Delft University of Technology

Distributed Systems (Group 7)

IN4391

# Final Report: Byzantine-CP

*Authors:*
Douwe Brinkhorst, Patrik Kron, Michael Leichtfried, Miguel Lucas

2020-04-24

# 1 Introduction

In the past weeks we have implemented the system described in *"A Byzantine Fault Tolerant Distributed Commit Protocol"* by Wenbing Zhao[1].

In the paper the author describes a commit protocol for transactions which run over untrusted networks. The Two Phase Commit (2PC) protocol addresses the issue of implementing a commit protocol for distributed transactions, and different approaches have been conducted in order to make it support byzantine behaviours. The main idea of the byzantine-commit-protocol is the replication of the coordinator and running a Byzantine agreement algorithm among the replicas. This protocol tolerates byzantine coordinator and limited faulty participant behaviour.

# 2 Objectives

We set objectives from the beginning in order to figure out and organise the work that had to be done. The objective list was divided into categories to state the priority of each objective.

- **Must have:** features the project must have in order to fulfill the basic requirements.
  - Implementation for coordinators.
  - Implementation for participants/initiators.
  - System testing infrastructure, including coordinator byzantine behaviours testing.
- **Should have/Could have**: features that might be implemented depending on time constraints.
  - Distributed deployment: test the system in multiple interconnected machines to simulate a realistic environment.
  - Message signing and signature checking.
- **Could have/Will not have**:
  - View change mechanism (this feature was not implemented by the paper authors either).

# 3 The Protocol

## 3.1 Byzantine Fault Tolerance - Why

There are multiple reasons to choose a byzantine fault tolerant distributed commit protocol.

[1] "A Byzantine Fault Tolerant Distributed Commit Protocol" by Wenbing Zhao (Department of Electrical and Computer Engineering, Cleveland State University), https://ieeexplore.ieee.org/document/4351387

In the basic 2PC protocol, a single coordinator is responsible for multiple participants. The coordinator is trusted by the participants and represents a single point of failure. If the coordinator expresses byzantine behaviour, for example by telling one participant to commit and another to abort, the participants will trust the coordinator and therefore do as it says. This would then lead to the participants having different views on what transactions are done, which defeats the purpose of the protocol to reach an agreement.

That is the main problem the byzantine fault tolerant commit protocol solves. It can handle compromised/byzantine coordinators that for example sends different messages to different participants. Another problem the byzantine fault tolerant commit protocol solves is that it's able to continue working even if some coordinators fail or become unavailable. Compare this with the 2PC protocol where the protocol would stop working if the coordinator stops working. Thus it improves the availability of the system compared to 2PC. Since the protocol introduces multiple coordinators, it becomes possible for the participant to send different messages to the coordinators. The authors have thought of this and made sure that the protocol detects and handles this.

### 3.2 Distributed Commit Protocol

In the distributed commit protocol presented in the paper the author address the problem of a byzantine coordinator by replicating the coordinator. One of these replicas is the primary. The resulting system works correctly so long as it has $3f + 1$ coordinator replicas where at most *f* coordinators are byzantine.

One of the participants, which we call initiator, initiates a transaction. This initiator is responsible for propagating the transaction to the other participants that are part of the transaction. Then the participants register with the coordinators. The protocol starts when the initiator has received confirmation from all participants, that they have registered with $2f + 1$ coordinators, and then sends a initiate commit request message to all coordinators.

The coordinators then sends a prepare message to all registered participants. The participants answer whether they can commit. If any of them could not prepare successfully, an abort will take place, otherwise the protocol proceeds.

The coordinators continue by creating an instance of the *Byzantine Agreement Algorithm*, where an agreement is attempted on both which participants are taking part in the transaction as well as the votes of the participants. This is described in more detail in the "Byzantine Agreement Algorithm" section.

After reaching an agreement, coordinator replicas send the agreement outcome to participants, which will only commit the transaction once $f + 1$ similar outcomes are received, to ensure that they reject the answer of byzantine coordinators. Since the protocol allows up to $f$ byzantine coordinators, when

$f + 1$ messages are received by the participants (including the initiator, which is a participant), they can be sure that at least one message is from a non-byzantine coordinator.

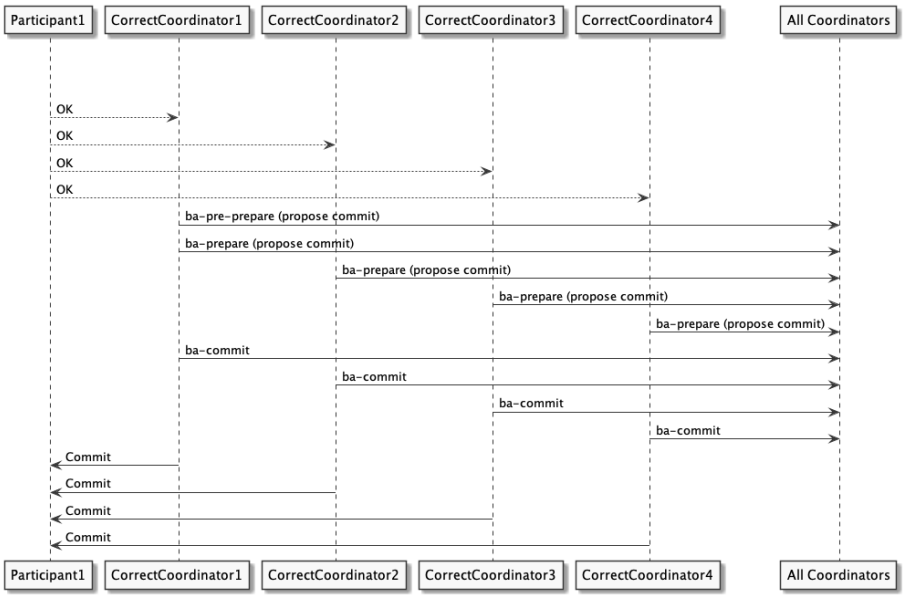


**Figure 1:** An example of the voting part of the BFTDCP protocol.

### 3.3 Byzantine Agreement Algorithm

Wenbing Zhao's algorithm is based on the BFT algorithm by Castro and Liskov. While the aim of the BFT algorithm is designed to make an agreement on the ordering of the requests received, the Byzantine Agreement Algorithm's objective is to agree on the outcome of a transaction. Byzantine Agreement Algorithm has three phases:

- **Ba-pre-prepare phase**: In this phase the primary sends a *ba-pre-prepare* message to all other replicas. The *ba-pre-prepare* message contains the following information: view id, transaction id, transaction outcome and decision certificate. The decision certificate is a collection of records of each participant's registration and vote for every transaction. A new view is requested and created if the *ba-pre-prepare* message fails any verification (signed by the primary, coherent transaction and view and has not accepted a *ba-pre-prepare* in this view-transaction).
- **Ba-prepare phase**: Once a replica has ba-pre-prepared it multicasts a *ba-prepare* message to all other replicas. The *ba-prepare* message contains the view id, transaction id, the digest of the decision certificate, transaction outcome and replica id $i$. The message is accepted if it is correctly signed by replica $i$, the receiving replica's view and transaction id match message view and transaction's id, message's transaction outcome matches receiving replica transaction

outcome and decision certificate's digest matches the local decision certificate. When a replica collects $2f$ matching *ba-prepare* messages from different replicas it can make a decision for the current transaction and sends a *ba-commit* message to all other replicas.

- **Ba-commit phase**: A *ba-commit* message contains the view and transaction id, decision certificate's digest, transaction outcome and sender replica id. A replica is said to have ba-committed if it receives $2f + 1$ matching *ba-commit* messages from different replicas. The agreed outcome is sent to every participant in the current transaction. *Ba-commit* messages are verified alike *ba-prepare* messages.

# 4 Design Decisions

As we're implementing a commit protocol which is based on messages, it makes sense to use an actor framewor. As we are restricted to Scala and *Akka* seems to be one of the most-common actor frameworks, we decided to use it and created two typed of actors, coordinators and participants. In the tests we created we initialize a couple of coordinators and participants (depending on the test case) and send a initialization message from one of the participants (the initiator) to the coordinators. After that the protocol starts.

# 5 Implementation Details

## 5.1 Message Signing

In order to avoid interference on the untrusted networ, messares are signed using public key cryptography. The signing is implemented using a master certificate that signs all the individual certificates (of the coordinators and participants).

As of now, the stated message origin is not cross-validated with the certificate yet, which means that a faulty actor may spoof messages.

## 5.2 Shortcomings of our Implementation

In the original paper, view changes were not implemented and we therefore declared their implementation out-of-scope.

Furthermore we did not ran the system in a distributed fashion, the reason is explained in the Discussion chapter. We do have some ideas however:

- Manually starting *Akka Actors* in different JVMs (optional: could be on the same or on different PCs).
- Getting the actors to communicate with each other using *Artery* (serialization of messages, actor discovery).
- Reworking key distribution (hard) or disabling signature checks

In addition to the tests explained below, the simulation of byzantine behaviour could be extended. The current implementation of byzantine behaviour only covers a fraction of the byzantine faulty space. Expanding this could be interesting, but simulating more byzantine behaviours would have a large impact on code complexity. Ultimately, we believe simulating all possible byzantine behaviours is infeasible and that we have to trust the author's proof.

# 6 Evaluation

## 6.1 Functional Requirements

Functional requirements were evaluated using Scala Tests with the *Akka Actor Test Kit*[2]. We covered:

- Basic Committing
- Aborting
- Unilateral aborting
- Byzantine behavior tolerance

Along with the development we have built a set of tests which tested every feature we implemented. This way we ensured that every module works properly.

We have built a total of 17 tests through which coordinators and participants exchange messages and perform the corresponding message verification and decision making processes. These tests ensure the implementation correctness. A different number of transactions, coordinator replicas and participants is used to test the system's resilience to multiple message passing.

The following tests were implemented:

- **Test 1:** Run with 1 coordinator replica and 1 participant, resulting in a commit.
- **Test 2:** Run with 4 coordinator replicas and 1 participant, resulting in a commit.
- **Test 3:** Run with 1 coordinator replica and 4 participants, resulting in a commit.
- **Test 4:** Run with 1 coordinator replica and 1 participant and have the participant unilaterally abort the transaction, resulting in an abort.

[2] `ScalaTestWithActorTestKit`

- **Test 5:** Run with 1 coordinator replica and 4 participants and have one participant unilaterally abort the transaction, resulting in an abort.
- **Test 6:** Run with 4 coordinator replicas and 4 participants and have one participant unilaterally abort the transaction, resulting in an abort.
- **Test 7:** Run with 1 coordinator replica and 1 participant and have the initiator abort the transaction, resulting in an abort.
- **Test 8:** Run with 4 coordinator replicas and 1 participant and have the initiator abort the transaction, resulting in an abort.
- **Test 9:** Run with 1 coordinator replica and 4 participants and have the initiator abort the transaction, resulting in an abort.
- **Test 10:** Run 2 instances of the protocol, both resulting in a commit.
- **Test 11:** Run with 4 coordinator replicas (of which 1 is nonresponsive) and 1 participant, resulting in a commit.
- **Test 12:** Run with 4 coordinator replicas (of which 1 is nonresponsive) and 1 participant and have the initiator abort, resulting in an abort.
- **Test 13:** Run with 4 coordinator replicas (of which 1 non-primary exhibits some byzantine behaviour) and 1 participant, resulting in a commit.
- **Test 14:** Run with 4 coordinator replicas (of which 1 non-primary exhibits some byzantine behaviour) and 1 participant and have the initiator abort, resulting in an abort.
- **Test 15:** Run with 4 coordinator replicas (of which the primary exhibits some byzantine behaviour) and 1 participant, resulting in a commit.
- **Test 16:** Run with 4 coordinator replicas (of which the primary exhibits some byzantine behaviour) and 1 participant and have the initiator abort, resulting in an abort.
- **Test 17:** Run with 1 participant and 1 slow coordinator which will exceed the timeout, resulting in a view change being suggested.

Tests 1 through 14 succeed as expected. Tests 15 and 16 fail, this is expected since a byzantine primary coordinator necessarily leads to a view change, which has not been implemented. Test 17 checks that the need for a view change is detected, not that it is actually performed. Hence it also succeeds as described.

## 6.2 Non-Functional Requirements

The latency was measured both with normal behaving nodes and with a single byzantine non-primary coordinator replica (a byzantine primary coordinator would require the out-of-scope view-changes to be implemented). If a non-primary coordinator replica is byzantine, a small performance reduction could occur since the algorithm might have to depend on other replicas to reach consensus.

The test consisted of starting a new transaction once the previous had committed, until 100 commits had been completed. The average latency of such a test constitutes one sample. 50 such samples were collected for each test configuration.

All tests were performed with 4 coordinators. The tests were carried out on a laptop with an Intel i3-5005U (dual-core operating at a fixed 2.0 GHz) with 8 GB of RAM.

Figure 2 shows the latency measured in these test. The error bars indicate 2 standard deviations, based on the variance between samples of 100 commits, not between the individual commits. The variance between individual commits is expected to be larger.

No performance difference could be discerned. This might be related to the observation that actors would often be running sequentially due to limited parallelism, which limited the benefit of early consensus.

The results show latencies in the same order of magnitude as stated by the author of the paper. Latencies grow linearly with the number of participants as well.

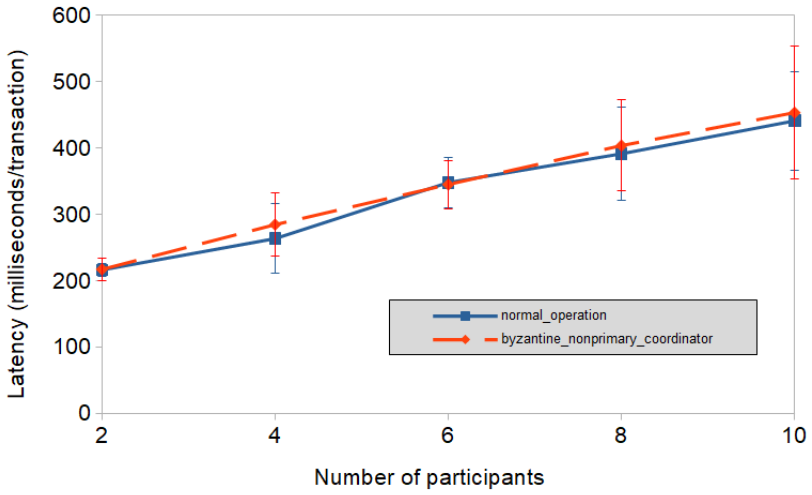


**Figure 2:** Latency comparison between normal operation and a byzantine non-primary coordinator

## 7 Discussion

The main challenge of the project was to fully understand the details of the paper. It was not very clear from the original paper that the system was very heavily depending on the WS-AT protocol, and that it therefore was crucial to understand it before understanding the byzantine fault tolerant distributed commit protocol. The coronavirus situation also made it necessary for three of us (exchange students) to return to our home countries urgently, that led us to loose some time in the last few weeks of the project, which we could not fully recover in the extra week we got, since the new courses had started then.

One of our team members, Miguel Lucas, was responsible for testing the system in a distributed fashion, but due to the coronavirus situation he had to return to his country and finish his studies at his home university. For this reason, he could not work on the project anymore so the system could not be tested in a distributed fashion.

## 8 Conclusion

To sum up, we managed to implement the protocol to a similar extent as the paper. The ability to tolerate some byzantine behaviour of a non-primary coordinator replica has been demonstrated. It has also been shown that the presence of this byzantine coordinator replica does not noticably affect the performance when tested in operation on a single machine.