SINF2345 Languages and Algorithms for Distributed Applications
Project report

Martin Trigaux
SINF22MS
martin.trigaux@student.uclouvain.be

Bernard Paulus
SINF22MS
bernard.paulus@student.uclouvain.be

May 7, 2012

Contents

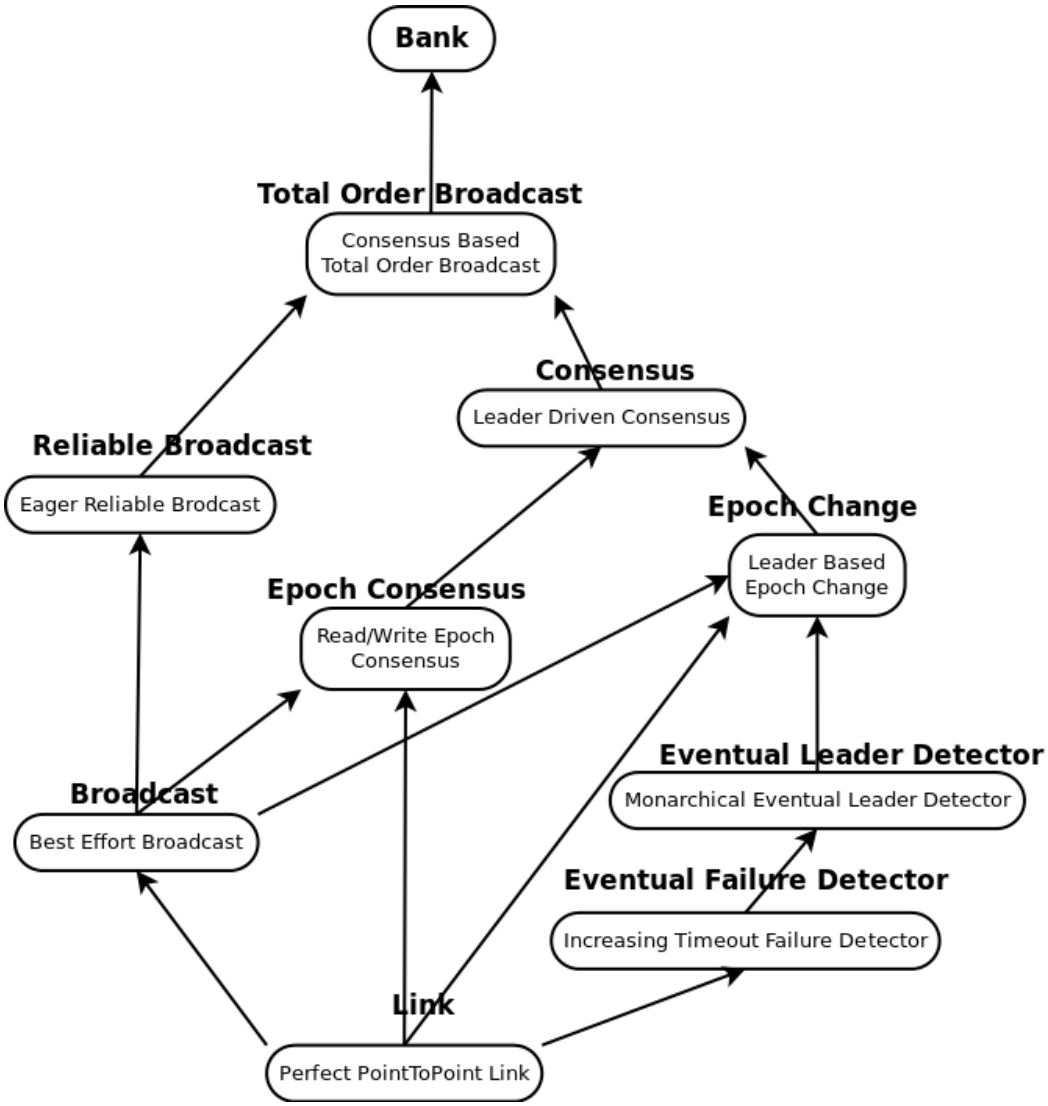
Introduction

The aim of the project is to build a distributed banking system. The system is distributed in a way such that any bank node is award of the full banking system state at all time but only atomics messages are transfered between nodes.

To build this project, we decided to use the programming language Erlang. We believe this language is highly adequate for concurrent applications and that is what motivated us to use it for this project. This language handles also very well events and made easier the implementation of proposed algorithms in the reference book.

1 Architecture

To realise the distributed system, we used the reference book "Reliable and Secure Distributed Programming" by C. Cachin, R. Guerraoui and L. Rodrigues second edition. As the modules and algorithms are tested and well defined, we tried to stick as much as possible to the book and implemented several algorithms from it. We used a architecture based on layers and encapsulation of messages. Each module is connected to other and behave upon events (received messages from other modules).



Basic abstractions

We used the **perfect link** module[1] for basic point to point communications. These are at the lowest level of our architecture and simply transmit messages to above modules. The perfect link ensures *reliability* and the *no duplication* property.

To handle failing nodes in a partially synchronous system, we used an **eventual failure detector** module[2]. This was implemented using the **increasing timeout failure detector** algorithm[3].

Once we have successfully detected faulty processes, we can focus on correct ones

[1]Module 2.3 p37 in reference book
[2]Module 2.8 p54 in reference book
[3]Algorithm 2.7 p55 in reference book

and elect a leader. The **eventual leader detector** module[4] allows to elect a reliable as a leader to perform certain computations on behalf of the others. We implemented the **monarchical eventual leader detector** algorithm[5]. This algorithm elect the leader with the highest rank among the alive nodes with the possibility to restore suspected nodes (eg: too slow to reply and were wrongly suspected as failing).

Broadcasts

We used the **best effort broadcast** module[6] to implement the broadcast messages between nodes. This module transmit messages to the adequate perfect links. As we want to handle failing nodes, we used a **reliable broadcast** algorithm[7]. To ensure the *agreement* property, we implemented the **eager reliable broadcast** algorithm[8].

Consensus

As we are working in a concurrent system, we need to reach to a **consensus** and decide the next state of the system. We used the **epoch-change** abstraction[9] to take a decision on a proposed value. This was implemented using the **leader based epoch change** algorithm[10]. We encounters majors problem using this algorithm due to an error in the reference book. This was discussed in a further errata of the book[11] we discovered only recently. However the authors made the non-explicit assumption that the local stack of messages is FIFO. That assumption was at first not verified and we realised only later it is crucial for the success of the algorithm. The implemented algorithm has been fixed using both the errata and the FIFO queuing process.

In the attempt to obtain a consensus, we used the **epoch consensus** module[12]. We implemented it using the **read write epoch consensus** algorithm[13]. The algorithm uses timestamps in a state value so as to serves the *validity* and *lock-in* or the epoch consensus. The method used relies on a majority if correct processes and assumes we have at least more than the half on non-failing nodes. In the **leader driven consensus** algorithm[14], we distinguish the instances of epoch consensus by their timestamp. Once we receive a new epoch event and need to switch the epoch, the algorithm aborts the running epoch consensus and initialize the next epoch consensus using the state of the previous one.

[4] Module 2.9 p56 in reference book
[5] Algorithm 2.8 p58 in reference book
[6] Module 3.1 p75 in reference book
[7] Module 3.2 p77 in reference book
[8] Algorithm 3.3 p80 in reference book
[9] Module 5.3 p218 in reference book
[10] Algorithm 5.5 p219 in reference book
[11] See http://distributedprogramming.net/docs/Errata.pdf
[12] Module 5.4 p221 in reference book
[13] Algorithm 5.6 p223 in reference book
[14] Algorithm 5.7 p225 in reference book

Total order broadcast

For our banking application, the delivery order of messages is crucial (in case of concurrent transactions). In the previous broadcast abstraction, we used FIFO-order broadcast but we had no assumption on the delivery order. The **total order broadcast** module[15] aims to fix this problem and order all messages, even those not from different senders. This module was implemented using the **consensus based total order broadcast** algorithm[16]. The system is based on consensus abstraction to be able, at any time, to decide on a set of unordered messages between two processes.

Banking system

Once we have implemented all our modules, we can build our banking module on top of a reliable system. Each bank node can handle users commands (account creation and money transfer). Based on the rules specified (negative account fee), the bank node broadcast the adequate atomic messages to the banking network. The transactions are validated only once the initial node receives and acknowledgment of the other nodes.

2 Manual

To launch the program you will need the `erlang` package installed on your computer. This as only been tested on Linux environments. Open a terminal and go to the folder containing the code. Compile the code using the command `make`.

To launch the code, use the following command :

```
$ erl -s test main
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:2:2] [async-threa\
ds:0] [hipe] [kernel-poll:false]

Eshell V5.9.1  (abort with ^G)
1>

Banking application with 3 nodes

Creating accounts 1 and 2
Transfering money from #1 to #2
2>
```

At this state, three banks nodes are running. The accounts 1 and 2 have been created with respectively 10 and 20 units of money (let's say € for the sake of simplicity). 5€ have then been transfered from the account 1 to 2.

[15] Module 6.1 p283 in reference book
[16] Algorithm 6.1 p285 in reference book

For deeper testing, the user can interact with the bank nodes using the erlang shell interface. Launch the shell prompt using simply the `erl` command and load the module using `test:main().` or `test:main(INT).` for a custom number of nodes. The result of the function needs is a list that needs to be stored in a list.

```
1> [A, B, C] = test:main(3).
```

To create a new account use the command `bank:create_account(#BANK, #ID, #MONEY).`. For example `bank:create_account(B, 3, 5).` to create a third account with 5€ on its account using the second bank (don't forget the ending dot).

To transfer money from an account to another, use the command `bank:transfer_money(#BANK, #IDFROM, #IDTO, #MONEY).`. For example `bank:transfer_money(C, 2, 3, 5).` to transfer 5€ from the second account to the third one using the third bank node.

To display the amount on the account of the user, use the command `bank:check_account(#BANK, #ID).`. For example `bank:check_account(A, 1).` to check the amount on the account of the first user using the first bank.

```
$ erl
...
1> [A, B, C, D] = test:main(4). %% 4 bank nodes, 2 accounts, 1 transfer
2> bank:create_account(C,3,10). %% create 3rd account
3> bank:transfer_money(D,3,1,5). %% transfer 5€ 3 -> 1
4> bank:check_account(B,3).
Account 3 has 5€ leftok
```

3 Conclusion

The choice of Erlang as the chosen programming language allowed us to build a fully concurrent system. On the other hand, we couldn't use the proposed framework and had to reimplement several algorithms in the book.

This project was quite interesting as it made us work with several different abstractions and different layers of our application. After this project, we believe we obtained a robust stack that could be reused to different applications.

Unfortunatly, we were still unable to get the last layer (total order broadcast) to work correctly and our banking application does not work correctly accordingly. We believe we could demonstrate you the working parts of our application during the project defense.