Distributed Systems Project, AA 2021-2022

Raft consensus algorithm:
simulation and analysis

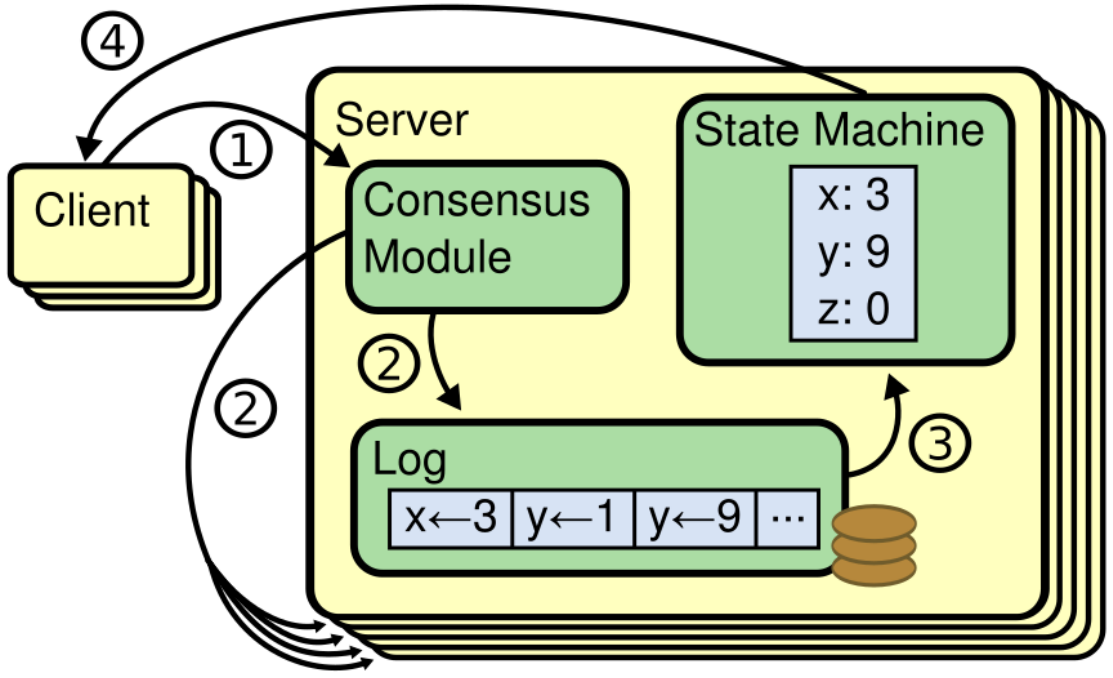
Francesco Cecchetti
Matteo Merz
Fabio Patella

The problem

The problem is to **replicate a log among a set of distributed servers**.

We have the following scenario:

1. A client sends a command to a server.
2. The server add the command to its log and share it with the other servers.
3. The command is run on the servers state machines.
4. A confirmation is sent to the client.



Replicated state machine architecture.

How to manage the replicated log?

We need a **consensus algorithm.**

Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members.

Raft solves this problem: it is a **consensus algorithm for managing a replicated log**.

Raft

Raft implements consensus by first electing a leader, then **giving the leader complete responsibility** for managing the replicated log. The leader accepts log entries from clients, replicates them on other servers, and tells servers when it is safe to apply log entries to their state machines.

Raft decomposes the consensus problem into three subproblems:

Leader election

A new leader must be chosen when an existing leader fails

Log replication

The leader must accept log entries from clients and replicate them across the cluster, forcing the other logs to agree with its own

Safety

If any server has applied a particular log entry to its state machine, then no other server may apply a different command for the same log index

We choosed to simulate the Raft algorithm with **OMNeT++**.

OMNeT++

OMNeT++ is a **discrete event simulator** widely adopted to simulate distributed systems and networks.

Simulations are structured through **components**, which are built with a C++ class library.

OMNeT++ proposes an infrastructure to assemble simulations (NED language) and provides an IDE for the designing, running and evaluating of simulations.

Simulation components

The main simulation components are the **server** and the **client**.

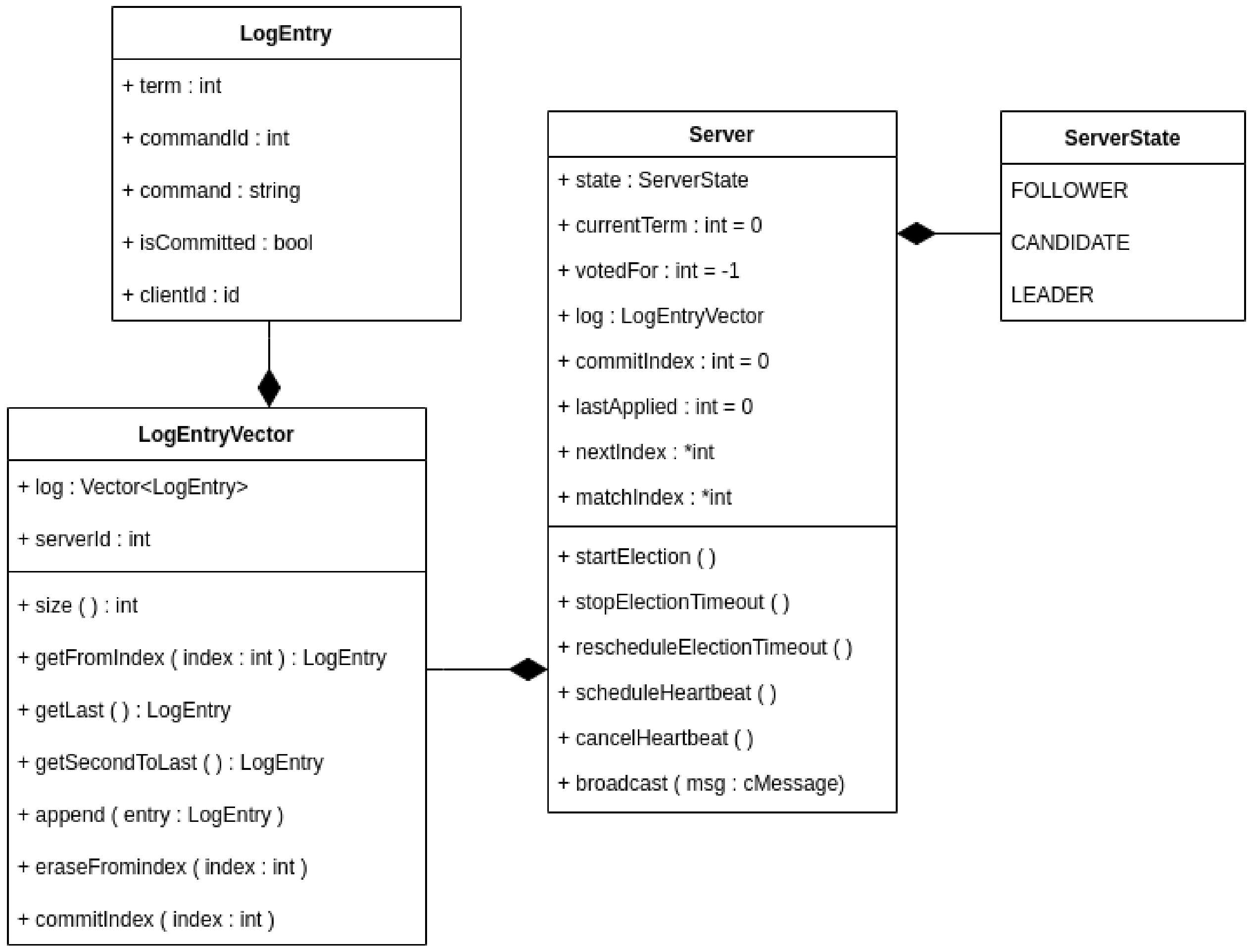
The simulation defines the number of servers and clients and all the connections: all servers are interconnected and connected with the clients.

```
simple Server
{
    parameters:
        volatile double electionTimeout;
        volatile double heartbeatPeriod;
        ...

    gates:
        input fromServer[];
        output toServer[];
        input fromClient[];
        output toClient[];
}

simple Client
{
    parameters:
        volatile double sendCommandTimeout;
        double resendCommandTimeout;
        ...

    gates:
        input fromServer[];
        output toServer[];
}
```

Implementation

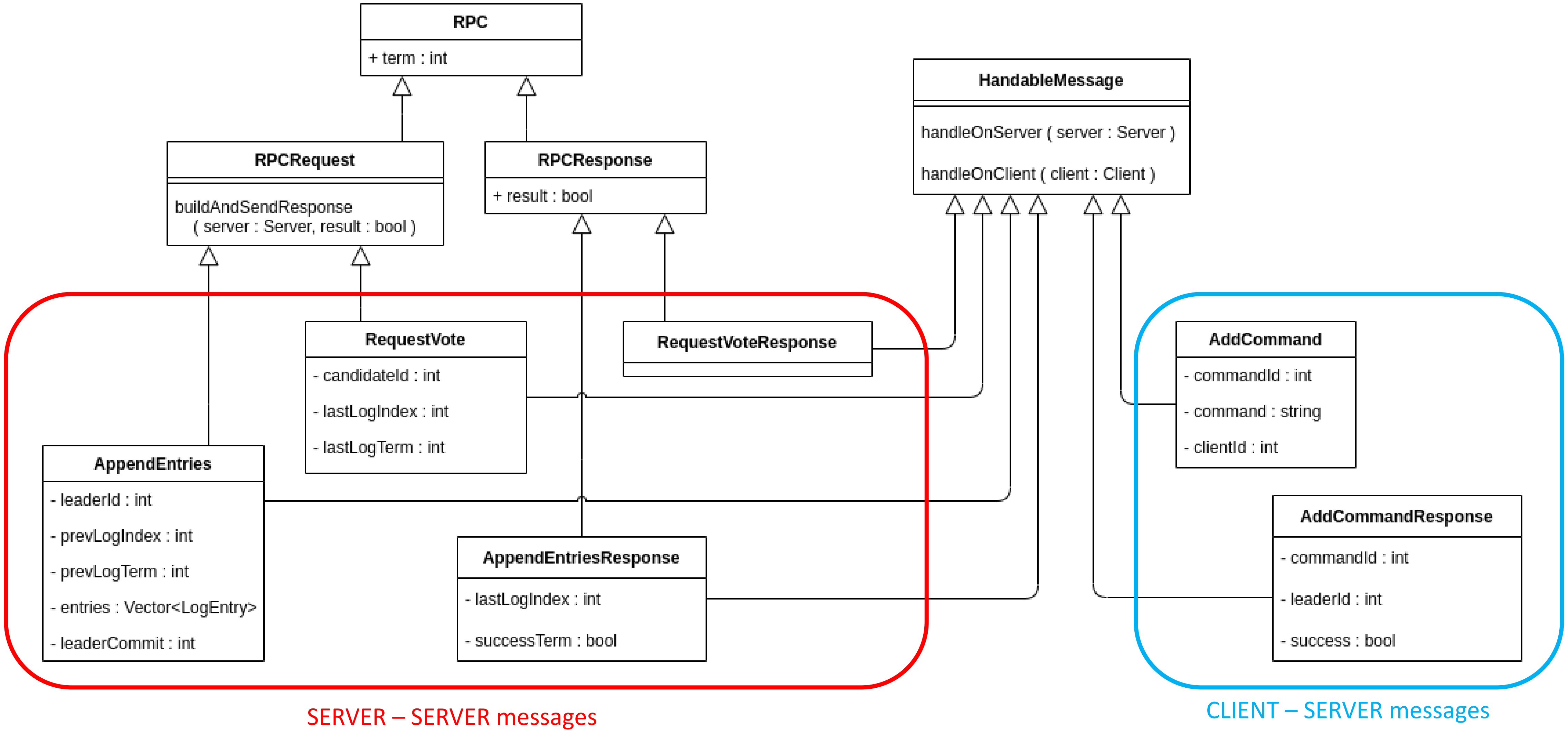
Each simple module is implemented through an associated **C++ class**; other classes have been added to facilitate the implementation.

Also **messages** are represented through classes.



Portion of the UML class diagram regarding the server.

Portion of the UML class diagram regarding messages.

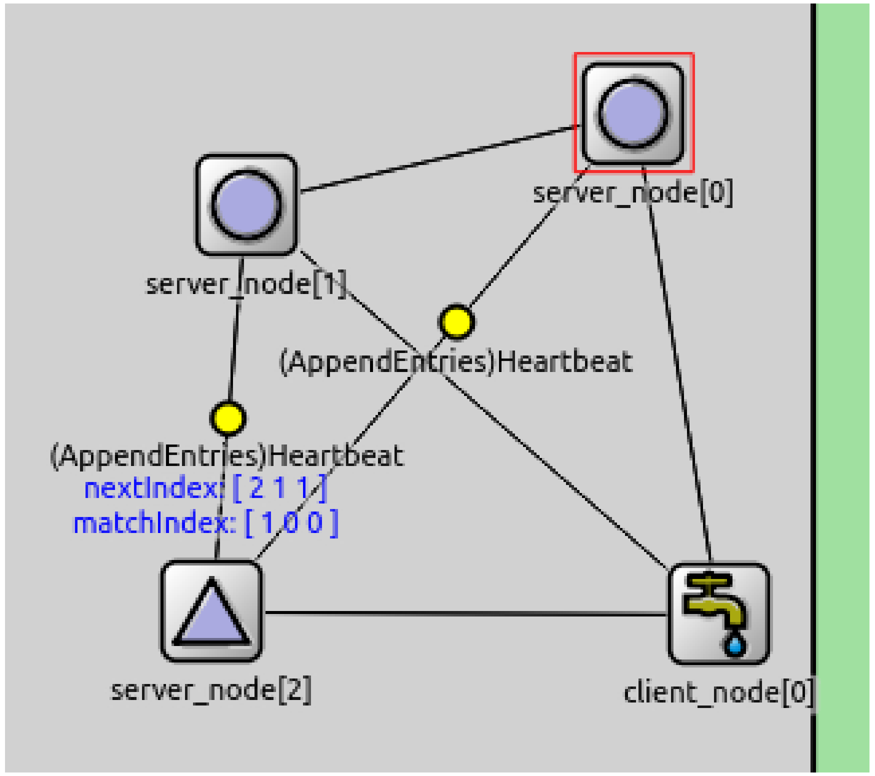


The *handleMessage* method of the **Server** manages explicitly just self-messages.

After some generics behaviours common for some message classes, the control is given to the message itself, by calling the method *handleOnServer* with the server instance as parameter.

The same reasoning can be applied to the **Client** class (here the called method is *handleOnClient*, with the client instance as parameter).

```
void Server::handleMessage(cMessage *msg)
{
    // *** SELF-MESSAGES ***
    if (msg->isSelfMessage()) {
        if (msg == heartbeatEvent) { ... } else
        if (msg == electionTimeoutEvent) { ... }
        return;
    }

    // *** EXTERNAL MESSAGES ***

    // Generic behaviour for RPC messages
    if (dynamic_cast<RPC*>(msg) != nullptr) { ... }

    // Generic behaviour for RPCRequest messages
    if (dynamic_cast<RPCRequest*>(msg) != nullptr) { ... }

    HandableMessage *handableMsg =
        check_and_cast<HandableMessage*>(msg);
    handableMsg->handleOnServer(this);

    cancelAndDelete(msg);
}
```



A screenshot from the running simulation.

Graphic elements

The server icon indicates the current server status.

- △ - LEADER
- □ - CANDIDATE
- ○ - FOLLOWER

The message icon gives a hint of the content:

- 🟡 - Request
- 🟢 - Positive response
- 🔴 - Negative response
- ● - Add command request/response

LIVE DEMO pt. 1

Basic running configuration of the implemented simulation.

How to stress the algorithm?

Raft is **robust** against any **non-byzantine failure**.

In order to understand if the algorithm is correctly implemented, we need to test it in an environment that admits failures.

For this reason, we implemented a mechanism that simulates non-byzantine failures at **process** and **channel** level.

Server crash

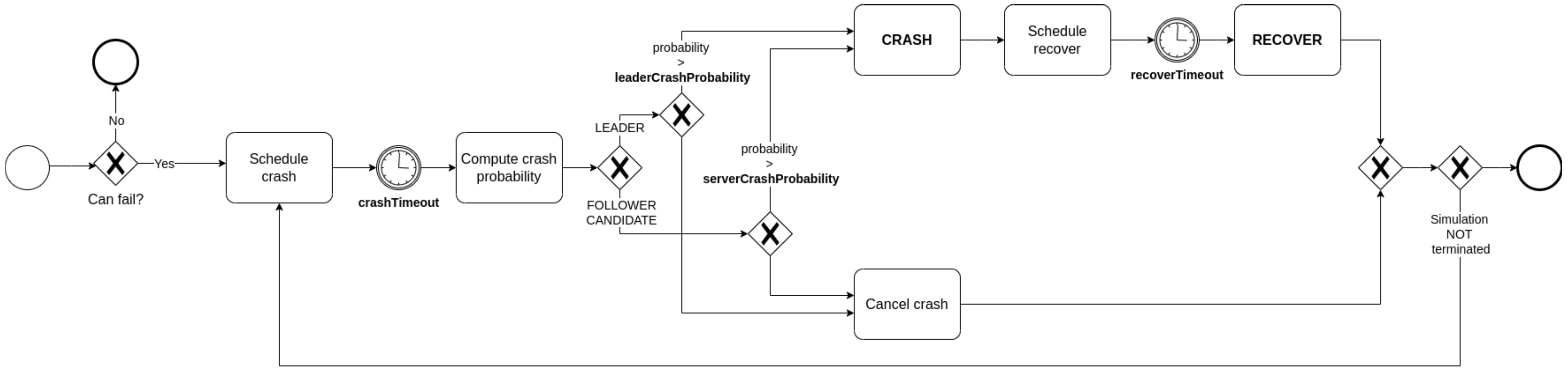
Servers can become unavailble at any time, and then recover after a while.

Channel delay and omission

Messages can be delayed or even lost during the transmission due to the channel.

Failure #1 – **Server crash**

When a server is crashed all the incoming **messages** are **ignored** and deleted.


No
Yes
Can fail?
Schedule crash
crashTimeout
Compute crash probability
LEADER
FOLLOWER CANDIDATE
probability > leaderCrashProbability
probability > serverCrashProbability
CRASH
Schedule recover
recoverTimeout
RECOVER
Cancel crash
Simulation NOT terminated

BPMN diagram showing the server crash and recovery processes.

Failure #2

Channel omission

To simulate channel omissions we **randomically delete external messages** received from servers and clients, depending on a given parameter.

```
void Client::handleMessage(cMessage *msg)
{
    ...

    // *** EXTERNAL MESSAGES ***

    double threshold =
        1 - channelOmissionProbability;

    if (uniform(0, 1) > threshold ) {
        cancelAndDelete(msg);
        return;
    }

    ...
}
```

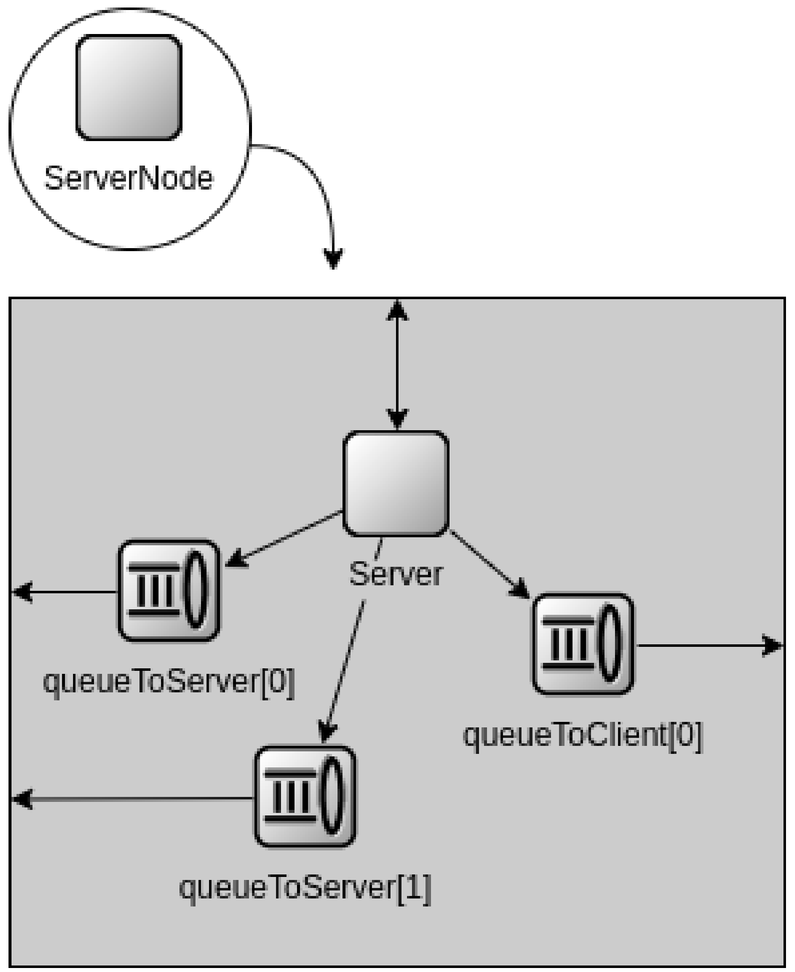
Failure #3

Channel delay

The simple modules of clients and servers has been extended: each **outgoing** channel has been replaced by a **queue**.

Messages in the queues are sent with a **random delay**, produced by an exponential distribution depending on a given parameter.

When a client or a server wants to send a message, it simply put the message on the right queue.



The structure of a ServerNode compound module
(the ClientNode one is very similar, we simply don't have any queueToClient).

LIVE DEMO pt. 2

Advanced running configurations of the implemented simulation.

How to ensure the simulation is working properly?

What happens if a leader crashes before committing an entry?
What if an appendEntries request is lost?

All those questions should be answered by generating **ad hoc tests.**

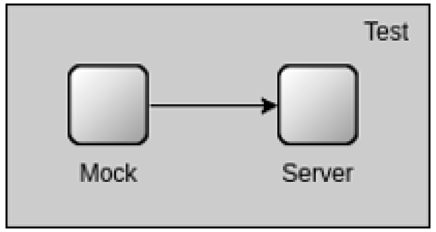
In each test an i**nitial configuration** should be provided, in order to put the simulation in the conditions we want to stress. Then, a set of **assertions** checks if the desired behaviour has been respected.

Network mocking

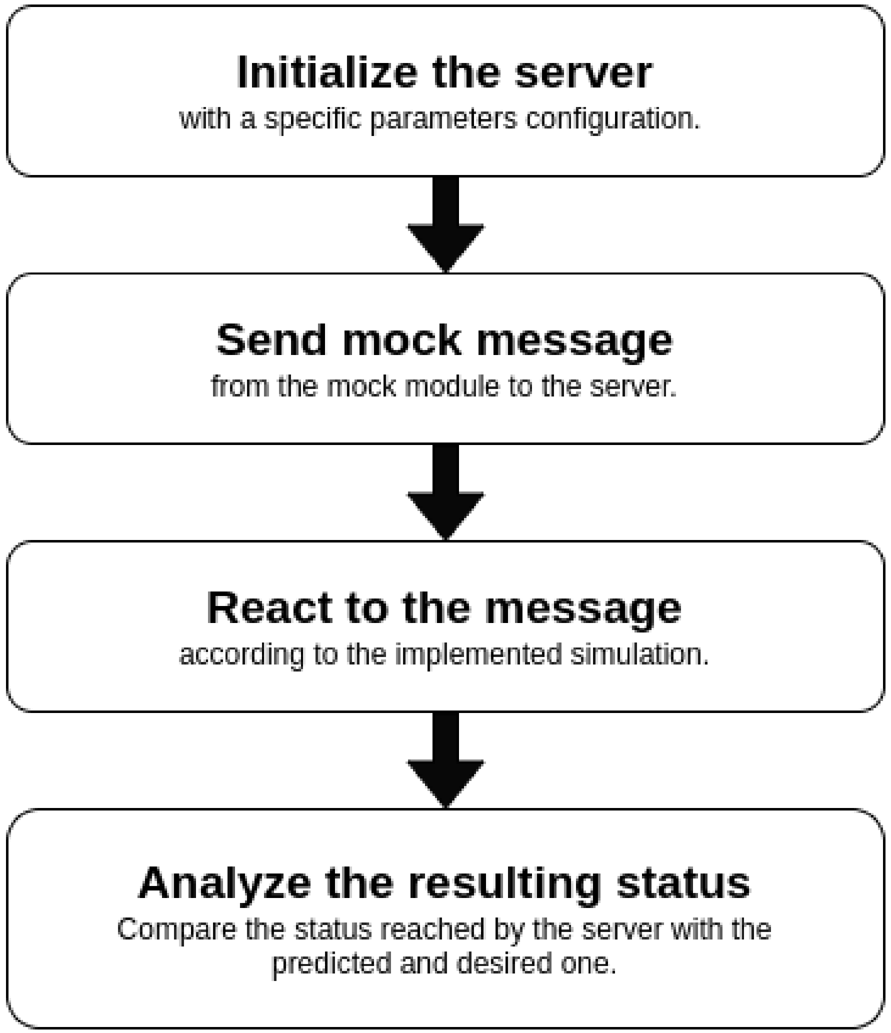
We implemented a **mock network** to run the tests.

The structure is very simple:

- on one side a server module
- on the other a mock module, created ad hoc to send customized messages.



Phases of a generic test



Initialize the server
with a specific parameters configuration.

Send mock message
from the mock module to the server.

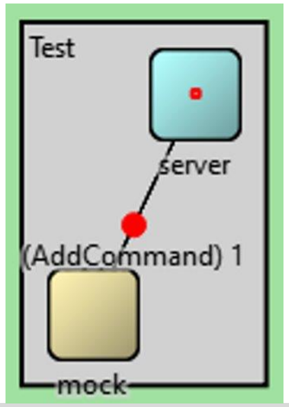
React to the message
according to the implemented simulation.

Analyze the resulting status
Compare the status reached by the server with the predicted and desired one.

Test example

Add command

Test what happens when a leader receives and addCommand request from a client.



```
State = LEADER;
log = new LogEntryVector();
nextIndex = new int[1];
matchIndex = new int[1];
nextIndex[0] = 1;
matchIndex[0] = 0;
currentTerm = 1;
nbOfServers = 2;
```

Phase 1

```
AddCommand *msg = new AddCommand(1, "test", 0);
```

Phase 2

Phase 3

Phase 4

```
AppendEntries *msg = check_and_cast<AppendEntries*>(msg);
if(msg->prevLogIndex != 0) getSimulation()->getActiveEnvir()->alert("FAIL: ...");
if(msg->prevLogTerm != 0) getSimulation()->getActiveEnvir()->alert("FAIL: ...");
if(msg->entries.at(0).command != "test") getSimulation()->getActiveEnvir()->alert("FAIL: ...");
```

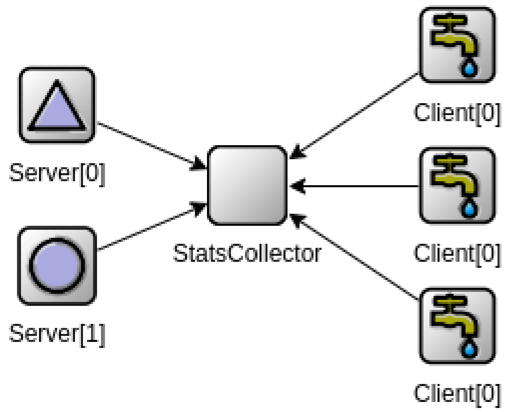
Statistics

After ensuring that the simulation was working properly, we moved to the following step, the **statistics gathering**. This process consists in selecting some quantities to be analyzed and then running a simulation for a long time in order to collect an adeguate amount of data about them.

OMNeT++ provides some **integrated tools** to solve in a easier way this task:
it includes classes to record statistics and organize them.

StatsCollector

Statistics have been collected with the support of an additional simple module, called **StatsCollector**.



```
simple StatsCollector
{
  parameters:
      @signal[consensusTime](type="simtime_t");
      @statistic[consensusTime](title="..."; unit=s;
        record=vector,mean,max,min;
        interpolationmode=linear);

      @signal[consensusMessages](type="long");
      @statistic[consensusMessages](title="...";
        record=vector,max,mean,min;
        interpolationmode=sample-hold);

      @signal[timeToRecoverLog](type="simtime_t");
      @statistic[timeToRecoverLog](title="..."; unit=s;
        record=vector,mean,max,min;
        interpolationmode=linear);

      ...

  gates:
      input fromServer[];
      input fromClient[];
}
```

Configurations

The statistics have been produced with the following **configurations**.

#server	#client	Server crash	Channel omission	Channel delay	Send command timeout	
5	20	-	-	-	0.9 s	Configuration **#1**
5	20	*10-20%	5%	-	0.9 s	Configuration **#2**
5	20	*0.5%	1%	1 ms	0.9 s	Configuration **#3**

Statistic #1

Consensus time

Type: **Duration**

Measures the **duration of the election process**. An election can involve more candidates and can be interrupted and restarted.

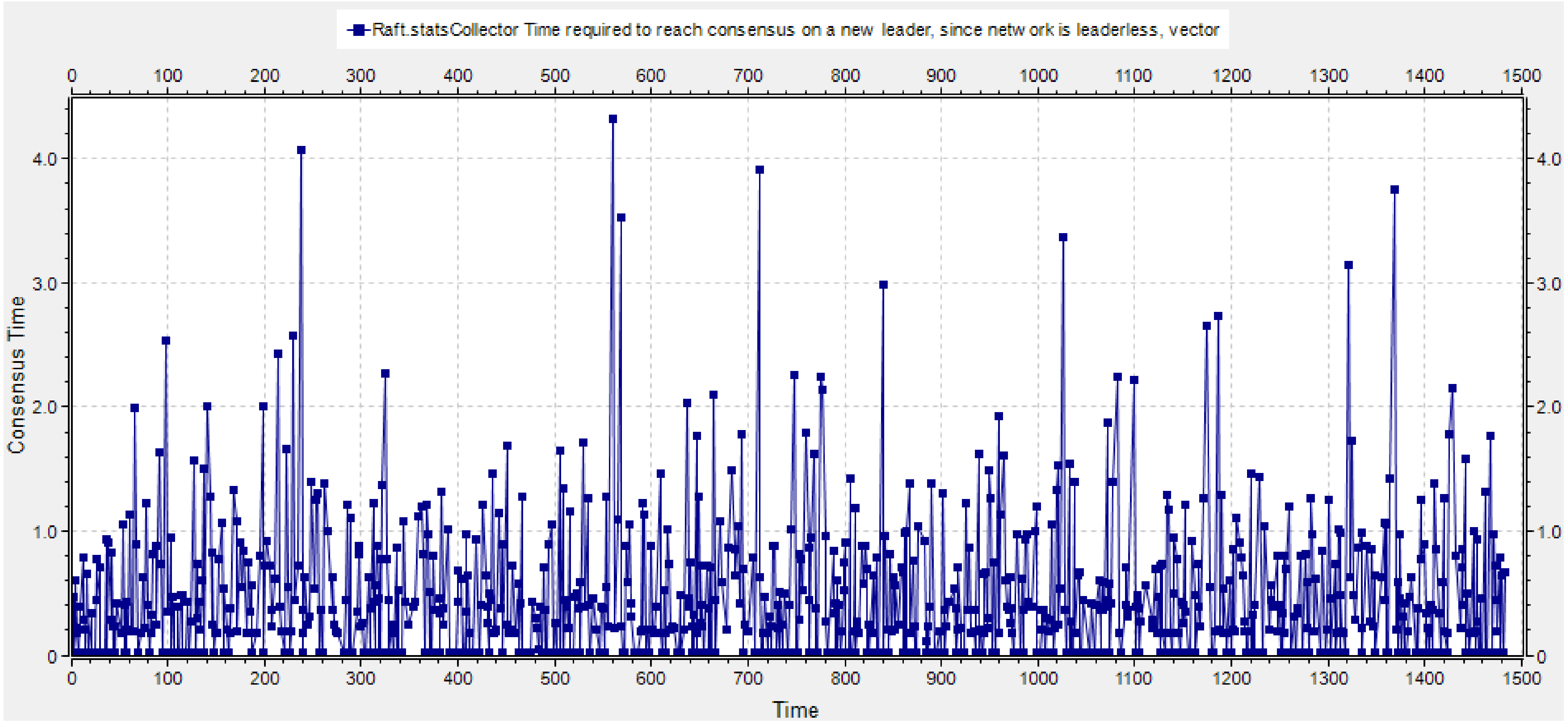
▶ **Start**

The measuring starts when a server detects the absence of the leader, or the leader crashes (immediately or after a while, a new election process will begins).

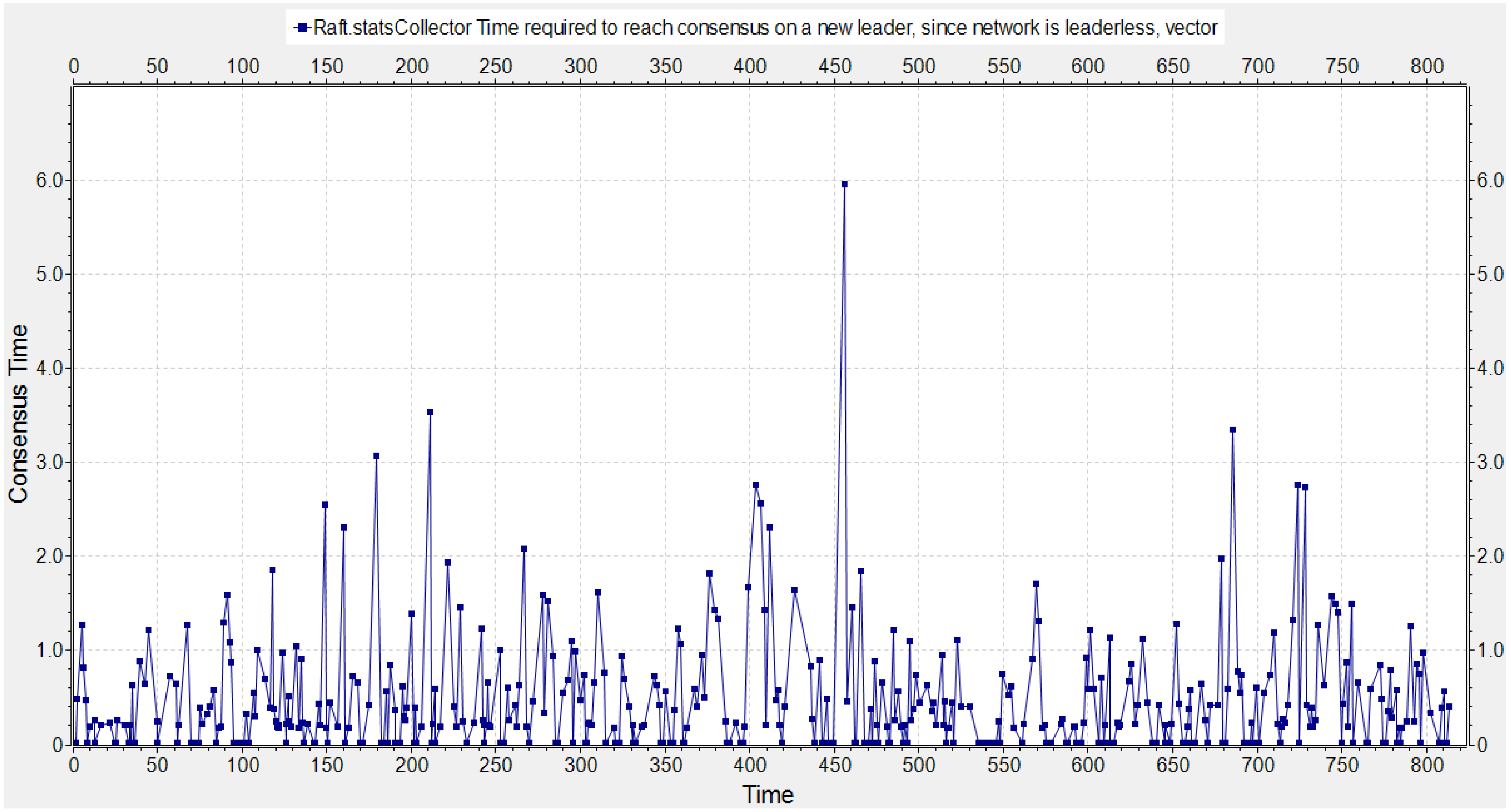
■ **Stop**

The measuring stops when a new leader has been elected.

Configuration **#2** Mean: 0.487 s Max: 4.321 s Min: 0.015 s



Configuration **#3** Mean: 0.460 s Max: 5.954 s Min: 0.016 s



Statistic #2

Consensus messages

Type: **Messages count**

Measure the **number of messages** sent among servers **to elect a leader**.
Only RequestVotes and RequestVoteResponses are counted.

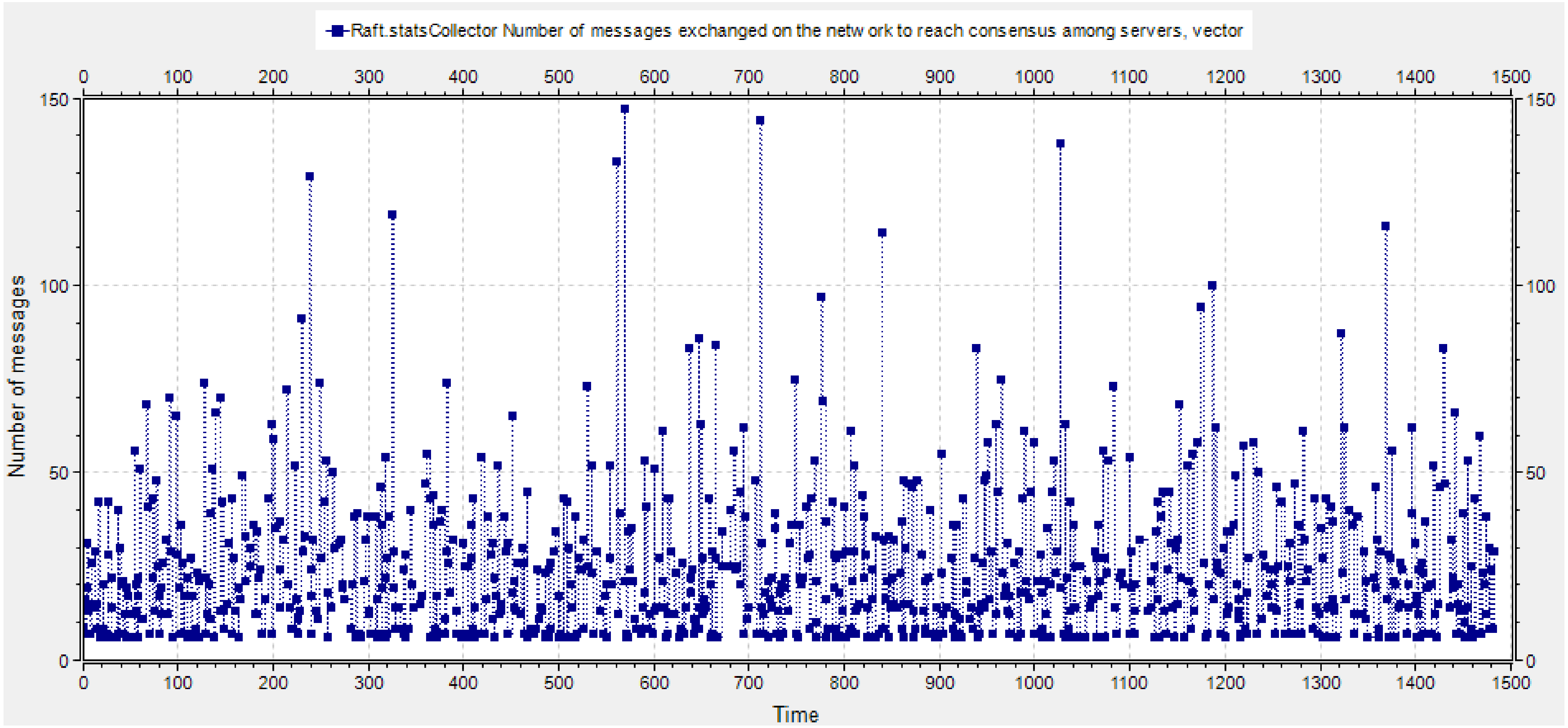
▶ **Start**

The measuring starts when a server detects the absence of the leader, or the
leader crashes (immediately or after a while,
a new election process will begins).

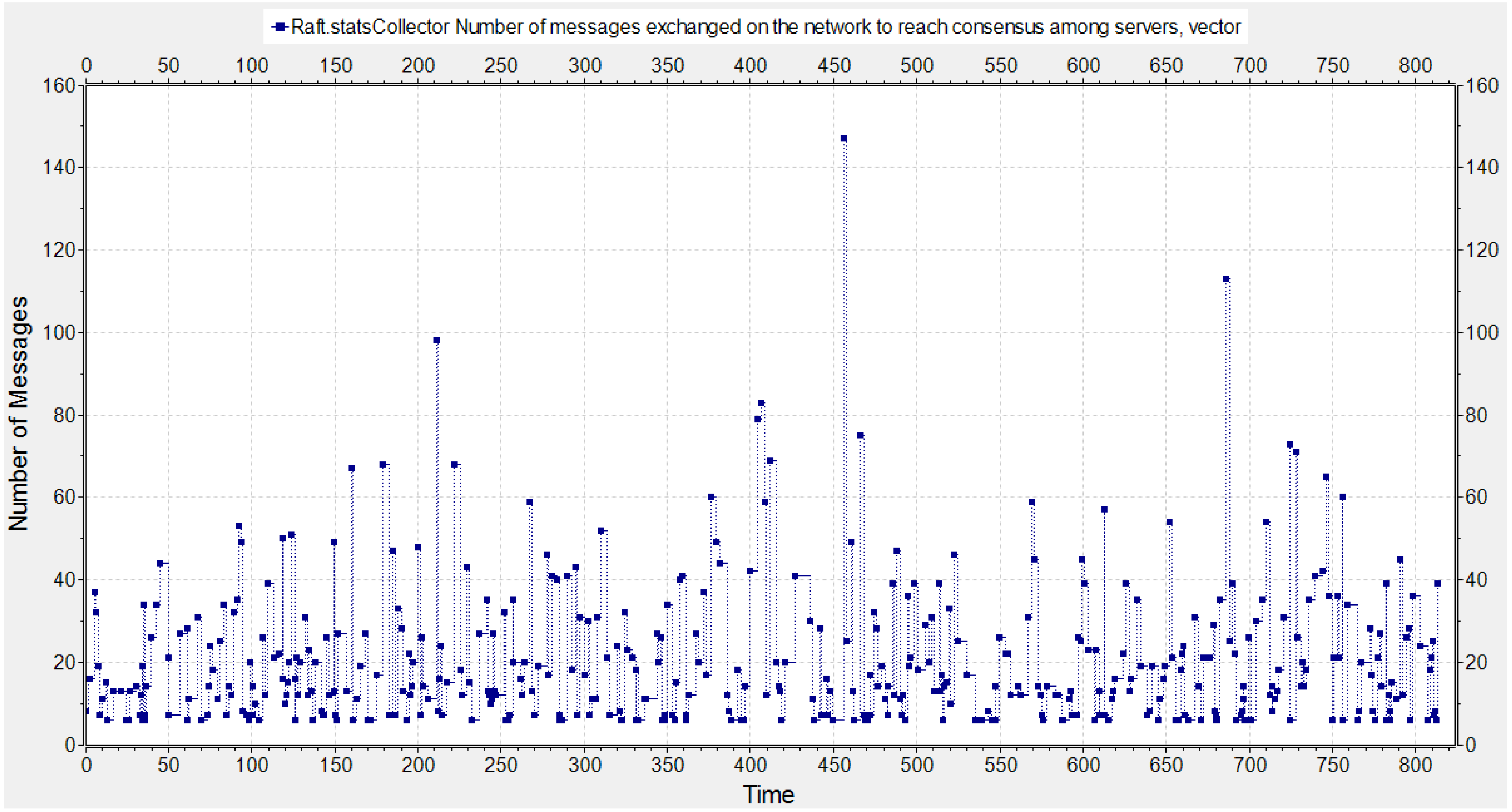
■ **Stop**

The measuring stops when the new leader has been elected.

Configuration **#2** Mean: 23.6 Max: 147 Min: 6



Configuration **#3** Mean: 20.5 Max: 147 Min: 6



Statistic #3

Time to recover

Type: **Duration**

Measures the time needed by a crashed server to fill up the log with the lost entries after a recovery.

▶ **Start**

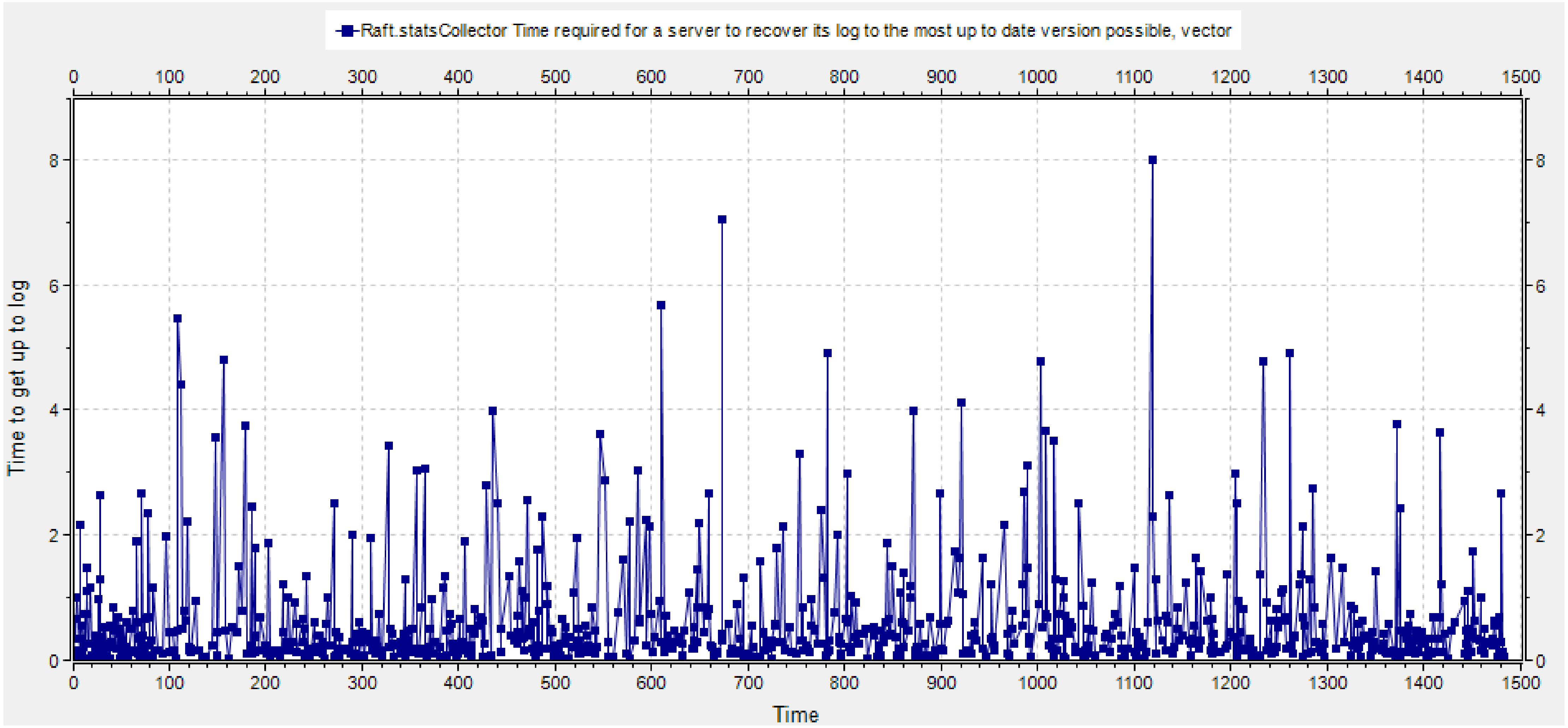
The measuring starts when a server recovers, after being down for a while.

🟥 **Stop**

The measuring stops when the server log is up to date with the last committed entry of the leader.

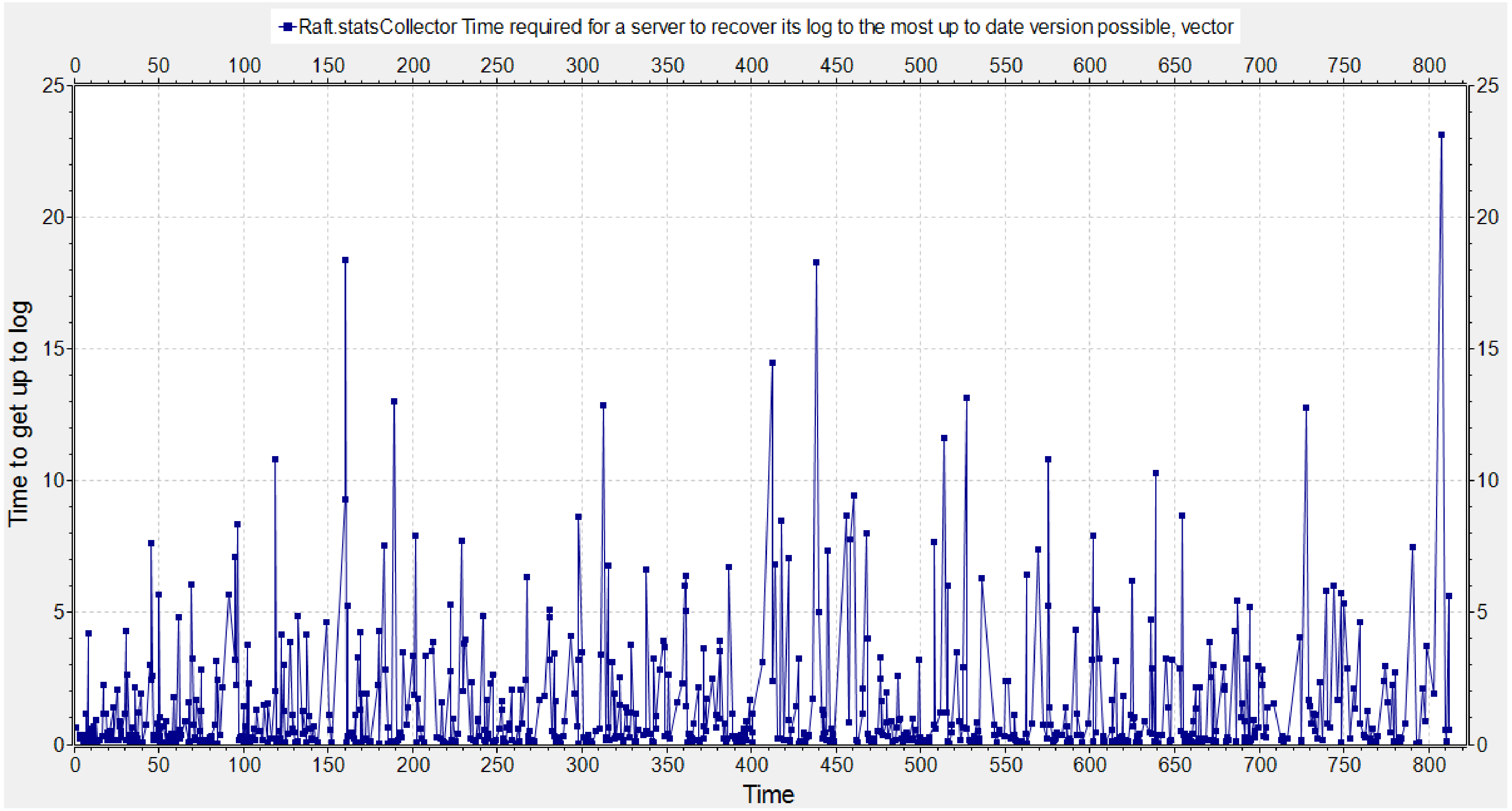
Configuration **#2**

Mean: 2.011 s   Max: 62.813 s   Min: 0.005 s



Configuration **#3**

Mean: 1.437 s Max: 23.125 s Min: 0.001 s



Statistic #4

Command response

Type: **Duration**

Measures the time to successfully **replicate a command** sent by a client and inform the latter.

Start

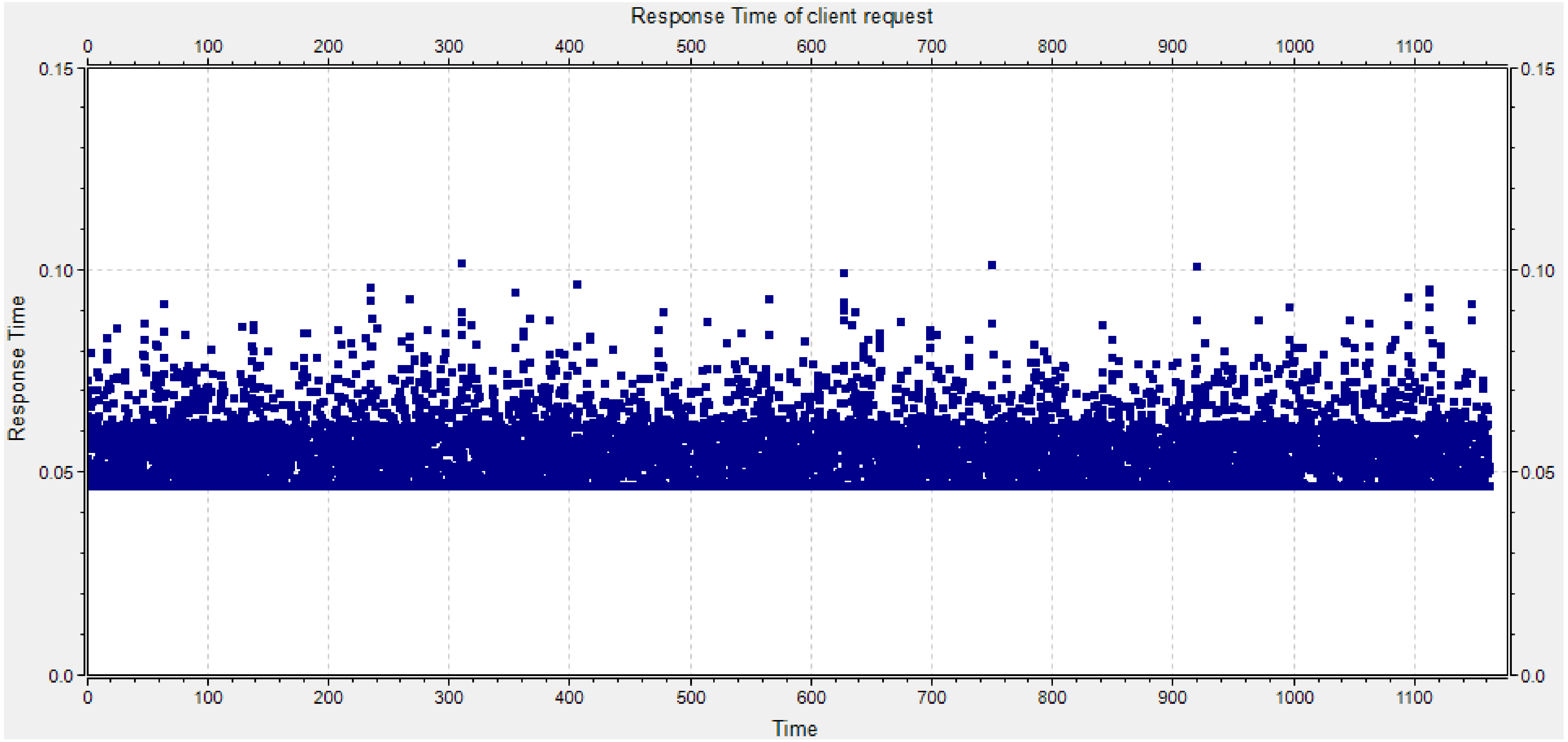
The measuring starts when a client sends the first addCommand request to a server (maybe not the leader).

Stop

The measuring ends when a leader send an addCommand response to the client, informing it that the entry related to the command has been committed.

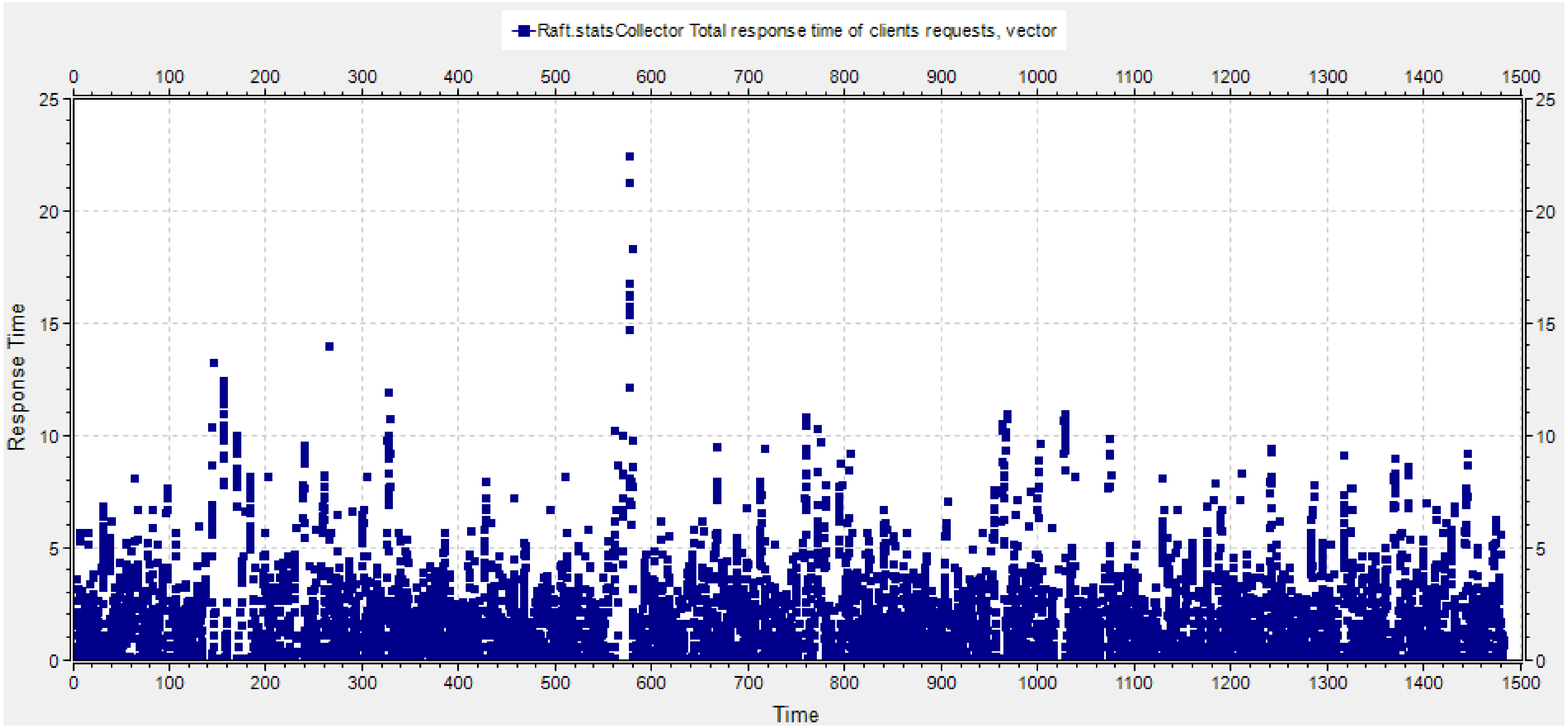
Configuration **#1**

Mean: 0.049 s Max: 0.562 s Min: 0.046 s



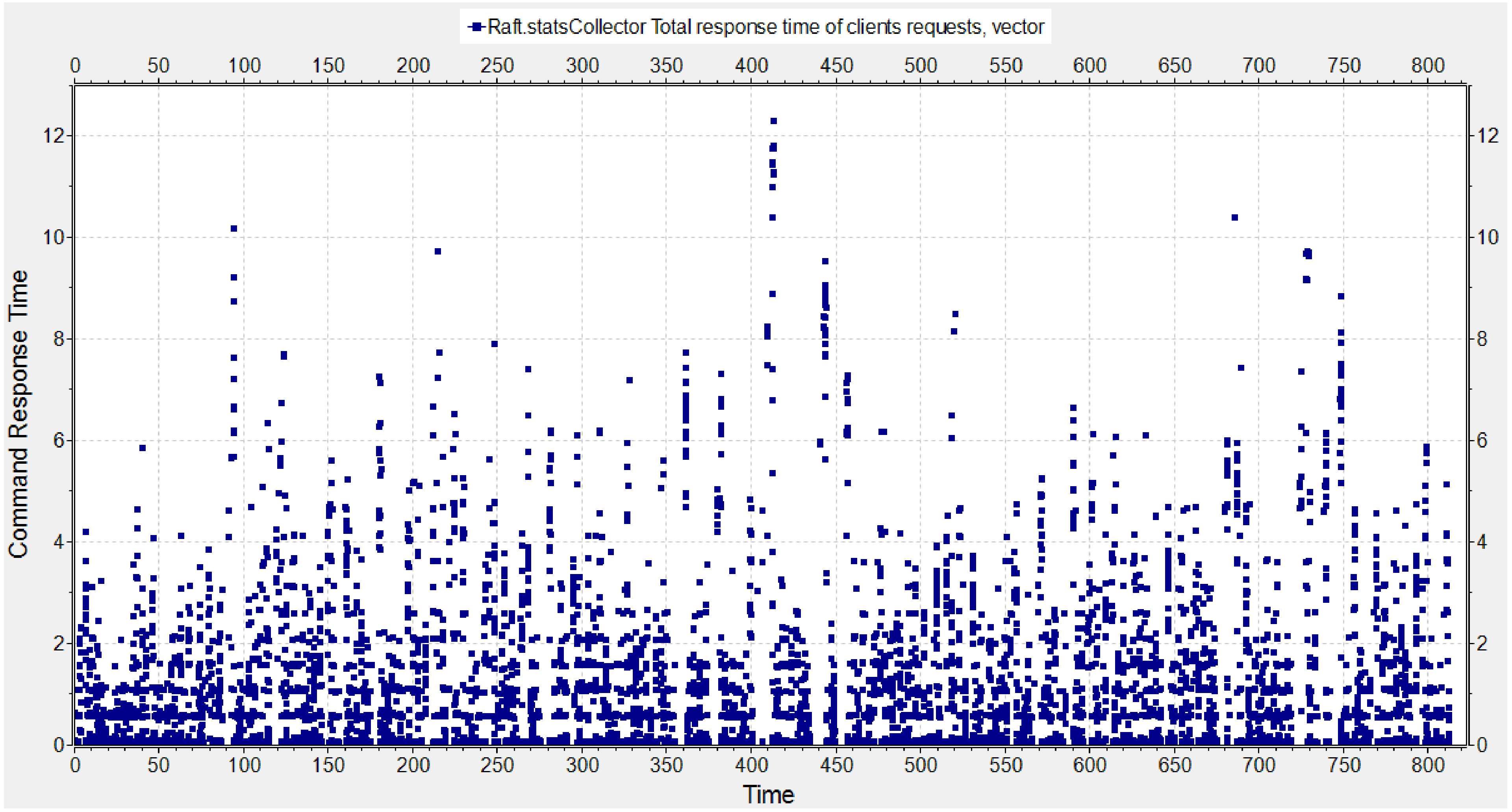
Configuration **#2**

Mean: 1.399 s Max: 22.402 s Min: 0.023 s



Configuration **#3**

Mean: 1.068 s Max: 12.296 s Min: 0.020 s



Statistic #5

Commit messages

Type: **Message count**

Measures the number of messages exchanged between servers to replicate a command sent by a client.

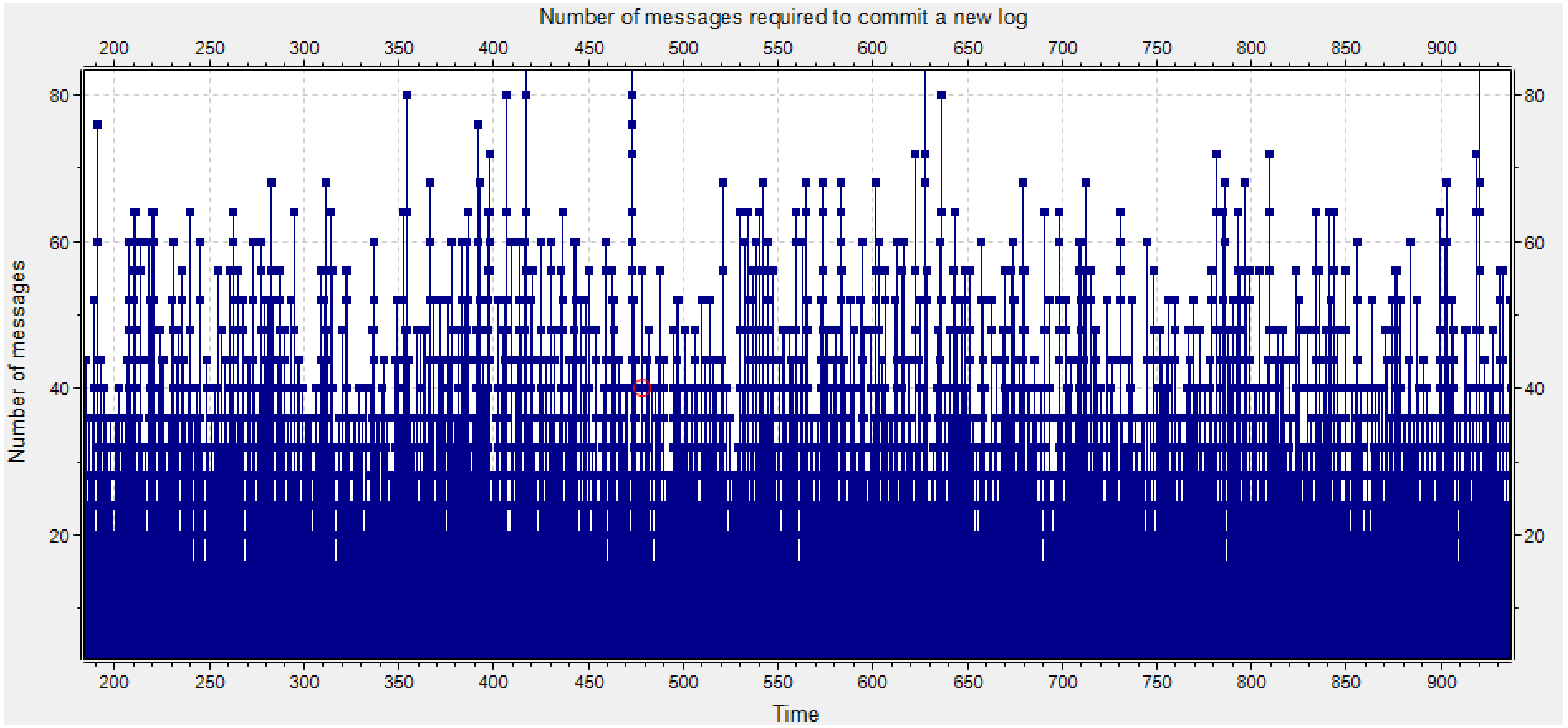
▶ **Start**

The measuring starts when a leader receives an addCommand request from a client and starts the replication process.

■ **Stop**

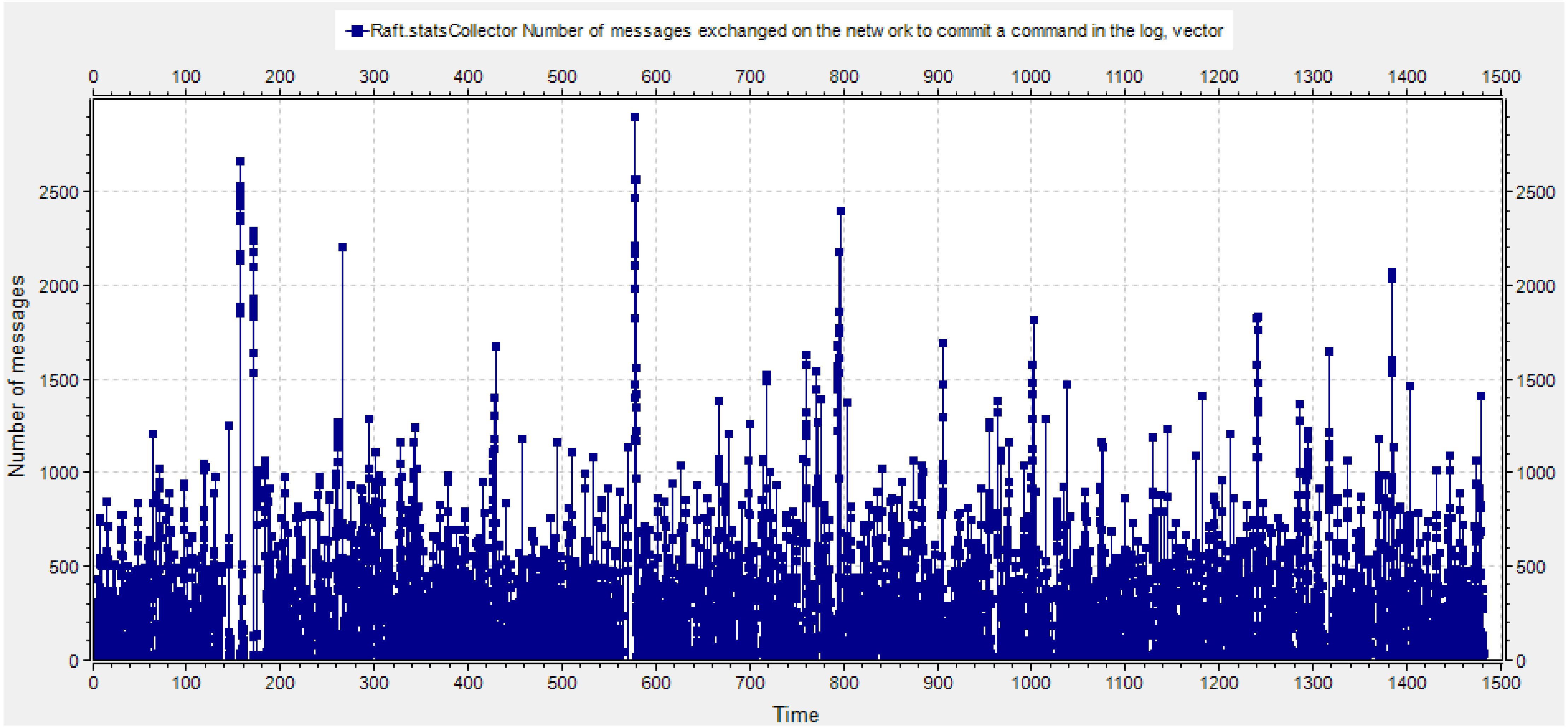
The measuring ends when a leader commits the entry related to the command received from the client.

Configuration **#1** Mean: 18.7 Max: 144 Min: 8

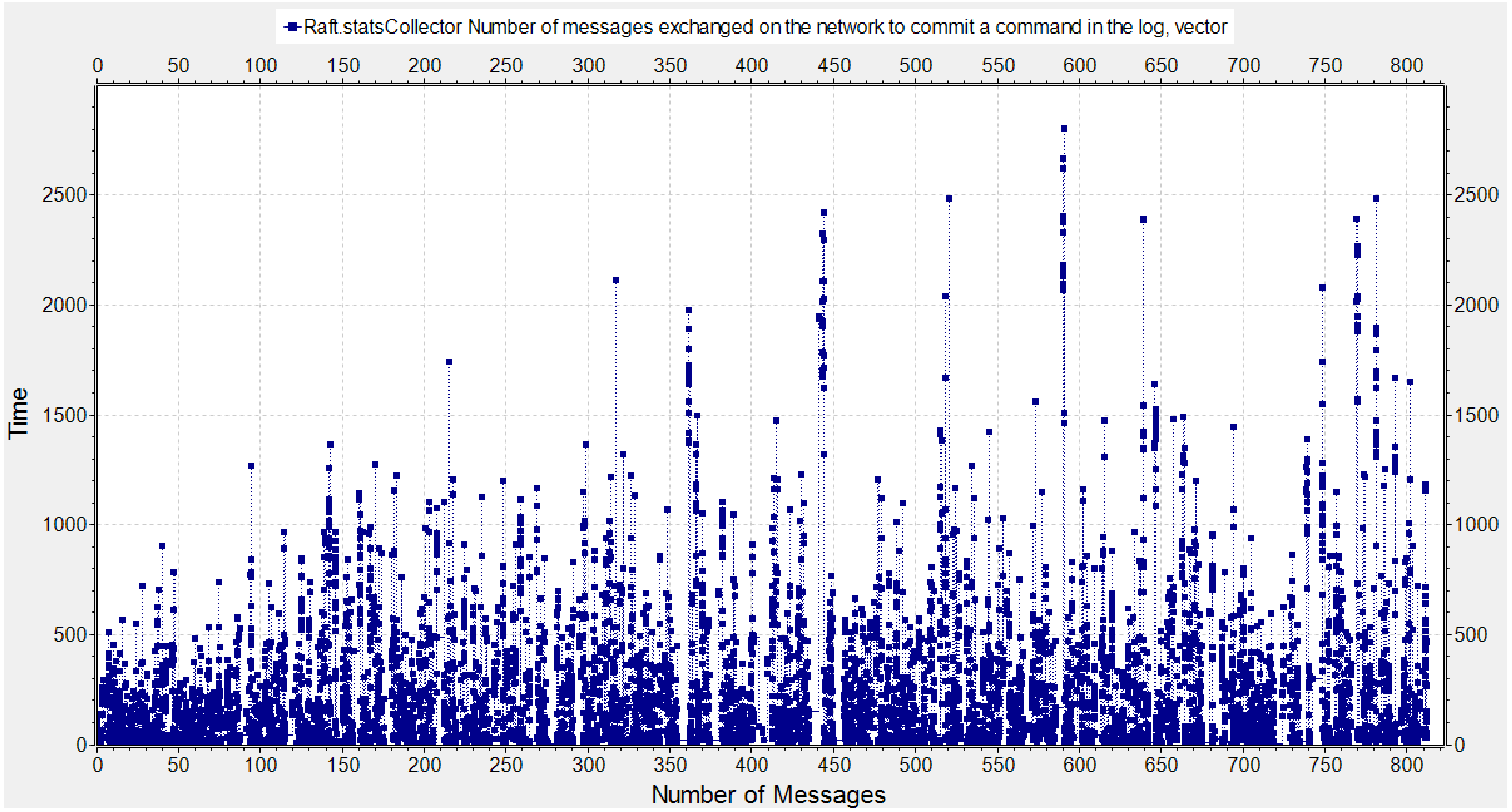


Configuration **#2**

Mean: 228.4 Max: 2897 Min: 2



Configuration **#3** Mean: 233.0 Max: 2801 Min: 2



Final Simulations

To measure the impact of server and clients cardinality

	#server	#client	Server crash	Channel omission	Channel delay	Send command timeout
Configuration **#4**	4	12	*0.5%	0.1%	3 ms	0.9 s
Configuration **#5**	12	12	*0.5%	0.1%	3 ms	0.9 s
Configuration **#6**	12	4	*0.5%	0.1%	3 ms	0.9 s

Final Simulations

To measure the impact of servers and client's cardinality

	Command Response Time (Mean)	Consensus Time (Mean)	Time to recover Log (Mean)
Configuration **#4**	0.9309s	0.6065s	1.2038s
Configuration **#5**	0.8745s	0.2383s	0.6123s
Configuration **#6**	0.7477s	0.18523s	0.4491s

Thanks for your attention