SWIM through the NATs

Fabrizio Demaria, <demaria@kth.se>
Samia Khalid, <samiak@kth.se>

1. Phase 1: Implementing SWIM

In the first phase of the project we implemented SWIM, a scalable decentralised membership protocol. The very first task was to implement the piggyback mechanism to disseminate information. When a node receive a ping message, it replies with a pong message that includes a “piggyback list” carrying the information about its knowledge of the network.

PiggybackList

In order to make the code more readable, we implemented a separate class for managing the piggyback information to be attached to the pong messages: *PiggybackList.* The main content of this class is a list of *InfoPiggyback* objects, each representing the changes in the system for a specific node; the main parameters inside *InfoPiggyback* objects are: *infoTarget* to indicate the node who changed status, *infoType* to indicate the type of status change (in the phase without NAT traversal, those are: NEWNODE, SUSPECTEDNODE, ALIVENODE, DEADNODE).
The *PiggybackList* object in a specific node also keeps track of the dissemination value for each single *InfoPiggyback* in the list: each time the entire list is attached to a pong message, all the dissemination values are decreased by one and the elements that reach zero are deleted from the list. The initial dissemination value, set when a new *InfoPiggyback* object is created, is equal to the *DISSEMINATION_VALUE* parameter found in *SwimComp.*
The *PiggybackList* object also implements a method called *deleteItem* that is able to delete one of the oldest pieces of information in the list (one of the *InfoPiggyback* objects with the lowest dissemination value); this is used to limit the amount of information exchanged via piggyback by setting a maximum size for the list using the parameter *MAX_LIST_SIZE* found in *SwimComp*. This approach eliminates the oldest entries and it works better than erasing random information.

MembershipList

The class *MembershipList* has been implemented to manage the information that each node has regarding the other nodes in the network.
The most important object inside the *membershipList* is the *neighboursNodes* map, that associates the address of every node in the list with a *MemberInfo* object. The *MemberInfo* object has information about the status of node, i.e. if the node is suspected or dead; it also carries the incarnation value for the node.
The *MembershipList* includes methods for different operations to be executed on the list: for example a method to return a random node from the list (*SwimComp* uses it to select who to ping), methods to update the parents of a nated node (used in Phase 2), etc.

Ping processing

Each node implements a timer called *PingTimeout* in order to schedule periodic pings. The target for the ping is chosen randomly among the nodes in the *membershipList,* excluding the ones that have not replied to a previous ping and have not yet being declared as suspected.
If a node B receives a ping from an unknown node A, A is added to the *membershipList* of B; if this is the case, B disseminates the information about the new node. Instead, if A was suspected by B, B unsuspects A and disseminates that A is still alive.
The final part of the ping processing consists of decreasing the dissemination values of the information inside the *PiggybackList,* copying the updated *PiggybackList* inside a pong message and send the pong to node A.

Pong processing

First of all when a node B sends a pong message to node A, if B was suspected by A it is declared unsuspected by A. Then the *PiggybackList* inside the pong is scanned. For each *InfoPiggyback* in the list, the *membershipList* is updated accordingly: if the information is new for node A (i.e. it causes a change in the *membershipList* of A), A adds it to its own *PiggybackList* with dissemination value set to maximum.

If the target of a *InfoPiggyback* inside the pong from B is A itself, then A processes the information differently: for example if B is declaring A as dead, A starts to disseminate *NEWNODE* messages about itself and if B is declaring A as suspected, A starts to disseminate ALIVE messages.
Message ordering is a main issue with the piggyback dissemination mechanism. Here we provide an example: if a node A is disconnected for a short enough period, it will be declared as suspected and soon after as alive by some nodes in the network; this means that the network will contain simultaneously both SUSPECTEDNODE and ALIVENODE messages about A. Consequently, the other nodes will continuously update the status of A in their list, thus disseminating the information that will never converge to the final correct status of A. To deal with this issue we implemented the incarnation value, presented in the next section.

Incarnation value

When an *InfoPiggyback* about a status change of node B is created by node A, it is associated to the current incarnation value of B in the *membershipList* of A. Let's consider the case in which node C has to update the information about the status of node B according to the *InfoPiggyback* contained in the pong message from A. At this point, the *membershipList* methods in C (namely *add, remove, suspectNode* or *unsuspectNode*) compare two incarnation values for B: the one in the *InfoPiggyback* with the one already present in the list of C (i.e. the one in the *MemberInfo* associated to B). If the information in the piggyback turns out to be newer for C, it updates the status and the incarnation value of B in its own members list (updating the *MemberInfo* of node B in the *neighboursNodes* HashMap).
The incarnation value for node B can be only incremented by B when it receives wrong piggyback information about itself. This is done in the pong handler in *SwimComp*: for example if node B receives SUSPECTEDNODE about itself, it sets its new incarnation value greater than the one received and disseminates ALIVENODE information with this new value. Similar considerations can be done for other type of status changes, i.e. NEWNODE and DEADNODE.
As a last remark, the incarnation value mechanism not only affects the piggyback processing, but also any method that deals with updating the *membershipList* of a node: it is mandatory to pass the incarnation value along with the targeted node to the methods *add, remove, suspectNode* and *unsuspectedNode* in *membershipList.*

K-indirect procedure

We implemented the k-indirect procedure by adding the following massage types:

- *NetPingReq* is sent to *k* random nodes by node A. The parameter *k* can be configured via *K_VALUE* in *SwimComp*. The content of this message includes the final target of the pinging request: node B;
- *NetPingResp* is sent back to node A if one of the *k* nodes successfully pings B.

When an indirect node C receives the *NetPingReq* message from A, it starts a normal pinging procedure towards B. It also starts a timer via *scheduleDeleteReq* (time value: *DELETE_REQ_TIMER*): until this timer is over, if C receives a message from B, C sends back a *NetPingResp* to A. Regarding A, upon sending the *NetPingReq* messages it also starts a timer *IndTimeout* (time value: *INDIRECT_PING_TIMER*): if no *NetPingResp* is received within the latter timeout, B is declared as dead by A.

2. Phase 2: SWIM through the NATs

Croupier samples

The first steps in phase 2 consisted in connecting Croupier in order to being able to retrieve open nodes samples in the *NatTraversalComp.* Each nated node in the system maintains a list of parents, whose length is determined by the parameter *MAX_NUMBER_PARENTS.* When a node is spawned, the bootstrap nodes are assigned as parents. Every time a node receives a sample from Croupier (via *handleCroupierSample*), if the current parent list has less parents with respect to the maximum allowed, one of the nodes in the sample is added (we check that the node to be added is not already part of the parent list).

Detecting dead parents

We tried two different approaches to detect dead parents. At first, we used the *SwimComp* to notify the *NatTraversalComp* (via *DeadNotification*) every time a node was marked as "dead" in the *membershipList*. We noticed that in this way we need to wait for the system to converge completely (in the worst case), in order for the nated node to detect the death of one of its parents. So we switched to the heartbeat mechanism, implemented right into the *NatTraversalComp:* in this case the ping/pong messages' content

is of type *ParentPing,* the pinging periodic timer is set via the *PERIODIC_PARENT_PING* parameter and the acknowledge timer is set via the *PARENT_ACK_TIMER* parameter. We do not implement suspicion mechanism in for the failure detection of the parents.

Disseminate information about the parents

When the parent list is locally updated in a nated node, this information has to reach all the nodes in the network in order for them to select the right relays. The information dissemination mechanism of SWIM is adopted also for this specific case: if a parent is removed/added, *NetTraversalComp* notifies *SwimComp via NetDeadParent/NetNewParent* messages, and consequently the DEADPARENT/NEWPARENT *InfoPiggyback*s start to be disseminated.
The dissemination and processing of this new kind of information greatly resembles the previously mentioned procedures for Phase 1 (we just added new switch cases in the pong processing of *SwimComp* and new methods for the parents' updating in the *membershipList* class).

Tabu list

An issue that we encountered in implementing the dissemination of the information about dead and new parents for nated nodes was very similar to the problem related to the ordering of messages and incarnation values: if a parent is only temporarily disconnected, it might be chosen again as parent soon after being declared as dead. In this way the network will simultaneously contain messages declaring the parent as removed and added, preventing the system from converging.
In this case we solved the issue implementing a tabu list: an open node can only be chosen once as parent for a specific nated node. This is a simple and effective approach to solve the problem and it can be considered acceptable if the parent nodes don't crash too frequently. Moreover, the oldest entries in the tabu list can be erased after a certain amount of time, when it is sure that the NEWPARENT won't mix with older DEADPARENT messages in the network, allowing for old parents to be chosen again.

NatEmulatorComp

During our development of Phase 2 of the project, we encountered the following problem: even if we were able to update the parents for a nated node and successfully disseminate this information to the other nodes, ping/pong through NAT was not working properly; if a node A wanted to ping or pong a nated node B by selecting as relay one of B's new parents, let's say parent C, the message was indeed relayed by C but it was later dropped by B at a layer lower than *NatTraversalComp.* Once we have been given the source code for the class *NatEmulatorComp,* we realised that the method *allowMsg* was dropping our messages because it was also performing the checking against the parents by using an older version of the parent list, i.e. the list provided during bootstrap that was never updated in this lower layer.
We solved this issue by performing the checking of the parents via the updated parent list (by inspecting the parent list inside the *NatedAddress* in the message itself).
This problem was originated because we had to perform deep copies of the *NatedAddress* elements in the system instead of updating the originally provided *selfAddress,* as we explain in the following section.

3. General problems and solutions

NatedAddress and deep copies

Our simulation is running in a single JVM, so by passing the various *NatedAddress* references to the nodes in the network, any modification of one of those in any component would have been applied to all the other references in the other components. This was not a problem in Phase 1, because *NatedAddress* objects were never modified. In Phase 2, we faced the problem when we started to modify the parent list inside the *NatedAddress* references of the nated nodes.
For example, in our implementation we have, in the *membershipList,* the following HashMap for the list of nodes: <*NatedAddress*,*MemberInfo*>; if we modified the parents of a nated node within the node itself, the list was updated for all the *NatedAddress* objects in the network. First of all, this was not correct since we need to properly disseminate the information in order to perform the update. Moreover, this modification would corrupt our HashMaps in the various *membershipList,* because the hash value of the *NatedAddress* elements would change, losing the reference to the corresponding *MemberInfo*.
In order to solve this problem, we extensively adopted *deepCopy* methods in order to emulate a real scenario. This method simply creates a new *NatedAddress* object with the updated information (about the parents, in our case). It is important to underline that this modification was rather intrusive and it caused many related issues even in the lower framework, as we mentioned in the previous sections. For example, when updating the *selfAddress* object in *NatTraversalComp* it was crucial to explicitly propagate this new

information into the corresponding *selfAddress* in *SwimComp,* to achieve consistency among the components in a simulated node.
Moreover, in order to compare two *NatedAddress* objects independently from their parents (just checking if the node is actually the same), we always used the node IDs for this check. As a last remark, we believe that the overall performance was slightly affected by the several deep copies operations and node IDs retrieval.

List of dead nodes

In order to be sure to always reach convergence, we had to keep in the *membershipList* all the information of the nodes even when they are declared as dead (we just flag them as "dead"). This was due to the incarnation value, that can't be lost by deleting the record from the HashMap. These values can be anyway deleted after a certain amount of time, when convergence is reached and no more piggybacks about the node exist in the network.
Please, note that in the *SwimScenario* only the methods to create and kill peers have been adopted: to simulate a temporary disconnection we actually killed and respowned the nodes (this means that they reenters the network with empty membership lists).

4. Experiments

Obtaining the graphs

We periodically send to the *AggregatorComp* a *Status* message containing the number of nodes in the *MembershipList* as well as the number of *InfoPiggyback* for each type of information, i.e. NEWNODE, DEADNODE, SUSPECTEDNODE, ALIVENODE, NEWPARENT, DEADPARENT. These values are printed to the log periodically with a fixed period set via the parameter *PERIODIC_STATUS.* The size of the membership list for each node, once plotted against time, tells about the convergence of the system upon nodes' spawning and removal. Moreover, all the piggyback values are a good indicator of the global dissemination procedure in the system. We also developed some scripts to perform the Simple Moving Average of the data in order to obtain smoother lines in the resulting graphs.
When the simulation is over, the log is automatically processed by ad-hoc bash and awk scripts that extract the interesting log entries and process them to finally generate a single file containing as a first column the timestamps and in the remaining columns the above mentioned information.
All this elaborated data is eventually processed by gnuplot to display the graphical results. In order to achieve this, some more scripts have been written for gnuplot.
The entire operation has been made straightforward by merging scripts into a single *Simulation.sh* capable of running the jar of the project with the chosen scenario and automatically starting the various scripts to reach the final graphs.

Tested scenarios

We are going to list the main tested scenarios, including the most relevant configuration settings and considerations, followed by the graphs. For more details about the various scenarios (timers values, k indirect-nodes value, etc.), it is possible to inspect the *SwimMain* and *SwimComp* classes.
Please, note that the x-axis of the graphs doesn't show the relative but the actual timestamp ([HH:MM:SS]) of execution (in some graphs the hours are omitted, too).

- **FIGURE 1 (simpleX):** testing the convergence of the system by varying network sizes (10, 50, 100, 500 nodes). All the nodes are open. Piggyback list size is 50 and dissemination value is set to 100. K value for indirect pinging is 5. The 500 nodes scenario converges in about 4 minutes. We can notice a reduction of performance when surpassing the 300 nodes: this is not due to a weak dissemination algorithm, but the problem is that after the introduction of deep copies for the *NatedAddress* objects, the spawning of the nodes is much slower (before deep copies, we managed to keep the same positive slope in the membership graph even with 2000 nodes).

- **FIGURE 2 (pbSizeX):** testing the effects of varying the amount of information exchanged: maximum size of the *PiggybackList* set to 5, 10, 25 and 100. The scenario involves: spawning 100 open nodes and temporarily disconnecting 20 them; even if this is not reported in the graph, we performed the test with a *PiggybackList* size equal to *n*log(n)* (i.e. 200), but in this case the performance doesn't increase with respect to the case of *pbSize=n* (i.e. 100).

- **FIGURE 3,4,5 (simpleDead100):** testing the case of 100 nodes reaching convergence, followed by 20 nodes removal. Figure 3 represents the overall convergence of the membership lists in the network,

Figure 4 shows the amount of newnode/deadnode information disseminated over time and Figure 5 shows the amount of suspected/alive information disseminated over time.

- **FIGURE 6,7,8 (simpleDeadRevive100):** similar to the previous scenario, but the 20 nodes are only temporarily disconnected. We disconnected the 20 nodes for a period short enough to cause the dissemination of ALIVE messages and test their correct behaviour, as shown in Figure 8. Moreover, this scenario shows that the entire incarnation number mechanism works, since the messages are now ordered and the overall system reaches convergence.
- **FIGURE 9,10,11,12 (natedBoot):** 100 nodes (50 open, 50 nated) are spawned. Then 10 random nodes are killed (8 open and 2 nated). At the beginning new parents are assigned to certain nated nodes (for example node 3 starts with only 2 as parent), to reach the maximum parent list size (5 parents): we can see that this information is disseminated by looking at the green line on the left side of the graph of Figure 11. Information about the updating of the parents for certain nodes is propagated and everything converges: *all the nated nodes ends up with 5 working parents, and all the open nodes are aware of the correct parents of each nated node in their membership list.*
- **FIGURE 13 (simpleKillX):** 200 open nodes are spawned, then a number of those is killed (10, 50, 100, 150). This scenario tests convergence of the membership list with different churn rates.
- **FIGURE 14 (natedBoot10open):** similar scenario with respect to *natedBoot,* but in this case system is made up of only 10 open nodes and 90 nated. In this case, 5 nated and 5 open are killed after the spawning of the initial 100 nodes. The system converges as in *natedBoot* (every node left in the system has 89 nodes in their membership list), but it takes a bit longer to stabilise perfectly.

Graphs

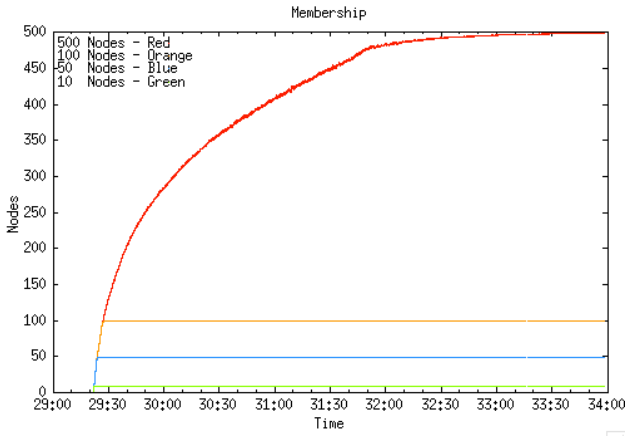

Figure 1

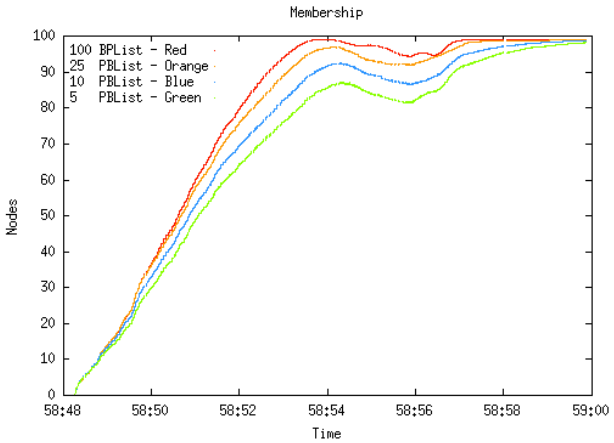

Figure 2

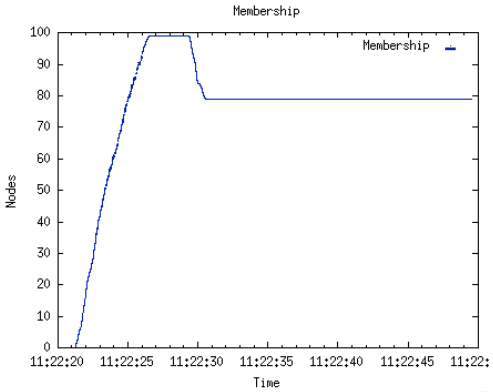

Figure 3

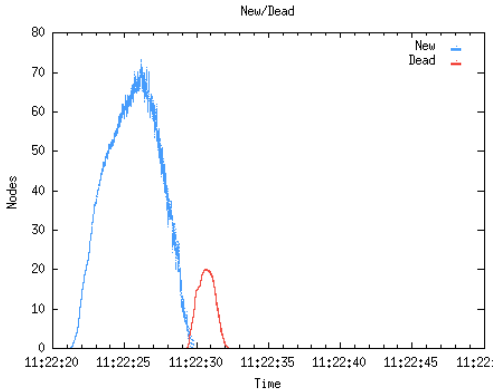

Figure 4

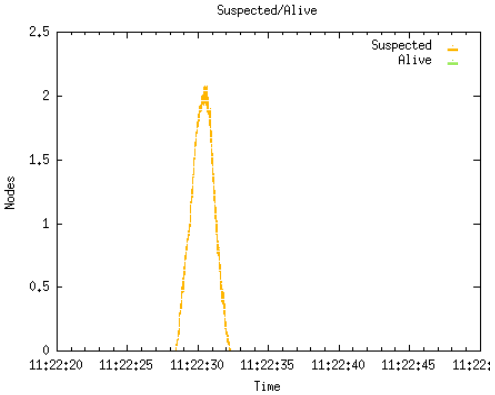

Figure 5

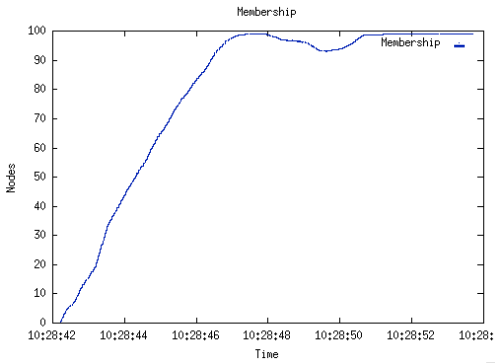

Figure 6

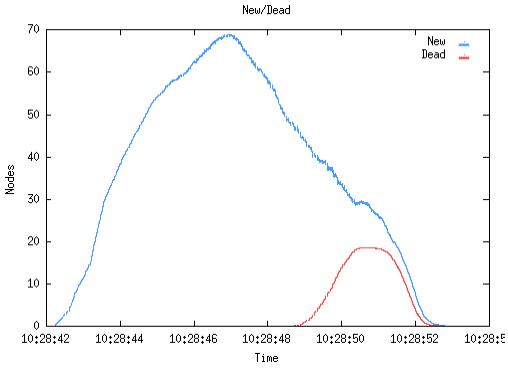

Figure 7

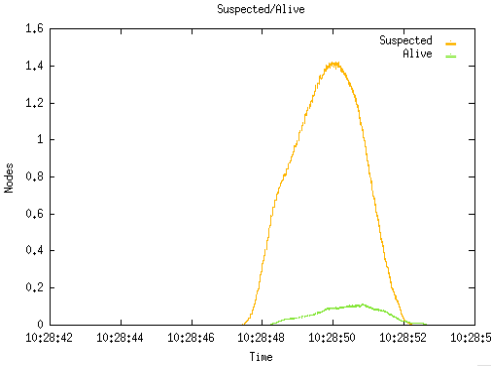

Figure 8

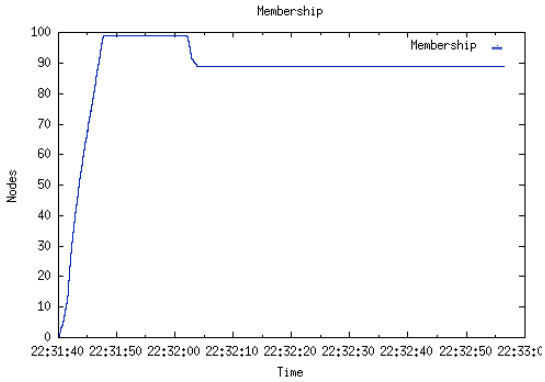

Figure 9

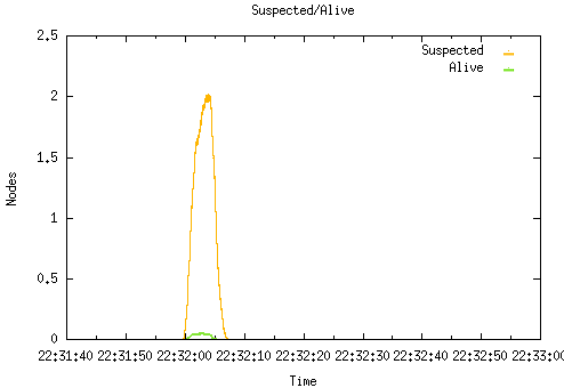

Figure 10

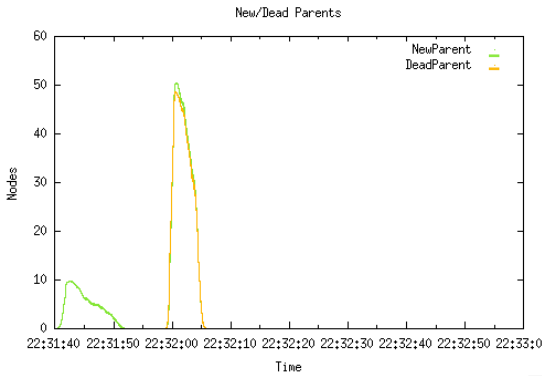

Figure 11

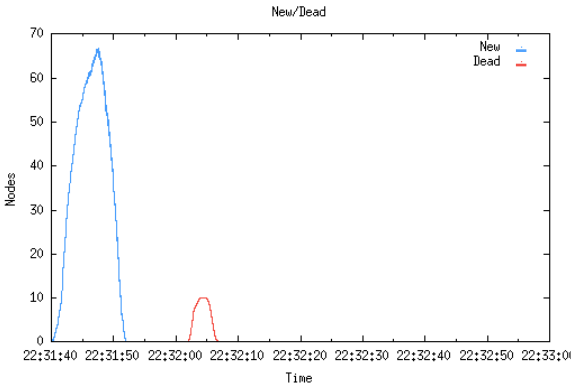

Figure 12

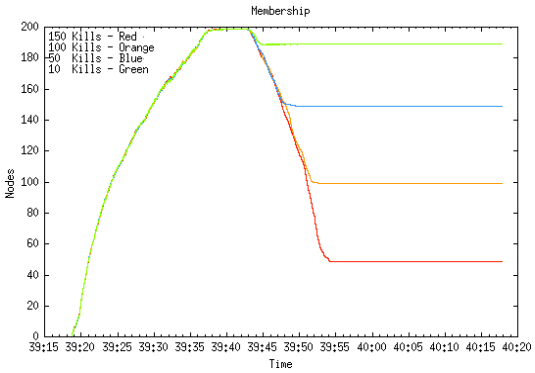

Figure 13

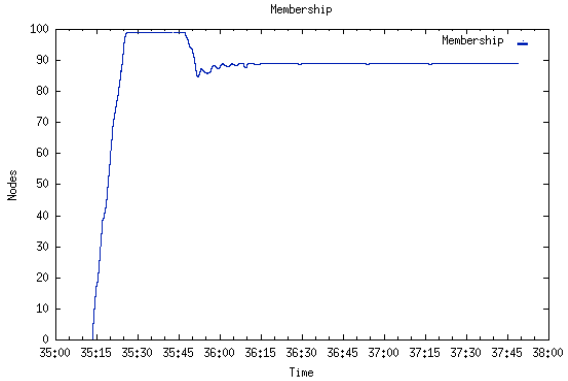

Figure 14

APPENDIX A

How to run the scenarios

We modified the *SwimMain* class in order to set various parameters according to the name of the scenario passed as first argument from command line; it is possible to run the scenarios listed in the report by passing as parameter to the SWIM jar the corresponding name (the name inside the parenthesis in the experiment section). If the name contains a "X", it means that different fixed values are selected for a certain variable in the scenario: substitute the "X" in the name with the desired value.

We prepared a unique a jar file and some scripts in order to easily test the project.
First of all it is necessary to install *gnuplot* on the machine. After having modified the permissions for the various scripts, it is possible to run *Simulate.sh* passing as first argument the name of the scenario. For example, to run the scenario called *simple* (by choosing 100 nodes to be spawned):

```
# ./Simulate.sh simple100
```

This script will generate the log file as well as the *result.dat* file, containing the data to be processed by gnuplot. To show the graphs, it is possible to simply run gnuplot passing the corresponding script to show the membership, suspected/alive, new/dead or newparent/deadparent graph. For example:

```
# gnuplot ./membersPlot.sh
# gnuplot ./newdeadPlot.sh
# gnuplot ./suspectedaalivePlot
# gnuplot ./parentsPlot.sh
```

The package includes the SMA.sh script to perform the Simple Moving Average of the data point in order to get smoother lines. To operate it, simply execute the script passing as first parameter the number of values over which perform the average.

Finally, to generate the plots of figure 1, 2 and 13 more complex scripts involving various steps have been adopted and are not included in the submitted package.