# *LSINF2345 – Languages and algorithms for distributed applications*
# Implementing a distributed transactional key-value store

Simon GUSTIN 11-71-1400     Loan SENS 71-07-1700

May 2018

## 1 Introduction

As part of the course *LSINF2345*, we were asked to implement a distributed transactional key-value store in Erlang. A key-value store usually simply maps a value to each key present in the store. The particularity here is that the store keeps a history of values for those keys along with a timestamp. Then, when queried a value, the store thus must be able to determine which value to send back in function of the local time of the process making the query. Our implementation also had to be able to distribute data over multiple data partitions and to send back this data when queried. After this, we had the choice to make enhancements to our implementation. We thus decided to implement a garbage collection system, so that the data added in the data store would not grow as much. Finally, we had to assess the performance and throughput of our system.

This report will thus first explain the architecture of our code (along with the improvements we implemented). We then summarize our assessment of the performances for different setups and quantities measured.

## 2 Architecture

The program is divided into two main parts: the clients and the data store, composed of transaction managers and data partitions processes. The client only talk to the transaction managers, which in turn make the link between them and the data partitions. In a real world system, the transaction managers would be servers and the data partitions a distributed database. Our system works for any positive number of data partitions, transaction managers and clients.

### 2.1 Data store

#### 2.1.1 Data partitions

Intuitively, the data partitions are encapsulations of dictionaries that contain the data of the system. Each data partition contains one and only one dictionary in which, each key (which represents a key of the key-value store) is linked to a list of tuples. Those tuples simply contain ta value and the timestamp at which this value was added in the data store. We could thus represent such a dictionary as follows:

```
KEY1 => [{TIMESTAMP11, VALUE11}, {TIMESTAMP12, VALUE12}, ...]
KEY2 => [{TIMESTAMP21, VALUE21}, {TIMESTAMP22, VALUE22}, ...]
...
```

Those processes are thus in charge of putting key-value pairs in the store and of retrieving them when asked. As we will explain later, they are also in charge of garbage collecting their dictionary. All those operations are triggered when the partition receives some particular messages. Note that in the whole program, we decided to send the PID of the sender in every message as it could be used for further improving the system, yet it is not always necessary.

On reception of an `update` message containing one key and one value to add or update in the store, a data partition first checks if the key is already present in its dictionary with this value as its most recent value. If it is, there is no need to do anything; otherwise, it appends a tuple with the value and the current timestamp to the list corresponding to the key in its dictionary. We get the timestamp from the function `os:timestamp/0`. Notice that we decided not to update the value corresponding to a key when it already has this value in order to use less memory, as the purpose of the system was not to be the fastest possible. Nevertheless, it could easily be changed if we were willing to sacrifice some memory to get a faster system.

Then, on reception of a `snapshot_read` message containing one key to read and the timestamp at which the value is wanted, the partition simply tries to find the value corresponding to that key at this time in its dictionary. If it can find it, the value will simply be sent back to the sender. Otherwise, in case the key is not present in its dictionary, the value returned will be `nil`, while in the case the key was present but not yet for the time of request (i.e. all the values for this key were set later), the returned value is `nil_yet`.

Lastly, an `exit` message will simply stop the process. (We called this at the end of an execution.) A gc message also exists, but we will explain it in the section dealing with garbage collection.

### 2.1.2 Transaction managers

All the messages we previously talked about are issued by transactions managers, which are in charge of making the link between clients (who fetch the data) and data partitions (that host the data). Actually, transaction managers handle the task of determining which data partition host the requested data in order to be able to update it or fetch it. It thus has access to each and every data partition of the system (as a list). Note that the instructions explicitly said not to implement a system that allowed to add or remove partitions while the system was running. Determining which data partition should host the values of a certain key is simply done using a deterministic hash function (the `phash/1` Erlang function) that returns the index of the data partition in the list. It also has access to all the other transaction managers for reasons that will become clear in the *garbage collection* section.

Thus, on reception of an `update` message containing a key and a value, the transaction manager hashes the key to determine which data partition will be sent an `update` message with this key and value. Knowing the update cannot fail, it doesn't wait for the `ok` answer of the data partition (but it is still able to receive it later on). This allows the transaction manager to treat pending messages, increasing the throughput of the system.

On reception of a `snapshot_read` message which may contain any number of keys, the transaction manager will determine which data partition should contain each of the keys, send a read message to each of them for the concerned key (along with the value of its current timestamp) and wait for the answers of all those data partitions. The values are then sent back to the sender as a list in the same order as the keys in the initial message.

As before, `exit` message allows to terminate the process after having requested all the data partitions to stop as well. The two last types of messages will be dealt with in the *garbage collection* section.

## 2.2 Clients

The clients are in charge of issuing requests to the transaction manager. They decide to update, read or ask for a garbage collection. A client can also sleep for a certain time (thus not issuing messages to any transaction manager).

We implemented two different behaviors for them: either, they can read a file from the disk, in which commands are written in the same format as the file provided on the Moodle of the course (strings that are actually interpreted by Erlang), or they can be used for performance assessment by issuing a given number of messages. For the latter, we can actually provide a list of numbers to the client that actually correspond to the probability of choosing a type of message to send to a random transaction manager (or sleeping for 10ms rather than sending a message).

In both cases, clients thus send requests to transaction managers, wait for the answer and print it to the console in the case of the first behavior (in the same format than the one in the output file provided on Moodle). Note that when the answer to a `snapshot_read` is composed of only one value, the Erlang console may wrongly interpret a value and print it as a different type than it was (for example, `[100]` will be printed as `"d"`). We could fix this by storing the type of the variable in the data store. As this is not a serious problem and the memory consumption would become way larger, we didn't implement this.

## 2.3 Improvements

### 2.3.1 Garbage collection

For such a system, garbage collection is necessary to free up useless memory: keys keep being updated by clients, appending values in the lists of values of the data store. The memory consumption thus can only grow.

Before implementing this, we need to determine in which cases a pair timestamp-value is stale and can be freed. This is actually rather simple: a timestamp-value is useless when no transaction will require it (that is, read it). This is the case only when the timestamp of all the transaction managers are larger than the one of the pair. Obviously, such a pair should be freed only if it is not the most recent pair for a key.

Note that we didn't take into account read messages that would be particularly slow. We did this because Erlang guaranteed that, if the messages sent by a process reach destination, they will arrive in the order in which they were sent. The problem can however still occur when there are several clients issuing messages: a client can send a slow `snapshot_read` message, and during its travel, another client can request a garbage collection which would remove some values requested at the time of the first message. This is actually a problem of scheduling and network speed. The instructions explicitly told us not to fix that kind of faulty behaviors.

Our system of garbage collection works as follows. A garbage collection is triggered by a client which sends a `gc` message to one of the transaction manager (at random). This transaction manager than queries the current timestamp of each transaction manager and then takes the oldest one among those and its own timestamp. It finally sends a garbage collection request to each data partition, along with this timestamp. Upon the reception of this message, a data partition will remove from its dictionary each and every pair timestamp-value that was added earlier than the timestamp contained in the message, unless this pair is the most recent one for a key.

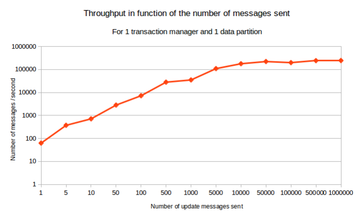
Notice that we didn't make an automatic call to garbage collection (on basis of a timer or a measure of the memory consumption), even though this could have been an expected feature. We decided not to do this simply because the instructions seemed to ask the garbage collection to be triggered only by clients.

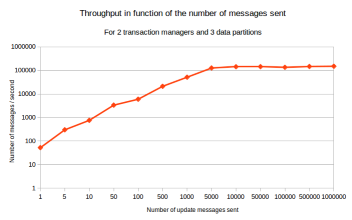
## 3 Performance assessment

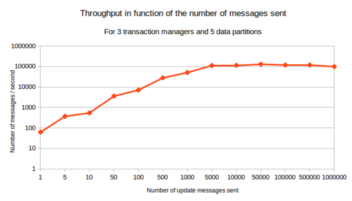
We first have to point out that, as our system runs locally on a single computer, those results depend on how fast it is. The computer used for the experiments used an Intel i5 4-core CPU at 3.1GHz and 6GiB of RAM. The graphs showed in this section are also available in a larger format in appendix.

In the following experiments, we computed the throughput of the system as the number of messages per second, averaged over three executions of the system. The latency is computed as the average of the waiting time between the sending of a message and the answer, that is, the total execution time divided by the number of messages. In other words, we have $throughput = \frac{1}{latency}$. Therefore and for a question of readability, we will only plot the throughput on the following graphs.

Our first experiments were made using one client sending different numbers of `update` messages. Each update inserted a brand new key in the data partitions. We show in the following graph the averages of the measured throughputs in function of the number of messages sent.



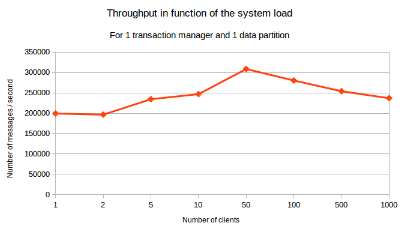


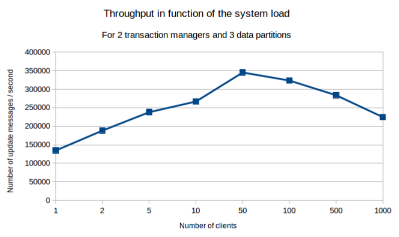


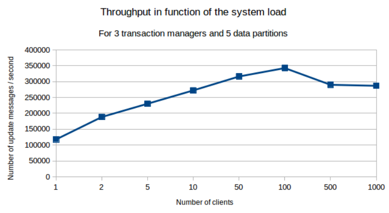
Unexpectedly, the more transaction managers and data partitions there are in the system, the longer it takes to execute the program (this was especially visible when the client had to sent a million messages between 3 transaction managers). We would have expected having more transaction managers and more data partitions would allow the messages to be sent to different processes, and thus be treated simultaneously. This is not what we observe simply because the client waits for the transactions to complete before sending another message.

We thus see that with one transaction manager and one data partition, for the tested cases, the throughput plateaus at about 240 000 messages treated per second, at about 140 000 messages per second and at about 110 000 messages per second, respectively. Using more transaction managers and data partitions made the execution very long for large numbers of messages to send.

We thus made the same test with more and more clients sending a total of 100 000 `update` messages (still without duplicate keys). The throughput of the system is thus the sum of the throughputs for each client. The following plots show it in function of the number of clients. We could say that the greater the number of messages sent at the same time, the larger the load of the system is. These graphs are thus really plots of the throughput in function of the system load.





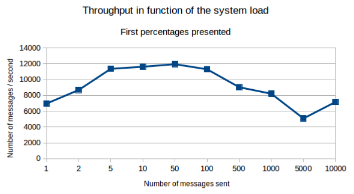


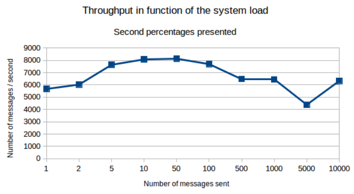
Notice those plots don't have a logarithmic axis anymore. We can see that the throughput goes up until it reaches a peak for a certain number of clients. Then, for more clients, the load of the system is too high for it to answer rapidly and the throughput thus goes down. For fewer clients, it seems logical that the throughput will be low, simply because all the client cannot issue enough messages to "saturate" the store.

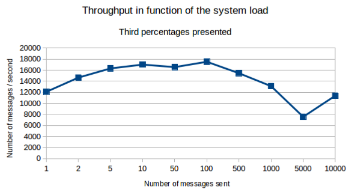
For one transaction manager and one data partition, the peak seems to be at about 300 000 messages per second; for 2 transaction managers and 3 data partitions, it is at about 350 000 messages per second; for 3 transaction managers and 5 data partitions, at about 350 000 messages per second as well but seems to not be going down as much as the other ones after the peak. Using more transaction managers and data partitions thus seems to (and logically does) cope better with a high system load.

We then made an experiment that would mimic a real world system: using 10 transaction managers and 50 data partitions, different numbers of clients sent all the types of messages with different ratios. The total number of messages sent was 10 000 (as this is significantly slower than the previous tests). For those experiments the `update` message could update the value of a key (it had a choice of 100 keys). The ones we present here are the following:

- 50% `update` messages, 40% `snapshot_read` messages, 10% `gc` messages
- 20% `update` messages, 40% `snapshot_read` messages, 40% `gc` messages
- 60% `update` messages, 10% `snapshot_read` messages, 30% `gc` messages





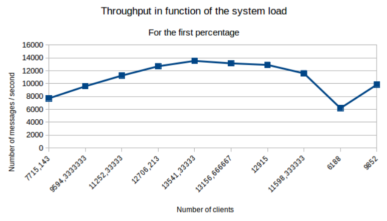


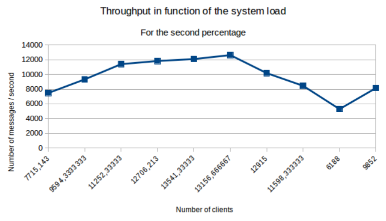
We thus observe that whatever the ratios, those curves always look very much alike. The throughput always seems to be higher between 5 and 100 clients. For fewer and more clients, the reason for the drops of the throughput are the same than before. The highest throughput is however different for each of those percentages (11 955 messages per second for the first one, 8132 messages per second for the second one and 17 513 for the third one).

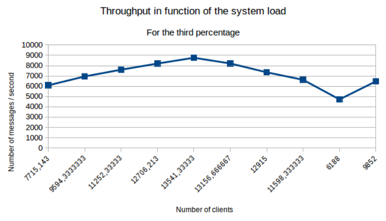
Our last experiment tests the influence of garbage collection. We made a system that uses 10 transaction managers and 50 data partitions, with clients issuing always the same percentage of reading requests, increasing the number of garbage collection requests and decreasing the number of update requests (for a total of 10 000 messages as before). In order to be able to see an influence of garbage collection, the `update` messages shouldn't contain a new key every time. The clients thus choose between 100 keys on each sending of an `update` message.

The different ratios we used are the following:

- 70% `update` messages, 30% `snapshot_read` messages, 0% `gc` messages
- 50% `update` messages, 30% `snapshot_read` messages, 20% `gc` messages
- 20% `update` messages, 30% `snapshot_read` messages, 50% `gc` messages







We thus get similar curves as before. However, it affects the overall performance of the system. Indeed, the three presented experiments have a mean performance of 10 851.9, 9686.1 and 7099.9 messages per second, respectively. We thus see that using garbage collection slows the execution time a bit. As this effect is way larger if we set the proportion of `snapshot_read` messages to 0, we can assume that this is affected by the ratio of reading/garbage collection request. Using garbage collection indeed makes reading requests faster on the long run.

Yet, the point of garbage collection is to use less memory. For the same examples (whichever the number of clients), we computed that the number of keys were the same, but the number of pairs timestamp-value were significantly different: 6984 pairs, 100 pairs and 100 pairs, respectively. We can easily understand that, while the first number grows as the program executes, this is not the case of the two other ones which get back down to 100 on each garbage collection (those being more frequent for the third example). For a system running forever, the first number will not be bounded, while the two others will be (the third one having the lowest bound). The code to compute those numbers is not present in the program we submitted since it made it less readable.

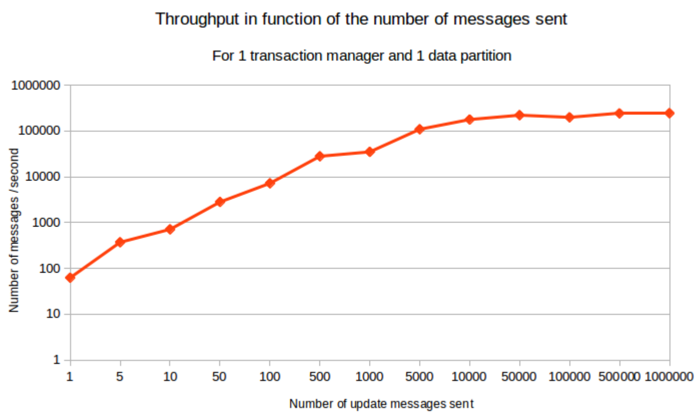
## 4 Conclusion

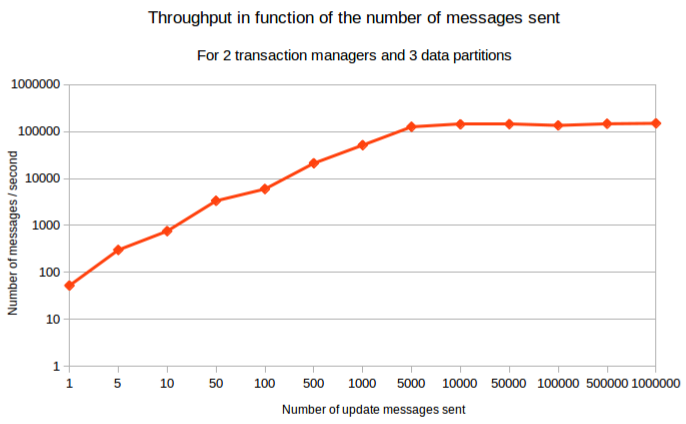
This assignment showed us what it was like to program in Erlang. We could use it to implement a simple key-value store that keeps a history of the values assigned to each key. We could also add a garbage collection system and measure its effect on the performance and the memory consumption of the whole program. We can safely say that our implementation fulfills the provided instructions.

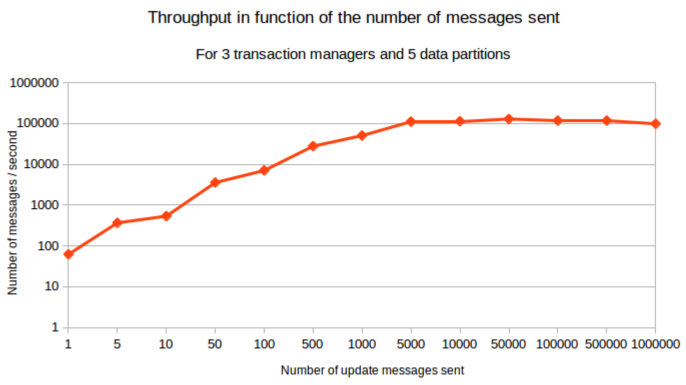
# Appendices

## A Larger graphs

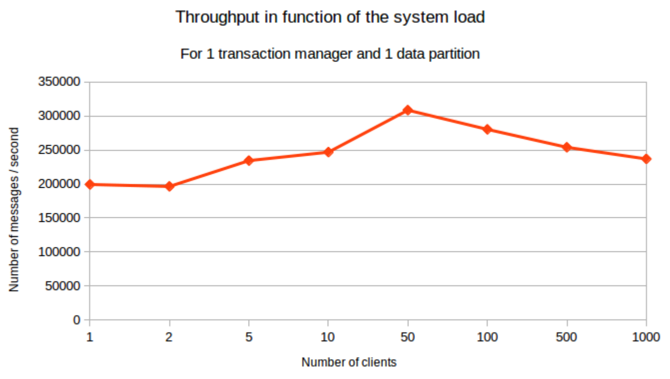
### A.1 One client and only update messages

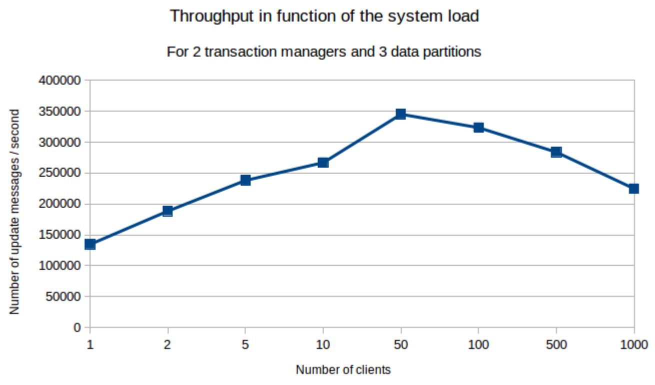


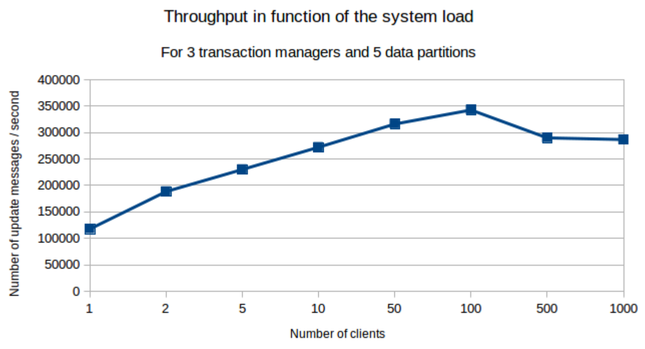




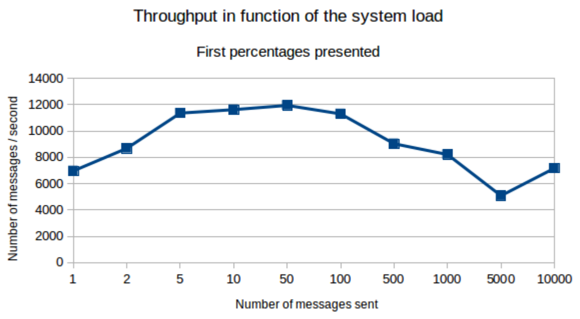
## A.2 Several clients and only update messages

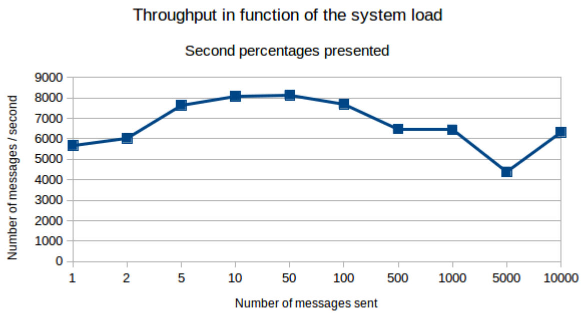


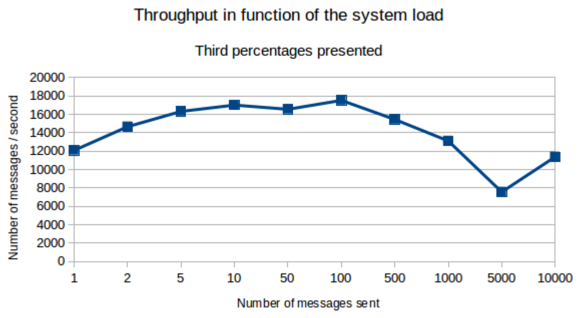




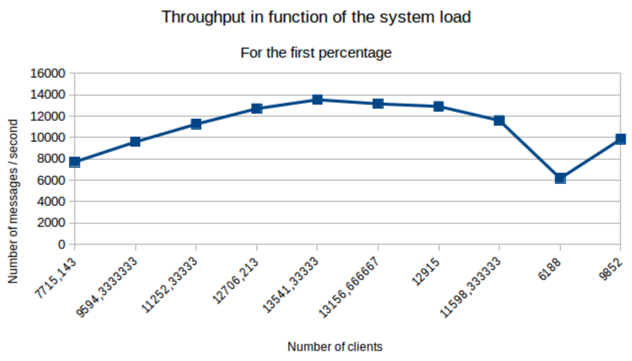
## A.3 Several clients and all kinds of messages

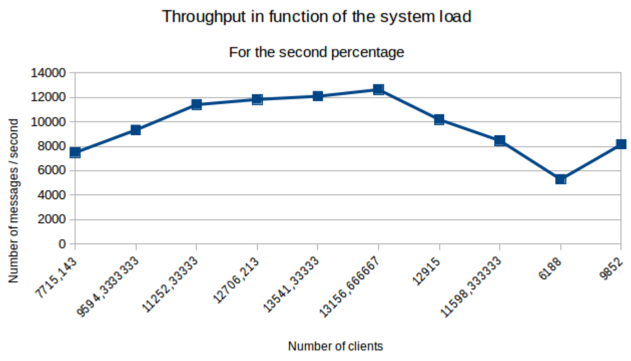


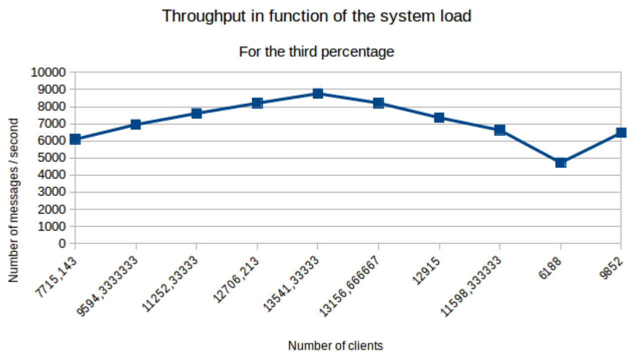




## A.4 Tests about garbage collection


Throughput in function of the system load
For the first percentage
Number of messages / second
Number of clients


Throughput in function of the system load
For the second percentage
Number of messages / second
Number of clients


Throughput in function of the system load
For the third percentage
Number of messages / second
Number of clients