Evaluation and validation of a distributed commit protocol

MASTER’s THESIS

Advisor
Professor Gustavo Alonso

Supervisor
Christian Buehlmann
(Supercomputing Systems AG)

Autumn Semester 2009

Candidate
Mirwais H. Tayebi
Acknowledgement

First of all, I would like to thank my supervisor Christian Buehlmann for offering me this challenge during my Master Thesis. I am very grateful for his prompt responses and solutions when difficulty arose and especially for his inputs about mycel system and the vision he has behind it.

I would also like to thank my mentor and advisor Prof. Gustavo Alonso, for his support during the whole Master Program at the ETH Zurich and his valuable guidance and advices during the thesis.

I am also thankful to my friend Ivan Krivulev, who took time to review the thesis and gave me some constructive feedbacks.

Last but not the least, I would like to thank the SCS company and all its members for their support and creating such a good ambiance at work.

Eventually, this master thesis would not have been completed without the invaluable support of my family.
Abstract

2PC has become standard for the commit protocols. This is however not the only one proposed in the literature. There is a wide range of these protocols available with different assumptions on the system model.

The purpose of this thesis is to address the transaction commit problem in mycel, which is a prototype for a transactional object persistence framework in a distributed environment. More specifically, a protocol should be proposed and integrated to it.

We do so by studying and investigating various commit protocols proposed widely in the literature. This includes the basic 2PC, its many variants, the non-blocking ones and those tolerating Byzantine failures. Where these protocols are extensively studied, proved and tested in the literature, we try to present them in a simple, intuitive and uniform manner. Then, based on the accumulated knowledge and the requirements of mycel, we selected a suitable decentralized commit protocol and integrated it to the prototype.
## Contents

### I General

1 Introduction ............................... 3
   1.1 Terminology ............................ 3
   1.2 Background ............................ 4
      1.2.1 Transactions ........................ 4
      1.2.2 Failures in distributed transactions 5

2 Atomic Commit Protocols ................. 9
   2.1 Background ............................ 9
   2.2 Basic 2PC ............................ 12
      2.2.1 Fault resilience .................... 12
      2.2.2 Performance ....................... 13
   2.3 Presumed-Abort ....................... 14
      2.3.1 Performance ....................... 14
   2.4 Presumed-Commit ..................... 15
      2.4.1 Performance ....................... 15
   2.5 Early-Prepare ......................... 16
      2.5.1 Performance ....................... 17
   2.6 3PC ................................ 17
      2.6.1 Performance ....................... 18
   2.7 1PC and its variants .................. 19
   2.8 Further optimizations ............... 19
   2.9 Consensus algorithms ................. 20
      2.9.1 Background ....................... 20
      2.9.2 Consensus and ACP ............... 22

### II mycel framework ..................... 25

3 mycel ................................... 27
   3.1 Background ............................ 27
CONTENTS

3.2 Vision ............................................ 27
3.3 Computational model and architecture .................. 29
3.4 Requirements and specification ......................... 30
3.5 Atomic commit in mycel .............................. 32

4 Proposed commit protocol .............................. 35
  4.1 Decentralized atomic commit ......................... 36

5 Tests and evaluation .................................. 39
  5.1 Test scenarios .................................... 41
    5.1.1 Scenario 1 ................................... 41
    5.1.2 Scenario 2 ................................... 41
    5.1.3 Scenario 3 ................................... 42
    5.1.4 Scenario 4 ................................... 43
    5.1.5 Scenario 5 ................................... 44
    5.1.6 Scenario 6 ................................... 44
    5.1.7 Scenario 7 ................................... 45
    5.1.8 Scenario 8 ................................... 46
  5.2 Evaluation ......................................... 46

6 Conclusion ........................................... 49

A dot-generated test scenarios ......................... 51

Bibliography ........................................... 59
Part I

General
mycel is a prototype for an object persistence framework in a distributed environment. It focuses on domain objects and transactions on them. The purpose of this thesis is to address the transaction commit problem in mycel.

This thesis is divided into two parts: The first part is general information and the review of the literature, where the second part is more about mycel and the proposed protocol.

In this chapter we give the background on transactions and failures that can occur during them. Then in the next chapter we discuss the different available protocols. This includes the basic 2PC, its many variants, the non-blocking ones and those tolerating Byzantine failures. Where these protocols are extensively studied, proved and tested in the literature, we try to present them in a simple, intuitive and uniform manner.

Once we have the necessary tools and knowledge, we introduce mycel. Based on the accumulated knowledge and the requirement of mycel, we then give a decentralized commit protocol and integrate it to the prototype. We consider the asymptotic complexity of the protocol for the evaluation purpose and do not use simulation and quantitative measurements.

1.1 Terminology

Here we introduce some of the often used terms in the rest of this thesis. This definition are given in the context of transactions.
Node: A distributed system can be seen as a graph, where a participating process or processor is called a node. A node in a real world is a computer attached to the network.

Site: (see Node)

Participant: Is a node that participates in a transaction, in distributed database systems corresponds to the resource manager.

Coordinator: Is a node that coordinates commit protocol among the participating nodes, in distributed database systems corresponds to the transaction manager.

Log: A file in stable storage (persistent medium like harddisk or flash).

Forced write: Also called synchronous or blocking I/O, is a form of I/O where the process waits until the write is completed.

1.2 Background

1.2.1 Transactions

A transaction is a unit of work. In Data Base Management Systems, it comprises a set of read and write operations.

Transactions in DBMS have to fulfill the following four requirements:

Atomicity: It has the effect of all-or-nothing, i.e. either all the operations are carried out thoroughly or neither of them.

Consistency: A systems is said to be in consistent state if it complies with the rules defined for it. Transactions therefore should leave the system in a consistent state, after their completion.

Isolation: If two or more transactions are performed concurrently, they should not interfere one with another. This is equivalent to the serializability of transactions.

Durability: The outcome of a transaction should be durable, i.e. even after a crash it should be possible to recover the effects of transaction.
1.2. BACKGROUND

Transactions in distributed systems are more complicated than in stand-alone systems. This is due to the fact that in distributed systems (Figure 1.1) more than one site is connected by an underlying communication network. In this case many sites can participate in a single transaction and if a transaction is successful, then its effects should be reflected in all of the participating sites. To guarantee this, that is atomicity of a transaction among all the sites, Atomic Commit Protocols (ACP) [BH87] are proposed.

1.2.2 Failures in distributed transactions

In any computer system, the chance that errors occur, exists. The task of system designers and developers is to reduce this chance and prevent the failures by various means. In distributed systems, the probability of failures is higher, as every component of the system can be the source of a failure. In this section we discuss the failure types that can occur during distributed transactions.

Figure 1.1: Components in a distributed system: different sites (nodes) connected by a network,
As discussed in [OV99], there are four types of failure which can occur during a transaction:

**Transaction failures:** A transaction can fail due to a deadlock or some other integrity constraint violation. Usually in this case, the transaction is aborted.

**Communication failures:** Nodes of a distributed system communicate through messages. The messages can get lost or duplicated. By using a reliable transmission protocol such as TCP, these failures can be avoided.

**System failures:** When these type of failures, which are also known as soft failures, occur, data in volatile storage is lost but stable storage remains intact. Extensive tests of the system can reduce the risk to these failures, but they can still occur non-deterministically. In such a case it is desirable that the system stops working: *fail-stop*.

**Media failures:** These type of failures, also known as hard failures, occur when a device fails. In this case data in stable storage is lost. However the advances in technology, has reduced this type of failure dramatically.

Two of the major requirements of a system namely reliability and availability are actually measures against these last two types of failures. The former requires a system to provide with the correct information and the latter requires it to give the information in a specified time frame. In the context of distributed transaction soft crashes are interesting to study. This is because a single transaction can be carried out by more than one node and if one of them crashes, the protocol should still guarantee ACID-ity of the transaction.

### 1.2.2.1 System failures

Henceforward only system failures will be considered. Communication link failures will be handled by the network, because we assume the network to be reliable. However a broken link, which is a special case of link failures will be also considered during commit protocols. In fact both of these types of failures are usually identified by timeouts.

Further, site failures are classified [RP93a] in two types: *omission* and *commission* failures. Nodes with fail-stop behavior are said to deal with
omission failures. That is an omission failure simply causes the node to stop and not to take any action. On the other hand if a fault causes a node to take actions not specified by the algorithm are said to be of type commission. This corresponds to the class of well known Byzantine failures.

A node crash is a failure of type omission, because it loses its state and stops working as specified. A node sending conflicting message to two different nodes is an example of commission failure.
CHAPTER 2

Atomic Commit Protocols

The problem of atomic commit arises during transactions in distributed systems. In such systems, a transaction is carried out by different sites. In order for a transaction to be atomic, it should leave the system and hence all the participating sites, in a consistent state after its completion.

The correctness criteria of such a protocol is that eventually all of the participants reach the same decision even in a failure case. To exclude the trivial case where everyone aborts, another criteria is added which states that any final decision reached should be one proposed by a participant. This is called atomic commit protocol (ACP).

In this chapter we describe some of the existing commit protocols and their properties. The goal is to present them in a clear and straightforward manner and outline their key properties.

2.1 Background

To understand how transactions work, it is important to know about the participating components and their interaction model. In a distributed DBMS, the sites participating in a transaction have a role: coordinator or participant. The coordinator, as its name suggests, coordinates the transaction between the participants of the transaction. Participants just follow the instructions given by the coordinator.

Transactions can be performed in flat or hierarchical manner (Figure 2.1). The former is characterized by one coordinator interacting directly with its
(a) In flat transactions there is just one level of interaction. (b) In hierarchical transactions root coordinator interacts with participants which can as well have the role of (sub-)coordinators.

participants, whereas the latter is identified when a transaction is formed by many subtransactions. The coordinators of subtransactions act as participants to the parent coordinator. Notice that by adding a level in the hierarchy, the coordination can be distributed but at the cost of higher message delay.

In addition there are two ways a protocol can be carried out \cite{Ske81}: centralized vs. decentralized (Figure 2.2). In centralized paradigm the coordinator follows different steps than a regular participant, whereas in decentralized paradigm each participating site executes the same steps of the protocol, in that everyone sends messages to all others and waits for their responses to begin with the next phase.

Each node has volatile and stable storage. A volatile storage is one where data does not survive system crashes, whereas if it is stored in stable storage, it can still be retrieved after a system crash. Interaction between the participants and coordinator is performed by sending and receiving messages. In order to keep a consistent state across all the participating sites, a mechanism is needed to handle the failure cases and recover from them. Usually it is done by writing a log of actions into stable storage.

The performance of an ACP is evaluated by metrics such as number of messages exchanged and force-writes (synchronous writes) in the log \cite{Wol90, DS83}. This however is not enough for the overall performance evaluation
2.1. BACKGROUND

![Diagram](image)

Figure 2.2: (a) Central model with one coordinator as a central component, where protocol steps in a participant is different than those of a coordinator (b) Decentral model with all nodes having the same role and protocol to carry out.

of a particular system. In this case quantitative measurements need to be carried out through simulation as in [LAA94, CSAH98, AU04]. This thesis gives however only the asymptotic complexity in the failure-free case. We do not evaluate the performance quantitatively by simulation or extensive experiments.

Many variants of ACP have simply tried to reduce number of message rounds and force-writes in the log. Others such as decentralized 2PC [Ske81] or linear 2PC [Gra78] try to exploit the topology of communication networks.

A well-known weakness of 2PC and its variants is their blocking aspect. This is because in case of a failure the operating sites must wait until the failed one recovers. To cope with it another phase is added or protocols based on consensus algorithm are proposed.

In the rest of this chapter we describe these different protocols, discuss their different underlying assumptions and outline their performance in asymptotic manner.
2.2 Basic 2PC

In classic two phase commit protocol [Gra78], as its name suggests transactions are carried out in two phases: prepare and decision phases. 

During the first phase, called prepare phase, the coordinator by sending a prepare message, asks all the participating sites if they are ready to commit. Participants force-write in the log the action they want to take. Notice that till this point they have the ability to unilaterally abort the transaction, but once they write in the log they can not revoke their decision. After the write in the stable storage every participant sends its vote to the coordinator. If its vote is to commit it enters the prepared state. But if its vote is to abort, it then aborts locally and releases all locks without waiting for the coordinator’s decision.

During the second phase, called decision phase, after the coordinator has received all the votes, it can decide. Its decision is to commit if all the participants voted YES, but if at least one votes NO it decides to abort. At the beginning of this phase, it force-writes the decision in its log and then sends it to all the participants who voted YES. Each participant upon receipt of the decision from the coordinator, takes the appropriate action and force-writes it in the stable storage then sends an ACK to the coordinator. Coordinator after receiving all the ACKs writes asynchronously an END to the log and forgets the transaction otherwise it retransmits the decision. (Figure 2.3)

2.2.1 Fault resilience

In synchronous distributed systems, site failures or communication failures are detected by timeouts. In case of 2PC and its variants this is done similarly.

To achieve consistency in case of system failures, a recovery process is executed once a crashed node restarts; it reestablishes the transaction context from stable logs.

To understand what happens, each time a node crashes during a transaction, we might consider the state diagram of a node (Figure 2.3 (a)(c)). The participant crashes and coordinator is in the following state:

- *waiting:* As coordinator has not yet decided it can safely decide to abort and forget about the transaction.
2.2. BASIC 2PC

- decided but no ACKs: if Coordinator has already decided but has not yet received some ACKs, then it simply retransmits the decision to those not yet acknowledged.

The coordinator fails and participant is in the following state:

- initial: As mentioned earlier a participant has the ability to unilaterally abort, in this case it aborts.

- prepared: This is a part where 2PC is known to be weak, as it blocks the participant until the coordinator recovers.

The blocking behavior of 2PC is known to be its drawback and many of its variants, as we see later, try to reduce it.

2.2.2 Performance

Let $n$ be the number of participants during a transaction without counting the coordinator. In order to be able to evaluate the performance of 2PC protocol in term of its communication cost and time, we have:
2.3 Presumed-Abort

To improve the performance of basic 2PC one could think of reducing the number of messages and force-writes to the stable storage. Mohan et al. [MLO86] show that by removing these informations, the fate of a transaction in some cases can be presumed.

In the basic 2PC when a coordinator crashes before writing the decision into the stable log, transactions will be treated as aborted. That is why the coordinator does not force-writes the beginning of a transaction. This idea of aborting when no information exists is integrated into basic 2PC to enhance the performance.

In PA when a participant votes abort, the coordinator sends abort to all participants and forgets about the transaction. In this case it does not await the ACKs from the participants. In addition a participant does not force-write the decision. On the other hand if the decision is to commit, it will follow the same steps as in basic 2PC. So in case a recovery process inquires about it and there is no information available, then it is safe to abort.

2.3.1 Performance

The presumption made by PA protocol reduces the number of messages and log writes only when the transaction aborts. As shown in Figure 2.4 (a), there is $3n$ messages and only $n$ force logs. Whereas in a commit case there is no performance gain comparing with basic 2PC. This make them optimal to use in systems where the probability of transactions to be aborted is higher than of those being committed.
2.4. Presumed-Commit

The counter-part of presumed-abort is presumed-commit protocol (PC) [MLO86] but it is not completely symmetric to PA. Suppose the coordinator crashes after receiving one YES vote. During the restart, as there is no information about the transaction in the log, the recovery process of the coordinator aborts and forgets about the transaction without informing others. In this scenario inconsistency arises when that prepared participant inquires about the transaction and as there is no information available about it, it commits (presumed-commit).

To avoid this problem, PC acts slightly different than PA (Figure 2.4). In that it force-writes the beginning of a transaction then sends prepare messages to the participant. Each participant force-writes its vote and sends it to the coordinator. Once all participants have voted, the coordinator then force-writes the decision and sends it to all the participants. If the decision is to abort it ends like basic 2PC, but if it is to commit participants write in the log commit and the transaction ends.

2.4.1 Performance

The optimization introduced here, performs well only in case of a commit. Otherwise it performs same as basic 2PC with an additional force-write in the log which marks the beginning of the transaction.
2.5 Early-Prepare

This is a protocol which combines PC with the following optimization: During the execution of a transaction each site makes sure it is in the prepared state. In that it immediately updates the data without waiting for the prepare message of coordinator. Coordinator can safely jump to the decision phase directly once all the operations of a transaction are executed. This phase is then similar to PC’s decision phase (Figure 2.6).

This idea removes the necessity for the first message round and logs, however it introduces some extra cost. In order to rollback, a participant needs to keep track of the changes it made. This is done by force-writing in the stable storage before any operation is completed. Additionally the coordinator needs to keep a list of all participants to contact in case a failure occurs before beginning the commit protocol.
2.6. 3PC

![3PC Diagram](image)

Figure 2.6: Message flow and writes into the stable storage in EP (a) abort case (b) commit case

2.5.1 Performance

Here another variable should be introduced, let it be \( o \) representing number of operations in a transaction. This protocol performs well when the number of participants are bigger than the operations in the transaction. It should be however noted that there are more forced-logs

<table>
<thead>
<tr>
<th></th>
<th>Commit</th>
<th>Abort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of messages</td>
<td>( n )</td>
<td>( 2n )</td>
</tr>
<tr>
<td>Number of message rounds</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Number of force-writes</td>
<td>( o + 2 )</td>
<td>( n + o + 2 )</td>
</tr>
<tr>
<td>Blocking</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>

2.6 3PC

This is an extension of 2PC in order to prevent its blocking behavior. This is done by adding another phase, called pre-commit [Ske81] between the two phases (Figure 2.7). By doing so, in the local state transition of each site a "buffered state" is introduced.

Once the coordinator reaches its decision it sends its intention in the pre-commit phase to all participants. Each participant record this intention and sends ACK but does not update yet. If the coordinator crashes, the participants block and a termination protocol is started. During this pro-
tocol another coordinator, called the backup coordinator is chosen among the participants. This new elected coordinator proceeds in two phases:

- ask all others in which state they are.
- based on the first phase’s outcome, it decide to commit or abort.

It should be noted that if the backup coordinator itself is in aborted or committed state, the first phase can be left out and the decision is sent directly.

### 2.6.1 Performance

Even if it removes the blocking behavior of 2PC in case of the coordinator failure, this protocol has a very high message and time complexity in the normal execution.

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of messages</td>
<td>$6n$</td>
</tr>
<tr>
<td>Number of message rounds</td>
<td>6</td>
</tr>
<tr>
<td>Number of force-writes</td>
<td>$3n + 1$</td>
</tr>
<tr>
<td>Blocking</td>
<td>NO</td>
</tr>
</tbody>
</table>
2.7 1PC and its variants

In a 1PC the voting phase is removed and the commit protocol has only one phase, namely decision phase [Gra90]. The idea is that when an operation of a transaction is executed, the participant acknowledges it directly and enters prepared state. The coordinator then sends commit only if every operation is acknowledged otherwise it aborts.

Recall that in 2PC it is still possible during the protocol that a participant aborts the transaction, whereas in 1PC the ability of a participant to unilaterally abort is removed.

This restriction and the strong assumptions behind 1PC protocols make them not so popular in practice. This includes the immediate constraint-checking, i.e. consistency is directly checked locally by each participant before acknowledging the operation. Secondly the strict two-phase locking is used in order to avoid cascading aborts.

EP mentioned before is actually a 1PC protocol. We consider briefly here two other variants of 1PC: Coordinator Log (CL) [SC93] and Implicit Yes Vote (IYV) [AHC00].

As we discussed, in EP participants keep record of every operation by force-writing it in their logs, which is used then used during recovery process. This introduces an overhead in the participant site. To avoid it, in CL protocol, a participant piggybacks the undo-, redo-log with the ACK then it sends it to the coordinator. The coordinator then records it to its stable storage. This makes participants dependent on the coordinator in case of failures and rollback.

In IYV participants sends their redo-logs and read locks to the coordinator and in the same time maintain a local log. In this way they are independent of the coordinator in case of a rollback but need to communicate with the coordinator if they crash.

2.8 Further optimizations

There are some other cases where commit protocols can be further optimized. One such optimization is the (partially) read-only transactions [MLO86]. During which, if a participant finds out that it has not updated any data, it sends back a READ vote to the coordinator, releases its locks and forgets about the transaction. The coordinator can then safely exclude
it from the second phase of the protocol.

In order to reduce the blocking aspect of 2PC, a termination protocol can be executed. One such protocol is cooperative termination protocol \([\text{BHG87}]\), in which, an undecided participant inquires other participants if they are in a decided state. If so it will decide accordingly, but if one of them has not voted yet then it can abort unilaterally and send all undecided ones its vote. They will block if all of them are in undecided state.

### 2.9 Consensus algorithms

#### 2.9.1 Background

The problem of consensus arises when building fault-tolerant systems. In distributed systems participating processors are considered to be either faulty or correct. In such an environment, problems which require coordination can be solved by consensus algorithms. This covers problems such as clock synchronization, ordered atomic broadcast and atomic commit. In general consensus algorithms allow the correct participating processors, not necessarily the faulty ones, to reach the same decision.

More formally following properties should be fulfilled:

- **consistency**: all correct processors decide irrevocably on the same value.
- **validity**: The output value will be some processor’s private value.
- **termination**: Eventually all the correct processors decide.

Generally distributed systems are classified under:

- **Synchronous Systems**: Synchrony in distributed systems is defined with respect to the message transfer delay. In case the message transfer delay and its processing time by receiving node is bounded by a constant \(\delta\) known to all sites, we have a synchronous distributed systems. During this period the site that started the communication, is blocked.

- **Asynchronous Systems**: Blocking aspect in a distributed system is often seen as a drawback, as other processes can not utilize the resources
during that period. Asynchronous distributed systems avoids this in that it does not contain a predefined $\delta$ as upper bound on message delay. After a message is sent, the process can continue with other operations.

When designing a protocol and analyzing its correctness one needs to understand the assumptions it is based upon. Usually it is impossible in some cases to design a protocol, but by weakening the assumptions solutions emerge:

In [FLP82] Fischer et al. proved the impossibility of a protocol which guarantees a consensus in an asynchronous system even with one faulty processor.

Dolev et al. [DDS87] shows with more relaxed assumptions that in some cases the consensus can be reached. First they define some system characteristics:

- Processors are either synchronous or asynchronous. In synchronous processors, after every step taken by a processor, every other processor will take at least one step. Asynchronous processors are known to wait arbitrary long between steps.

- Communication is asynchronous when messages can take an unbounded delay to be delivered, it is synchronous if messages are delivered in bounded time.

- Message can be ordered or unordered.

- Transmission mechanism can be point-to-point or broadcast.

With these system characteristics, there are three cases when consensus can be solved:

<table>
<thead>
<tr>
<th>Case</th>
<th>Processors</th>
<th>Communication</th>
<th>Messages</th>
<th>Transmission</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>synchronous</td>
<td>synchronous</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>synchronous</td>
<td>ordered</td>
<td>broadcast</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>synchronous</td>
<td>ordered</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In [FLP82] the assumptions are different: Message are unordered, communication and processors are asynchronous.
2.9.2 Consensus and ACP

The FLP impossibility results from the difficulty to determine if a process has crashed or is simply slow. Chandra and Toueg [CT96] give a solution to consensus problem in asynchronous systems using unreliable failure detectors. These failure detectors find out about failures that occur in the system but they might mistake as well. Guerraoui discusses in [Gue95] if a non-blocking atomic commit (NB-AC) can be as well solved in asynchronous systems\(^1\) using failure detectors. The fact that in NB-AC all processes should reach the same agreement and not only the correct one as in consensus, makes it a harder problem than consensus. This makes it impossible to solve NB-AC with unreliable failure detectors in asynchronous systems. However in this paper another practical solution which is a weaker version of NB-AC called NB-WAC, is presented.

Paxos [Lam98] is a family of protocols used to solve the consensus problem. Transaction commit protocol belongs to the consensus problem. As most of the 2PC based commit protocols block, there is a wide range of commit protocols using consensus algorithms to avoid the blocking behavior. Paxos commit algorithm [GL06] is indeed a non-blocking protocol, which uses \(2F + 1\) coordinators and tolerates \(F\) failures. The classic 2PC is then a special case \((F = 0)\) of Paxos commit algorithm. The authors then show that it is possible to implement such algorithm with the same write delay as 2PC, same message delay during failure-free case but with more messages exchanged.

Consensus algorithms tolerating Byzantine failures are also known as Byzantine agreement. As discussed in [Gra90], even though Byzantine agreement generates more overhead than atomic commit problem, they both are identical in fault-free case. They are however different when failures occur.

Following table summarizes the main differences:

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Fault tolerance</th>
<th>Agreement</th>
<th>Blocking</th>
<th>Message complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commit Protocol</td>
<td>Many faults</td>
<td>All processors</td>
<td>YES/NO</td>
<td>Low</td>
</tr>
<tr>
<td>Byzantine Agreement</td>
<td>(N/3)</td>
<td>Some processors</td>
<td>NO</td>
<td>High</td>
</tr>
</tbody>
</table>

Integrating Byzantine agreement in the commit protocol adds more overhead to the protocol and is not always practical. However in some case where untrusted components participate in transactions then it becomes unavoidable. The internet as an untrusted network is one example where Byzantine agreement can’t be ignored.

\(^1\)Recall that 3PC, also a non-blocking protocol, is designed for synchronous systems.
In [MSF83] a non-blocking commit protocol is proposed, where the second phase of the protocol is replaced by a Byzantine agreement algorithm. Doing so prevents the coordinator to send conflicting messages to the participants. Another Byzantine fault tolerant protocol is presented in [Zha07], where coordinators are replicated and a Byzantine agreement is run among them. This protocol tolerates Byzantine faults in the coordinators and in a subset of participants.

Similarly to tolerate Byzantine failures, but without using Byzantine agreement algorithm, Rothermel et al. [RP93b] propose a protocol in open systems, where nodes are either trusted or non-trusted. Their protocol tolerates any number of Byzantine failures in non-trusted nodes and terminates consistently in at least the trusted nodes.
Part II

mycel framework
In this chapter we start by giving the motivation behind mycel and its requirements. We then discuss its computational model and architecture shortly. Since this thesis addresses only the commit protocol used in mycel we focus on requirements and properties of such a protocol together with properties of mycel which influences the choice.

3.1 Background

Dissatisfaction with the current object relational semi-solutions and their steep learning curves triggered the search for a persistence silver bullet. The fact, that current persistence solutions still lack seamless interoperability with object oriented languages and the inability of object oriented databases to leave their niche supported the idea. The zoo of alternative persistence solutions apart from relational databases is enormous, but seems to remain at an academic or prototype level.

3.2 Vision

The vision for the mycel persistence framework ist to provide a simple, flexible and scalable persistence solution for mainstream applications. The framework should close the object relational gap by completely replacing the object relational mapper and the relational database. To achieve this
goal, several properties of relational databases have to be provided by the new framework. Mainly the ACID properties.

The ideal use case scenario looks as follows

```java
void transfer(int amount, int fromAccount, int toAccount) throws VariousExceptions {
    PersistenceManager pm = new PersistenceManager([Discovery Method]);
    Transaction trx = pm.beginTransaction();
    try {
        domain.Account a = trx.findDomainAccountInstanceByNumber(fromAccount);
        domain.Account b = trx.findDomainAccountInstanceByNumber(toAccount);
        if (a.getValue() >= amount) {
            a.withdraw(amount);
            b.deposit(amount);
        }
    } catch (mycel.api.Exception ex) {
        trx.rollback();
    }
    trx.commit();
}
```

Envisioned features of the mycel persistence framework are:

**ACID:** Relational databases are used in object-oriented application mainly for their ACID properties. mycel must guarantee ACID properties to compete OR architectures.

**Ease of use:** Developer-friendly interface. Language neutrality. Focus on data and not on technology.

**Agility:** Easy to modify and migrate datastructures and data.

**Scalability:** Gained through distribution.

**Availability:** Gained through redundancy and distribution.

These features lead to the following technical requirements:

**Fully replicated data:** Location transparency is achieved by fully replicating data. Provides fault-tolerance. Provides location transparency to users. ACID durability.

**Lockless operation:** To scale, the number of possible locks / synchronization points has to be as low as possible.
3.3 Computational model and architecture

The computational model in mycel follows the same distributed system model found in literature which is formed by nodes interconnected through a communication network. The set of nodes that participate in a transaction or to whom the transaction is relevant forms a cluster. Inside a cluster each node has its own copy of all objects hosted by the cluster.

Each node can initiate a transaction on objects and should an object be updated it should be done consistently in all the nodes of the cluster, i.e. it is replicated in all other nodes of the cluster. We notice that it is not the same as in distributed databases, where data is distributed between nodes and in which a transaction might contain operations to be executed by different sites and hence different data subsets (Figure 3.1 and Figure 3.2).

Applications using mycel framework (Figure 3.3), will access and update the domain objects which are replicated in all nodes of the cluster. Each node has a transactional object memory where the current instances of objects reside. A client application will access a proxy object, make its updates and if it wants to commit the changes, it will make sure it is done in all the nodes of the cluster. If the commit is successful, each transactional object memory of the nodes will contain the same objects.

We specify the replication in mycel according to the 3-parameters classification of database replication discussed in [WSP+00], in which any replication technique can be distinguished by its server architecture (primary copy vs. update everywhere), transaction termination (voting vs. non-voting schemes) and server interaction (constant vs. linear):

- **Server architecture**: It is of type *update everywhere*, in which there is no primary or master server and all the servers are updated without any discrimination.

- **Transaction termination**: To guarantee consistency at all the sites, the transaction termination is carried out by *voting* such as 2PC.

**Transactional modifications**: ACID atomicity and consistency.

**MVCC**: ACID isolation is achieved through multiversion concurrency control.
Figure 3.1: A cluster of four nodes where data objects are replicated. A sample transaction reads/writes its local objects but then needs to synchronize its modifications with all other replicas.

- **Server interaction:** Interacting with other servers is performed in *constant* manner, where a set of operations is packed into one message and then sent.

### 3.4 Requirements and specification

In this section we discuss the requirements and specification of mycel and its transaction model in order to be able to validate one suitable atomic commit protocol.

As mentioned before in mycel any node on the cluster can initiate a transaction. This, if not correctly handled, leads to inconsistent read and write during concurrent transactions. To ensure isolation among concurrent transactions, multi-version concurrency control (MVCC) [BG81] mechanism is used. One of the advantages of MVCC is that it does not need to lock data, where it provides every transaction with a current valid copy of the data.
3.4. REQUIREMENTS AND SPECIFICATION

Figure 3.2: A cluster of four nodes in a usual distributed DBMS where each node contains a subset of the whole data. A transaction can have operations executed on different nodes, i.e. different data subsets, but the outcome should be the same in all participants.

Commit is then only successful when the data version is still the same as when transaction borrowed it.

From the first part we know that logging the transaction states is required by the recovery process, which reestablishes a transaction’s context so that consistency is ensured in the crashed node. In mycel however logging is not required, it assumes that a recovering node synchronizes itself with other nodes upon recovery. Recovery and synchronization are however not addressed in this thesis.

The fault model of mycel covers only failures of omission type, i.e. failures that are also known as fail-stop and caused usually by node crashes or a broken link in the communication network. As all the nodes are trusted and there is no reason to believe that they can send each other malicious or forged messages, we exclude Byzantine failures or more generally known as faults of type commission. In addition communication in mycel is in synchronous manner.
3.5 Atomic commit in mycel

Here we discuss an important difference between atomic commit in mycel and commit protocols we studied in the first part.

In atomic commit in distributed DBMS (Figure 3.2) the decision should be unanimous. That is when a transaction wants to commit, all of its participant should vote YES and if even one of them votes NO then the transaction is aborted. This is because in distributed DBMS, data is distributed across different nodes and the transaction is only then atomic when all of the participants commit their part of the transaction.

In mycel however we can safely assume that if the majority votes YES then the decision is to commit otherwise it is to abort. This is due to MVCC and the fact that mycel transactions operate on the local objects and if locally a transaction is prepared to commit, its vote-request message merely means checking if other nodes agree on those updates.

This difference in the mycel transaction model will allow us to design a better suited protocol based on the usual atomic commit protocols.
Let’s now analyze why a majority of votes suffices. To do so let’s see why a NO vote can occur and why we can ignore some of them, that is if there are $n$ participants $\left\lfloor \frac{n-1}{2} \right\rfloor$ votes can be ignored:

1. Newer version of objects are available, i.e. some other transaction updated those objects. If this is true then there is the majority of the nodes that will vote NO, hence the final outcome will be abort.

2. A concurrent transaction has borrowed a copy of objects before the current transaction, there are chances that it commits first. In this case one of the transactions eventually wins over the other if it collects the majority YES vote, but both will abort if they receive an equal number of votes. This can occur only when the number of participating nodes in the cluster is even. In this case it can be seen as a drawback if these two transactions are retried and again they face the same situation which can go on and on until one of them eventually wins.

3. No votes at all (due to link failure or node crash) can also be treated as a NO vote. In this case again if majority has voted YES then commit. As far as the disconnected node is concerned it will synchronize itself with the rest upon recovery.
Proposed commit protocol

We discuss first the protocols seen in the first part and their weaknesses if integrated in mycel.

1PC protocols perform well during commit time as there is less message rounds. This comes with a cost for the participants, who hold the locks on resources for a longer period of time, that is, from beginning of the transaction till the commit protocol is started and carried out. This is an undesirable feature in mycel. 2PC and its variants have the well known problem of blocking when coordinator fails. They could however still be considered for mycel. 3PC removes blocking aspect of the 2PC but introduces additional overhead by adding another phase.

Atomic commit protocols based on consensus have different assumptions, namely they are proposed to tolerate commission failures. mycel however excludes this type of failure. In addition these protocols have a very high overhead in terms of message exchanged and message rounds.

In the previous chapter, we mentioned how a majority of votes can be sufficient to reach an agreement in the transaction. This means also that the ability of unilateral abort as in ACP’s studied in the first part is not required.

In this chapter we propose a commit protocol based on decentralization idea of [Ske81] 2PC. This simple yet elegant protocol removes the blocking aspect and reduces the execution time.
4.1 Decentralized atomic commit

We discussed in the first part that decentralizing an atomic commit protocol, has two advantages:

- everyone follows the same steps in the protocol, i.e. there is no coordinator specific protocol and no participant specific protocol.
- communication time is reduced because of just two rounds of messages but this comes in expense of communication cost where for \(n\) nodes, \(n(n - 1)\) messages are exchanged.

In addition, our assertion that a majority of votes is required for a decision, together with the idea of decentralizing the protocol, reduce and even remove the blocking aspect which is found in most of the 2PC protocols. Recall that in 2PC, if the coordinator crashes before sending its decision to the prepared participants, then they are blocked.

The node which started a transaction is called the initiator. When a transaction wants to commit, the initiator first prepare itself and broadcasts its vote (YES or NO) along with a request-for-vote to the rest of cluster members (Figure 4.1).

Each member will then record the initiator’s and its own vote and broadcasts its vote to all others. Every member node does so and whenever it receives the majority of votes it will decide on the fate of the transaction locally and independent of others.

A node will vote YES if its objects have the same versions as when the transaction started and if no other (concurrent-) transaction is running on those objects. Otherwise it will vote NO. Recall that in case of mycel, even if a node votes NO, it does not mean that the transaction will abort. So when a node votes NO, it will still keep the information related to that transaction which probably commits at the end.

When omission failures occurs, that is, when a communication link is broken or a node crashes, the disconnected or crashed node has to synchronize itself with the rest upon restart. If the minority of nodes is faulty, the protocol can still reach a decision. In this case we can follow one of two heuristics: consider the vote of crashed node as a NO vote, or the recompute the majority among the existing connected nodes. This flexibility contributes to the non-blocking behavior of this protocol in case of failures.
4.1. DECENTRALIZED ATOMIC COMMIT

Figure 4.1: Message rounds during decentralized ACP, (a) A is the initiator of the transaction and sends its vote to others implicitly asking them to vote, (b) other nodes of the cluster multicast their votes, and whenever a node has the majority of YES votes, it commits.
Tests and evaluation

Having opted for the decentralized ACP and integrated it to mycel, in this chapter we add some test scenarios to examine the behavior of the protocol.

In the test environment (Figure 5.1) we have a cluster of 3 nodes which exchange messages through a (emulated-) network. To simulate and control the delivery of messages the network uses a scheduler. The network’s communication primitives are unicast and broadcast. The former sends a message to a single node whereas the latter does it to all the members of the cluster. In a real world application, the nodes build machines which form a cluster and interconnected by a reliable network (such as TCP/IP). We consider all the plausible -but not superfluous- scenarios which can happen during transactions in a distributed system. This includes:

- non concurrent transactions in failure-free cases
- concurrent transactions initiated from the same or a different node
- transactions during which failures such as node crash or link failure occur

It becomes soon cumbersome to describe with words the behavior of transaction and order of messages sent and received by nodes in a distributed fashion. To alleviate it we use the dot tool\(^1\) to generate a graph of the transaction, where nodes, messages, node crashes and time-points of sent and received messages will be visualized.

\(^1\)http://www.graphviz.org
Figure 5.1: Simulated test infrastructure with a cluster of 3 nodes connected by a network which in turn uses a scheduler to deliver messages to cluster nodes. The control unit is used to perform the simulation and record the steps during a transaction so that it can be visualized with a graph.

In a second step to evaluate the performance of the protocol, we will focus on the time complexity and message complexity. We will not measure the latency directly as this can’t be generic for all the network types, but give the number of message rounds during commit. Message complexity will be of course to give the number of messages exchanged.
5.1 Test scenarios

In all these scenarios we have a cluster of 3 nodes unless stated otherwise. Any node can start a transaction in which case we call it the initiator of the transaction. As specified in the protocol, a majority of votes is needed to reach an agreement, that is if 2 out of 3 nodes are prepared to commit, then all will commit.

5.1.1 Scenario 1

This is a normal scenarios where no node crashes and no concurrent transaction is carried out. All the nodes have the same state at the beginning. Node 1 starts a transaction and sends YES vote to the other two asking them for their votes. Node 2 and 3 record the vote of initiator (Node 1) and votes themselves as well. As soon as they issue their votes they have two YES votes and commit the transaction. When Node 1 receives the first vote it commits as well.

![Diagram of a transaction without any failure](image)

Figure 5.2: One isolate transaction without any failure

5.1.2 Scenario 2

In this scenario there are two concurrent transactions running on node 1. When transaction A wants to commit, node 1 sends its YES vote to the rest
and requests them for their votes. In the meanwhile transaction B wants to commit as well. In this case node 1, knowing there is already a concurrent transaction, will issue a NO vote and asks the other 2 nodes for their votes as well. Notice that no node has the ability to unilaterally abort, they can vote NO but it does not count for the transaction to abort immediately. A transaction aborts only if it has a majority of NO votes.

Now each of the 2 nodes will vote YES for the transaction they first receive a prepare command. Suppose Node 2 receives prepare for transaction B first and node 3 for transaction A. Node 2 will vote YES for transaction B and NO for transaction A. Node 3 will vote YES for transaction A and NO for transaction B. Finally we have:

<table>
<thead>
<tr>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction A</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Transaction B</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
</tbody>
</table>

![Figure 5.3: Concurrent transactions initiated in the same node](image)

5.1.3 Scenario 3

This scenario is the same as previous one except both nodes receive, prepare command for transaction B first. In this case they vote YES to transaction
B, but No to transaction A. Finally transaction B wins, even if the initiator (node 1) votes NO.

<table>
<thead>
<tr>
<th></th>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction A</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>ABORT</td>
</tr>
<tr>
<td>Transaction B</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Figure 5.4: Concurrent transactions initiated in the same node. The transaction which started last, commits.

### 5.1.4 Scenario 4

In this scenario we have two concurrent transactions, however started from different nodes. Transaction A is initiated at node 1 whereas transaction B at node 2. It is clear that each of these nodes will vote YES on their own transaction but NO on the concurrent transaction. Node 3 in this case will votes YES for the first prepare received. Suppose it receives prepare for transaction B first. It will issue a YES vote for transaction B and a NO vote on transaction A. Finally transaction B wins over transaction A:

<table>
<thead>
<tr>
<th></th>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction A</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>ABORT</td>
</tr>
<tr>
<td>Transaction B</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>
5.1.5 Scenario 5

This is a scenario where one node crashes. There are two concurrent transaction started by different nodes of the same cluster. One node crashes after sending its votes. The others can still reach an agreement (commit/abort). Note that if that node crashes before sending its vote, the others will consider its vote as a NO vote. They can find out about a node’s failure by using timeouts.

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>COMMIT</td>
</tr>
<tr>
<td>B</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>ABORT</td>
</tr>
</tbody>
</table>

5.1.6 Scenario 6

In this scenario there is one transaction running and minority of nodes (in this case 1) crashes before sending its vote. As the majority of nodes is alive, they can still reach an agreement. If they all votes YES, the transaction commits, but if at least one votes NO then it will abort. In this scenario both nodes will vote YES and transaction commits.
5.1. TEST SCENARIOS

Figure 5.6: Concurrent transactions with a node crash

Figure 5.7: Isolated transaction and the minority of nodes crash

5.1.7 Scenario 7

As in the previous scenario, there is one transaction running but the majority of nodes (in this case 2) crashes before sending their votes. Transaction in this case aborts after node 1 times out for the votes of others.
5.1.8 Scenario 8

This is an interesting scenario as well where number of nodes in the cluster is even. Let’s assume there are 6 nodes, where the majority is 4. When there are concurrent transaction, it might happen that half of the nodes vote YES for transaction A and the other half for the transaction B. In this case both transactions abort as they fail to obtain a majority of votes.

<table>
<thead>
<tr>
<th></th>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
<th>Node 4</th>
<th>Node 5</th>
<th>Node 6</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction A</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>ABORT</td>
</tr>
<tr>
<td>Transaction B</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>ABORT</td>
</tr>
</tbody>
</table>

5.2 Evaluation

We evaluate the proposed protocol by giving its time complexity (message rounds) and message complexity. In decentralized fashion there is obviously more messages exchanged. This is because every one sends a message (in this case its vote) to everyone else, which means that there are \( n(n - 1) \) messages to be carried by the network during commit protocol. This can be an overhead for the network, if the number of nodes are high (e.g. for 100 nodes it is in the order of 10000 messages), but we assume that in mycel’s environment there are no more than half a dozen of nodes, in which case the number of messages exchanged can not become network overhead.
This higher communication cost however enables a lower communication time: to reach an agreement only 2 message rounds are needed. The first round is when the initiator of the transaction sends its vote and requests the others for their votes. The second round is when every node of the cluster broadcasts its vote to the rest.

Finally notice that from \((n - 1)\) messages awaited by each node, only \(\lfloor \frac{n}{2} \rfloor\) messages (either all YES or all NO) are needed to reach an agreement. This reduces the blocking aspect in the protocol, that is whenever a majority of votes is received, there is no need to wait for the rest of votes.
Conclusion

The existing wide range of distributed commit protocols in the literature, makes it difficult for practitioners to choose among them. In this thesis in order to tackle the transaction commit problem in mycel, a prototype for transactional object persistence, we studied different existing protocols.

Each of these protocol have similarities and differences in the assumptions behind them, their requirements and the class of failure they are resilient to. In order to evaluate a protocol, we gave the message complexity (number of messages exchanged), time complexity (number of message rounds) and their blocking behavior.

Based on the requirement of mycel, we then designed and implemented a decentralized commit protocol. This protocol is based on the idea of decentralizing the protocol and use of (majority) quorum-based decision. Both these ideas reduce blocking aspect and the overall communication time of the commit protocol.

We then integrated and validated the proposed protocol in the system, ran it in different scenarios that we identified could occur during transaction commit. We observed the behavior of the protocol during these scenarios and it matched our expectations.

**Future work** In order to evaluate performance of the commit protocol, only asymptotic complexity is not enough for a particular system. It should be evaluated quantitatively based on discrete-event simulation. System consistency after failures is reached by recovery protocols. We discussed that in mycel, a crashed node upon restart, synchronizes itself with
others. This issue should be further studied and developed as it is an im-
portant component of any distributed system.
dot-generated test scenarios

These scenarios as described in the last part of the thesis are generated with the dot tool. They correspond to the same 8 scenarios discussed in the thesis.
Figure A.1: One isolate transaction without any failure
Figure A.2: Concurrent transactions initiated in the same node
Figure A.3: Concurrent transactions initiated in the same node. The transaction which started last, commits
Figure A.4: Concurrent transactions initiated in the different nodes
Figure A.5: Concurrent transactions with a node crash
Figure A.6: Isolated transaction and the minority of nodes crash

Figure A.7: Isolated transaction and the majority of nodes crash
Figure A.8: Concurrent transactions on an even number of nodes in the cluster


