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 6, 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 motivated us to use it for this project. This language handles also very well events and made easier the implementation of proposed algorithm 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 (recieved 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 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 alived 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] for 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 *agreeement* 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]. However the book makes the assumption that the local stack of messages is fifo which may not be true at all time.

In the attempt to obtain a consensus, we used the **epoch consensus** module[11]. We implemented it using the **read write epoch consensus** algorithm[12]. The algorithm

[1] Module 2.3 p37 in reference book
[2] Module 2.8 p54 in reference book
[3] Algorithm 2.7 p55 in reference book
[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] Module 5.4 p221 in reference book
[12] Algorithm 5.6 p223 in reference book

uses timestamp 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[13], 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 consensu using the state of the previous one.

Total order broadcast

For our banking application, ne need to rely on a delivery order 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[14] 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[15]. 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 recieves and acknowledgment of the other nodes.

2 Manual

3 Conclusion

[13] Algorithm 5.7 p225 in reference book
[14] Module 6.1 p283 in reference book
[15] Algorithm 6.1 p285 in reference book