Politecnico di Milano - Department of Electronics, Information and Bioengineering

POLITECNICO
MILANO 1863

BarbequeRTRM - Distributed Manager

[Coding Project]

Student Claudio Scandella
ID 853781

Course Advanced operating systems
Academic Year 2018-2019

Advisor Giuseppe Massari, Michele Zanella
Professor William Fornaciari

October 3, 2019

Contents

1 Introduction 3
- 1.1 Problem statement . . . 3
- 1.2 Summary of the work . . . 3

2 Design and implementation 4
- 2.1 Fully vs Hierarchical distributed system . . . 4
- 2.2 Proto file . . . 5
- 2.3 Discover messages . . . 6
- 2.4 Ping messages . . . 6
- 2.5 DistributedManager Task . . . 7
- 2.6 Instance tags . . . 9
- 2.7 Discover and DiscoverInstances functions . . . 9
  - 2.7.1 DiscoverInstances . . . 9
  - 2.7.2 Discover . . . 9
- 2.8 Ping and PingInstances functions . . . 10
  - 2.8.1 PingInstances . . . 10
  - 2.8.2 Ping . . . 10

3 Experimental evaluation 12
- 3.1 Experimental setup . . . 12
- 3.2 Results . . . 12
  - 3.2.1 Fully distributed system . . . 12
  - 3.2.2 Hierarchical distributed system . . . 15

4 Conclusions and Future Works 20

1 Introduction

The Barbeque Run-Time Resource Manager (BarbequeRTRM) is a framework that manages platform resources at real-time in order to give to running applications the needed amount of computational power accordingly to their demand in power consumption, target performance, heat dissipation and possibly many other characteristics.

1.1 Problem statement

BarbequeRTRM is currently able to manage different platforms and execution environments but it only manages resources on a local machine. Since in real world many applications run on different, and likely remote, machines the framework can't fit such situations, thus its use has a narrow field of application.

1.2 Summary of the work

This work wants to pave the implementation of a distributed framework that is able to communicate with different instances on different machines in order to widening the range of applications in which it can be helpful.

2 Design and implementation

Currently, the BarbequeRTRM framework can be configured either to work in a fully distributed way or in hierarchical distributed way. The two ways share some characteristics, but both need an ad-hoc implementation. In the fully distributed system all instances act in the same way while in the hierarchical distributed system all instances have a role, that change at run-time under certain circumstances, so they act in a different way accordingly to it. An instance configured to run in a fully distributed mode can't communicate with an instance configured to run in a hierarchical mode, and vice-versa.

The BarbequeRTRM Daemon is the core of the framework and contains all the logic split into different components which have a specific role. Between them there is the so called AgentProxyGRPC, a module that aim to make an instance communicate with other ones. Currently, AgentProxyGRPC is partially implemented with server functionality (listen to other instances requests) and client functionality (queries other instances).

In this work AgentProxyGRPC functionality is expanded with two functions Discover and Ping:

- Discover: makes an instance to open a connection with another instance through GRPC on a certain IP address. If the connection is established then the other instance is running, thus discovered;
- Ping: makes an instance test the quality of the connection with a previously discovered instance. It collects round trip time and availability (percentage of pings returned).

2.1 Fully vs Hierarchical distributed system

In both configurations each instance maps each other in a double map data structure (sys_to_ip_map and ip_to_sys_map), with an identifier (ID) and an IP address (IP) on which the instance is listening on. The act to update sys_to_ip_map and iP_to_sys_map is called track and untrack of an instance that must be inserted into them and removed from them, respectively.

In a fully distributed system each instance has a mapping that is independent of the one of the other instances. Each instance assigns its ID = 0 (so every instance has a map ID->IP that is unique over the instances) and autonomously assigns other instances ID. When the local instance discovers a new remote instance it assigns an ID that is the lower number greater than 0 that is not yet assigned (i.e. the first free ID). When the local instance does not discover a remote instance that it is tracked then it is untracked. The ID of the remote instance that is no longer tracked is freed up and can be assigned to the next new discovered remote instance. The fully distributed system has a quite simple mechanism. Things get harder in the hierarchical configuration.

In a hierarchical distributed system it is a requirement that each instance keep the same mapping between IDs and IPs. In this configuration instances can have different roles:

- NEW: this is the role to which every instance that is just launched is initialized at;
- MASTER: this role is assigned to exactly one instance at a time in the whole system. The assignment is done in different ways based on situation;
- SLAVE: this role is assigned to any other instance that is no more NEW and can't be MASTER.

Further details of roles are explained later.

When a new instance is launched and it discover no remote instance then it declares itself as MASTER.

The successive remote instances launched will discover the MASTER which will set them as SLAVEs giving them an ID. When the MASTER dies then the remote instance with the lower ID (all instances knows the exact ID of each of them) becomes the new MASTER. If a NEW instance tries to discover remote instances while the new MASTER is being assigned it wait until the assignment is done. It is important to specify that an instance knows the changes in system topology only when it sends a discover message. Thus, every instance has the same system view if and only if every instance has done at least a discover after the last topology change.

2.2 Proto file

Two new RPC functions are needed in RemoteAgent service: Discover and Ping. Discover send a new type of message, DiscoverRequest, as request and wait a new type of message, DiscoverResponse, as response. In GenericRequest and ResourceStatusRequest messages dest_id field is no longer needed so it is deleted.

Figure 1 and **Figure 2** show the new RPCs and new messages.

```
rpc Discover(DiscoverRequest) returns (DiscoverReply);
rpc Ping(GenericRequest) returns (GenericReply);
```

Figure 1: Discover and Ping RPCs added to the RemoteAgent service in agent_com.proto file.

```
message DiscoverRequest {

  enum IAm {
    INSTANCE  = 0;
    NEW   = 1;
    MASTER = 2;
    SLAVE = 3;
  }

  IAm iam = 1;
}

message DiscoverReply {

  enum IAm {
    INSTANCE  = 0;
    SLAVE   = 1;
    MASTER = 2;
  }

  IAm iam = 1;
  uint32 id = 2;
}
```

Figure 2: New messages used in Discover RPC defined in agent_com.proto file.

2.3 Discover messages

The discover routine is the most important one because allows instances to know each other so they can "see" the topology of the system in the time instant they execute it. As said before, the framework can run in two different modalities: fully and hierarchical.

In fully mode each instance has the same role called INSTANCE. When an INSTANCE sends a discover message it set the iam field of DiscoverRequest message to INSTANCE so the other instances knows that that message is from an instance that in running in fully distributed mode. As a response, the instance that receive the DiscoverRequest message replies with a DiscoverResponse message setting iam field to INSTANCE and id field to its own id (that in a fully distributed system is always 0, so it ignored by all the instances).

In hierarchical mode when NEW sends DiscoverRequest message it sets iam field to NEW. The instances that receive that message vary their behavior accordingly to their role. If they are SLAVE they ignore the message. If they are MASTER they reply with the first free ID and it sends it back to the sender in the DiscoverReply message along with its own role. The original sender will then set its ID with the one received from MASTER. When a SLAVE sends DiscoverRequest message it sets iam field to SLAVE. The instances that receive that message will reply with their role and ID. The original sender will keep the IDs received in order to keep track of which instance in running and their IDs as assigned by MASTER. When MASTER sends DiscoverRequest message it sets iam field to MASTER. The instances that receive that message will reply with their role and IDs. The original sender (i.e. the MASTER), counter-intuitively, will keep the information that receive from the instances that reply with DiscoverReply because it also becomes aware of the instances that are running just when it executes discover routine although it is him that give to instances the IDs. This is since in GRPC it is not possible to knows the sender IP so when MASTER assigns ID to an instance it does not know to which IP address is giving the IP to.

If an INSTANCE receives a message from either NEW, SLAVE or MASTER it returns an error. The same happens when a NEW, SLAVE or MASTER receives a message from an INSTANCE.

Figure 3 summarize the exchange of discover messages for the hierarchical distributed system. Fully distributed system exchange is omitted due to its simplicity.

2.4 Ping messages

Ping routine is simpler than discover routine. It uses GenericRequest and GenericReply messages to calculate mean RTT and availability. In fully mode all instances can (and should) send ping messages and they also reply to ping messages just returning a positive return value. In hierarchical mode only MASTER sends ping messages that are replied with a positive return value just by SLAVES. There is incompatibility between fully and hierarchical mode in ping messages too. Ping reply messages of both fully and hierarchical modes are just a simple message because all the job is done by the request sender that calculate the RTT for each ping message it sends.

In fully distributed system all instances ping other instances. In hierarchical distributed mode only MASTER pings other instances.

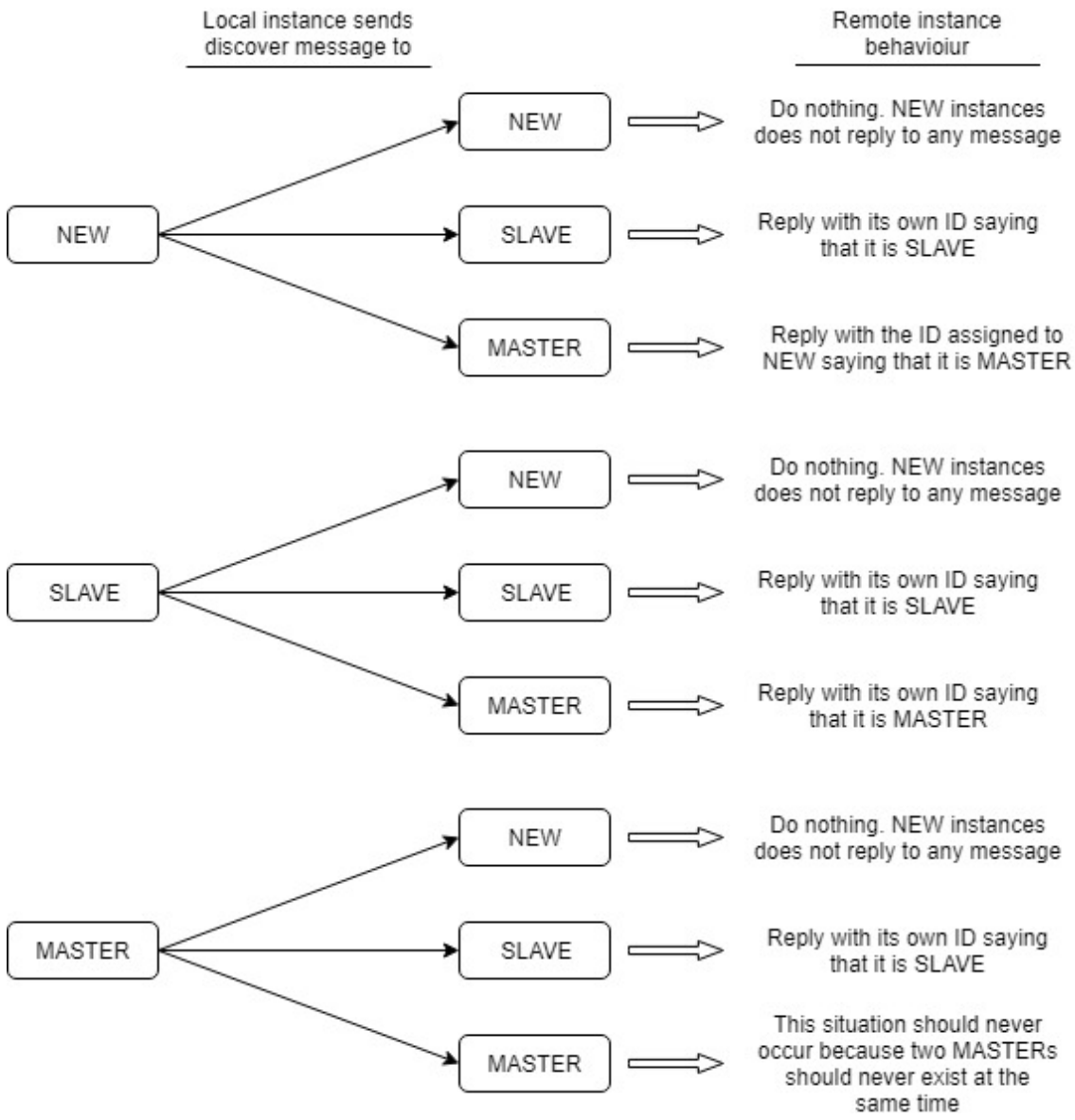


Figure 3: Behaviour of instances according to their role in hierarchical distributed system.

2.5 DistributedManager Task

DistributedManager is the main class of the distribution management system and it is a singleton that can be accessed anywhere. It configures the local instance with the settings that are needed to perform discovery of remote instances. DistributedManager inherits from Worker the Task function that is the heart of it. The general functioning of DistributedManager is an infinite loop, with a certain period, of discover and ping routines that is started in Task activated by the ResourceManager during the setup. Discover and ping routines (called DiscoverInstances and PingInstances) are executed with a certain period that can be different of each other. The periods are defined into configuration file under the DistributedManager options (discover_period_s and ping_period_s) and they define period in seconds. If the two periods are different the manager needs to decide the Task period. This is done calculating the greatest common divisor (GCD) between discover and ping periods. GCD becomes the Task period. In this way, with two counters it can be decided when to call discover and ping routines.

Sequence diagram in **figure 4** shows an overview of the distributed manager functioning.

As you can see on the above sequence diagram the discover and ping functions are called with the

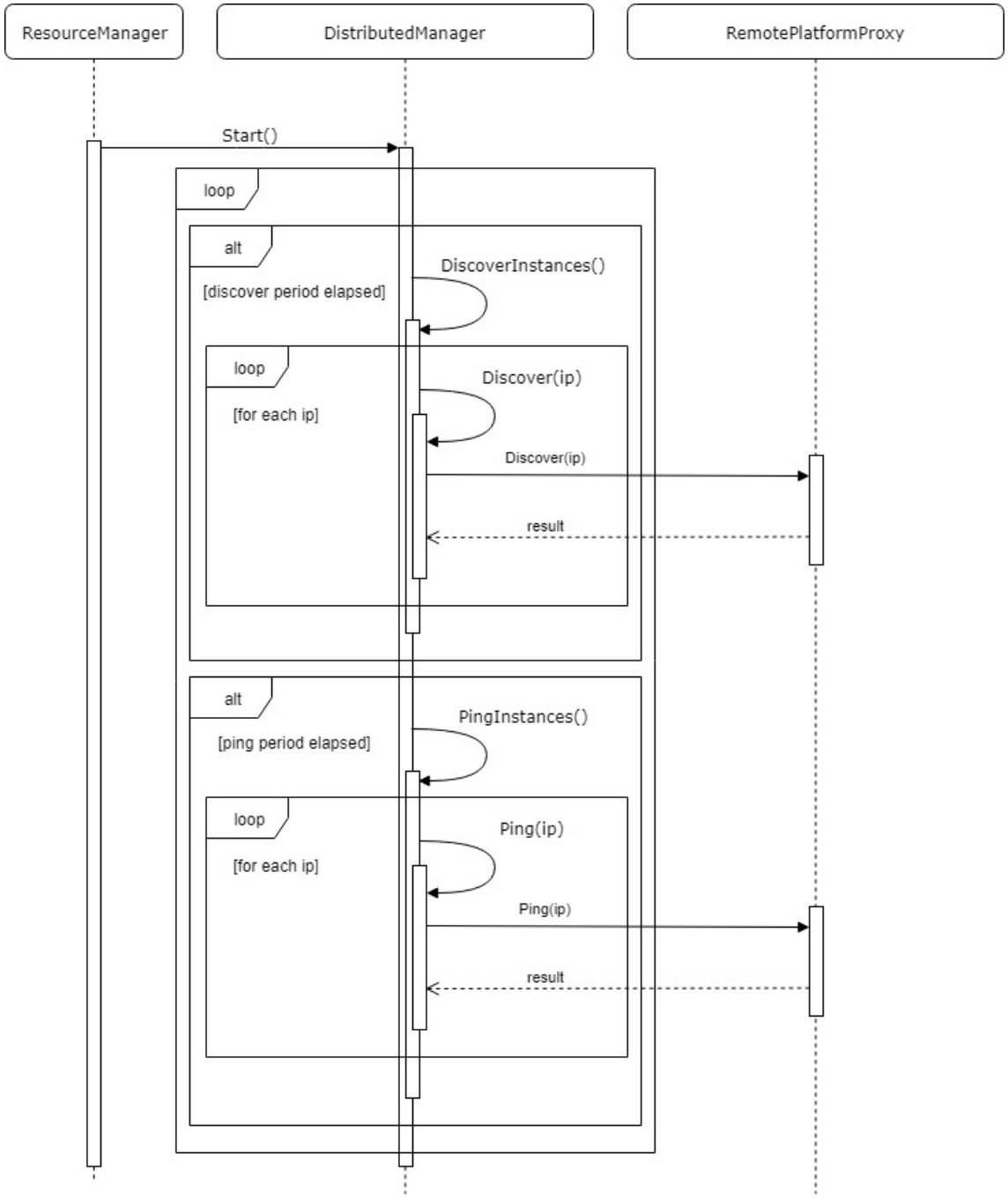


Figure 4: DistributedManager's Task function overview. What happens in RemotePlatformProxy is omitted for simplicity.

parameter ip. The DistributedManager knows the IPs to which send the discover and ping messages because they are defined in the configuration file. In reality, in configuration file is defined a range of IPs with the variables start_address and end_address so that during the configuration the Distributed-Manager builds all IPs that are within them, saves the IPs into a local vector variable called ipAddresses and it can use them in discover and ping routines.

2.6 Instance tags

A tag is an identification of running status. Each instance can have one of three different tags:

- DISCONNECTED: when the instance is not running so it does not reply to discover messages;
- OK: when the instance is running and replies to discover messages;
- SLOW: when the instance is running, replies to discover messages but timeout expires when comes to reply to ping messages.

DistributedManager sends a configurable number of ping messages to each instance. If an instance does not reply to any ping message, then it is tagged as SLOW. Thus, it suffices to reply to one ping message in a single routine to be tagged as OK.

2.7 Discover and DiscoverInstances functions

Discover and DiscoverIntances are two functions of DistributedManager. DiscoverInstances is called from Task function while looping and it is the entry point to discover other instances. Both DiscoverInstances and Discover functions are different based on the configuration although they share some sections.

2.7.1 DiscoverInstances

Common part: it creates a thread (main thread) for each known IP address in the range of addresses as found in configuration file and start them with the Discover function. Thus, each thread starts knowing just the IP address to which send the discover message. Then, join all the threads.

Fully mode: after joined all the threads it returns;

Hierarchical mode: after joined all the threads it checks if the local instance is alone in the system (i.e. no other instance is running). If it is alone then it set itself as MASTER giving itself ID=0 because master has always ID=0. If it is not alone then check if MASTER has been discovered by the threads. If MASTER has not been found, then check which instance has the lower ID and set it to MASTER (i.e. changing its tracking ID).

2.7.2 Discover

Discover function is more complicated and contains many differences between the fully and hierarchical modalities. From now on it is considered a single main thread.

Common part: a thread (message thread) is created to send a discover message with the proper values as explained in previous sections (especially section 2.3). The main thread, instead, wait for it to return some value while setting a timeout.

Fully mode: if the message thread returns a success, then the remote instance is tracked with the ID received in the reply message. Otherwise, if the timeout expires or the message thread returns an error, then the remote instance is untracked, if currently tracked.

Hierarchical mode: since there are different roles, in this modality things are slightly harder. Figure 5 comes in handy to understand how different roles act in different situations. As you can see on the figure, nothing different, with respect to the fully modality, happens when the message thread returns an error or the timeout expires. If the message thread returns a success, then each role act differently. It is worth to notice that the main differences are when a NEW instance discover message is replied

by MASTER, so NEW get assigned an ID, and when MASTER replies to another MASTER, that it is a theoretically impossible thing. In this last case, the local instance exit with an error.

In Figure 5 you can see a chart that summarize Discover function behaviour in different situations.

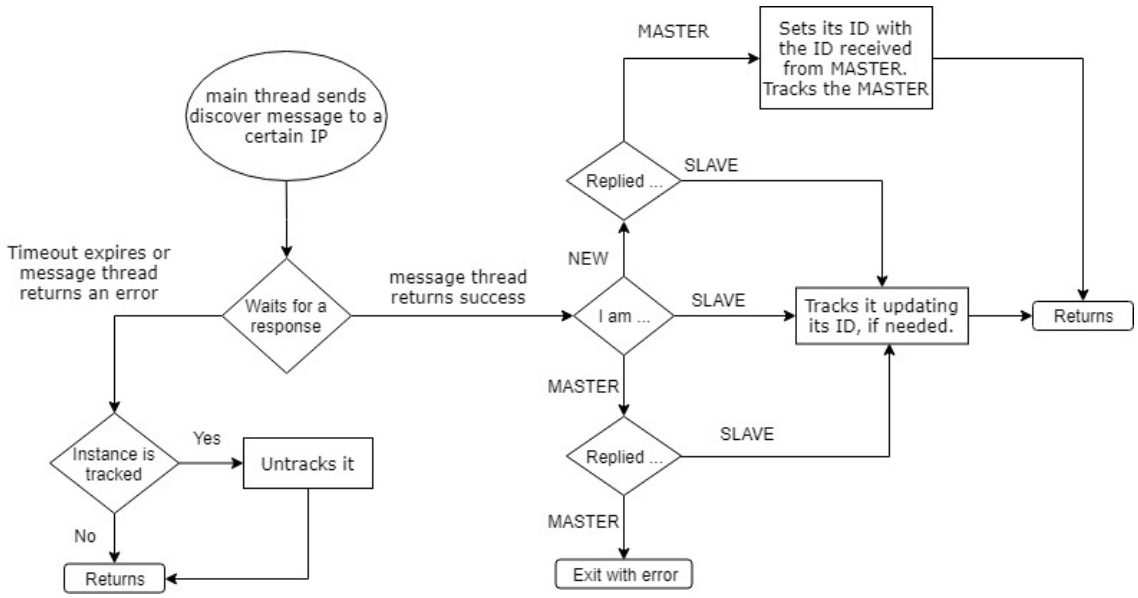


Figure 5: Behaviour of Discover function.

2.8 Ping and PingInstances functions

Ping and PingIntances are two functions of DistributedManager. PingInstances is called from Task function while looping and it is the entry point to ping other instances. These functions share the structure with the previously explained Discover and DiscoverInstances ones.

2.8.1 PingInstances

It creates a thread (main thread) for each tracked instance and start them with the Ping function. Thus, each thread starts knowing just the IP address to which send the ping message. Then, join all the threads and returns.

2.8.2 Ping

Before illustrate Ping function it is fundamental to know how mean RTT and availability are calculated.

DistributedManager class has two data structures called instance_private_stats_map of type Instance_Private_Stats_t and instance_public_stats_map of type Instance_Public_Stats_t.

Instance_Public_Stats_t just contains mean RTT and availability calculated. It is public so other classes can get the instances as calculate by the distributed manager.

Instance_Private_Stats_t contains an array (last_pings) of a settable dimension through two macros (PING_NUMBER and PING_CYCLES) and an integer pointer. PING_CYCLES defines the number of times PingInstances is called from Task function to which take into consideration the ping values in

order to calculate mean RTT and availability. PING_NUMBER defines the number of ping messages to be sent to an instance for each cycle. Example: If PING_NUMBER = 3 and PING_CYCLES = 5 then the instance ping 3 times in a row an instance (this is 1 cycle). For the successive 4 cycles the old pings are kept and used along with the 3 new ones to calculate the RTT. At the 6th cycle the 3 values of the first cycle are overwritten with the 3 new ones. And so on...

The last_pings array serves to store the ping values calculated in each cycle. Ping_pointer is the pointer to the position to which the last ping has been written into last_pings (exactly as a stack).

The mean RTT is simply calculated as the mean value of all the positive values contained in last_pings array. Availability is calculated as the percentage of positive values over values different to 0 into last_pings (i.e. how many ping messages returned a success over the number of sent ping messages, for each instance).

Back to the Ping function...

A certain number of threads are created (let's call them ping threads). That number is settable with a macro called PING_NUMBER. For each ping thread, the main thread set up a timeout. Each ping thread sends a ping message to the remote instance that the main thread is considering (i.e. the instance with IP the one passed to it as parameter).

For each ping thread, if it returns an error or the timeout expires, a ping value of -1 is saved into instance_private_stats_map->ping_values for that remote instance, otherwise the ping value returned by ping thread is saved into it. If no ping thread returned a success in a cycle then that remote instance is tagged as SLOW, inserting it into slowv_instances set. slow_instances set is emptied at every ping cycle. instance_public_stats_map is erased at every ping cycle.

3 Experimental evaluation

The system functioning must be tested making instances to communicate with each other through the discover and ping messages.

3.1 Experimental setup

Since there has been no possibility to test a real distributed system, the system has been tested locally. Many instances of BarbequeRTRM are started, but on the opposite as should happens in a real distributed system, every instance is started making it listen on the same IP address. In order to mimic a distributed system with different instances on different IP addresses, they listen on different ports. This way, a single machine is enough to test the system.

3.2 Results

In this section it is shown step by step what each instance in the system logs with PrintStatusReport function. Each instance is called with the port to which is listening on (e.g. instance that listen on 8850 port will be called 8850).

Steps are explained in figure captions.

3.2.1 Fully distributed system

Step 1:

```
==========================================================================
|          IP           | Sys |   RTT   | AVAILABILITY |    STATUS    |
|-----------------------+-----+---------+--------------+--------------|
|       127.0.0.1:8850  |   0 |    -    |       -      |    MYSELF    |
|       127.0.0.1:8851  |   - |    -    |       -      | DISCONNECTED |
|       127.0.0.1:8852  |   - |    -    |       -      | DISCONNECTED |
|                       |     |         |              |              |
==========================================================================
```

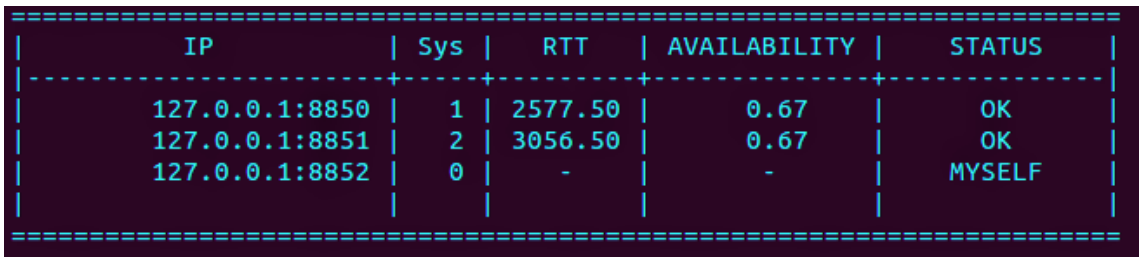
Figure 6: 8850 is the first launched. It finds no other insance. The report status shows only itself.

Step 2:

```
==========================================================================
|          IP           | Sys |   RTT   | AVAILABILITY |    STATUS    |
|-----------------------+-----+---------+--------------+--------------|
|       127.0.0.1:8850  |   1 | 1594.00 |      1.00    |      OK      |
|       127.0.0.1:8851  |   0 |    -    |       -      |    MYSELF    |
|       127.0.0.1:8852  |   - |    -    |       -      | DISCONNECTED |
|                       |     |         |              |              |
==========================================================================
```

Figure 7: 8851 is launched nad it founds 8850 mapping it to ID=1. Note the mean RTT and availability statistics.

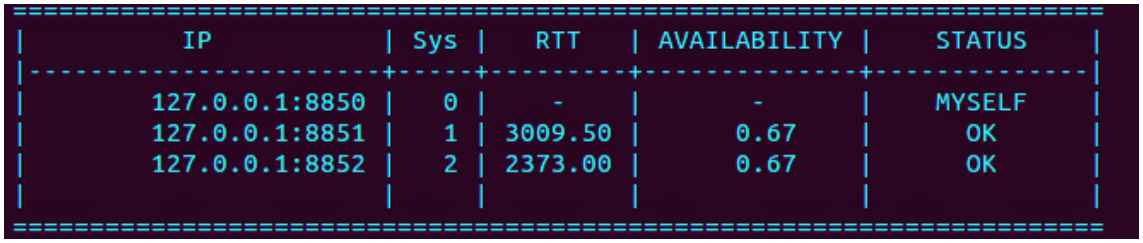
Step 3:



IP	Sys	RTT	AVAILABILITY	STATUS
127.0.0.1:8850	1	2577.50	0.67	OK
127.0.0.1:8851	2	3056.50	0.67	OK
127.0.0.1:8852	0	-	-	MYSELF

Figure 8: Now 8852 is launched and so it discover 8850 and 8851 and map them to ID=1 and ID=2m respectively. So far it is fairly straightforward.

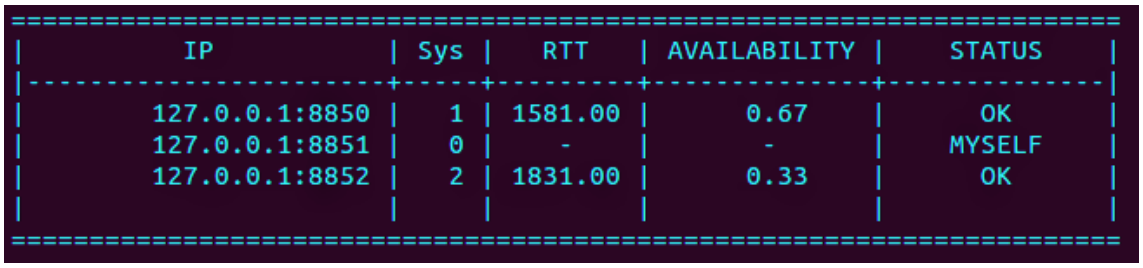
Step 4:



IP	Sys	RTT	AVAILABILITY	STATUS
127.0.0.1:8850	0	-	-	MYSELF
127.0.0.1:8851	1	3009.50	0.67	OK
127.0.0.1:8852	2	2373.00	0.67	OK

Figure 9: 8850 refresh doing a new discover and ping cycle so now it finds 8851 and 8852 mapping them to ID=1 and ID=2, respectively.

Step 5:

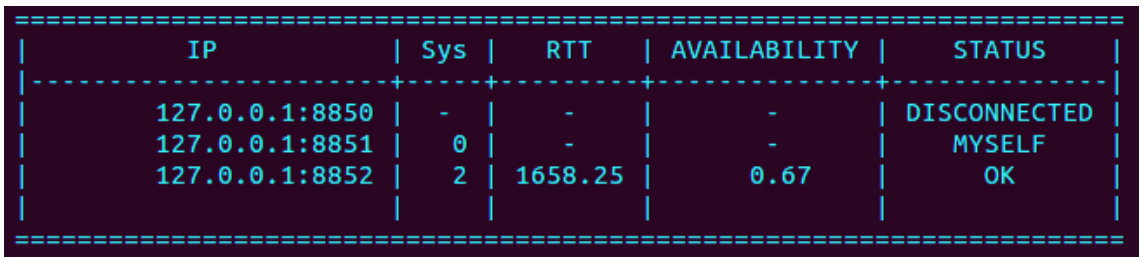


IP	Sys	RTT	AVAILABILITY	STATUS
127.0.0.1:8850	1	1581.00	0.67	OK
127.0.0.1:8851	0	-	-	MYSELF
127.0.0.1:8852	2	1831.00	0.33	OK

Figure 10: 8851 refresh too, so it finds 8852, mapping to the first free ID (i.e. 2), and 8850 (already found in the previous cycle).

Step 6: 8850 is turned off so from now other instances should not discover it anymore.

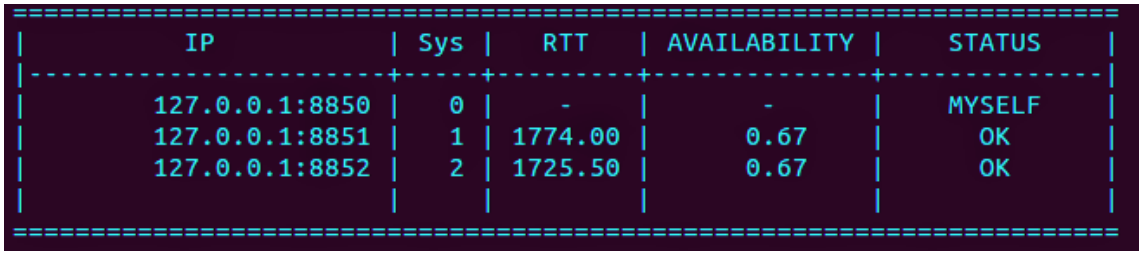
Step 7:



IP	Sys	RTT	AVAILABILITY	STATUS
127.0.0.1:8850	-	-	-	DISCONNECTED
127.0.0.1:8851	0	-	-	MYSELF
127.0.0.1:8852	2	1658.25	0.67	OK

Figure 11: 8851 refresh and it does not found 8850 anymore because it is no longer running. 8852 is found as in the last discover cycle.

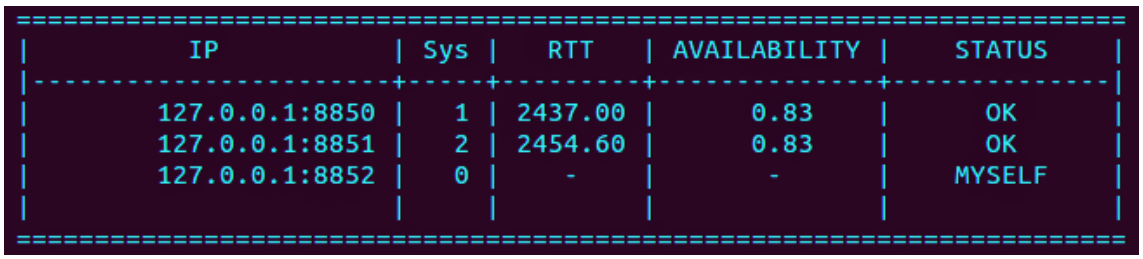
Step 8:



IP	Sys	RTT	AVAILABILITY	STATUS
127.0.0.1:8850	0	-	-	MYSELF
127.0.0.1:8851	1	1774.00	0.67	OK
127.0.0.1:8852	2	1725.50	0.67	OK

Figure 12: 8850 comes back and do a discover cycle finding the other instances. Note that the IDs assigned to instances are the same as before just by chance.

Step 9:

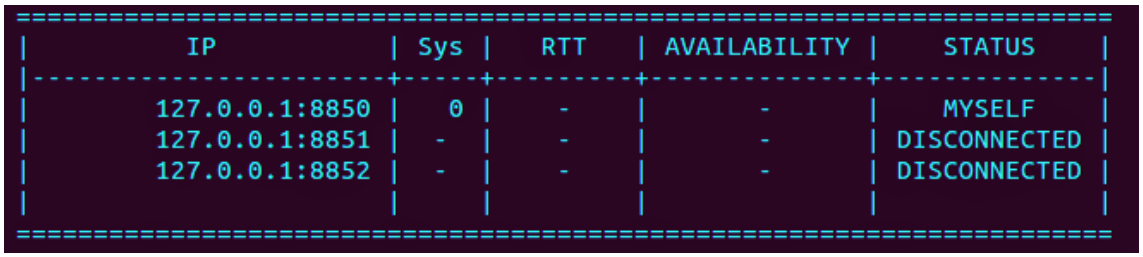


IP	Sys	RTT	AVAILABILITY	STATUS
127.0.0.1:8850	1	2437.00	0.83	OK
127.0.0.1:8851	2	2454.60	0.83	OK
127.0.0.1:8852	0	-	-	MYSELF

Figure 13: In this last step 8852 refresh with a new cycle and it finds 8850 and 8852 as before. Note that 8852 did not realise that 8850 died and reborn. Note also that each instance has a different mapping between IPs and IDs and that they set themselves as the instance with ID=0.

3.2.2 Hierarchical distributed system

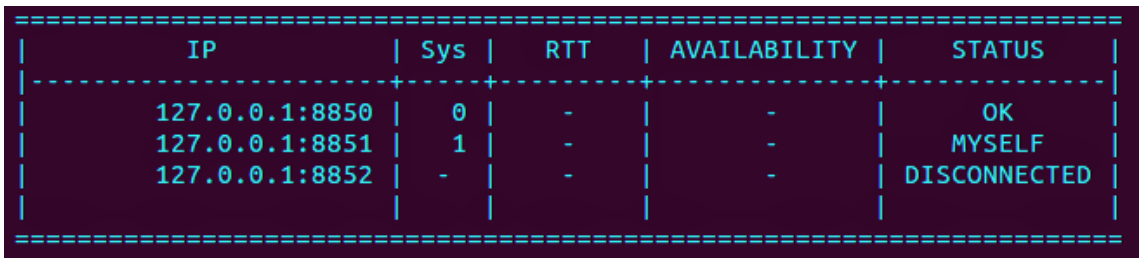
Step 1:



IP	Sys	RTT	AVAILABILITY	STATUS
127.0.0.1:8850	0	-	-	MYSELF
127.0.0.1:8851	-	-	-	DISCONNECTED
127.0.0.1:8852	-	-	-	DISCONNECTED

Figure 14: 8850 is the first launched so it finds no other insance. Thus, it set itself as MASTER (ID=0 means MASTER).

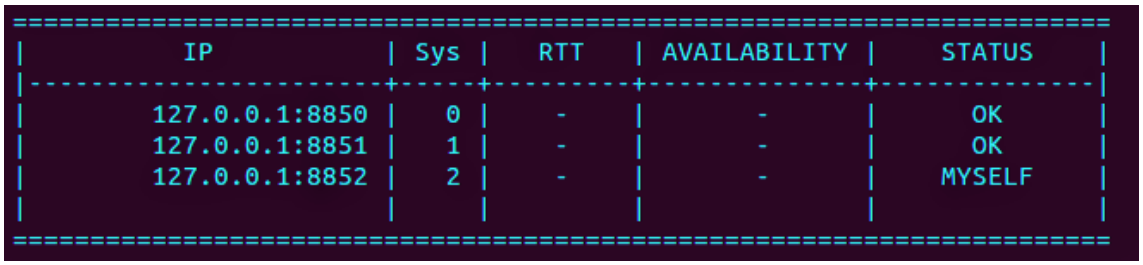
Step 2:



IP	Sys	RTT	AVAILABILITY	STATUS
127.0.0.1:8850	0	-	-	OK
127.0.0.1:8851	1	-	-	MYSELF
127.0.0.1:8852	-	-	-	DISCONNECTED

Figure 15: 8851 is launched and it finds 8850 that is MASTER which gives him the ID=1.

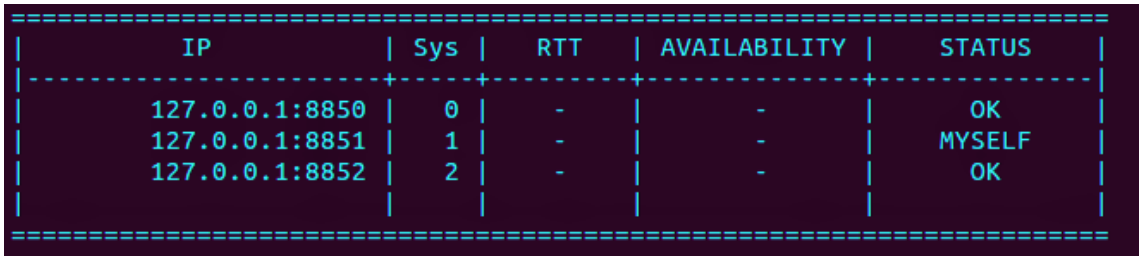
Step 3:



IP	Sys	RTT	AVAILABILITY	STATUS
127.0.0.1:8850	0	-	-	OK
127.0.0.1:8851	1	-	-	OK
127.0.0.1:8852	2	-	-	MYSELF

Figure 16: 8852 is launched and finds 8851 and 8850 that as MASTER gives him ID=2. Note that both 8852 and 8851 did not ping other instances.

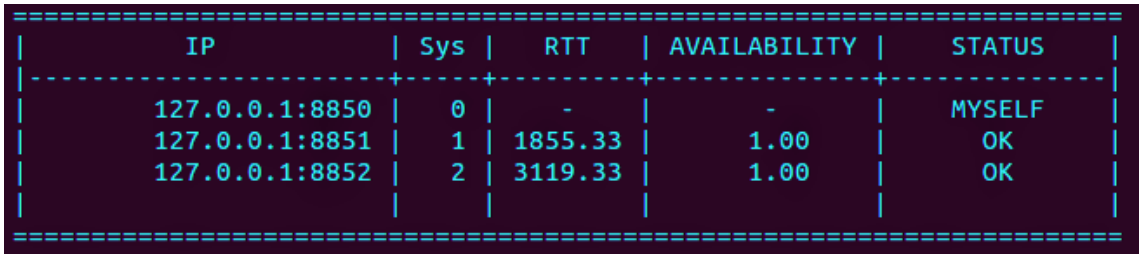
Step 4:



IP	Sys	RTT	AVAILABILITY	STATUS
127.0.0.1:8850	0	-	-	OK
127.0.0.1:8851	1	-	-	MYSELF
127.0.0.1:8852	2	-	-	OK

Figure 17: 8851 refresh so it founds 8852 too.

Step 5:

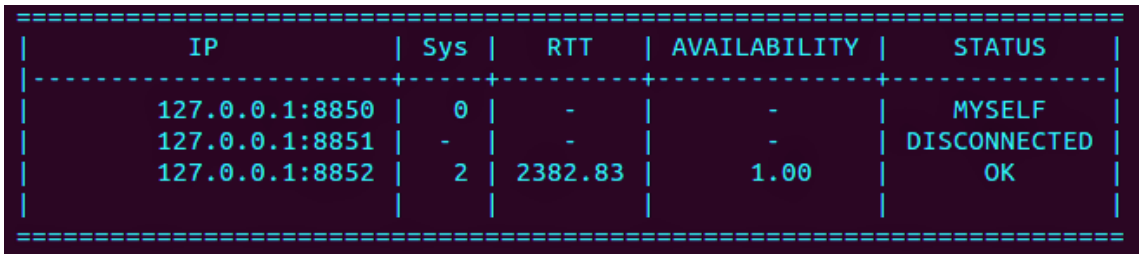


IP	Sys	RTT	AVAILABILITY	STATUS
127.0.0.1:8850	0	-	-	MYSELF
127.0.0.1:8851	1	1855.33	1.00	OK
127.0.0.1:8852	2	3119.33	1.00	OK

Figure 18: MASTER finally do a discover and ping cycle so it founds 8851 and 8852. Note that MASTER is the only instance that sends ping messages.

Step 6: Now 8851 is turned off so it should not be found anymore by other instances.

Step 7:

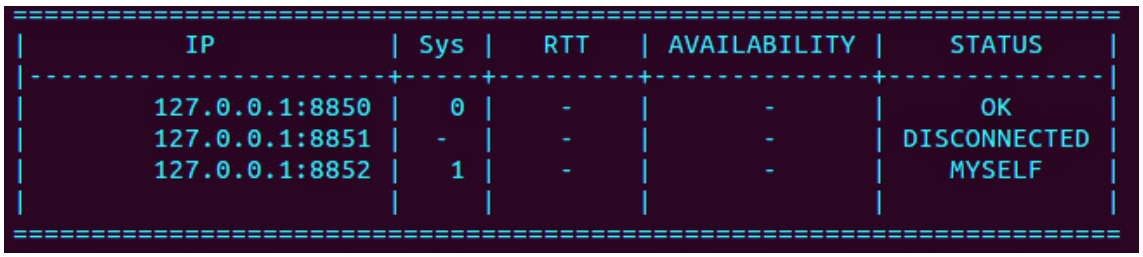


IP	Sys	RTT	AVAILABILITY	STATUS
127.0.0.1:8850	0	-	-	MYSELF
127.0.0.1:8851	-	-	-	DISCONNECTED
127.0.0.1:8852	2	2382.83	1.00	OK

Figure 19: MASTER refresh and it does not find 8851 anymore.

Step 8: 8852 is turned off, too.

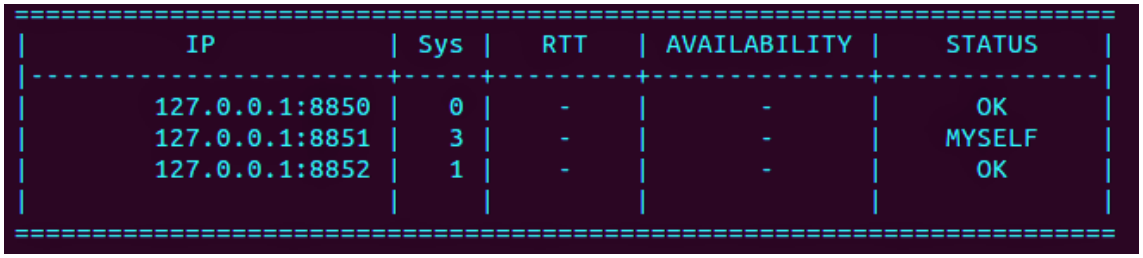
Step 9:



IP	Sys	RTT	AVAILABILITY	STATUS
127.0.0.1:8850	0	-	-	OK
127.0.0.1:8851	-	-	-	DISCONNECTED
127.0.0.1:8852	1	-	-	MYSELF

Figure 20: 8852 is launched again. MASTER gives him ID=1 that is different than the one that it had before it died because MASTER freed up ID=1 when 8851 died.

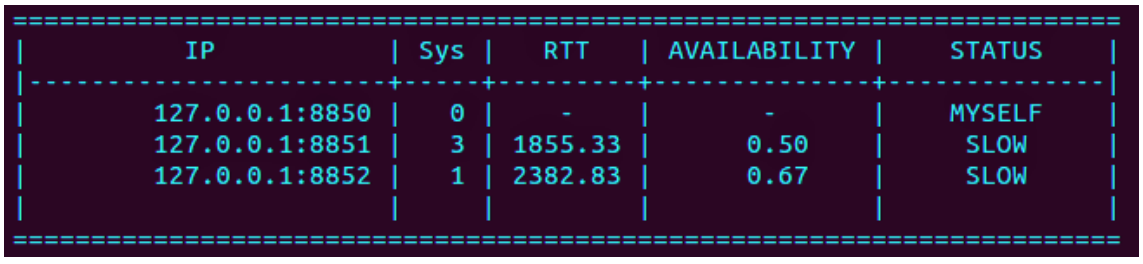
Step 10:



IP	Sys	RTT	AVAILABILITY	STATUS
127.0.0.1:8850	0	-	-	OK
127.0.0.1:8851	3	-	-	MYSELF
127.0.0.1:8852	1	-	-	OK

Figure 21: 8851 is launched again. MASTER gives him ID=3, not ID=2, because MASTER does not know that 8852 with ID=2 has died (it realise it only when it do a discover cycle) and 3 is the smaller free ID greater than 0.

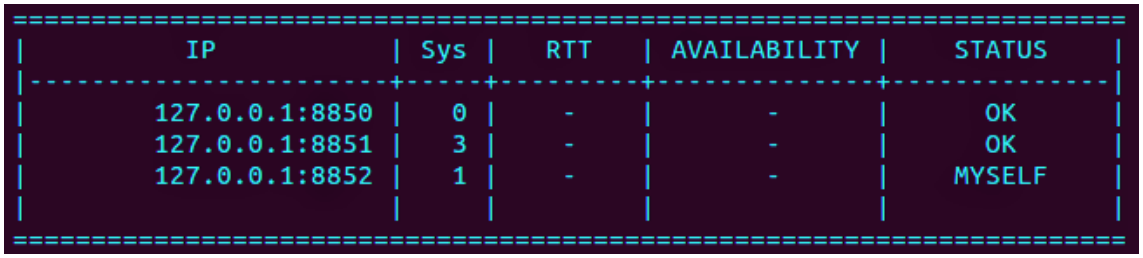
Step 11:



IP	Sys	RTT	AVAILABILITY	STATUS
127.0.0.1:8850	0	-	-	MYSELF
127.0.0.1:8851	3	1855.33	0.50	SLOW
127.0.0.1:8852	1	2382.83	0.67	SLOW

Figure 22: 8850 refresh so it realise that the instance with ID=2 is not living anymore so it frees up ID=2. At the same time it founds 8851 and 8852 with the new IDs. Note that in this case both 8851 and 8852 are tagged SLOW because they did not reply to any ping message. RTT and availability are calculated nevertheless.

Step 12:

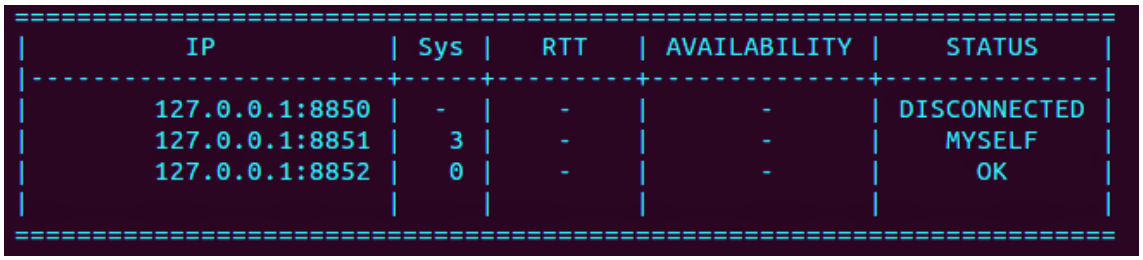


IP	Sys	RTT	AVAILABILITY	STATUS
127.0.0.1:8850	0	-	-	OK
127.0.0.1:8851	3	-	-	OK
127.0.0.1:8852	1	-	-	MYSELF

Figure 23: 8852 refresh so now all the instances has the same IP->ID mapping.

Step 13: Now MASTER dies so a new MASTER needs to be instaured.

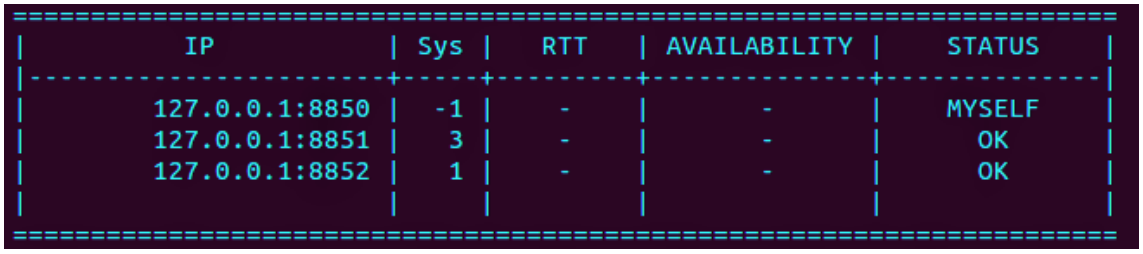
Step 14:



IP	Sys	RTT	AVAILABILITY	STATUS
127.0.0.1:8850	-	-	-	DISCONNECTED
127.0.0.1:8851	3	-	-	MYSELF
127.0.0.1:8852	0	-	-	OK

Figure 24: 8851 refresh so it realise that MASTER is died. It automatically set the instance with lower ID as MASTER. 8852 has ID=1 while 8851 has ID=3 so 8852 becomes MASTER. Note that 8852 does not know it yet.

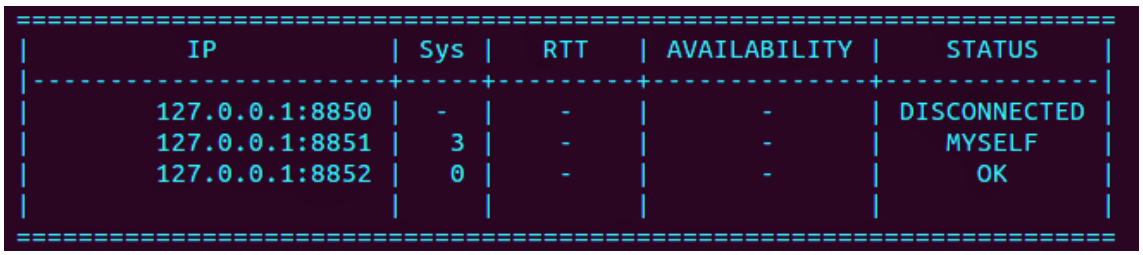
Step 15:



IP	Sys	RTT	AVAILABILITY	STATUS
127.0.0.1:8850	-1	-	-	MYSELF
127.0.0.1:8851	3	-	-	OK
127.0.0.1:8852	1	-	-	OK

Figure 25: 8850 is launched again and as NEW it does not find MASTER bacause 8852 did not realise it yet. So 8850 wait until the next discover cycle. Not that 8850 has ID=-1 because it is the ID of NEW roles.

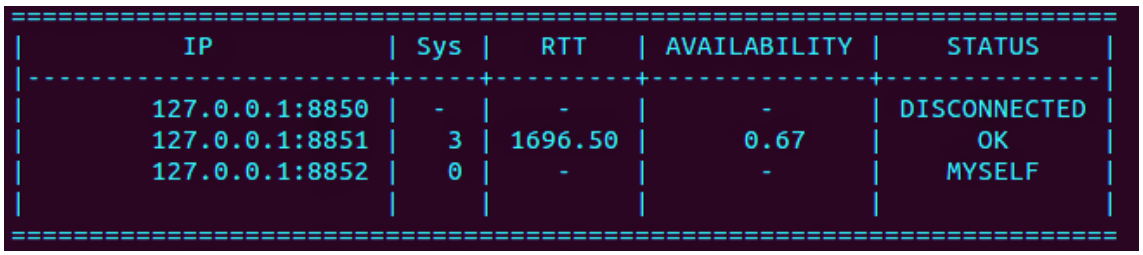
Step 16:



IP	Sys	RTT	AVAILABILITY	STATUS
127.0.0.1:8850	-	-	-	DISCONNECTED
127.0.0.1:8851	3	-	-	MYSELF
127.0.0.1:8852	0	-	-	OK

Figure 26: 8852 refresh and founds only 8852 because NEW instances do not reply to discover messages so it can not be found.

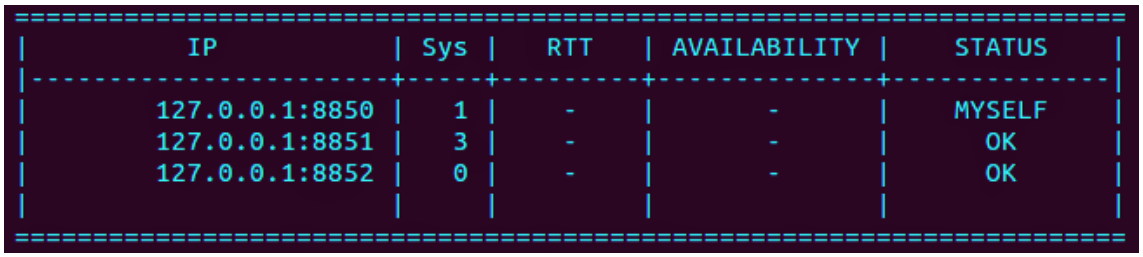
Step 17:



IP	Sys	RTT	AVAILABILITY	STATUS
127.0.0.1:8850	-	-	-	DISCONNECTED
127.0.0.1:8851	3	1696.50	0.67	OK
127.0.0.1:8852	0	-	-	MYSELF

Figure 27: Finally 8852 refresh so it realise that it is MASTER. Now it starts to ping instances, too.

Step 18:



IP	Sys	RTT	AVAILABILITY	STATUS
127.0.0.1:8850	1	-	-	MYSELF
127.0.0.1:8851	3	-	-	OK
127.0.0.1:8852	0	-	-	OK

Figure 28: 8850 finally found MASTER so it get the ID=1. Now all instance has the same IP->ID mapping.

4 Conclusions and Future Works

BarbequeRTRM is now capable to buils a network of remote instances in two different modalities. Both modalities support topology changes at run-time.

The next step is to implement functions that allow instances to exchange their own system architecture in order to give remote instances the possibility to know the distributed architecture and resources as they were a big complex architecture with a lot of different resources.