YaleNUSCollege

Modelling and Testing Composite Byzantine-Fault Tolerant Consensus Protocols

Daniel Lok

Capstone Final Report for BSc (Honours) in Mathematical, Computational, and Statistical Sciences

Supervised by: Dr. Ilya Sergey

AY 2018/2019

Acknowledgements

This capstone would not have been possible without the support of numerous friends and faculty.

We thank Dr. Ilya Sergey for his invaluable advice on every aspect of this project, as well as his eagerness in teaching and recommending additional sources of information. We would also like to thank Dr. Olivier Danvy for his course *Functional Programming and Proving*, which provided a rich foundation on which much of this capstone was built.

Additionally, we would like to thank the residents and ex-residents of Elm College's 17th floor for emotional support, and in particular, Yong Kai Yi and Cephas Tan for their helpful comments and advice on the direction, structure, and wording of this manuscript.

YALE-NUS COLLEGE

Abstract

B.Sc (Hons)

Modelling and Testing
Composite Byzantine-Fault Tolerant
Consensus Protocols

by Daniel LOK

Implementing distributed systems is a notoriously difficult task. Programmers that attempt to build executable versions of published consensus protocols often run into numerous bugs along the way, in part due to the often informal nature of the publications.

This project provides a framework to conveniently implement and test models of distributed consensus protocols, with the intention of assisting programmers in formulating "correct" system sematics prior to actual implementation. This paper describes the internal workings of the framework, its API, and how to use it to model complex phenomena such as Byzantine faults, modular protocol composition, and asynchrony.

Keywords: Distributed Systems, Consensus Protocols, Byzantine-Fault Tolerance, Simulating, Modelling, Testing

Contents

Chapter 1

Introduction & Motivation

Distributed systems are ubiquitous in the modern world. They are the key enabler of scalability for any internet-based service, and unsurprisingly, they are notoriously complex [22]. A fundamental problem with distributed systems is that their operating environment is inherently error-prone. Even if the computing units run completely bug free software (often not the case), the computing units themselves could be subject to power failures, hardware faults, or even control by an adversary.

These problems are generally expected to be solved by a fault-tolerant consensus protocol, but unfortunately, as noted by Rahli et. al., such protocols are mostly only published in pseudocode or otherwise non-executable formats [18]. As a result, individuals or organizations that wish to use these protocols are left to formulate their own implementations. Given the complexity of the algorithms, this can often lead to bugs.

Unfortunately, modelling and testing distributed protocols is not a convenient task. Current methods include using formal methods to produce verifiably correct programs (e.g. Disel [21], Velisarios [18], TLA+ [12]), or employing domain-specific languages to build and test executable models of the protocol (e.g. P [8], ModP [6]). While these techniques have been used

to a good degree of success in some companies [8][16], they have not seen wide-spread industry adoption in general.

This project aims to provide a more accessible framework for modelling and simulating consensus protocols, with the intention of testing their safety properties. The framework allows for the simulation of complex phenomena such as Byzantine faults, modular protocol composition, and asynchrony. Our intention is to provide a tool to help programmers reduce bugs in the early stages of protocol implemention, by making it easier to formulate and test node structures and communication semantics.

Additionally, this framework could be used to inform formal models. As mentioned above, many protocols are only described informally, and figuring out how to formalize these statements is a crucial first step in the verification process. Our work aims to ease the proof effort in two ways:

1. **Reducing conceptual load.** By providing an existing conceptual framework, programmers only need to think of how nodes should be structured, and how they should respond to messages. This would reduce the time taken to go from paper to prototype.
2. **Reducing implementation time.** Since the framework is written in OCaml, and the user-specified functions have pre-defined type annotations, translating the protocol to Coq for theorem-proving should be relatively convenient.

To demonstrate the simulator's capabilities, we provide an example implementation of a composite Byzantine-fault tolerant protocol: Sharded Byzantine Atomic Commit (SBAC) from a recent paper published by Chainspace [1]. The code can be found on GitHub[1].

[1]https://github.com/daniellok/protocol-simulator

In summary, we present the following contributions:

1. A framework for simulating consensus protocols, which is expressive enough to model (Chapter 3):
    - Byzantine faults
    - Asynchrony
    - Composite protocols
2. A method for testing **protocol invariants** (i.e. predicates which are true at every stage of a protocol's execution) using the framework (Section 4.3).
3. A sample implementation of the SBAC protocol (simplified) in the framework, which tests the "SBAC Theorem 2" safety property from the original Chainspace paper [1] (Chapter 5).

Chapter 2

Background Information

This chapter assumes that the reader has little to no knowledge of distributed systems and consensus protocols. We provide a brief introduction to some relevant concepts, and describe the internal workings of some popular protocols. In-depth knowledge of these protocols is not strictly required to understand the structure of our simulation framework, but we do reference them in some of our later examples.

2.1 Introduction to Distributed Systems

As businesses grow larger and larger, their computational needs increase in tandem—they require more processing power to handle requests to their services, and more storage space for their data. To help deal with these needs, one can choose to upgrade to a more powerful machine (vertical scaling), or to purchase more machines to distribute the load (horizontal scaling) [15]. Since there is a limit on how much processing power a single machine can have, as demand grows arbitrarily large, horizontal scaling is often the only realistic choice.

A system that uses multiple networked processing units in this manner is known as a **distributed system** [22], and ensuring that the system is reliable is a non-trivial task. Realistic networks can be unreliable, so messages between individual components of the system may be lost, delayed, or corrupted. On top of this, the computers themselves may be subject to failure in the form of crashes (benign failures), or control by an adversary (Byzantine failures) [23].

Due to the various problems that can arise, any implementation of a distributed system must include some sort of **consensus mechanism** [3]. This mechanism is expected to guarantee that the internal state of each component in the system is consistent, or at least that it will *eventually* be consistent, even in the face of failures.

As an example of why consensus is important, one can imagine a distributed database composed of three machines: one acting as a "leader" from which clients can read and write, and the other two acting as "followers", from which clients can only read. When a client writes to the leader, the leader propagates the result to the followers, which update their local copy of the database. Now consider the following sequence of events:

1. Suppose all three machines track a value n, initially equal to 1.
2. A client writes $n = 2$ to the leader.
3. Before the leader can propagate the result, one follower crashes.
4. The leader propagates $n = 2$ to remaining follower.

When the crashed follower comes back online, it will still have the value $n = 1$, while the rest of the machines will have $n = 2$. Without a consensus mechanism to prevent this sort of inconsistency from occuring, clients may

receive different values when they read n from the system. This is an example of a *crash-recovery* failure, and several popular protocols (e.g. Paxos [11], Raft [17]) exist to address it. However, it is still considered relatively benign, since components of the system fail in a fixed way.

Such a failure model is sufficient for closed networks (e.g. internal servers in a company), but consider if the distributed system is meant to allow public participation. The Bitcoin network is a popular example of such a system, since anyone can join and (attempt to) contribute blocks to the Bitcoin ledger. In this case, failures may take an arbitrary form, since a malicious participant may attempt to subvert the system in any number of ways.

2.2 Byzantine Fault Tolerance

Failures in which nodes exhibit arbitrary behavior are known as **Byzantine faults**. The name stems from the "Byzantine Generals Problem", an allegory from a paper written by Lamport et. al. in 1982 [14]. The paper describes a hypothetical situation in which a commanding general must disseminate instructions to multiple lieutenant generals. However, there is a twist—some of the generals (of either type) could be traitors. The end goal, as stated in the original paper, is as follows:

> **Byzantine Generals Problem.** A commanding general must send an order to his $n - 1$ lieutenant generals such that:
> **IC1.** All loyal lieutenants obey the same order.
> **IC2.** If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.

The fact that the commander might not be loyal is what makes the Byzantine Generals Problem difficult. If the commander were always loyal, then IC1 and IC2 could be satisfied trivially—a loyal commander would send the same order to all lieutenants, so the loyal lieutenants only need to follow the order. However, since the commander is not guaranteed to be loyal, they could send different orders to each lieutenant. In such a situation, the loyal lieutenants must find some way to collectively achieve IC1. For example, they might communicate amongst themselves and devise a fallback plan if they received different orders (this is tricky, since traitorous lieutenants might relay incorrect information as in Fig. 2.1).

One important result from the paper is a proof that the Byzantine Generals Problem is unsolvable if traitors make up at least one third of the total population. Thus, Byzantine consensus algorithms in general have the assumption that the total number of nodes in the system must be at least $3f + 1$, where f is the maximum number of faulty nodes the system can tolerate. Such algorithms usually achieve consensus by performing a majority vote after having all nodes compute their outputs independently.

Ultimately, the problem is meant to be an analogy for achieving consensus in an adversarial environment. The commanding general represents an information source, while the lieutenant generals represent processing units which rely on said information. The notion of being a "traitor" corresponds to nodes being faulty in an unpredictable manner.

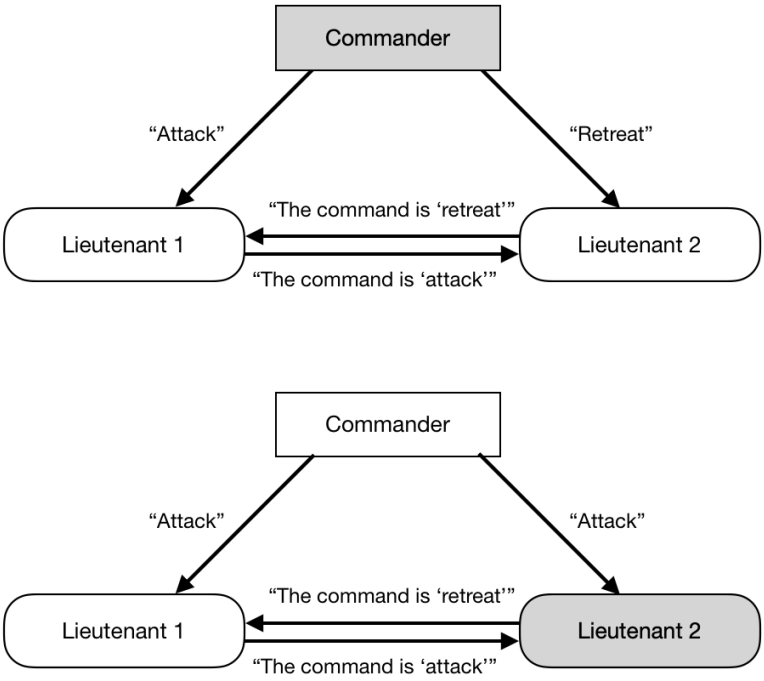

FIGURE 2.1: Example of an unsolvable configuration from the original paper [14]. Both situations look the same from the view of Lieutenant 1.

2.3 Relevant Consensus Protocols

As mentioned in the Introduction section, we provide a sample implementation of the Sharded Byzantine Atomic Commit (SBAC) protocol. This protocol is interesting because it is a composition of two others—the Practical Byzantine Fault Tolerance (PBFT) and Two-phase Commit (2PC) protocols. In this section, we will briefly describe the semantics of all three, and introduce relevant concepts that will be referenced in the rest of this project.

2.3.1 Practical Byzantine Fault Tolerance (PBFT) [4]

State Machine Replication

PBFT is a Byzantine-fault tolerant consensus algorithm for **state machine replication**. From a high level, a **state machine** can be thought of as a construction which has [13]:

1. a set S of user-defined states
2. a set C of user-defined commands
3. a set R of user-defined responses
4. a mapping e from $(C, S) \rightarrow (R, S')$. In other words, a transition function. As Lamport writes, "the relation $e(C, S) = (R, S')$ means that executing the command C with the [state machine] in state S produces the response R and changes the [state machine] to state S'". It is worth noting that this mapping must be deterministic, i.e. given the same C and S, the same R and S' will be generated.

State machine **replication**, as the name implies, is simply the process of making multiple copies (replicas) of an original state machine. This is useful for applications such as the distributed database example in Section 2.1, where we may want to store the same information on multiple computing units.

PBFT Semantics

In PBFT, membership of nodes is fixed, meaning that the protocol does not support arbitrary addition and removal of nodes, and every replica knows the address of every other replica. These replicas move through a series of configurations known as **views**. In a single view, one replica is designated

the **primary** (i.e. the leader), and the rest are **backups** (followers). The primary has a few distinct responsibilities, such as receiving, ordering, and relaying requests from external clients. However, since the algorithm operates under a Byzantine failure model, it is possible that the primary is faulty or malicious. To deal with this situation, PBFT has two modes:

1. Normal-case (when the primary is not faulty)
2. View change (when the primary is suspected to be faulty)

For this project, only the normal-case operation has been implemented, so this section will omit details of the view change algorithm. Hence, for all phases below, we assume that the primary is not faulty. Additionally, the algorithm makes use of cryptographic primitives (e.g. hash functions, authentication, signatures) to ensure message integrity and unforgeability. However, for simplicity, we will omit details here as well.

During normal-case operation, replicas go through five stages of message-passing in order to process client requests: request delivery, pre-prepare, prepare, commit, and finally, reply delivery. In the following sections, we will briefly explain each phase.

Request Delivery

A request to a system running the PBFT protocol takes the form `Request`(o, t, c), where o is the operation to perform, t is the timestamp at which the request was sent, and c is the client's identifier or address.

When a client wants to make a request to the system, they send a message of the above form to the primary. If they do not know which replica is

the primary, or if the primary has recently changed, then they can alternatively send the message to multiple replicas. Non-faulty replicas will then relay the request to the primary.

Pre-prepare

Once the primary has received the request m, the request is assigned with a sequence number n. Sequence numbers are used to enforce a total order on the execution of operations. This is important because the network is assumed to be asynchronous, so different replicas may receive requests in different orders.

After this is done, the primary will send a "pre-prepare" message of the form `Pre-prepare`(n, m)[1] to every other replica. The primary also adds this message to its internal log—a list of all messages it has received. This log is used to check some predicates which will be introduced later on.

Prepare

When a replica i receives the pre-prepare message, it begins the next phase by sending a "prepare" message to all other replicas. This message has the form `Prepare`(n, m, i)[1], where n and m are taken from the pre-prepare. The replica then adds both messages (the pre-prepare and the prepare) to its log.

The predicate `prepared`(n, m, i) is defined to be true when replica i has the following in its log:

1. One `Pre-prepare`(n, m) message.
2. At least $2f$ different `Prepare`$(n, m, \_)$ messages from other backups.

[1] This is a simplification of the original message from the paper, omitting details relating to view change and low-level memory/network optimization (e.g. hashing, message piggybacking).

Commit

Once `prepared`(n, m, i) becomes true, replica i sends a "commit" message to all other replicas. This message takes the form `Commit`(n, m, i)[2] (same variables as in the `Prepare`). Much like the in previous phase, the predicate `committed-local`(n, m, i) is defined to be true when:

1. Replica i's log contains $2f + 1$ `Commit` messages from different replicas.
2. Replica i has executed all previous requests. More formally, all requests from pre-prepares with sequence numbers $n' < n$.

Once `committed-local`(n, m, i) becomes true, replica i executes the operation o contained in m. Note that because of Condition 2 above, requests can be committed out of order, since they will only be executed (by non-faulty replicas) in the order dictated by the sequence numbers.

Reply Delivery

After the non-faulty replicas have finished executing the operation, they individually send some reply to the client, c. This reply is of the form `Reply`(t, c, i, r)[2], where t is the timestamp from m, c is the client's address or identifier, i is the index of the replica, and r is the result of the operation.

The client must wait for $f + 1$ replies from different replicas with the same t and r before accepting the result. Since at most f replicas are faulty, the client can be sure that the reply is valid.

[2] See footnote on previous page

2.3.2 Two-phase Commit (2PC) [2]

Atomic Commitment

2PC is an **atomic commitment** protocol (ACP). This class of protocols allows a set of nodes to either unanimously commit (execute) or abort a transaction. In other words, if *any* node decides that the transaction cannot be committed, then *no* node will execute it. Nodes communicate their decisions by voting—they can either vote `Yes` or `No` when presented with a transaction. ACPs are generally *not* Byzantine fault tolerant, since they do not aim to handle arbitrary failures.

In *Concurrency Control and Recovery in Database Systems* [2], Bernstein et. al. lay down the following properties that are expected of ACPs:

1. All processes that reach a decision reach the same one.
2. A process cannot reverse its decision after it has reached one.
3. The Commit decision can only be reached if all processes voted Yes.
4. If there are no failures and all processes voted Yes, then the decision will be to Commit.
5. Consider any execution containing only failures that the algorithm is designed to tolerate. At any point in this execution, if all existing failures are repaired and no new failures occur for sufficiently long, then all processes will eventually reach a decision.

In order to satisfy Property 5 for benign failures, the usual version of 2PC defines **timeout actions** and a **recovery protocol**. Timeout actions are default procedures that nodes perform if a set amount of time has passed without making progress on a transaction. A recovery protocol is a set of

instructions that enable a node to "catch up" with the decisions it has missed during a crash.

For the purposes of this project, we need only concern ourselves with Properties 1-4. This is because the variant of 2PC used in the SBAC protocol effectively assumes *no* node failures. Thus, we will not describe the timeout actions and recovery protocol below.

2PC Semantics

Like PBFT, the nodes in 2PC are set up in a leader-follower fashion. One node is designated the **coordinator**, and the rest are **participants**. The coordinator is in charge of collating votes from (and relaying the final decision to) all participants. Step-by-step, the algorithm is:

1. For a given transaction, the coordinator requests votes from the participants by sending the `VOTE-REQ` message. This assumes that participants know what the transaction is, which can be achieved practically by including the transaction in the message.
2. Upon receiving the `VOTE-REQ` message, a participant decides internally whether or not it can commit the transaction. If so, it sends `YES` to the coordinator. If not, it sends `NO`. If its vote is `NO`, it no longer needs to listen to messages from the coordinator, since the final decision will be to abort.
3. Once the coordinator receives all votes, it can make a decision. As mentioned above, if all votes are `YES`, then it decides to commit, and sends the `COMMIT` message to all participants. Otherwise, if there is at least one `NO`, it sends `ABORT`. At this point, the coordinator's job is done.

4. If a participant receives the `COMMIT` message, then it executes the transaction and updates its state to reflect its execution. Otherwise, it aborts. At this point, the protocol is complete.

2.3.3 Sharded Byzantine Atomic Commit (SBAC) [1]

Sharding

SBAC is a **sharding** protocol. As opposed to state machine replication, in which all nodes share the same state, each node in a sharded system (hereby referred to as a shard) possesses a disjoint subset of the data. This method of partitioning is useful in applications where the amount of data is large enough to make storage on a single machine inefficient.

When we shard a system, we also need a way to determine which data belongs to which shard. Naturally, we'd prefer that each shard has about an equal amount of data—if the subsets are imbalanced, then the performance gains of sharding would be diminished. Solutions to this problem are generally left as individual design decisions, but most involve taking the cryptographic hash of some part of the data, and using that as a key to determine the shard.

SBAC Architecture

Shards in the SBAC protocol are groups of nodes which maintain a common state through PBFT (see Fig. 2.2). In typical sharding fashion, each of these groups contains a different subset of data. This data is a list of "objects"—the exact structure of which would be implementation-specific. The purpose of these objects is to be consumed and created through user-supplied

"transactions". The general flow of a transaction through the SBAC protocol is as follows:

1. Before sending the transaction to the system, the user computes the expected outputs locally.
2. The transaction T (which contains a list of the necessary input objects, the procedures to be performed on those objects, and the expected list of output objects) is sent to all concerned shards in the SBAC system. Concerned shards are simply shards which are in charge of any objects involved in the transaction.
3. Within each concerned shard, the transaction is validated independently (we will omit details here for simplicity). If it is deemed acceptable by all shards, and if all input objects are in the "active" state, then the input objects will be consumed (i.e. rendered inactive such that no further transactions may use them), and the output objects will be created. If even one shard finds it unacceptable, then the transaction is aborted.
4. Each shard informs the user of the status of the transaction.

Differences from 2PC

As it was implied above, the shards process transactions through a variant of the 2PC protocol. There are a few key differences between SBAC's atomic commit protocol and 2PC:

1. There is no centralized coordinator. The user/client initiates a transaction T by sending a `Prepare`(T) message to all concerned shards. This replaces the `VOTE-REQ` message.

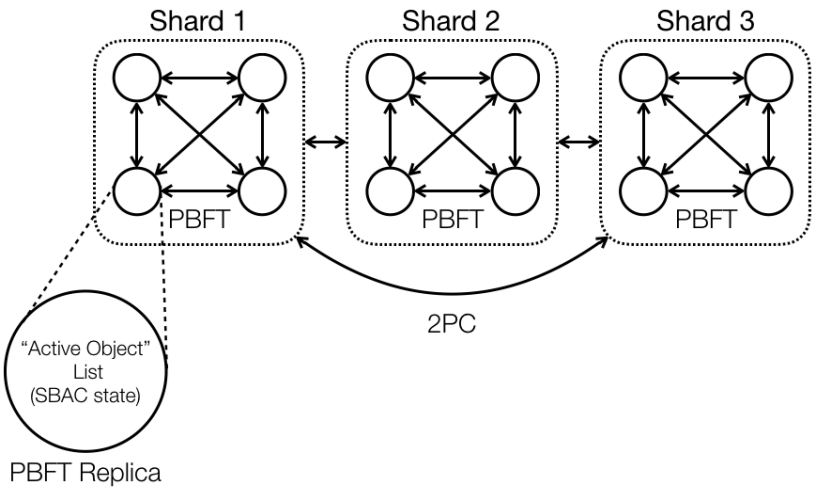

FIGURE 2.2: SBAC system architecture

2. As a consequence of 1., shards send their votes to all other concerned shards through the `Prepared(commit/abort,`T`)` message. This replaces the `YES/NO` messages.
3. Once shards receive all votes from all other concerned shards, they either commit T if all votes were `commit`, or abort if there was at least one `abort`.

Failure Model

Up to this point, we have not defined the failure model for SBAC—if it is to be Byzantine fault tolerant, why would we use a non-Byzantine fault tolerant protocol for inter-shard communication? In the original paper, the authors address this concern by defining two separate fault models:

1. **Honest Shards.** Under the Honest Shards failure model, at most f nodes in each shard can be faulty. Since each shard is of size $3f + 1$, we can effectively consider all shards to be non-faulty.

2. **Dishonest Shards.** Under the Dishonest Shards failure model, more than f nodes can be faulty (i.e. shards may not work properly). In this case, SBAC cannot guarantee correctness. However, the authors describe methods through which faulty nodes can be identified and banned/blacklisted from participating in the protocol.

For simplicity, we will only be considering the Honest Shards threat model in this project. Hence, we omit all details related to identifying and removing faulty shards.

Chapter 3

Simulator Semantics

In this chapter, we provide a detailed rundown of our simulation framework's internals. We describe the high-level architecture, what is required of the user, and how to perform a simulation run.

3.1 Overview

Our simulator's implementation takes the form of a module which is separately defined for each protocol we want to model (since every protocol will have different notions of nodes and messages). In OCaml, this can be achieved cleanly by declaring a module type (see `Sim_type` in Fig. 3.1), and using it to parameterize a module (see `Sim` in Fig. 3.1). This type of parameterized module is also known as a **functor**. The user does not need to modify the functor itself—only the module type.

There are four types, and four functions that a user needs to define in order to begin simulating a protocol. The types are:

1. `_node` : The structure of a node in the network
2. `_request` : Client-to-system request messages (e.g. `Request` in PBFT)
3. `_internal` : Internal node-to-node messages (e.g. votes in 2PC)

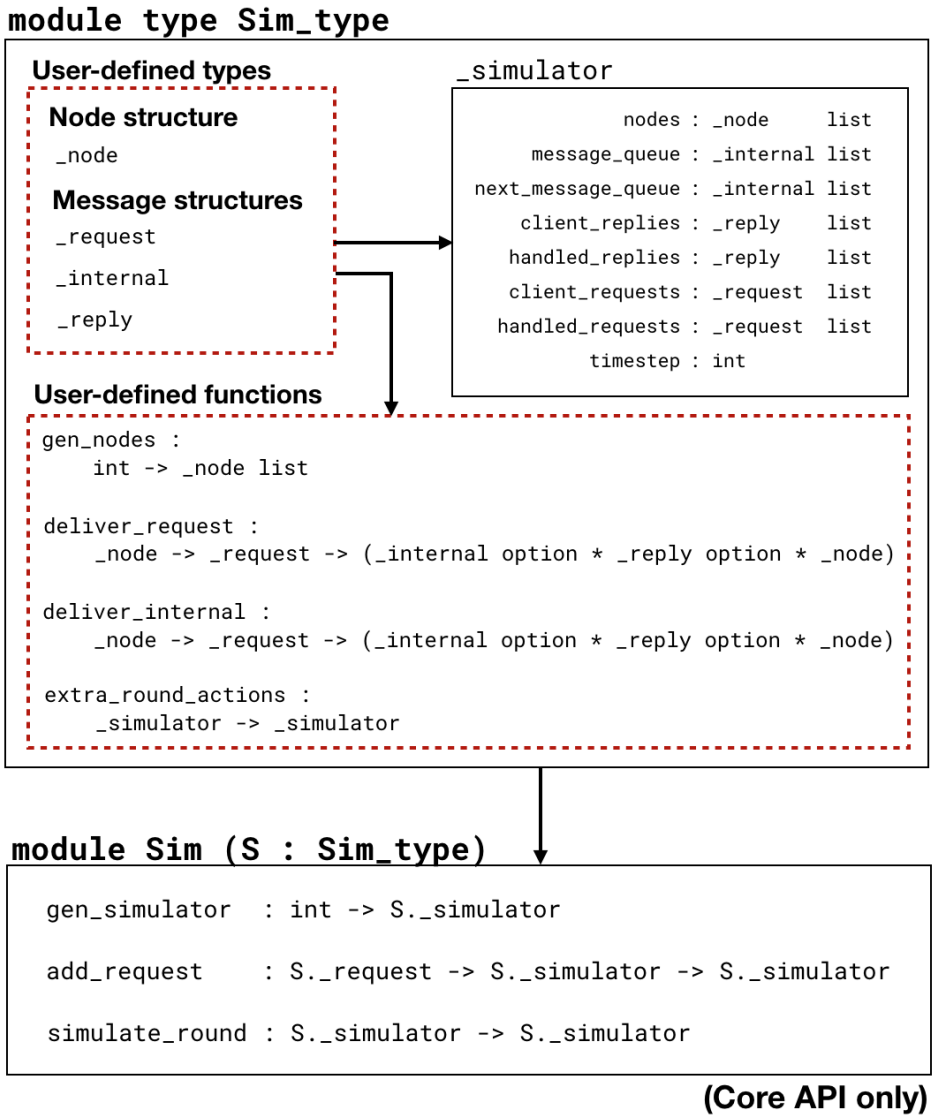

FIGURE 3.1: High-level overview of the framework. User-defined elements are indicated with red dashed lines. The arrows indicate dependency; e.g. the module `Sim` is dependent on the `Sim_type`.

4. `_reply` : System-to-client reply messages (e.g. `Reply` in PBFT)

And the functions (refer to Fig. 3.1 for explicit type annotations):

1. `gen_nodes` : A function to generate a list of nodes in their initial states (i.e. before any interactions have taken place).

2. `deliver_request` : How a node responds to a client request. Practically, this means that the function takes a request and a node, and returns a response (optional), a client reply (optional; only if the protocol has finished its last stage), and the updated node.
3. `deliver_internal` : How a node responds to an internal message—similar to 2.
4. `extra_round_actions` : A function to perform arbitrary actions at the end of a round. We use this function to execute state updates which do not cleanly fit into the message-response framework. For example, one rule in PBFT is that a node is only allowed to execute an operation after all operations with lower sequence numbers have been executed. To capture this behavior, we could use this function to check for, and execute, any pending commits in our nodes. This particular behavior *could* be baked into the `deliver_internal` function as well, but sometimes we might want to manipulate the simulator itself (rather than the nodes), in which case we have no alternative but to use this function.

Once the user defines these types and functions, they can instantiate the `Sim_type` module signature, and use it to define a `Sim` module for their protocol (see Fig. 3.2 for an example).

```
module PBFT_type : (Sim_type
                    with type _node     = replica
                     and type _request  = pbft_request
                     and type _internal = pbft_internal
                     and type _reply    = pbft_reply) =
  struct
  ...
  end;;

module PBFT_simulator = Sim (PBFT_type);;
```

FIGURE 3.2: Example instantiation of `Sim_type` and the `Sim` module for a PBFT simulator (truncated for brevity).

3.2 Simulator Module API

At this point, we have described how the simulator is structured, but not how it works—what exactly does a simulation run look like? In this section, we describe the internal logic of the `Sim` module, and explain the three methods which are relevant for users. Note that in the following sections, the word "simulator" refers to an instance of a `Sim_type._simulator` record. This structure encapsulates the system's state throughout a simulation run.

3.2.1 `simulate_round`

Nodes in a distributed system communicate via a network, but our framework is meant to run locally. Hence, we have chosen to simulate the network via synchronous "rounds" of message passing, much like in Charron-Bost and Schiper's Heard-Of model [5].

At each "round", the simulator delivers a set of messages to all nodes,

approximating a one-to-all multicast. These messages might be client requests, or internal messages generated by nodes in the previous round. For simplicity's sake, if a node is not meant to receive a message, it simply ignores it. Otherwise, the node immediately generates a response according to the simulated protocol's rules (i.e. the user-defined `deliver_request` and `deliver_internal` functions). These responses are collected, and will be delivered in the next round. In the case where the response is meant to be delivered to the client, it is put in a separate list (`client_responses`) where it can be handled appropriately.

This series of actions is performed by the `simulate_round` method contained in the `Sim` module. The argument to this function is a `_simulator` record, which we shall name `sim` (with a lowercase "s" to distinguish it from the module). To simulate one round, the function does a few things:

1. For every request *r* in `sim.client_requests`, and for every node *n* in `sim.nodes`, call the user-defined `deliver_request` function using *r* and *n* as arguments.
2. For each *r* and *n*, the `deliver_request` function will output a triple of:

    i. An optional `_internal` message (a response)
    ii. An optional `_reply` message (an update for the client)
    iii. A mandatory `_node` (the node's updated state after processing *r*)

3. For the outputs above, do:

    i. Append the response (if it exists) to `sim.next_message_queue`
    ii. Append the reply (if it exists) to `sim.client_replies`
    iii. Replace *n* in `sim.nodes` with the updated node

4. Repeat 1-3, but with `sim.message_queue` and `deliver_internal` instead of `sim.client_requests` and `deliver_request`.
5. Once all requests and messages have been delivered, prepare for the next round by setting:

   i. `sim.message_queue = sim.next_message_queue`
   ii. `sim.next_message_queue = []`

6. Call `extra_round_actions` on `sim` to obtain an updated simulator.
7. Increment the `sim.timestep` field, indicating that the round has ended.

3.2.2 `gen_simulator`

Naturally, before we can begin simulating rounds, we need to have an initial value for `sim`. To obtain this initial value, we simply call the `gen_simulator` function with an integer n (the number of desired nodes) as an argument. Internally, it uses the user-defined function `gen_nodes` to produce a list of nodes as an initial value for `sim.nodes`. All other fields in the simulator are initialized as the empty list, and `sim.timestep` begins at 0.

3.2.3 `add_request`

In order to keep the simulated protocol separate from its potential applications, we do not include the external client as part of the model. Instead, the user is free to add requests at any timestep via the `add_request` function. This function takes a `_simulator` and a `_request`, and adds the request to the simulator's `client_requests` field.

By separating the client from the protocol, the user can use `sim` as a "black box"—interacting with it only by adding requests, simulating rounds, and periodically inspecting `sim.client_replies` for status updates.

Chapter 4

Modelling & Testing Complex Phenomena

Now that we have introduced our simulator's internal mechanisms, we can demonstrate how it can be used to model more complex phenomena such as benign/Byzantine faults, composite protocols, and asynchrony. Furthermore, we describe a way to test protocol invariants by performing predicate-checking on randomized simulation runs.

4.1 Fault Tolerance

Fault-tolerance is one of the central concerns of distributed computing, and so any model of a consensus protocol must be able to simulate the notion of failures. In our model, this can be achieved in multiple ways, but perhaps the most convenient method is to add an `is_faulty` field to the node structure (see Fig. 4.1).

```
type replica = {
  id        : int;
  ... (* transaction-specific data types *) ...
  log       : pbft_internal list;
  seq_num   : int;
  last_exec : int;
  primary   : bool;
  is_faulty : bool;
};;
```

FIGURE 4.1: An example structure for a PBFT replica

The value of this field can be checked in the `deliver` functions, and cause the functions to follow different execution paths. For example, simulating a crashed node could look like:

```
let deliver_internal (m : _internal) (n : _node) =
  if n.is_faulty
  then (None, None, n)   (* no responses, no updates    *)
  else (...);;           (* perform appropriate actions *)
```

FIGURE 4.2: Example `deliver_internal` function simulating a crash failure

Byzantine failures can be modelled in a similar fashion, except instead of returning `None`s, the functions could be made to return random or intentionally confusing messages. However, this alone would not be enough freedom for a Byzantine adversary—recall that a key aspect of a Byzantine node is the ability to send different messages to different nodes. Therefore, to model a Byzantine node, we need two things:

1. The ability to send messages to specific nodes (since messages are multicasted to all nodes by default).

2. The ability to send multiple messages per round (since the `deliver` functions only return a single message option).

Fortunately, our framework is extensible enough to allow such behaviors (see Fig. 4.3 for a concrete example). For the concerns above, we could:

1. Declare a message sub-type which includes a list of recipients.
2. Specify `_internal` as a list of the messages declared in 1.

```
type order =
  | Attack
  | Retreat;;

(* message sub-type *)
type message =
  | Command of id list * order   (* sent by Commander   *)
  | Relay   of id list * order;; (* sent by Lieutenants *)

(* to be used as _internal in the simulator *)
type internal = message list;;
```

FIGURE 4.3: Example message structure that could be used in a model of the Byzantine Generals Problem (see Fig. 2.1). The `id list` type would be used to indicate the IDs of the message's intended recipients.

From here, we can modify the `deliver` function much like we did in the benign case. For every message m contained in the `_internal`, if a node n's ID is not in the list of m's intended recipients, we simply move on to the next message without performing any state updates on n.

4.2 Modular Composition

A powerful feature of our framework the ability to nest simulators. This functionality can be used to model composite protocols in a modular manner. For example, to build our model of the SBAC protocol, we first built a standalone PBFT simulator (see Fig. 3.2 for the truncated definition). Then, we included the PBFT simulator type in the specification of an SBAC node:

```
type shard = {
  id          : int;
  log         : internal list;
  pbft_nodes : PBFT_type._simulator;
};;
```

FIGURE 4.4: The structure of an SBAC node (a shard). The `PBFT_type` module type is independently defined.

In the parent `shard` structure, the `pbft_nodes` field represents the collection of PBFT replicas which make up the shard (see Fig. 2.2). Whenever the shard needs to make a decision or execute a transaction, it does so by initiating requests to the PBFT subsystem. Effectively, the `shard` acts as the client to its internal `pbft_nodes`.

Consequently, this means that we need to synchronize the larger SBAC simulator with the internal PBFT simulators—when we simulate a round of message-passing *between* shards, we should also simulate a round *within* the shards. Fortunately, this can be done conveniently in our framework by using the `extra_round_actions` function. At every round, we simply call `PBFT_sim.simulate_round` on the internal PBFT simulators in the shards, and possibly handle any replies that might appear. A sample in OCaml might look like:

```
let advance_shard (sd : shard) : shard =
  let new_nodes = PBFT_sim.simulate_round sd.pbft_nodes in
  ... (* checking for replies in PBFT simulator *) ...
  { sd with pbft_nodes = new_nodes;
            ... (* other updates *) ... }

(* to be used in the Sim_type module definition *)
let extra_round_actions (sim : _simulator) : _simulator =
  let new_shards = List.map advance_shard sim.nodes in
  { sim with nodes = new_shards;
             ... (* other updates *) ... }
```

FIGURE 4.5: Advancing the internal PBFT simulators

Note that this deviates slightly from the real-world implementation—the model treats shards as a separate entity, when actually the `shard.id` and `shard.log` fields would be baked into each individual PBFT replica. However, separating the components as we have done allows us to build and test components of the system independently, which would be particularly important for large, complex protocols.

4.3 Predicate Checking

Ultimately, the goal of modelling and simulating protocols is to verify that they *work*. Therefore, we should be able to define and test notions of what it means to be "correct". Such a notion could come in the form of a statement describing some guarantee of the system. For example, the authors of the Chainspace paper state the following of SBAC (SBAC Theorem 2):

> Under the 'honest shards' threat model, no two conflicting transactions, namely transactions sharing the same input will be committed. [1]

This is an example of a **safety property**—a guarantee that a certain event will never happen (committing two conflicting transactions). In addition to safety properties, protocol designers are also interested in **liveness properties**—guarantees that an event will *eventually* happen (e.g. that some transaction will be committed).

Safety properties can be formulated as predicates on the system's state—for example, "for all committed transactions in `sim.client_replies`, no two transactions share any inputs (no conflicts)". This is a statement whose truth we can programmatically assert at every round of a simulation run. To produce a negative test run, we might manually add two conflicting transactions to the system, and verify that only one of them gets committed after a certain number of rounds.

Unfortunately, liveness is a bit more complicated to model, since temporary violations of a predicate can be tolerated as long as it will hold at some point in the future. Taking the example above, it would be acceptable that no transactions are currently committed, as long as some will be committed in the future. As of now, our framework only supports modelling safety properties, and the sections below will describe how it might be done.

4.3.1 Schedule Generation & Execution

While manually constructing test cases might work for small sanity checks, we would eventually want to scale up and test that the predicates still hold for a series of random conflicting and non-conflicting transactions. In order to do this, we need to build a randomized "schedule"—a list-of-lists of instructions for the simulator to perform:

```
type instruction =
  | Add_transaction of transaction
  | ... (* other instructions *) ...;;

type schedule = (instruction list) list;;
```

FIGURE 4.6: Type definition of a schedule

These instructions can be anything—for example, one could have instructions to add transactions, drop messages, or even to crash/revive nodes. The programmer simply needs to define a `execute_instruction` function, which describes how to modify a `_simulator` given an `instruction`.

Executing a schedule involves the following:

1. Before round i, execute all the instructions (if any) contained in the i-th schedule element. For example, if the instruction is `Add_transaction` t, then before the round starts, we would construct a request to process t, and add it to `sim.client_requests`.
2. Call the `simulate_round` function on the simulator. This brings us to round $i + 1$.
3. For all predicates we want to test, check that they hold.
4. Repeat 1-3 with $i = i + 1$, until all elements in the schedule have been executed.

Using a random transaction/instruction generator, one can generate a schedule of arbitrary length, which would allow for complex execution flows that would be difficult to create manually.

4.4 Asynchrony

Asynchrony is one of the main confounding factors of a distributed protocol. Therefore, having fully synchronous rounds in the simulator may not be desirable, as certain behaviors would never be simulated (e.g. transactions being committed out of order, timeouts). However, asynchrony can be built into the framework in a few ways:

1. **Dropping messages.** In the `extra_round_actions` function, randomly delete some messages from the queue. This could also be done by adding a `Drop_messages` instruction to the schedule.
2. **Delaying messages.** In the `extra_round_actions` function, randomly move a subset of messages from `sim.message_queue` to `sim.next_message_queue`. This could also be done by adding a `Delay_messages` instruction to the schedule.

Of course, moving from a synchronous model to an asynchronous model means that we have to include some notion of time tracking in order for the nodes to execute timeout actions. One convenient method might be to add a "timeout counters" field to the node structure. This field could be defined as a `(int * transaction_id) list`, where the integer represents the number of elapsed rounds since a relevant message was received for the associated transaction.

At every round, this integer would be incremented, or reset to 0 if a relevant message was received. If the counter grows too large (i.e. the node has been waiting for too long), then a timeout action can be executed. Otherwise, if the transaction is committed or aborted, then we can simply remove the tuple from the list.

Chapter 5

SBAC Implementation & Findings

As mentioned in the introduction, we have built a simplified implementation of the SBAC protocol using our framework. In this chapter, we explain how the instantiation was done for the PBFT module (the SBAC module is relatively simple, and is more or less covered through Section 4.2 and Fig. 5.1). We also describe our findings from performing automated random testing of a safety property from the Chainspace paper.

5.1 PBFT Module

Each shard in the SBAC system uses a collection of PBFT nodes in order to maintain its state. In this section, we detail the implementation of the PBFT module, and how it is used to support SBAC's transaction framework.

5.1.1 Simplifications

In this sub-section, we detail the assumptions made in our implementation of PBFT in order to simplify the protocol to a manageable level. These assumptions can be addressed in future improvements to the work.

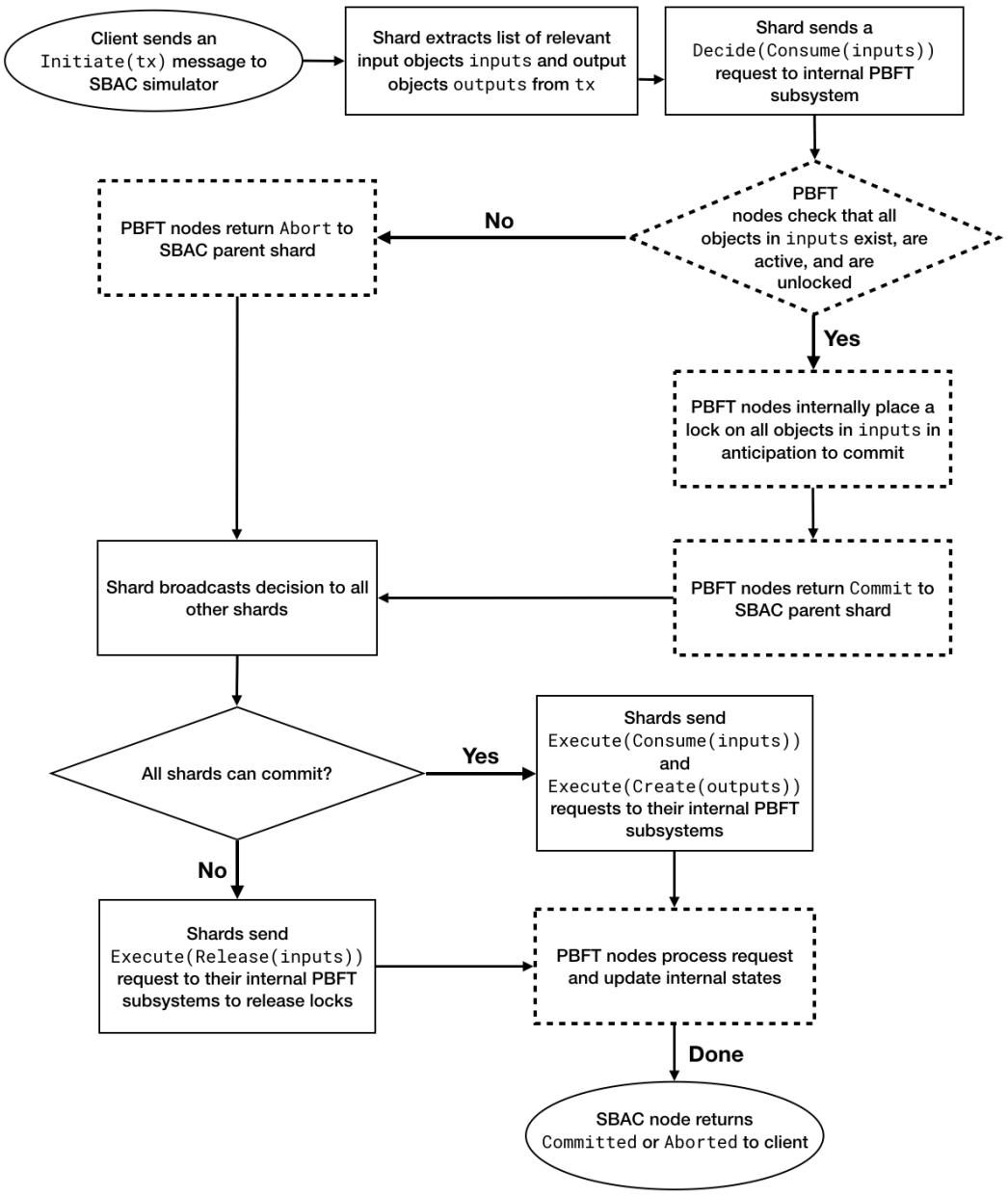

FIGURE 5.1: Flow of an SBAC transaction. Dashed outlines are actions that occur within the PBFT nodes. Solid outlines are actions that occur in the SBAC shards.

1. **Only normal-case operation.** We do not implement PBFT's "view-change" algorithm, which means in particular that leaders cannot be faulty.

2. **Byzantine nodes behave randomly.** We do not implement "smart"

Byzantine behavior in the nodes, so a Byzantine node will simply send out random messages.

5.1.2 Node Structure

Figure 4.1 has a sketch of a PBFT replica's structure, but with the transaction-specific data types omitted. This subsection will fill that gap, and describe exactly what fields are necessary for an SBAC transaction. In SBAC, transactions are defined over **objects**, which we define arbitrarily like so:

```
type obj = {
  id    : id;
  owner : id;
  value : int;
};;
```

FIGURE 5.2: Object type definition. The `owner` and `value` fields are arbitrary, and do not have an impact in our implementation. However, in a real system, such fields might be used for transaction validation.

Additionally, to prevent transactions from using the same objects, we must introduce the notion of **locks**. Once a shard begins processing a transaction, it places a lock on the transaction's input objects. If another transaction wishes to use a locked object, it is immediately rejected. These locks are held until the transaction is either committed or aborted. In our implementation, we define a lock as:

```
type lock = {
  obj_id : int;
  tx_id  : int;
};;
```

FIGURE 5.3: Lock type definition. The `obj_id` field refers to the ID of the locked object, while the `tx_id` field refers to the ID of the transaction that is currently holding the lock.

Putting the two together, the transaction-specific data types that we need are simply a list of objects, and a list of locks!

```
type replica = {
    id        : int;
    objects   : obj list;   (* active objects *)
    locks     : lock list;  (* locks held     *)
    ... (* See Fig. 4.1 *) ...
 };;
```

FIGURE 5.4: Transaction-specific data types for a PBFT node in an SBAC system

5.1.3 Request Structure

The PBFT module is responsible for the heavy lifting when it comes to executing SBAC transactions. In other words, an SBAC shard relies upon its internal PBFT subsystem to check whether a transaction is possible, and if so, to execute it.

As such, we have defined a few **procedures** that the replicas can perform. When the client (i.e. the SBAC shard) makes a request to the PBFT system, it can either ask the replicas to **decide** if a procedure is possible, or to actually **execute** the procedure.

The procedures and actions involved to execute them are as follows:

```
type pbft_procedure =
  | Consume of obj list     (* consume inputs *)
  | Create  of obj list     (* create outputs *)
  | Release of obj list;;   (* release locks on objs *)

type operation =
  | Decide  of pbft_procedure
  | Execute of pbft_procedure;;

type pbft_request =
  | Request of (id * operation);;   (* tx_id * operation *)
```

FIGURE 5.5: PBFT request structure. The `id` in the `Request` type refers to the ID of the transaction in progress. Refer to Fig. 5.1 to see how these actions fit together.

1. `Consume(inputs)`. Replica removes `inputs` from its list of objects.
2. `Create(outputs)`. Replica adds `outputs` to its list of objects.
3. `Release(objs)`. Replica removes all locks which contain `objs` from its list of locks, but only if the `tx_id` from the request matches the `tx_id` holding the lock.

5.1.4 Reply Structure

The replies that the PBFT nodes provide are quite straightforward. If the request was to decide whether a procedure is possible, then the nodes respond with a **decision** (either `Commit` or `Abort`). If it was to execute a procedure, then the nodes reply with their updated state after the procedure has been completed (either `New_values` if the object list was modified, or `New_locks` if the lock list was modified).

In accordance with PBFT's specification, the client waits for a supermajority of nodes to emit the same reply before accepting the result as valid.

```
type decision =
  | Commit
  | Abort;;

type pbft_status =
  | Decision   of decision
  | New_values of obj list
  | New_locks  of lock list;;

type pbft_reply =
  (* replica id * tx_id * status *)
  | Reply of (id * id * pbft_status);;
```

FIGURE 5.6: PBFT reply structure.

5.2 Automated Testing & Findings

As mentioned in the previous chapter, the authors of the SBAC protocol state that under the "honest shards" threat model, no two committed transactions will share any input objects (SBAC Theorem 2 [1]). We have encoded this notion as a predicate on the `handled_replies` field of the SBAC simulator (this is where transactions go after they have been committed/aborted).

In order to check the predicate, we traverse the list of transactions contained in `handled_replies`, maintaining a list of "seen" input objects as we go (we name this list `objs`). For all transactions `tx` with the status `Committed`, we assert that `tx.inputs` and `objs` are disjoint. If this assertion succeeds, then we set `objs = objs @ tx.inputs`, and continue traversing the list of handled replies. If it fails, then the predicate does not hold, and we can stop execution early.

In order to automate test runs of the protocol, we generate a schedule (see Chapter 4.3.1). The schedule has only one instruction: `add_transaction`. However, due to the nature of the SBAC system, adding randomly-generated transactions results in very few actual commitments. This is because in its initial state, the system has no active objects—hence, any transaction that requires inputs will be rejected.

To solve this problem, we either need to initialize the system with a set of active objects, or we need to have a notion of **genesis transactions**—transactions which produce outputs without any inputs. We have chosen to take the latter approach, as it seemed to be the more flexible option.

When generating a schedule of length n, we define three phases:

1. **Genesis.** The first 10% of elements in our schedule consist only of genesis transactions. These serve to give the SBAC system an initial population of active objects, which results in more successful transactions being committed down the line. In order to know what object IDs exist in the system (and to ensure they are unique), we maintain an integer reference `obj_id` which we increment with every output object generated.
2. **Settling.** The next 10-12 elements of the schedule are just the empty list. This phase is intended to allow all transactions from Phase 1 to finish committing, as 10-12 is the amount of rounds necessary for the system to finish processing a transaction.
3. **Normal operation.** The rest of the schedule is composed of random transactions. We define a maximum of three `add_transaction` instructions per schedule element. In each of the transactions, the list

of input object IDs is sampled uniformly from the range $[0, \texttt{obj\_id}]$. The output objects are generated as in the genesis transactions.

In 50 independent executions of randomly generated schedules (of length 1000), we found no violations of the above predicate. This gives us confidence that under the "honest shards" threat model, the safety property as laid out by the Chainspace authors does indeed hold.

For further research, it would have also been interesting to investigate the property under the "dishonest shards" threat model to verify that the property does *not* hold. However, demonstrating this would likely require an implementation of "smart" Byzantine behavior, which would have been slightly out of scope for this project.

Chapter 6

Related Work

This chapter will focus on how our project relates to existing work done on modelling, simulating, and validating distributed systems. Generally, work done in this field follows one of two methodologies: **formal methods**, and **systematic testing** [6]. Our framework falls into the latter category, but below, we will provide some context for both.

6.1 Formal Methods

Formal methods are an active area of research which employ mathematical methods to prove properties about computer programs [19]. To formalize and verify a program is to construct a mathematical model of it (i.e. devise a **formal specification**), and prove that it exhibits some desired properties. This is quite an involved process, but proof assistants such as Coq [24] exist to help to ease the burden by mechanizing some steps of the proofs.

However, the authors of Verdi [25] note that there is usually a significant difference between programs and their formalizations. While specifications may aid in designing systems, their practical construction often differs from what was planned. Furthermore, protocols which have been designed and

published in specific ways are often tweaked to fit an application's individual needs.

To help bridge this gap, the authors have used Coq to develop a framework which allows programmers to produce specifications and proofs about *executable* distributed algorithms. An implementation of a distributed program using Verdi reduces the necessary verification effort by allowing the developer to reason separately about the correctness of their application, and the fault model under which it operates.

Another recent development in the space of verified distributed systems is Disel [21], a framework for compositionally reasoning about applications. In software development, large programs are typically composed of many independent modules which interact with each other. Keeping a large project modular is a good way to separate concerns and reduce the burden of knowledge on developers, who only need to know *what* a module does, without worrying about the *how*.

However, as the authors of Disel describe, efforts in formal verification rarely possess the modularity of large software projects. This is a problem because two systems which rely on the same protocol would likely have to be verified independently, even though the verification effort should ideally only be focused on what was built on top of the shared protocol. Disel addresses this problem, and using their framework, programmers can implement and verify primitive constructs such as distributed protocols, and proceed to build applications on top of them (as long as they satisfy certain specifications).

The projects above have made invaluable contributions toward reasoning about protocols that handle benign failures. However, the focus of this

project is on Byzantine fault tolerant protocols. In this space, the authors of Velisarios [18] have produced a verified implementation of the PBFT protocol [4], along with a framework to aid in reasoning about arbitrary faults.

6.2 Systematic Testing

Systematic testing generally refers to the act of executing of a program in order to find bugs. Since programs often have complicated control flows, the goal is to generate test cases which validate as many execution paths as possible. If a suite of test cases can reach a large percentage of branches in the code, then it is said to have good **code coverage**. For small programs, good tests can be generated manually. However, as programs grow larger and more complex, the number of branches in the execution tree grows exponentially. In order to feasibly verify complicated systems, automated testing is a necessity.

One method to automatically generate test cases is, of course, to simply generate them at random. However, this naturally results in many redundant tests, since we are likely to generate inputs for which the execution paths are exactly the same [7][20]. Since at least 1976, work has been done on generating smarter test cases through a technique called **symbolic execution** [10]. Rather than executing the program with real inputs (concrete execution), the symbolic interpreter leaves the inputs as variables (symbols), and keeps track of all the different branching conditions and constraints. At the end, it is possible to determine real values for the symbols (i.e. construct a test case that leads to a specific branch of code) by solving equations derived from the constraints.

While this was a large advancement in automated testing, it is often computationally intractable to analyze the entire control flow for large programs. Additionally, distributed systems give rise to a whole host of other complications, such as non-deterministic outputs, deadlocks, and liveness issues. Thus, researchers have been actively exploring ways to build the execution tree for testing distributed systems, and one of the better-known projects in this area is the SPIN model checker [9]. In order to deal with the extremely large set of potential system states, SPIN uses many optimizations such as **partial order reduction** and **on-the-fly checking** to reduce the problem's complexity.

More recently, there has been the P programming language [8], and the ModP system [6] built on top of it (and from which our project draws inspiration). P is a language for programmers to implement and test distributed systems code. From the original paper:

> A P program is a collection of *machines*. Machines communicate with each other asynchronously through *events*.

In order to implement a protocol, the programmer must specify the structure of the machines and events. P programs can be verified via the built-in PTester, or compiled to C as an executable. ModP is an extension of P, which allows for more complex programs to be built. In ModP, users can implement systems as individual modules, and have them interact as a larger whole.

6.3 Our Project

The intent of our framework is to provide a tool to support rapid prototyping of protocol implementations. As compared to the work above, it is not as powerful nor as fully-featured. However, we have endeavored to make it accessible and intuitive—asking little of the programmer, while still allowing for a large degree of expressivity.

Chapter 7

Conclusion & Future Work

As distributed systems become more and more critical in our day-to-day lives, the ability to be sure about what they do becomes necessary. This project was created to assist programmers in their pursuit of that goal.

While our framework provides neither the ability to produce full formalizations nor executable code, we believe that it will be useful for those who want to prototype and test ideas quickly, without the learning curve associated with some of the more established works we have described.

We see this tool as a good fit for programmers who are in the ideation phase, prior to the actual implementation or formal verification of a protocol. The framework's ability to test protocol invariants makes it useful for quickly identifying errors in the early stages of experimentation.

Further work on this project would be focused around finding a way to automatically translate the simulator's semantics into Coq, which would be a large step toward building formally verified distributed systems conveniently. Additionally, we could investigate ways to model and test liveness properties, which would add a great deal of value even falling short of formal verification.

Bibliography

[1] Mustafa Al-Bassam, Alberto Sonnino, et al. “Chainspace: A Sharded Smart Contracts Platform”. In: *CoRR* abs/1708.03778 (2017). arXiv: 1708.03778. URL: http://arxiv.org/abs/1708.03778.

[2] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987. Chap. 7, pp. 226–240. ISBN: 0-201-10715-5.

[3] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. 2nd. Springer Publishing Company, Incorporated, 2014. ISBN: 3642423272, 9783642423277.

[4] Miguel Castro and Barbara Liskov. “Practical Byzantine Fault Tolerance”. In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. OSDI ’99. New Orleans, Louisiana, USA: USENIX Association, 1999, pp. 173–186. ISBN: 1-880446-39-1. URL: http://dl.acm.org/citation.cfm?id=296806.296824.

[5] Bernadette Charron-Bost and André Schiper. “The Heard-Of model: computing in distributed systems with benign faults”. In: *Distributed Computing* 22.1 (2009), pp. 49–71. ISSN: 1432-0452. DOI: 10.1007/s00446-009-0084-6. URL: https://doi.org/10.1007/s00446-009-0084-6.

[6] Ankush Desai et al. *Compositional Programming and Testing of Dynamic Distributed Systems*. Tech. rep. UCB/EECS-2018-95. EECS Department, University of California, Berkeley, 2018. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-95.html.

[7] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA: ACM, 2005, pp. 213–223. ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065036. URL: http://doi.acm.org/10.1145/1065010.1065036.

[8] Vivek Gupta et al. *P: Safe Asynchronous Event-Driven Programming*. Tech. rep. 2012. URL: https://www.microsoft.com/en-us/research/publication/p-safe-asynchronous-event-driven-programming/.

[9] Gerard J. Holzmann. "The Model Checker SPIN". In: *IEEE Trans. Softw. Eng.* 23.5 (May 1997), pp. 279–295. ISSN: 0098-5589. DOI: 10.1109/32.588521. URL: https://doi.org/10.1109/32.588521.

[10] James C. King. "Symbolic Execution and Program Testing". In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252. URL: http://doi.acm.org/10.1145/360248.360252.

[11] Leslie Lamport. "Paxos Made Simple". In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), pp. 51–58. URL: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/.

[12] Leslie Lamport. "Specifying Concurrent Systems with TLA+". In: (1999), pp. 183–247. URL: https://www.microsoft.com/en-us/research/publication/specifying-concurrent-systems-tla/.

[13] Leslie Lamport. "The Implementation of Reliable Distributed Multiprocess Systems". In: *Computer Networks* 2 (1978), pp. 95–114. URL: https://www.microsoft.com/en-us/research/publication/implementation-reliable-distributed-multiprocess-systems/.

[14] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *ACM Transactions on Programming Languages and Systems* (1982), pp. 382–401. URL: https://www.microsoft.com/en-us/research/publication/byzantine-generals-problem/.

[15] M. Michael et al. "Scale-up x Scale-out: A Case Study using Nutch/Lucene". In: *2007 IEEE International Parallel and Distributed Processing Symposium*. 2007, pp. 1–8. DOI: 10.1109/IPDPS.2007.370631.

[16] Chris Newcombe, Tim Rath, et al. "Use of Formal Methods at Amazon Web Services". In: (2014). URL: http://lamport.azurewebsites.net/tla/formal-methods-amazon.pdf.

[17] Diego Ongaro and John Ousterhout. "In search of an understandable consensus algorithm". In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014, pp. 305–319.

[18] Vincent Rahli, Ivana Vukotic, et al. "Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq". In: *Programming Languages and Systems*.

Cham: Springer International Publishing, 2018, pp. 619–650. ISBN: 978-3-319-89884-1.

[19] John Rushby. *Formal Methods and the Certification of Critical Systems*. Tech. rep. SRI-CSL-93-7. Also issued under the title *Formal Methods and Digital Systems Validation for Airborne Systems* as NASA Contractor Report 4551, December 1993. Menlo Park, CA: Computer Science Laboratory, SRI International, 1993.

[20] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: A Concolic Unit Testing Engine for C". In: *SIGSOFT Softw. Eng. Notes* 30.5 (Sept. 2005), pp. 263–272. ISSN: 0163-5948. DOI: 10.1145/1095430.1081750. URL: http://doi.acm.org/10.1145/1095430.1081750.

[21] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. "Programming and Proving with Distributed Protocols". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 28:1–28:30. ISSN: 2475-1421. DOI: 10.1145/3158116. URL: http://doi.acm.org/10.1145/3158116.

[22] Maarten van Steen and Andrew S. Tanenbaum. "A Brief Introduction to Distributed Systems". In: *Computing* 98.10 (2016), pp. 967–1009. ISSN: 1436-5057. DOI: 10.1007/s00607-016-0508-7. URL: https://doi.org/10.1007/s00607-016-0508-7".

[23] Andrew S Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, 2007. Chap. 8.

[24] The Coq Development Team. *The Coq Proof Assistant, version 8.8.0*. Apr. 2018. DOI: 10.5281/zenodo.1219885. URL: https://doi.org/10.5281/zenodo.1219885.

[25] James R. Wilcox, Doug Woos, et al. "Verdi: A Framework for Implementing and Formally Verifying Distributed Systems". In: *SIGPLAN Not.* 50.6 (June 2015), pp. 357–368. ISSN: 0362-1340. DOI: 10.1145/2813885.2737958. URL: http://doi.acm.org/10.1145/2813885.2737958.