COQ-BFT

Formal proof of Byzantine Fault Tolerance
for distributed replication systems

Stefan Prisca - 201703265

Introduction

The aim of this project is to give a formal proof that a replication system can achieve byzantine fault tolerance under the condition that the primary replica is not faulty, and that all replicas are connected with one another. The algorithm is based on the work of Miguel et.al. [1]. The report is structured as follows: first the preliminary concepts are introduced, then the project objective is formalized. After that, a short analysis of the problem is presented, and the system model is shown together with informal proofs for the main Theorem and additional Lemmas required. A short experimental section is included after, presenting a few tests done with QuickChick to illustrate that the properties hold when executing the algorithm. The report ends with a conclusion and analysis of the implemented system, and the lessons learned during this project.

Replication systems

Replication systems are distributed computing systems which ensure consistency by sharing and storing information around the system. Because of out-of-order message delivery in the network, such systems need a consensus algorithm to decide on the order in which to store the information.
Many algorithms are based on a primary-backup model. In such a model, one of the replicas (the primary) decides on the order in which to commit transactions, and all other replicas respect this order. A very famous example is the RAFT algorithm.

Byzantine fault tolerance (BFT)

Byzantine behavior refers to cases when a component of the system is neither functioning correctly nor offline. It participates in the system, but it’s behavior is inconsistent. This behavior can cause a lot of issues in replication systems, as faulty replicas can affect the order in which transactions are committed in the other replicas, leading to inconsistencies in the system.

Related work

The project is based on the "Practical Byzantine fault tolerance and proactive recovery." publication by Miguel et.al. [1].
Miguel and Barbara present an implementation of byzantine fault tolerant systems, which allows the system to resist BF as long as at most $(n - 1)/3$ out of a total of n replicas are faulty. Their work is based on previous replication algorithms (e.g. Paxos by Lamport), and extends those with BFT.

Replication is done using a primary backup mechanism (primary node) and quorums (groups of nodes) to order the incoming requests. A node is selected as primary node, and decides the order in which to execute requests.
Replicating a request is then a three stage process (Pre-Prepare, Prepare, Commit). In order to go from one stage to the other, nodes must obtain approval from a quorum (group of nodes).
In the end, a request can only be committed if there is a quorum which agrees on the ordering of all requests committed so far and on the sequence number assigned to this request.
Replicas operate in the context of a view. The primary decides the order of requests within the view. When the primary becomes faulty, the other replicas change the view in order to select a new primary.
The authors claim, and informally prove that the view change algorithm ensures liveness and safety (keep exact transaction record between changing views) of the system.

Project Description

This project aims to take the system described by Miguel et.al. and formalize part of it in the Coq programming language. The focus is on the first part of the algorithm, where it is assumed the primary replica is not faulty. In this scenario, it can be said that the system is in a stable state after processing the request if all the other replicas have agreed to assign the same sequence number. In this case, the ledgers will be consistent.

Project Objectives:

- Model a distributed replication system in coq
- Implement part of the suggested BFT algorithm, proving that the system is fault tolerant under the axiom that the primary replica is not faulty.

Problem analysis

The condition for a system to have tolerance to BF is that, after processing a request all the non faulty replicas in the system have agreed to assign the same sequence number. In a primary-backup replication system, this sequence number corresponds to the one which the primary replica assigned. This can be stated as the following theorem:

[Theorem BFT]

For a given request R, if the primary replica P assigns sequence number N, after replicating the request in the system all non faulty replicas assign sequence number N for R.

Miguel et.al. argue that their system achieves Byzantine Fault Tolerance under the condition that the majority of the replicas are non faulty. The system presented consists of 3*F+1 total replicas (F = number of faulty replicas). The condition under which a replica accept a given request is that it gathered 2*F+1 quorum votes from all the other replicas in the system. This means that 2*F+1 of the other replicas have agreed to assign the same sequence number to the request.
Based on this, the Byzantine Fault Tolerance theorem holds if the quorum certified condition above holds. Knowing this, and knowing that the system has 2*F+1 non-faulty replicas (from above), the Quorum Certified Property can be stated as:

[Property QCertified]

A sequence number N is Quorum Certified for request R if at least all the other non-faulty replicas have assigned N to R.

Having the BFT Theorem and the QCertified property, it remains to prove that at the end of processing the request, the QCertified property holds for all replicas in the system, therefore the whole system will meet the condition of the BFT Theorem.

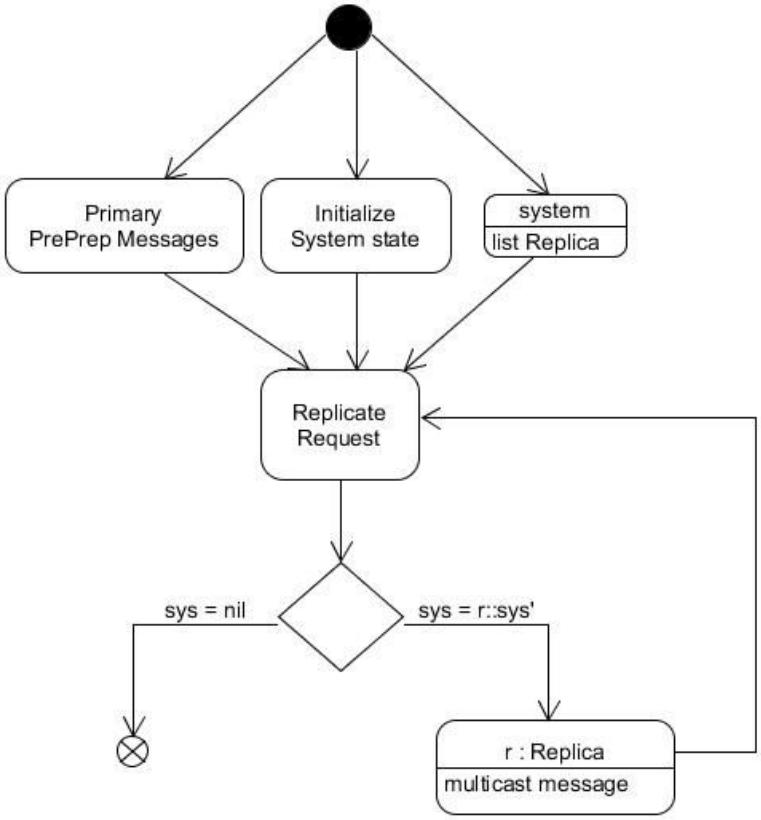
System Design

To design the system in a functional manner, the replicas are treated as functions which update the system state by multicasting their messages. The system therefore is a simple list of such functions. The system state is then represented as a matrix where each row represents a replica, and the values on that row correspond to the messages the replica received.
Note that the primary is not part of this matrix. Having it separately made it simpler to state the Axiom that the primary is not faulty. For example, the state of a system with 4 replicas looks like this:

	R1 messages	**R2 messages**	**R3 messages**	**R4 messages**
R1	1	2	3	4
R2	1	2	3	4
R3	1	2	3	4
R4	1	2	3	4

The above table is to illustrate how the state of the system looks like in the case R1 appended value 1 to the message lists of all the other replicas, R2 appended 2 and so on. Appending a value to the matrix is the equivalent of multicasting that value in a distributed system.

The flow of processing a request is presented below:



The main function is ReplicateRequest. This function takes in a system, the initial system state and the prePrep messages. The initial system state corresponds to an empty matrix of size *N*N* where N is the number of replicas in the system (except the primary). The prePrep messages correspond to the messages sent by the primary function.
The ReplicatRequest function iterates over the system and calls each Replica function on the system state. The result of processing a request is a matrix (State) in which all replicas multicasted their value. The values which replicas multicast depend on whether the replica is faulty or not:

- Faulty replicas: These always multicast a faulty value.
- Non Faulty Replicas (NFR) : these take the value sent to them by the primary and multicast that.

In order to make it simple to argue about the system, the correct value is assumed to be known and is stored as a constant: *nfrValue*. The faulty value is also stored as *faultyValue*. Having these constants makes it possible to state the *PrimaryNonFaulty* axiom:

[Axiom PrimaryNonFaulty]

Forall systems, all the prePrep messages are equal to the nfrValue.

The *nfrValue* is also used to check whether or not a replica has gathered the right number of votes.

Theorems and proofs

The main Theorem [Th.BFT] is stated in coq by saying there is a ledger agreement on the state produced after replicating a request:

```
Theorem BFT : forall (sys:System),
      LedgerAgreement (ReplicateRequest sys (prePrep sys) (initState sys)).
```

The *LedgerAgreement* property holds under the condition that the *quorumCertified* [Prop.QCertified] property holds for all replicas in the state:

```
Definition quorumCertified (nfrSS : State) (r: (Replica * list nat)) :=
      match r with
      | (_, msgs) => countNFR_msgs msgs >= countNFR_state nfrSS
      end.

Definition LedgerAgreement (st : State) := Forall (quorumCertified st) st.
```

Where `countNFR_msgs` counts the number of *nfrValue*s in the list of messages a replica recived, and `countNFR_state` counts the number of non faulty replicas in the final state.

Core proof

Since the full proof is long, and has a lot of small lemmas arguing about correctness in different cases, in the following pages a explanation of the core proof concepts is presented.
The idea behind proving BFT [Th.BFT] is to show that each replica has the number of correct messages equal to the number of NFRs in the system. To do this, one needs to show that the number of NFRs in the system corresponds to the number of NFR rows in the final state. Then show that each of the NFRs increment the number of NFR messages by one. The final part is showing that, given two states *st* and *base*, incrementing the number of messages in *st* compared to *base* by the number of NFRs in the state *st* means that *st* is quorum certified. These three steps are described in more details bellow, leading to the final theorem.

[Proof Count NFR state - system]

The aim here is to show that the number of NFRs in the system corresponds to the number of NFR rows in the final state. Informally, this is proved as:

[Th. countNFR_state_system] For any sys:System,
countNFR_system sys = countNFR_state(ReplicateRequest sys (prePrep sys) (initState sys))
[Proof.]

- By definition of ReplicateRequest, it holds that for any sys, msgs, st:
  `countNFR_state (ReplicateRequest sys msgs st) = countNFR_state st.`
  (Proof simple, by induction on the system).
  So the proof now becomes:
  countNFR_system sys = `countNFR_state` (initState sys)
- By definition of initState it holds that
  countNFR_system sys = `countNFR_state` (initState sys)
  (Proof simple, by induction on the system).

[Proof ReplicateReq - NFR Inc]

The goal is to prove that replicating a request increments the number of NFR messages by the number of NFR replicas in the system.
The proof is made under the assumption that the messages are an extension of the original messages in the system. More specifically:

```
msgs = prePrep sys \/ exists sys', msgs = prePrep (app sys' sys).
```

This allows the use of Axiom PrimaryNonFaulty inductively on the system:

```
Axiom PrimaryNonFaulty : forall (sys:System) (msgs : total_map nat),
      prePrepExt msgs sys -> forall (id : ReplicaId), msgs id = nfrValue.
```

The property can be used because ReplicateRequest iterates over the system only, passing the original messages. An informal proof is presented below, showing the main steps:

[Th. procReq_nfrInc] for any (sys :System) (st : State) (msgs : total_map nat),

```
prePrepExt msgs sys ->
nfrMsgsIncrementedBy (ReplicateRequest sys msgs st) st (countNFR_system sys).
```

[Proof.] By induction on the system

- *[sys = nil]* in this case, it needs to be shown that for any st:State,
  `nfrMsgsIncrementedBy st st 0.`
  This follows directly from the definition of *nfrMsgsIncrementedBy* and induction on the state.
- *[sys = r::sys']* Knowing that
  `<nfrMsgsIncrementedBy(ReplicateRequest sys msgs st) st (countNFR_system sys)>`
  Prove that it holds for <*sys = r::sys'*>
  In this case it needs to be show the effect of replica R on the state by arguing about each type of replica.
  For state < *st* > and < *n = (countNFR_system sys)* >:
  - *[r = FaultyReplica]*: From the definition of FaultyReplica, it needs to be shown that knowing < `nfrMsgsIncrementedBy st base n` > it follows that < `nfrMsgsIncrementedBy (multicast st faultyValue) base n.` >. The first follows from the induction hypothesis and the second can be proved by induction on the state.

- *[r = NFReplica]*: From the definition of NFReplica, it needs to be shown that knowing <`nfrMsgsIncrementedBy st base n`> it follows that <`nfrMsgsIncrementedBy (multicast st nfrValue) base ( S n )`>
  The first comes from the induction hypothesis. The second comes from the <`PrimaryNonFaulty`> axiom.

[Proof Quorum certified state]

The goal here is to prove that if a state has required number of NFR messages than it is quorum certified and there is a Ledger Agreement for it. This is done by comparing the number of messages in the state to a base. In order to simplify the argument, the proof is made under the condition that the number of NFR messages in the base is equal to 0. This holds for the initial state of the system, which is the base for the final proof. Informally, it can be said that:

[Th. quorumCertifiedState] For any st, base : State,

```
Forall (msgsEq 0) base
      -> nfrMsgsIncrementedBy st base (countNFR_state st)
      -> LedgerAgreement st.
```

[Proof.] From the definition of Ledger Agreement, it hold if < `Forall (msgsGte n) st` >. This is proved by induction on <`st`> *(Lemma nfrIncN_nfrMsgs)*:

- *[st = nil]* follows directly from the definition of Forall.
- *[st = (r, msgs)::st']*: It must be proved that < `Forall (msgsGte n) ((r, msgs)::st')` >. From the definition of Forall, this holds if < `(msgsGte n) (r, msgs)` > and < `Forall (msgsGte n) st'` >:
  - < `(msgsGte n) (r, msgs)` > follows from the Hypothesis < `nfrMsgsIncrementedBy st base (countNFR_state (r, msgs)::st')` >
  - < `Forall (msgsGte n) st'` > follows from the Induction Hypothesis, and simple proofs that the other hypotheses for the induction also hold.

[Proof BFT]

[Th.BFT] For a given request R, if the primary replica P assigns sequence number N, after replicating the request in the system all non faulty replicas assign sequence number N for R.

Having the three parts, it is now possible to prove the main hypothesis.

[Th. BFT] For any sys:System,
`LedgerAgreement (ReplicateRequest sys (prePrep sys) (initState sys))`

[Proof] From *[Th. quorumCertifiedState]* it holds that
`< LedgerAgreement (ReplicateRequest sys (prePrep sys) (initState sys)) >` if:

- `<Forall (msgsEq 0) (initState sys)>` : This follows directly from the initState and destructing the system.
- Considering `<stF = (ReplicateRequest sys (prePrep sys) (initState sys)) >`, `<nfrMsgsIncrementedBy stF (initState sys)(countNFR_state st)>`.
  This follows from *[Th. countNFR_state_system]* and *[Th. procReq_nfrInc]*.

Testing - QuickChick

In order to verify that the system functions properly and that the LedgerAgreement property holds at the end of processing requests, testing has been done with QuickChick. The results are as expected.
The test for correctness under the assumption that the primary is not faulty has been done using the following Definition:

```
Definition LedgerAgreementReplicateReqState (F : nat) (sys:System) :=
  CorrectSystem F sys ==>
       let st := ReplicateRequest sys (prePrep sys) (initState sys) in
       (LedgerAgreement st)?.
```

All QuickChick tests have passed as it was expected.

Testing for failure when the primary replica is faulty has been done using the following:

```
Definition ReplicateFaultyPrimary :=
       let size := 10 in
       let sys := (newSysGen size) in
       let vals := (vectorOf size (choose(0,100))) in
       liftM3 ReplicateRequest sys (liftM2 newPrePrepL sys vals)
                                        (liftM initState sys).
```

The Ledger Agreement property was verified on this, and as expected failed after 1 test:

```
QuickChick (forAll ReplicateFaultyPrimary (fun st => (LedgerAgreement st)?)).
```

Discussion

It took awhile to get to the point where I could prove anything related to the actual theorem and the goal of the project. At the beginning I implemented the system and then started on the proof. But I could not connect the proof to anything because the data structures and the logic were too complex. (i.e. using too many maps, and pairs and structured data like you'd use in any other language).

I then had to start over with a much simpler system where replicas would just append values to a list. I managed to give an inductive proof that the replicas put a certain value after ProcessRequest, but the system was to simple and I could not argue about the number of messages each replica received from the others. But this was a step forward because I managed to argue about the replica behaviour between faulty and non-faulty replicas. This argument was possible only after I had a function which matched the replica to a NFR or FaultyReplica (the GetReplicaFun).

Then I started the matrix implementation but again it got too complex. I represented the system as a function of Replicas -> State -> Messages -> System. Until I split it up and had all of them separate I could not prove anything, since I was also recursing on the state when going through the replica functions. After I split it up and flattened some of the other functions to have a more simple system I managed to give an inductive proof of LedgerAgreement. But this was not good. It did not capture the fact that replicas gather the correct number of good messages. All I proved was that if the system cannot be empty, and the previous system had LedgerAgreement so did the one with another replica. But this was not strong enough.

Already having an algorithm on which I could proof things, I then started the proof for counting the replicas in the system and relating them to the number of messages. This is the actual proof from the paper also. There were still some issues with the system which I had to fix in order to obtain the final argument. The biggest was that I stored the messages from a replica as being an output of that replica and not directly as an input of the others. This means that when counting at the end I had to go and find messages by ids in the output lists. Only after using the current multicast function and adding the messages directly in all the other replicas, I could argue about the message counts.

As a conclusion, coq is very different from any other language. It really forces you to simplify everything as much as possible. Until I had small functions which did one thing each, and flat data structures, I could not connect anything to the proofs. Modeling is very different in Coq. Also, I found that it is best to implement the system from both ends. Going just bottom-top or top-bottom won't work. Not only once have I found myself arguing about a sub-part of the system which was not needed actually, or starting a proof for the main theorem which could not be connected to any of the other parts.

Bibliography

1. Castro, Miguel, and Barbara Liskov. "Practical Byzantine fault tolerance and proactive recovery." ACM Transactions on Computer Systems (TOCS) 20.4 (2002): 398-461.

Resources

Source code: https://github.com/stefanprisca/coq-bft