Gossip-based dissemination, Peer Sampling Service

Large-Scale Distributed Systems

Sébastien Vaucher
sebastien.vaucher@unine.ch

29 October 2015

Contents

MASTER IN COMPUTER SCIENCE

unine
UNIVERSITÉ DE NEUCHÂTEL

1 Introduction

This report presents the results obtained in the first assignment of the Large-Scale Distributed Systems course taught at the University of Neuchâtel. The goal of the assignment is the implement two gossip-based dissemination algorithms, namely anti-entropy and rumor mongering. A second part of the assignment consists in implementing a peer-sampling service. The implementations are programmed in the Lua programming language, and use the Splay framework.

Apart from this report, a number of files are supplied:

gossip.lua
: Contains the entry-point of the program and the implementations of both gossip algorithms.

pss.lua
: Contains the implementation of the peer-sampling service.

gossip.sh
: Bash script to launch the program on a local machine.

***.txt**
: Raw logs generated by the program running on the cluster of the university.

parse_log.pl
: Perl script to parse logs produced by the program and generate files readable by the Gnuplot utility.

gnuplot_*.gp
: Gnuplot scripts generating the graphs found in this report.

pss_check_*.rb
: Ruby scripts provided at the beginning of the assignment by the tutor, they are used to compute different metrics relative to the peer-sampling service. They were not modified.

generate_graphs(.sh|.bat) Script (.sh for POSIX-compliant systems, .bat for Windows) to generate the plots found in this report from the raw logs. Note that only the .sh version is able to generate the plots relative to the peer-sampling service.

All the data presented in this report is the result of executions on the Splay cluster of the university. The conditions are near-ideal because all nodes join the network at the same time and never leave it.

For a better reading comfort, all figures are disposed at the end of the document.

2 Gossip-based dissemination

As part of the assignment, the anti-entropy and rumor mongering algorithms were implemented. The source code of both is contained in the file gossip.lua. The goal of this particular implementation is to disseminate an infection to 40 individual nodes.

2.1 Plots

In Figure 1 and Figure 3, we compare the speed at which the infection reaches the nodes. Figure 1 shows the difference when we vary the HTL parameter of the rumor mongering algorithm. This *Hops To Live* value specifies how many nodes a message traverses before it stops being transmitted further. Figure 3, in the other hand, shows the effect of the F parameter. It states to how many random peers a given node has to propagate an incoming message. Both figures also display the performance of the anti-entropy algorithm.

Figures 2, 4 and 5 display the number of duplicates metric. Figure 2 and 4 show the same execution as Figure 1 and 3, respectively. The number of duplicates metric is applied to the rumor mongering algorithm. It is incremented by 1 every time a node receives a message that it already knows about. Figure 5 shows the number of duplicate messages needed to reach a certain number of nodes. Note that the x axis is a logarithmic scale.

2.2 Analysis

The first metric we are interested in is the time it takes for all nodes to be infected. With the help of Figures 1 and 3, we can compare the performance of anti-entropy and rumor mongering. In a standard implementation of rumor mongering, the dissemination is performed immediately after receiving a message. In our case, the messages are buffered to provide a better point of comparison with anti-entropy.

The clear advantage of anti-entropy is that it will eventually reach all nodes. It is also quite fast to notify about 70% of the nodes. The notification of the first 20% and last 10% of the nodes are slower.

With rumor mongering, it is crucial to set the stop condition according to the environment. When using the Hops To Live condition, the size of the network has to be roughly known. Figures 1 and 2 display the effect of HTL on performance. When using $F = 2$, that is sending messages to 2 partners at a time, the HTL parameter has to be large enough, 7 is our case. The number of duplicates messages generated when using a large value grows rapidly. It almost doubles when transitioning from $HTL = 5$ to $HTL = 7$. With small values like $HTL = 1$, no duplicates are recorded. Rumor mongering with $F = 2$ and $HTL = 7$ is faster than anti-entropy.

The second parameter accepted by rumor mongering is called F. It states to how many partners a given node has to propagate received messages. Its effect on performance can be observed in Figures 3 and 4. The executions with $F = 2$ and $F = 3$ have suffered

from a sort of "bug" during their executions. By looking at the logs[1], it seems that all nodes in the cluster basically decided to take a "quick nap" of 30 seconds in the middle of the execution. A second execution was performed, but yielded even worse results, with longer pauses in the execution. For the sake of the analysis, the long horizontal line in the plots should be discarded.

As far as F is concerned, we can see that increasing it has a similar effect as increasing HTL. Increasing either parameter accelerates the dissemination at the expense of more duplicated messages. The execution with $F = 1$ and $HTL = 3$ did not yield any duplicates but its coverage is very poor at about 15%. Increasing F to 2 greatly ameliorates the coverage with only a minor impact on the number of duplicates. Such a value can be a good starting point to kickstart an eventually complete gossip-based algorithm, like anti-entropy.

Figure 5 shows the number of duplicates needed to reach a number of nodes. Obviously, only the rumor mongering algorithm is tested. Please note that the x axis is logarithmic. This plot provides us with one conclusion: rumor mongering yields only few duplicates when we try to reach only a fraction of the nodes. However, if we want to reach all the nodes, we need to cope with a large number of duplicated messages. The effect of F versus HTL on the number of duplicates is undetermined, apart from the fact that higher values for either parameter bring in more duplicates.

3 Peer-sampling service

The second part of the assignment consists in implementing a peer-sampling service. The PSS is capable of keeping a partial view of the global system by exchanging information about peers with other peers. In our context, it is used to replace the complete view of the system used to analyze gossip-based algorithms. In accordance with the given instructions, the gossip algorithm tested with the PSS is the combination of anti-entropy and rumor mongering. For rumor mongering, $F = 3$ and $HTL = 3$ were taken as parameters based on the observations in subsection 2.2.

One execution used the *rand* partner selection strategy, with all the others using *tail*.

3.1 Plots

Figures 6 and 7 show two metrics relative to the performance of the peer-sampling service, namely clustering and in-degrees. Figures 8 and 9 display the result of integrating the PSS in the combination of anti-entropy and rumor mongering. Different couple of parameters for H and S are used, as well as using the complete list of nodes given by Splay.

3.2 Analysis of PSS-related metrics

There are three metrics solely related to the PSS in which we are interested. The first – and most important – is the partition of the graph of nodes. If the graph is partitioned,

[1] Files rm_f2_h3.txt and rm_f3_h3.txt.

it means that the network is segmented in multiple parts, thus preventing some nodes to contact some other nodes. We measure the partitioning twice: once at the earliest time, when the view is composed of a random sample from the complete view; and a second time at the end of the execution, when the PSS has completed some cycles. The result that we get is that the network was only once partitioned. In the case of $H = S = 0$ using the *rand* strategy, one node was left out of the system at the beginning of the execution. However, the PSS was able to eventually resolve the issue, as the network was re-united at the end of the execution. This proves that the PSS algorithm is self-healing. This characteristic is crucial for real deployments where nodes come and go as they please.

The second metric is the clustering ratio. A high value (max. 1) means that the network contains aggregates, which are bad for gossip-based algorithm. On the other hand, a value of 0 indicates a perfect random graph, leading to more chances of contacting a node that never received the message in dissemination. The plot in Figure 6 shows the cumulative distribution of clustering for each node. The better curves are the ones that are shifted to the left. We can clearly see that a smaller clustering is offered by enabling the *Swapper* removal strategy ($S > 0$).

The last metric is the in-degree of the nodes. The in-degree of a node is the number of other nodes that have a pointer to the node in question. The metric is represented in Figure 7. The cumulative in-degree for each node is shown. In order to have gossip-based dissemination perform at its best, the distribution of the in-degrees across nodes has to be a Gaussian around C (8 in our case). On the graph, the best curve is the most vertical one, centered around 8. The presence of either the *Healer* ($H > 0$) or *Swapper* ($S > 0$) strategy tends to provide a better distribution, with a combination of both being the best.

3.3 Analysis of gossip-based dissemination using the PSS

The two interesting metrics to compare executions of gossip-based dissemination algorithms are explained in subsection 2.2. They are the time it takes to complete the dissemination and the number of duplicate messages yielded by the rumor mongering algorithm.

The time taken by our combination of algorithms to complete the dissemination seems to be affected by how we configure the peer-sampling service. Not using either the *Swapper* or the *Healer* is the worst case encountered. The usage of the *rand* or *tail* partner selection strategy has only limited effect, with *rand* being worse throughout the whole execution. The different values for H and S do not have an effect on the time taken by the dissemination (apart when $H = S = 0$), as all executions complete in the same 2-seconds span. Using the peer-sampling service or the complete list of nodes in the network is roughly equivalent, thus indicating that a well-functioning peer-sampling service implementation is a good alternative to the utopia of possessing the complete list of nodes.

The number of superfluous messages created by the rumor mongering algorithm is definitely affected by the peer-sampling service. While all other metrics described the

rand partner selection strategy as the worst, it clearly shines when we need to reduce the number of duplicates messages. Using the complete view of the network is equally as good for the first two thirds of the nodes. Using only the *Swapper* is the most duplicate-generating configuration. The other configurations curves are too noisy to draw meaningful conclusions about them.

4 Conclusion

The implementation of the anti-entropy and rumor mongering gossip-based dissemination algorithms presented in this report is functioning. It is able to effectively propagate a message to a network of individual nodes. The implementation of the peer-sampling service is also fully functional and capable of integrating with the two above-mentioned gossip algorithms.

The advantage of anti-entropy over rumor mongering is its eventual completeness, while the latter has an advantage in the early execution. A good working strategy is to start with rumor mongering, and then let anti-entropy finish by reaching the last nodes. Rumor mongering can be easily customized to will by tweaking its two parameters. Requiring rumor mongering to be complete inevitably yields a rapidly-increasing number of duplicated messages.

The peer-sampling service is necessary in a lot of situations involving large-scale distributed systems. In large-scale deployments, obtaining the complete list of nodes is not reasonably feasible. A PSS is a good alternative, being even better than the complete list in some cases. Configuring the PSS parameters has only a limited impact on the performance of the whole system, compared to the power of those of rumor mongering, for instance.

The analysis found in this report have to be carefully interpreted. They are the result of only a single execution on a shared platform for each set of parameters. They nonetheless provide a good estimation of the performance of each protocol.

List of Figures

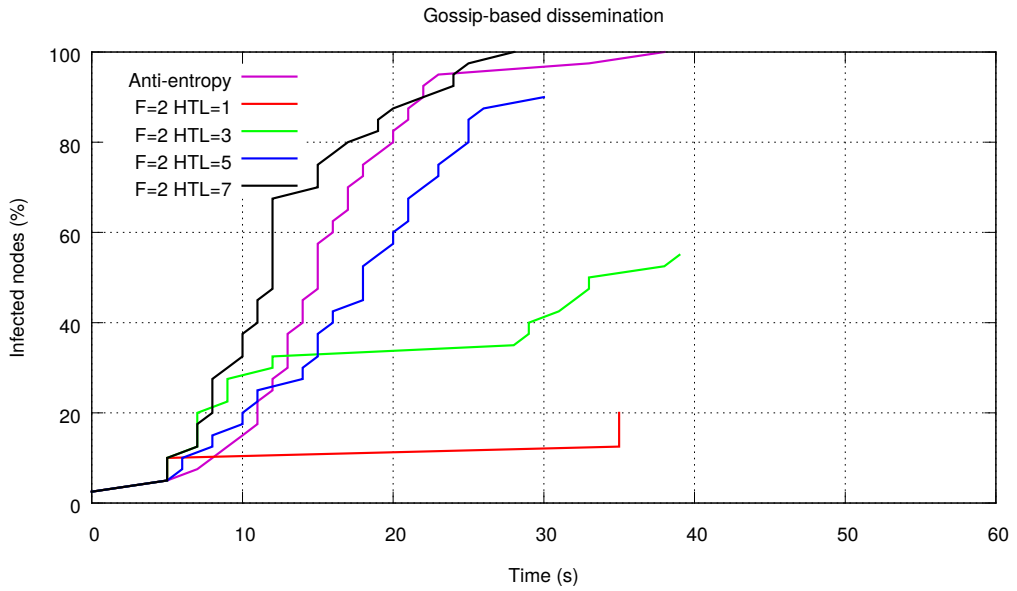

Figure 1: Variation of the HTL parameter, percentage of infected nodes over time

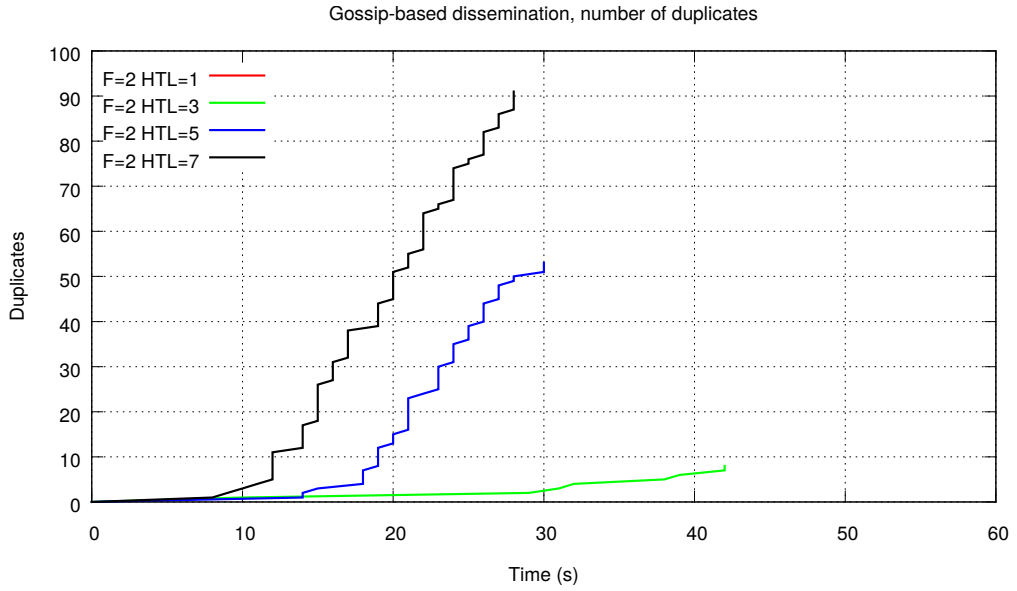

Figure 2: Variation of the HTL parameter, duplicate messages over time

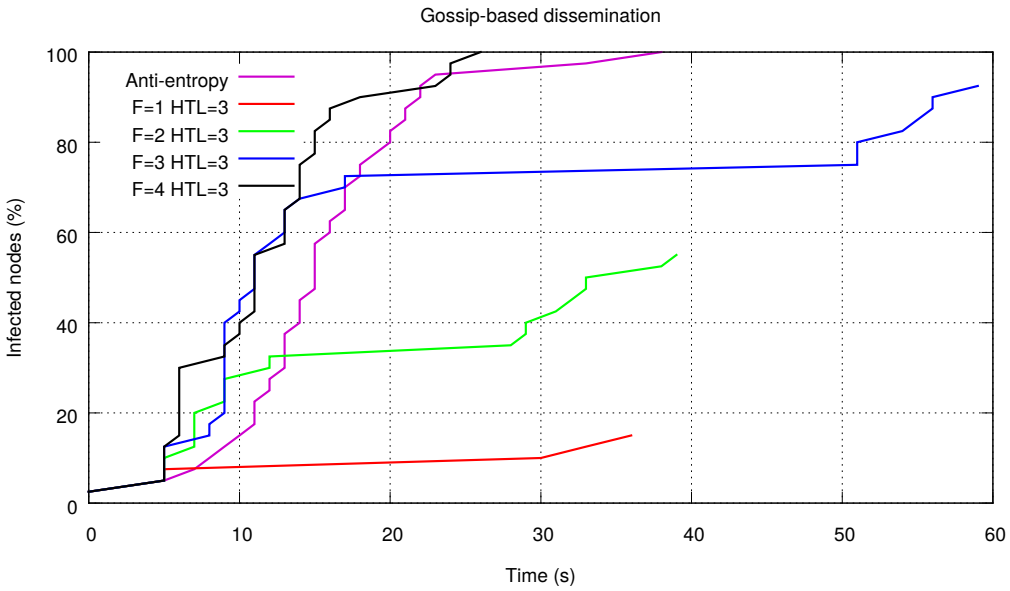

Figure 3: Variation of the F parameter, percentage of infected nodes over time

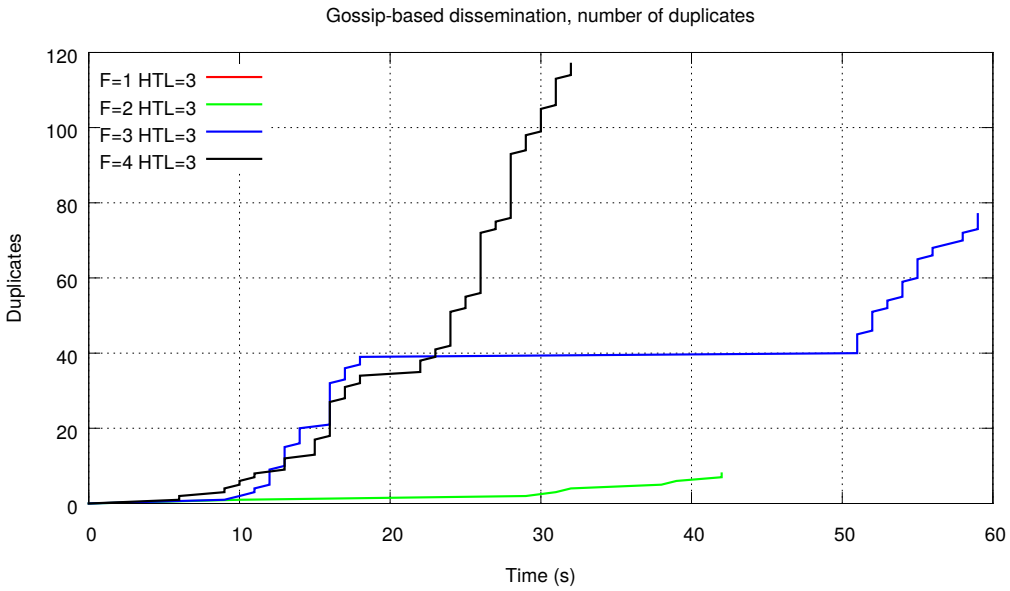

Figure 4: Variation of the F parameter, duplicate messages over time

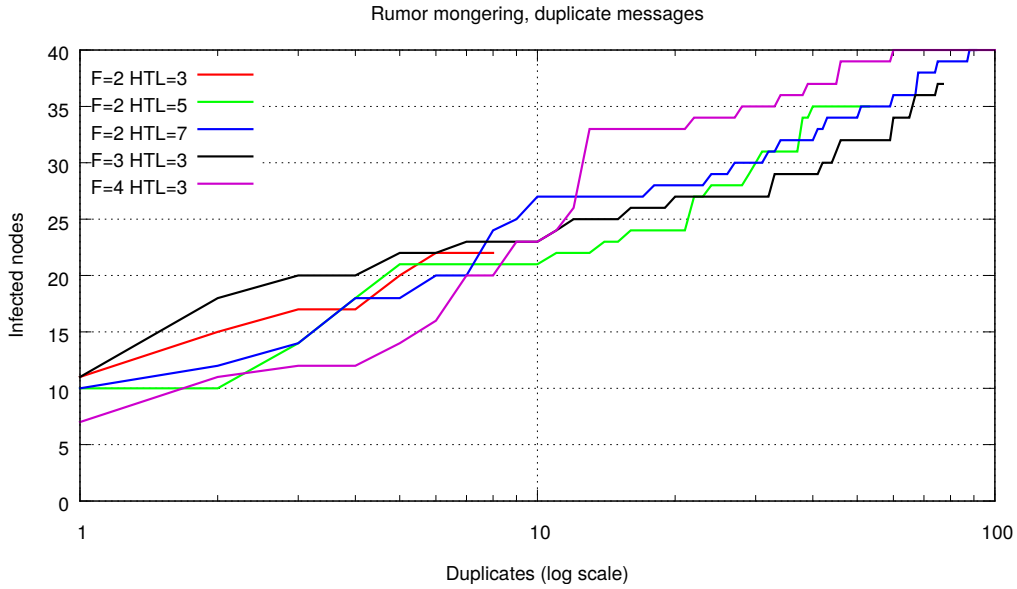

Figure 5: Number of duplicate messages needed to achieve a given number of infections

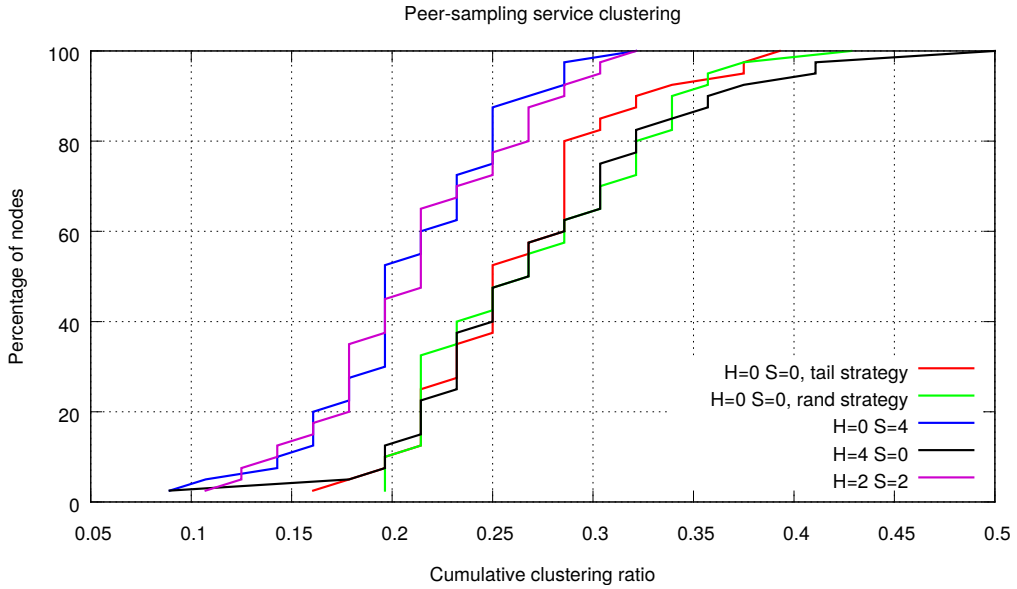

Figure 6: Clustering of the nodes

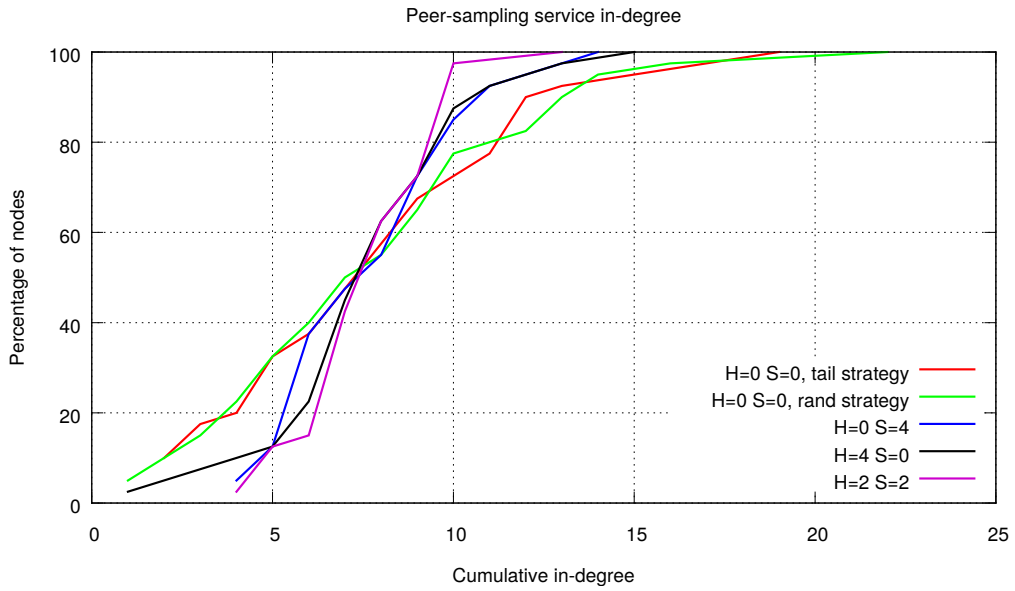

Figure 7: Cumulative in-degree of the nodes

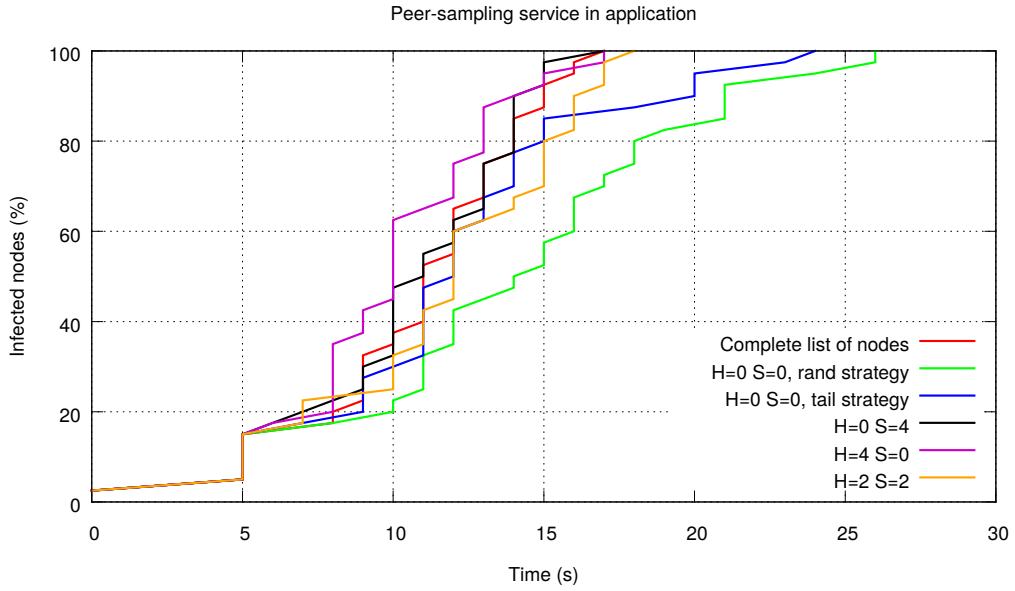

Figure 8: Dissemination using the peer-sampling service

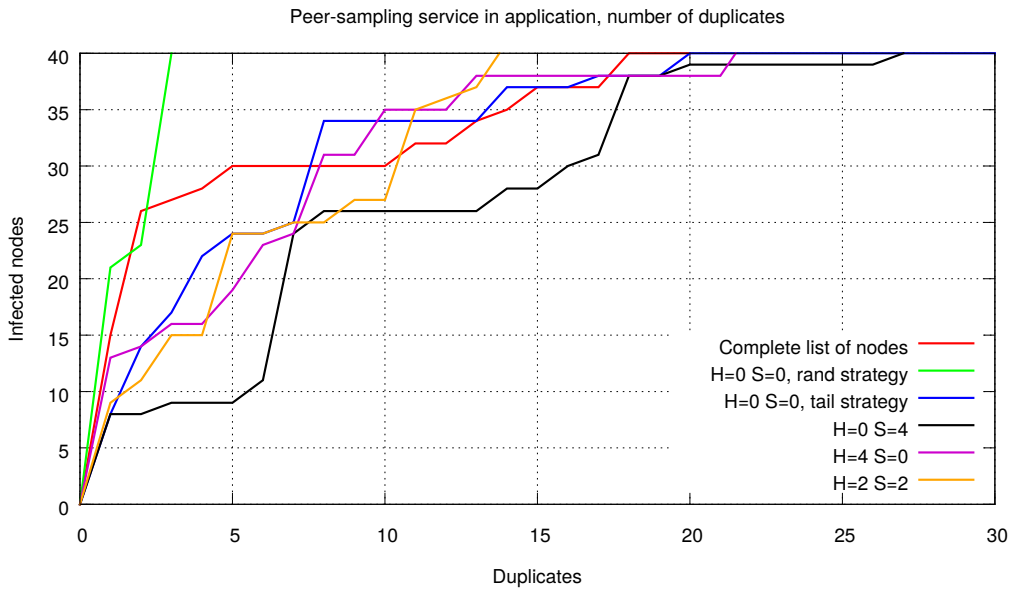

Figure 9: Duplicated messages while disseminating using the peer-sampling service