Groupy: a group membership service

Anestos Nikolaos - Ektoras

October 8, 2014

1 Introduction

In this seminar we will implement a group membership service that provides atomic multicast.

The system consists of several nodes where one of them is the leader. Each node has an application layer, an interface layer and a multicast layer. The application layer either receives messages from other nodes and changes his state accordingly, or sends a message (through the multicast layer) to every other node in the system telling them to change their state. The interface layer (gui) receives the state change and represents it visually in a colored box. The multicast layer consist of a leader and slaves. Any node that wants to broadcast a state change will send a message to the leader and the leader will do a basic multicast to all the members in the group.

2 Main problems and solutions

This exercise was in a totally different philosophy from the others in this course. Most of the code (if not all) was ready for us, we had to understand how it works and write appropriate tests in order to observe its behavior.

After a detailed examination of the code, I understood how to start the "leader" node and the "slave" nodes linked to it. I wrote a small test to get them running easily without having to write a lot of commands in the shell.

The main problem was that even though the window was visible, I didn't see any state change on it. I injected some printouts in the gui module and found out that the new colors were available to the window but the window didn't show them. I searched for available commands in the library and I used wxWindow:refresh(Window) after the window's color change which fixed this problem.

3 Evaluation

The solution has three versions. Each newer version fixes some of the earlier version's issues.

3.1 Start with the basics

In gms1 module, everything is done as simple as possible. A node (the one that is started first) is the leader. The leader keeps track of all the slaves in the system and their linked application layer. The slaves send a message to the leader and he broadcasts it to the whole group. Since the leader receives and handles those messages in FIFO order, each node receives the messages in the same order as well (total order).

Closing a window (shutting down) of a random node doesn't affect the system at all. All the other nodes continue to send messages to the leader and he broadcasts them. Though, since we don't use acknowledgment in the sending of the messages, no one is informed about the node crash.

On the other hand, if the leader crashes, the system stops working because no one is there to broadcast the message for the nodes. We will address this issue in the next implementation.

3.2 Leader is dead, I repeat leader is dead...

In gms2 module, we try to work around the possibility a leader dies. The slaves monitor the leader process and will be informed if the leader dies. The slaves then enter an election state where the "older" slave (the first in the Slaves list) is elected as leader. At first sight our system seems to work just fine. If we close the leader's window, a new leader is elected and the system continues to run.

What happens though if the leader dies before he has completed a broadcast? We use the provided crash function which can simulate a random crash. After using this function and letting leaders crash, we see that there are cases where the system isn't affected, and cases where some nodes are out of sync. The latter happens when the leader crashes before sending the broadcasted message to every node in the system. This means that the basic multicaster is not the best for this system and we will replace it with a reliable multicaster.

3.3 Trust me, I 'm reliable

In order to make sure that all nodes are in sync even if a leader crashes in the meantime of a broadcast, the new (elected) leader, will resend the last message to everyone. This will cause more problems, if the nodes that had already seen it don't ignore it. To avoid that, we need to number the messages sent by the leader so that every node knows what to expect.

While testing this module, we don't come up to any weird synchronization problems. Even if the leader crashes while broadcasting, the new leader is there to make sure tha everyone in the system is in sync.

The slave tests the incoming message's number, if the number is the same as the slave expects, he forwards it to the application layer and we see

a color change. If the number of the new message is lower than the one the slave expects, he ignores the message since he already has seen it.

3.4 Can we do any better?

What will happen though if a message is lost? It might be the case that a message sent by the leader, might reach **some of the slaves** in the system. To avoid that we could:

- Have the leader expect for an acknowledgment (for how much time?) from all the nodes and resend the message if a node doesn't reply. This will complicate the leader quite a bit, because we need to make him wait for a reply from every node, and maybe even make him count how many times he tried to send a message to a node, because that node might have crashed and he will be in a deadlock.
- We could also have the slave asking the leader to resend the lost message and process them in the correct order. This means that the leader would have to keep a history of the messages he sent (how many messages?).
- Another idea might be that when a node sees a message with a higher number than the one he expects, to ask the leader to exclude him from the group and include him again. This will make him start over with a state equal to the other nodes.

Any solution we come up, depends on what we want to achieve with our application. Hopefully, in our system, we started with the assumption that messages are reliably delivered and we do nothing about that issue.

4 Conclusions

This exercise was very interesting. Even though most of the code was given, it was quite challenging to understand how it works and then go one step further, and think of what can be done better. This lab made me understand in depth what the problems of an atomic multicast might be and how to solve them.