Large-Scale Distributed Systems
Project 1: Gossip-based dissemination, Peer Sampling Service

Laurent Hayez

October 29, 2015

Table of contents

1 Introduction

The aim of this project is to implement the gossip-based dissemination protocol seen in class. We will evaluate its performance in terms of delays when anti-entropy and rumor mongering are used separately or combined. The next part is devoted to the Peer Sampling Service (PSS) and its implementation. In the first part of the project, we assume that each node knows the entire system, but this assumption is not realistic. We will therefore evaluate PSS's performance, behaviour and its interaction with the two gossip-based dissemination protocols in order to understand how it would work in a real life deployment.

2 Gossip-Based Dissemination

In this section, the objective is to implement the anti-entropy and rumor mongering mechanisms. For the sake of simplicity, we made the following assumptions.

- Each node propagates a single message.
- A node is infected, or is not. To see if a node is infected, we used a string, “yes” if the node is infected and “no” if the node is not infected.

- We are interested in the following metrics:
  - the speed of message propagation, the number of infected nodes as a function of time;
  - the completeness of message propagation: do all nodes eventually receive the message?
  - the number of duplicates: how many messages are redundant for each node (only for 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.
- For anti-entropy, we consider push-pull update: if any of the two nodes is infected, then both nodes get infected after the update.
- For rumor mongering, we consider the “Hops To Live (HTL)” stopping condition.

2.1 Anti-entropy

We will start with the anti-entropy mechanism, which implementation is presented in Listing 1.

Listing 1: Anti Entropy

```lua
function selectPartner()
  partner=nodes[math.random(1, #nodes)]
  if partner == nodes[job.position] then
    selectPartner()
  end
  return partner
end

function selectToSend()
  return infected
end

function selectToKeep(received)
  infect=nil
  if infected=="no"and received=="no" then
    infect = "no"
  else
    infect="yes"
  end
  return infect
end

function receive(state)
  old_infected = infected
  infected=selectToKeep(state)
  if infected=="yes" and old_infected=="no" then
    log:print("i_am_infected")
  end
  return infected
end

function gossip()
  gossip_partner=selectPartner()
  received = rpc.call(gossip_partner, {"receive",
   selectToSend()})
  receive(received)
end
```

We use a gossip period of 5 seconds and a population of 40 peers to test the implementation. The results are provided in Figure 1.

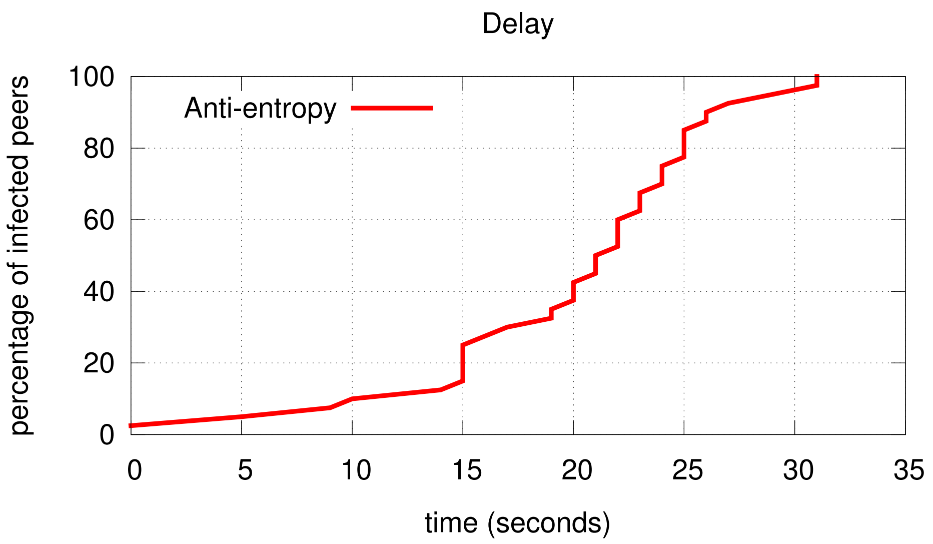

Figure 1: Proportion of notified peers in function of the time with anti-entropy

We see on the plot that at the beginning, the propagation is not efficient, and at 15 seconds, the peers are infected faster and faster, until approximately 90% of the peers are infected. At this moment, the last peers take a little bit more time to get infected. The growth is exponential. It is due to the fact that more and more peers are infected, thus more and more peers send the message `i_am_infected`.

2.2 Rumor mongering

We would like to compare the performances of anti-entropy and rumor mongering, but for that we have to adapt a bit the mechanism of rumor mongering. We will do the following to be able to compare both protocols: when a node receives a message for the first time, instead of immediatly sending it to f other nodes, it waits by buffering the message until the next period forwarding.

We provide an implementation of the rumor mongering mechanism in Listing 2 and we use a gossiping period of 5 seconds, a population of 40 peers with values $f = 2$ and $HTL = 3$.

Listing 2: Rumor mongering

```
------------------------------------------------------------------
function rm_notify(h)
    -- invoked by a remote node to infect the current node
    -- buffer only if (1) not infected or (2) rumor with larger
     HTL
  log:print("node "..job.position.." ("..infected..") was
     notified with hops "..h.." (HTL="..HTL..")")

  -- TODO: infect if necessary
  if infected == "yes" then
```

```
      log:print(os.date("%X").." duplicate_received")
    else
      infected = "yes"
      log:print("i_am_infected")
    end

    if (not buffered) or (buffered and ((h-1) > buffered_h)) then
      buffered = true
      buffered_h = h-1
    end

    if (h < HTL) or (buffered and ((h + 1) < buffered_h)) then
     buffered = true
     buffered_h = h + 1
    end
  end
-----------------------------------------------------------------

-----------------------------------------------------------------
function rm_activeThread()
  current_cycle = current_cycle + 1

  -- do I have to send something to someone?
  if buffered then
    log:print(job.position.." proceeds to forwarding to "..f.."
    peers")

    -- select nodes with misc.random_pick (excluding self from
    the random pick)
    copy_nodes = nodes
    table.remove(copy_nodes, job.position)
    selected_nodes = misc.random_pick(copy_nodes, f)

    -- sending rpc to all nodes
    -- we invoke the method rm_notify(h) with h=buffered_h
    for i, node in ipairs(selected_nodes) do
       rpc.call(node, {"rm_notify", buffered_h})
    end

    buffered = false
    buffered_h = nil
  end
end
-----------------------------------------------------------------
```

We now present in Figure 2 the number of duplicates needed to reach 39 out of 40 peers with different values.

We see on plot that the worst performance is when $f = 2$ and $HTL = 3$ while the best is when $f = 5$ and $H = 5$. Although there is no evident correlation between the values of HTL and f, we deduce from the algorithm that if we set a high value for f, there can be more redundance. This is a direct consquence of choosing f peers in the list of all peers: the highest the value of f, the more chances there is to choose some peers that have already been infected. Moreover, the system seems to be more efficient with $HTL = 5$ than lower values, but the respectively third and fourth best results are with $HTL = 2$, so we can't tell from this example that it is better to have a high HTL or not.

We have to put into perspective that we used a population of 40 peers, and a maximal time of 120 seconds, which clearly does not represent a real life network. For further

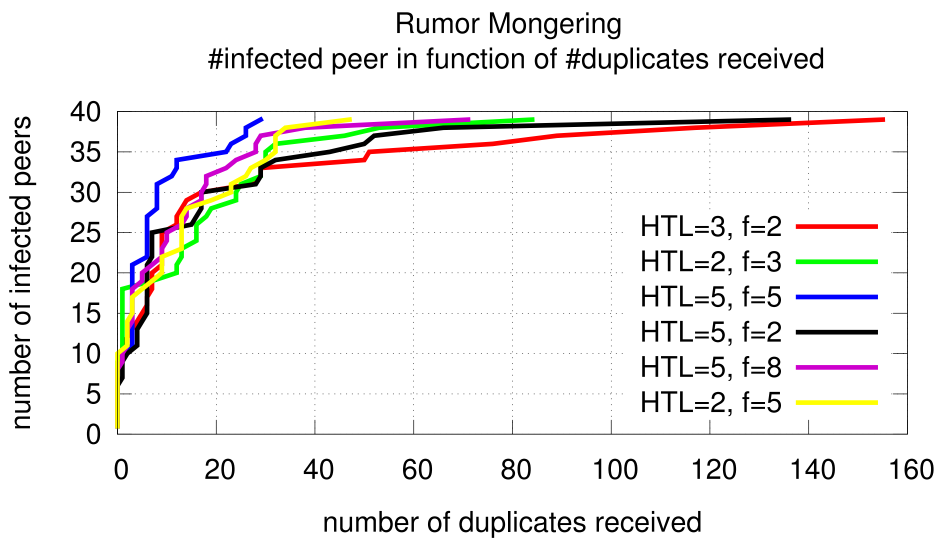

Figure 2: Number of duplicates to reach 39 out of 40 peers with different values of f and HTL

investigation, it would be better to launch computations on more peers for a longer time.

We compared anti-entropy and rumor mongering by plotting the evolution of the number of infected peers on the same plot (with several values of f and HTL). This is shown in Figure 3.

We see on this plot that as soon that $f > HTL$, the infection propagates faster with rumor mongering than with anti-entropy, but as soon as $f \leq HTL$ it takes more time to infect all the nodes with rumor mongering than with anti-entropy. We can't see a general relation between the increase of HTL or f and the increase of the propagation speed. For example, $HTL = 5$ and $f = 8$ is faster than $HTL = 5$ and $f = 2$, but $HTL = 5$ and $f = 2$ is slower than $HTL = 3$ and $f = 2$.

We can also combine both protocols to run at the same time, resulting in the plot shown in Figure 4.

On this example, the fastest is still rumor mongering with $HTL = 5$ and $f = 8$, although the combined version of rumor mongering and anti-entropy is not much slower. Both rumor mongering examples propagate faster than the combined version, and all propagate faster than anti-entropy alone. It is strange that the combined versions are not faster. Theorically, rumor mongering should seed the network at initiation and anti-entropy ensures that all nodes are infected. This might be due to the fact that we did not launch the computations in the same conditions. It is possible that for the combined version, the cluster was also used by other persons, resulting in slower computations, or simply we did not have luck for the random picks. It would certainly be more precise to launch for example 20 times the same script and then doing the average instead of considering just one example.

3 Peer Sampling Service

In this section, we provide an implementation for the Peer Sampling Service (PSS). We used a simplified generic push-pull implementation. This implementation is provided in

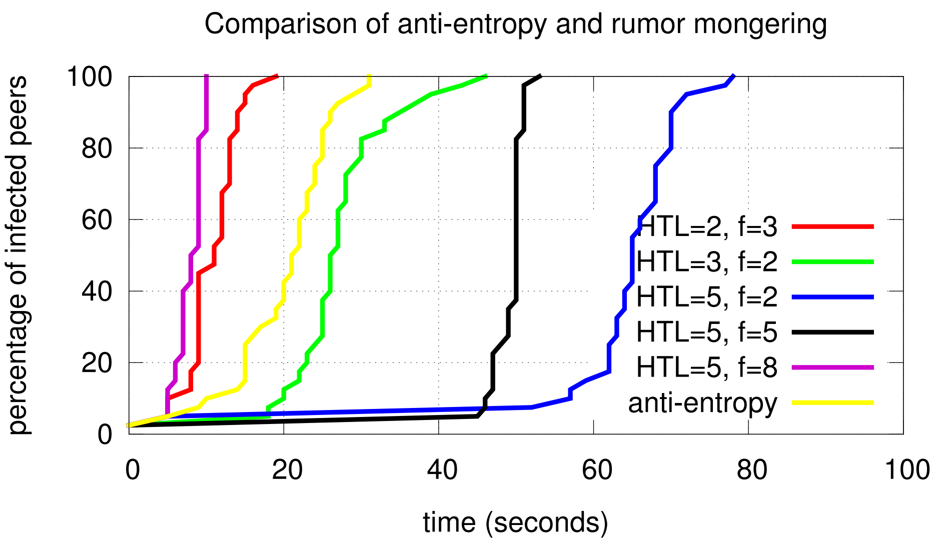

Figure 3: Comparison of the anti-entropy and rumor mongering mechanims for several values of f and HTL

the file `pss.lua`.

We tested the PSS implementation using the three Ruby scripts provided. All the tests were run with a population of 50 peers, a view size of $c = 8$, $exch = 4$ and the following values of H and S: $(H = 0, S = 0)$, $(H = 0, S = 4)$ and $(H = 4, S = 0)$.

1. With the `pss_check_partition.rb` script, we always got that the nework was connected when running the script.

2. With the `pss_check_indegrees.rb` script, we produced the plot shown in Figure 5. The objective of the PSS is to have a gaussian distribution of indegrees around c, where $c = 8$ in our case. For the Healer, we don't really see a gaussian distribution. We have a lot of variations (if we were to draw a tendancy curve, we would be tempted to take a sinusoidal one). For $H = 0$ and $S = 0$, it already looks a little bit more like a gaussian distribution, although is it not exactly centered in c. If we were to draw the tendancy curve of the Swapper, we would certainly obtain a curve similar to a gaussian, which is was the intended objective.

3. With the `pss_check_clustering.rb` script, we produced the plot shown in Figure 6. We see on the plot that the clustering factor is rather similar for the three different choices of H and S. The objective is to have a minimal clustering in order to maximize the randomness of the graph, which is best achieved with the Swapper. The clustering informally reprensents the fact that "my neighbours are neighbours themselves" (an analogy of this would be "my facebook friends are friends on facebook themselves").

   Mathematically speaking, we can see this graph as the cumulative distribution of the density functions f_1, f_2 and f_3 where f_1 (respectively f_2 and f_3) are the probability distribution of the clustering factor of the PSS with $(H = 0, S = 0)$ (respectively with $(H = 0, S = 4)$ and $(H = 4, S = 0)$). Thus if we denote by F_1 (respectively F_2 and F_3) the cumulative distribution of f_1 (respectively f_2 and

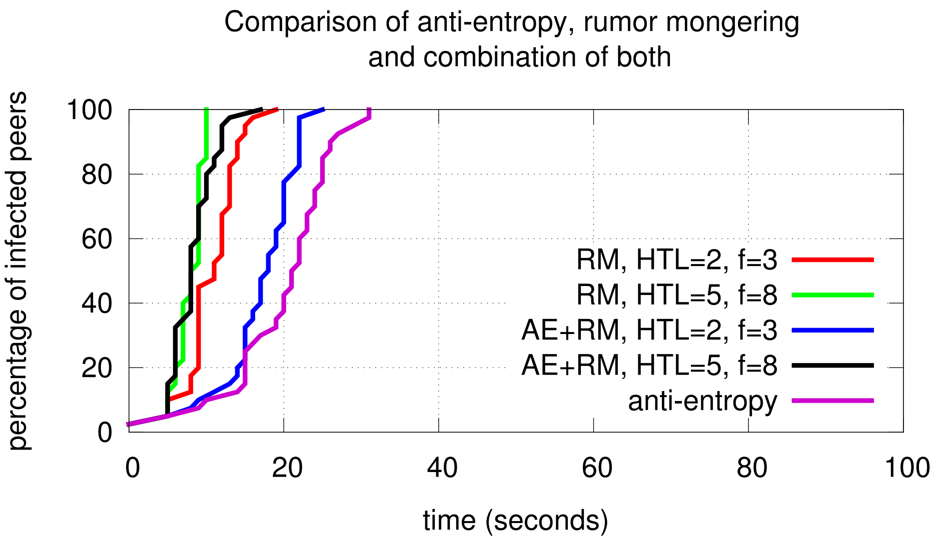

Figure 4: Comparison of the anti-entropy, rumor mongering and the combination of both protocols

f_2), we have that

$$F_1(a) = \int_{-\infty}^{a} f_1(x)dx$$

and it represents the proportion of peers that have a clustering factor lower or equal to a. Based on this observation, we now clearly see that the Swapper is better than the two others, because we have that

$$F_2(a) \geq F_1(a) \geq F_3(a),$$

which means that the proportion of peers with a clustering factor lower or equal to a is greater for the healer, then when $(H = 0, S = 0)$ and finally for the Healer.

4 Plugging everything together

In this section, the objective is that anti-entropy and rumor mongering use the PSS in order to select their gossip partners.

At first, we added a few boolean values like `pss_activated` in the `gossip_pss.lua` script (which is based on the combined anti-entropy and rumor mongering script) and then required the `pss.lua` script as a module. These booleans permit to choose whether to use the PSS, anti-entropy and rumor mongering or not. We just had to comment a few lines in `pss.lua` and it worked quite well, but it was not practical to launch it on the cluster. So we basically made a new file `gossip_pss_web.lua` which was the concatenation (plus or minus a few details) of the two previous mentionned files. We directly implemented a `getPeers(n)` function that returns n peers (it is more practical for the rumor mongering mechanism).

With this script, we could produce the plot in Figure 7. We let the PSS run 60 seconds before launching the anti-entropy and rumor mongering mechanisms.

We actually see that the Swapper is the best. But with what we explained in section 3, it is not surprising. It had the best indegrees and clustering properties. The fact that

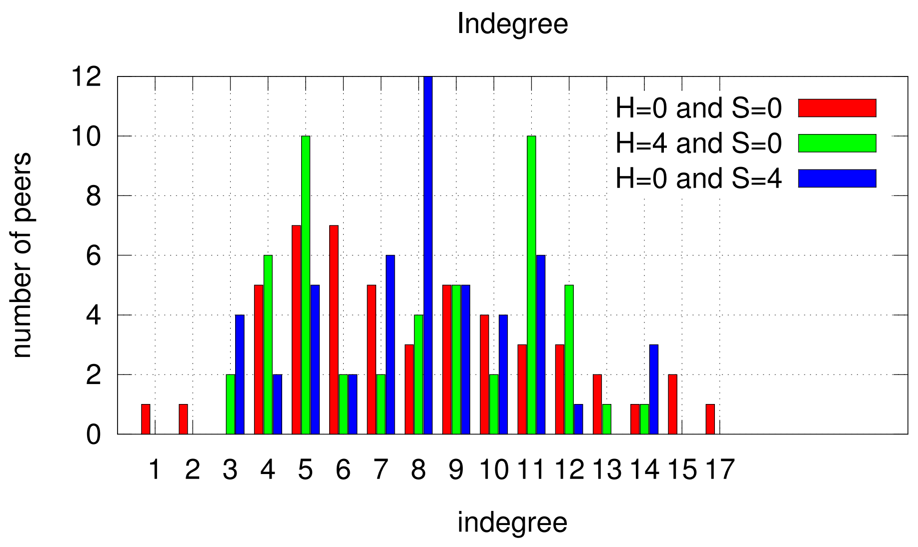

Figure 5: Indegrees for different values of H and S

the clustering factor is low implies that there are not too much redundant messages with anti-entropy, and there are more chances to choose a peer owning a new message with anti-entropy.

If we compare Figure 7 with Figure 4, we see that on the former, the nodes are infected faster (we used a population of 50 peers for the PSS and a population of 40 peers for task 2.3). Although it is hard to compare both plots further as the parameters are not the same, we can see that the PSS helps the propagation, because the resulting graph has good properties.

5 Conclusion

The first part of the project was consecrated to the implementation of two gossip-based dissemination protocols, namely anti-entropy and rumor mongering. We evaluated their performances in terms of delay when they were used separately or at the same time. We found that depending on the values of f and HTL, anti-entropy was more efficient than rumor mongering. Still depending on the values of f and HTL, the combination of both mechanisms was faster than rumor mongering and anti-entropy. In this part, we made the unrealistic assumption that each node in the system has a global view of the network.

This lead us to the second part of the project, namely implementing the Peer Sampling Service. We had to test the properties of the graph produced by the PSS:

- is the network connected?
- how clustered is it?
- what is the indegrees of the nodes?

We found out that the Swapper protocols ($H = 0$ and $S = 4$) had the best properties, the cluster factor was lower and the indegree distribution was similar to a gaussian distribution centered at c.

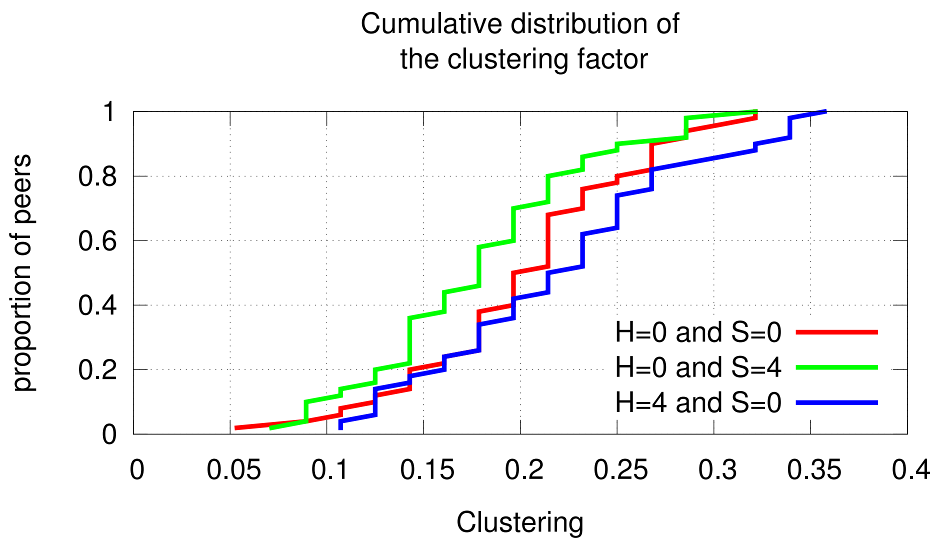

Figure 6: Cumulative distribution of the clustering factor for different values of H and S

Eventually, we could use our implementation of the PSS to ameliorate the gossip-based dissemination protocols seen in the first part. The observation was that the PSS helped the dissemination because of the graph's good properties.

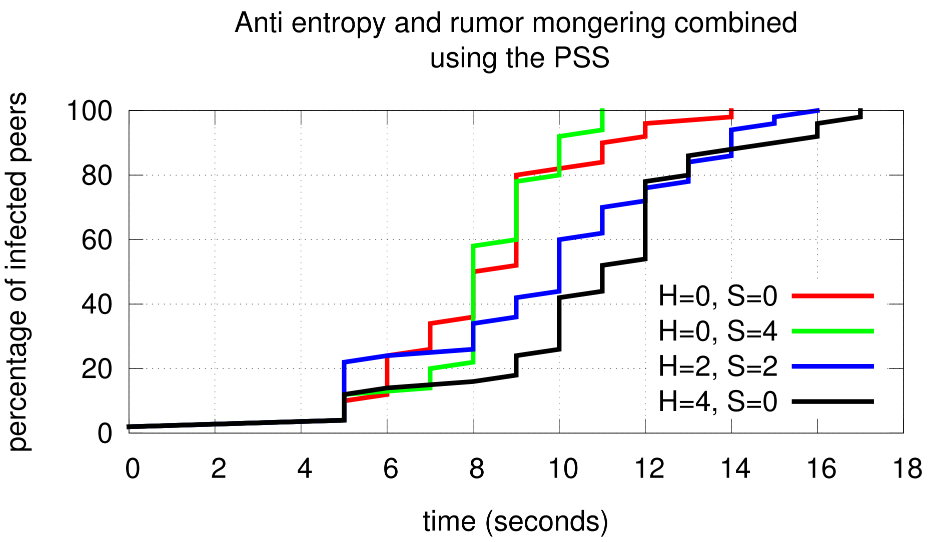

Figure 7: Delay of reception of one message using the PSS with various values of H and S and the combined version of the anti-entropy and rumor mongering mechanisms.