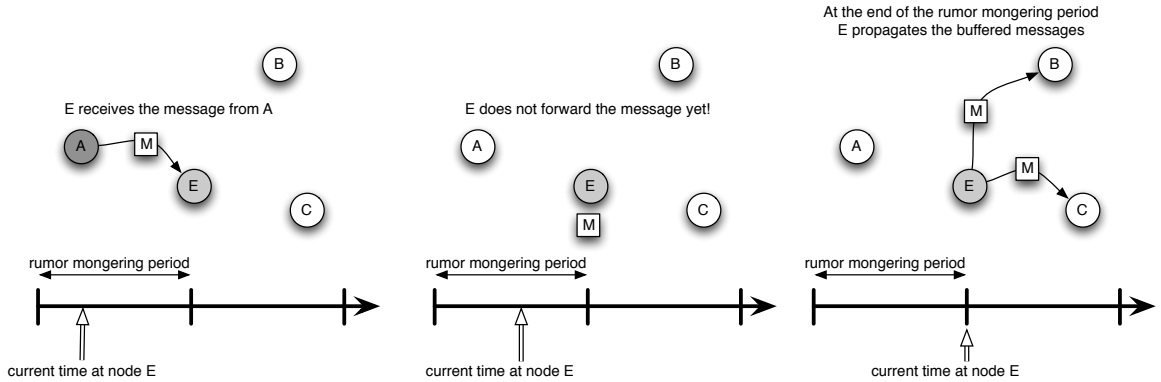

Figure 2: Synchronized rumor mongering: principle

Task 2.2.1: Implement the rumor mongering mechanism presented in Algorithm 2 and test the dissemination using a gossiping period of 5 seconds and a population of 40 peers, using values $f = 2$ and $HTL = 3$. You can adapt the skeleton from the anti-entropy task.
Task 2.2.2: Output, for each node, when receiving a message, the current time (don't

forget the first infected node, the source!). Do not ignore duplicates: output a message (e.g., `duplicate_received`) to the log when one is received.

Task 2.2.3: Experiment with several values of f and HTL. How many duplicates do you need in order to reach 39 of your 40 nodes? You can present these results as a table, or as a plot.

Task 2.2.4: Adapt the gnuplot script from Task 2.1.4 and the parsing script from Task 2.1.3, so that on the same plot you can compare rumor mongering and anti-entropy. You can present several evolutions for several values of f and HTL on the same plot.

Algorithm 2: Rumor mongering

Variables

infected: boolean, initially `false`
job.nodes: set of all nodes in the system
buffered: boolean, initially `false`, describes if the node needs to propagate the infection at the next propagation period
buffered_h: integer, hops-to-live of the message to forward (only valid if *buffered* is `true`)

Constants

f: integer, how many nodes to forward the message to
HTL: integer, hops to live

notify(h)

```
/* invoked by a remote node to infect the current node                    */
/* buffer only if (1) not infected or (2) rumor with larger HTL           */
```

if $\neg$ *infected* **then** $infected$ = `true`
if $(\neg$ *buffered*$)\vee(buffered \wedge ((h-1) > buffered\_h))$ **then**
    buffered = `true`
    buffered_h = $h-1$

periodic thread (every 5 seconds): forward()

if (buffered) **then**

```
    /* note the following should ideally exclude the source               */
```

    destinations = f random nodes from *job.nodes*
    foreach *neighbor* $\in$ *destinations* **do**
        invoke notify(*buffered_h*) on node *neighbor*

```
/* the node propagates the message only once                              */
```

buffered = `false`

2.3 Combining anti-entropy and rumor mongering

Combine both protocols to work at the same time, and update your gnuplot so as to display on the same graph:

- the anti-entropy delays for propagation;
- the rumor mongering delays for propagation;
- the combination of both.

3 Implementing the Peer Sampling Service

During this session, implement the generic peer sampling service presented in Algorithm 3.

You will need to operate on tables for storing the view. Remember the ordering of the elements on the table matters, hence it shall be used as an array (see first lab session). Amongst the useful library calls, you will have to use, are Splay's `t = misc.shuffle(t)` and Lua's `table.insert(some_table)` and `table.sort(some_table,comparison_function)`, whose documentation is to be found in the books or on the Internet Lua documentation.

Implementing the Peer Sampling generic protocol requires some operation on tables. It is highly recommended that each such operations are unit-tested in a separate Lua file before being used as part of a Splay deployed job.

Task 3.1: Implement the selectToSend() and selectToKeep() functions in a separate test file and test them using some examples. Fake nodes/age pairs can be created as follows:

```
view = {}
view[#view+1] = {peer={ip="127.0.0.1",port=4255},age=4,id=1}
view[#view+1] = {peer={ip="127.0.0.2",port=1234},age=30,id=2}
view[#view+1] = {peer={ip="127.0.0.3",port=55},age=2,id=3}
...
```

Task 3.2: Implement the Peer Sampling Service and test it using 50 peers, a view size of $c = 8$, $exch = 4$, and the following values of H and S: $(H = 0, S = 0)$, $(H = 4, S = 0)$, $(H = 0, S = 4)$ (as seen in class).

To initialize the view, use a random pick in the complete list of peers given by Splay (do not include the current node!).

Task 3.3: In order to assess the properties of randomness, and the non-partition of the created graph, you will use the following **provided scripts**:

- `pss_check_partition.rb` outputs the number of partitions in the random graph created by the peer sampling service. More than one partition means the network is not connected, which is bad.
- `pss_check_indegrees.rb` outputs the cumulative distribution of indegrees for all nodes in the system. Use a gnuplot script similar to the one used for Task 2.1.4 to plot the cumulative indegree distribution for all nodes for the various values of H and S.
- `pss_check_clustering.rb` outputs the cumulative distribution of the clustering factor for all nodes. Similarly, use a gnuplot script to plot it for the various values of H and S.

In order for the scripts to work, you will need to use the following output format to output the content of the view on each peer:

`VIEW_CONTENT [peer id] [neighbor id]*`

for instance, peer 31 with neighbors 12, 5, 9, 1, 27, 22 in its view should output:

`VIEW_CONTENT 31 12 5 9 1 27 22`

(and nothing else).

Also, it is necessary for all nodes to output their view nearly at the same time. Output the content of the view in a separate periodic thread that runs at a larger period than the

active thread of the PSS itself. To have this thread scheduled at approximately the same time on all nodes (to get a *snapshot* of the system), its creation should be the first action taken on each node.

The scripts are available on Ilias. Feel free to modify them to check other properties (include the modified scripts in the source package you will email to the assistant for grading, and explain the modifications you did).

4 Plugging everything together

Last, you will plug things together and see how the dissemination delay is impacted by the peer sampling service *vs.* the use of the global list of all peers in the system.

It is recommended that you let the peer sampling run for c cycles at least before engaging in dissemination, or you will not be able to witness differences.

Task 3.1: Implement a getPeer() function in the peer sampling service. Implement the selectPartner() function of the dissemination protocol to allow either selecting from the complete list or selecting from the Peer Sampling Service.

Task 3.2: Using only the combined rumor-mongering and anti-entropy protocol, plot the delay of reception of one message (boolean infection) using the PSS with various values of H and S, and using the complete list given by Splay. Also, take note of the duplicates count and report it in your report.

5 Optional tasks

These tasks are optional, feel free to implement, experiment and have fun with them. Include your results and code in the package for bonus grading (optional tasks are not required to get a full mark though – completing other tasks is enough).

- Using the trace file "massive_fail.churn_trace" on Ilias, evaluate the behavior of the peer sampling service when half of the network fails (the trace uses a set of 60 peers, and 30 peers fail after 3 minutes).
- Implement multi-message dissemination: nodes do not exchange a boolean value but a set of messages. Messages are sent by any node and all nodes are interested in all messages.
- Make the dissemination and Peer Sampling Service combination work under churn. Use the trace file "continuous.churn_trace" on Ilias. This trace has a maximal number of peers of 80, and lasts for 20 minutes.
- Experiment with the other stop conditions introduced in class for stopping rumor mongering propagation.

Algorithm 3: The Peer Sampling Service: simplified generic implementation, push-pull

Variables
- *view*: set of nodes of size c, initialized from random list of size c from Splay
- **Each entry of the view q contains:**
  - $q.age$: age of the peer item
  - $q.peer$: a Splay peer structure (ip,port) for calling RPCs on the peer
  - $q.id$: for logging convenience, the original position of the node.

Constants & Parameters
- $exch$: integer, how many peers are exchanged at each round
- H: integer $\in [0 \dots exch - S]$, healer parameter
- S: integer $\in [0 \dots exch - H]$, shuffler parameter
- $SEL \in \{rand, tail\}$: partner selection policy

selectPartner()
- **if** $SEL = rand$ **then** `return` random peer from *view*
- **else if** $SEL = tail$ **then** `return` peer from *view* with largest age

selectToSend()
- toSend = `new` buffer of size *exch*
- toSend.append(job.me,0) ; `// add current node with age 0`
- shuffle(*view*) ; `// randomize before applying H/S rules`
- move oldest H items to the end of *view* ; `// apply healer rule`
- toSend.append(*view*.head($exch - 1$))
- `return` toSend

selectToKeep(received) (received is a buffer of (peer,age) entries)
- *view*.append(received)
- remove duplicates from *view*
- remove the min(H,*view*.size-c) oldest items from *view*
- remove the min(S,*view*.size-c) head items from *view*
- remove max(0,*view*.size-c) random items from *view*

periodic thread (every 5 seconds): activeThread()
- partner = selectPartner()
- buffer = selectToSend()
- received = `remotely call` passiveThread(buffer) on partner
- selectToKeep(received)
- **foreach** $p \in view$ **do** $p.age = p.age + 1$

passiveThread() (called remotely)
- buffer = selectToSend()
- selectToKeep(received)
- `return` buffer