Dependable Distributed Systems- Project 1 - Phase 3

João Leitão, Nuno Preguiça, and Alex Davidson

NOVA Laboratory for Computer Science and Informatics (NOVA LINCS)

and

Departamento de Informática

Faculdade de Ciências e Tecnologia

Universidade NOVA de Lisboa

V 0.8

April 13, 2023

1 Overview

This document discusses the first phase of the first project for the CSD project (2022/23 edition). The project will be presented as three phases, that will focus on specific aspects of the project as a whole. Throughout the execution of the project each group will always be working towards the final goal, which is already discussed in this document (and might be refined in the next specifications of the project).
Overall, the first project of CSD aims at building a replicated system whose goal is to offer an open goods market. The idea of this market is to operate akin to an auction system (think on the lines of `https://ebay.com`https://ebay.com) but with some twists. In practice in this system, users are able to register offers of goods, including the type of good, quantity, and price per unit that they are willing to sell, and they can also register their interest in acquiring goods, by indicating the type of good, quantity they want to acquire, and the maximum price they are willing to pay for each unit. The system will match these offers and requests to effectively offer a market system for trading general goods. While the specificities of this system, in particular what features could be provided to end users to make it more attractive, usable, and dependable can be discussed in a future phase of the project, the generic architecture of the system is fixed. Throughout the labs in the Dependable Distributed Systems(Confiabilidade de Sistemas Distribuídos) we will be discussing different aspects and features of the implementation of this project. This will assist groups in building the most effective system and also in devising mechanisms to validate the correctness and the performance of their developed solutions. The project will be implemented in Java, taking advantage of the Babel Framework that was already presented in the first lab.
In the following sections, we present some of the details of the project that should guide your implementation efforts. In particular, this document presents the architecture (and some of their interfaces) that should be constructed for the first project (Section 2). The programming environment for developing the project is briefly discussed next (Section 4). Finally, the document concludes by providing some information on operational aspects and delivery rules for this phase of the project (Section 5). Notice that the document do has literature, which is recommended that students read before starting to implement their solutions.

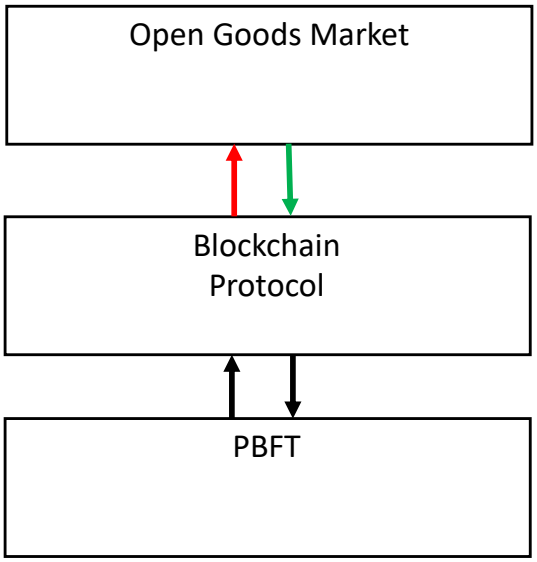

Figure 1: Architecture of a Process.

2 Solution Architecture

Figure 1 illustrates the layering of protocols that you will have to develop for the first project[1]. In a nutshell, each replica of our Open Goods Markey system will be composed of three layers (or protocols). From top to bottom, we will have our application logic (which is labelled *Open Goods Market* in the diagram); we will have a *Blockchain Protocol*; and the lowest layer we will have a Byzantine-fault tolerant agreement protocol, in particular we will be using the well known Practical Byzantine Fault Tolerance Protocol (*PBFT*), which was studied in the lectures. We now provide an overview of the main responsibilities of each of these layers that groups will have to implement. Further details in this document are provided on the interface used by the Blockchain protocol and PBFT to interact.

2.1 Application Logic (Open Goods Market)

The application logic protocol will be responsible for two primary tasks: i) interactions with clients, receiving their operations and replying to them to confirm that those operations where received and will be processed; and ii) maintaining state about current offers and requests, and matches that might be executed among them, this will also require that the application logic exposes operations to clients that allow them to gather information (asynchronously) about the state of their current made (and confirmed) offers and requests for goods. The application will have to submit operations received by the clients to the blockchain protocol, such that these operations can be registered in the replicated ledger (and potentially be matched). Additional details about this protocol will be provided on future releases of this problem statement.
Clients will execute transactions among them using a token managed by the blockchain that we will call CSDs (Coin Simple Dollar). There is a special client that we name *Exchange* that is known to all replicas of the system (and that should have a well known public key), which represents a currency exchange in the real world, where clients can Deposit dollars which are then translated to CSDs in our system. Client evidently can also execute a Withdrawal of their funds from the system through the exchange. This exchange is the only entity in the system that can create and destroy CSDs in the system through the operations DEPOSIT and WITHDRAWAL both of which should contain an unique identifier (UUID), the identify the client (through their public keys) whose funds are being manipulated, the amount of coin to be adder or removed to the client balance of CSDs in the system, and these operations should be

[1]For evaluation purposes this layering is *mandatory*.

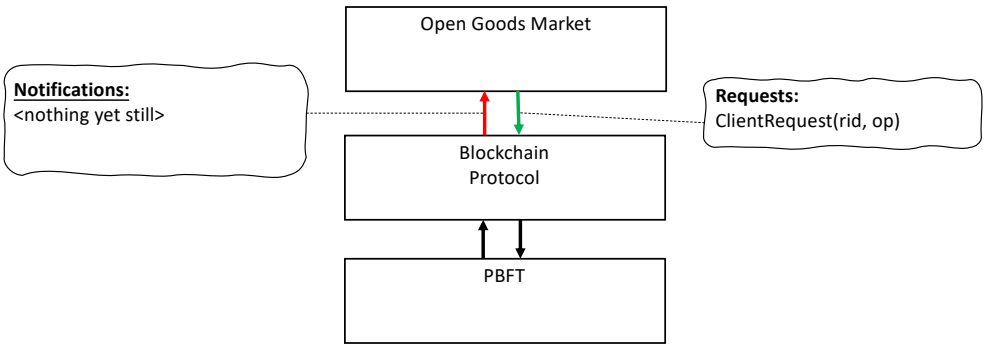

Figure 2: Interface between the Blockchain Protocol and PBFT.

Off

signed by the Exchange. Evidently the Withdrawal operation should only success if the client has enough founds. The status of these operations can be checked by any client using the checkOperationStatus that is described further ahead. The clients can issue multiple operations to any of the replicas. Some operations modify the state of the system while others only observe the current state of the system. The operations that modify the state of the system are: ISSUEOFFER that contains a unique operation identifier (i.e., a UUID), a type of resource being offered (can be of type String), a quantity of resource being offered (integer), and a price per unit (float, represents CSDs); and a ISSUEWANT that also contains a unique operation identifier (i.e., a UUID), a type of resource being bought (String), a quantity of resource being bought (integer), and a minimum price per unit (float, represents CSDs); finally, clients can also cancel and Offer or a Want by issuing a CANCEL operation that should contains the unique identifier of their own operation they want to cancel. Notice that the effect of this last operation only happens if the state in the system (as recorded in the blockchain) as not executed the operation being cancelled. The operations that only read the state of the system are: CHECKOPERATIONSTATUS, which carries a unique operation identifier for a Offer or a Want, and should return to the client either the state *Unknown*, *Pending*, *Executed*, or *Cancelled*, with these states translating to the following situations:

Unknown: The operation has not been registered in the system yet (i.e., not in the blockchain).

Pending: The operation is registered in the system but has not yet been executed (only applicable for Offers and Wants).

Executed: The operation is registered in the system has has had all of their effects, in the case of the Offers and Wants it should be reported the public keys of the clients that were involved in the operation and the amount of CSDs that were transferred between them.

Cancelled: The operation is registered in the blockchain and was cancelled.

In addition to this, a client can also check his balance of CSDs in the system through a CHECKBALANCE operation that should identify the client that is issuing the request using his public key, and with the message being signed by that client. The reply should report the balance of the client or zero if no record exists for that client.
Students should rely on the memory of the process to keep all relevant data that allows them to process these operations efficiently, this can include maps to hold for instance the balance of CSDs for each client. Naturally, the correctness of the application can depend on this state, since the state should be possible to rebuild by using the blockchain maintained by the underlying protocol as a log.

2.2 Blockchain Protocol

The blockchain protocol will maintain a replicated ledger that will register all client operations, and potentially management operations such as adding a new replica or removing a replica from the system. The replicated ledge will be

composed of an ordered set of blocks of operations, that include the operations described previously. The blockchain protocol has to aggregate these operations in blocks which are then proposed to the agreement protocol for being ordered. Notice that the blockchain protocol is operating also as a byzantine-tolerant state machine replication, meaning that it is the responsibility of the blockchain protocol to manage operations that allow to change the system membership (commonly denominated by *view*). The blockchain protocol will act as the client for the PBFT agreement protocol. In more detail the Blockchain protocol receives operations from (external clients) that are received by the Application logic (as captured in Figure 2). It then will generate blocks containing sequences of these operations which are then proposed and agreed on through PBFT.

Block format and Execution: The format of the blocks that compose the blockchain can be fine tuned by students. Evidently, it is expected that each block will contain the hash of the previous block, a sequence number for that block (in the blockchain), an ordered list of operations, the identity of the replica that generated the block and a signature. We reinforce that the students can apply the knowledge their acquired about the Bitcoin and Ethereum blockchain in lectures, to adjust this format. The Genesis block can also be defined by students, and can be read from a file as a parameter of the protocol. Notice that it suffices that all replicas trust the genesis block. The genesis block should have a block number of zero (and since it does not encode any relevant information for the system, it is a special case that does not need to be executed). When blocks are decided by PBFT, replicas should validate the block, and if it is valid then the block can be appended to their local blockchain, and the operations within the block can be executed. Notice that executing the operations might lead to changes on state maintained by the block chains, or might require notifying the application layer above that the operation was executed, such that the application can appropriately update its state. To speed up the validation of operations by the blockchain (which is required to validate the block) students can maintain any auxiliary state in memory that they deem necessary. Notice that operations that are propagated to the application must respect their order in the block to ensure the determinism of their execution. Similarly, the validation of operations in the block should consider the effects of operations that are in the block before the operation being executed.

Invalid Blocks: If a block is decided by PBFT that is invalid, that block should not be appended to the blockchain. In that case, if the block was generated by the current leader, that block is a proof of misbehaviour of that leader, and that should lead to a view change, in such case the blockchain protocol has enough information to inform its local PBFT replica that a view change should be started.

Interation with PBFT: Notice that in this Blockchain protocol, since we are relying on PBFT for agreement, only the current leader of the system (as defined by views installed by PBFT) generates and proposes new blocks. This implies that the Blockchain protocol must know who is the current leader and, for replicas that are not the leader, redirect client requests to the leader (while keeping track that these operations are pending). Additionally, since the Blockchain protocol is the effective client of PBFT this brings some implications:
i) The first implication is that the Blockchain protocol must monitor the execution of client request received by it, and in case these operations are not executed within a given time interval, then the Blockchain protocol should start measures that can lead to a view change in the system. These are explained in more detail in Lab Two, however a quick summary is provided here for completeness. If a client request received by a given node and redirect to the current leader is not present in a committed block in a given time window, the node should start to send that request to all replicas in the system, indicating that the request is pending for too long (e.g., in a message that could be named *ClientRequestUnhandledMessage*. Upon receiving such message, and since the replica sending it might be byzantine (for instance it might never send it to the leader) all replicas should echo that message to all other replicas using some other message (e.g., *StrtClientRequestSuspectMessage*). When a replica receives enough copies of this last message, it becomes aware that the leader must have received this client request, and hence they should start a timer for the time they are willing to wait for the leader to commit that client operation in a block. If the leader does so, than we cannot prove that it is misbehaving and hence the timer is simply cancelled. Otherwise, the leader is potentially misbehaving, and the Blockchain protocol will request a view change to the PBFT protocol using the request SUSPECTLEADER, that will start trigger the appropriate behaviour of PBFT (as discussed on Lecture three).
ii) The second implication is that when receiving a notification of a block that is committed, and following the original specification of PBFT, all replicas should send that notification to each instance of the Blockchain Protocol in all replicas. However, this is not a good approach due to two reasons: a) it breaks the conceptual layering model of

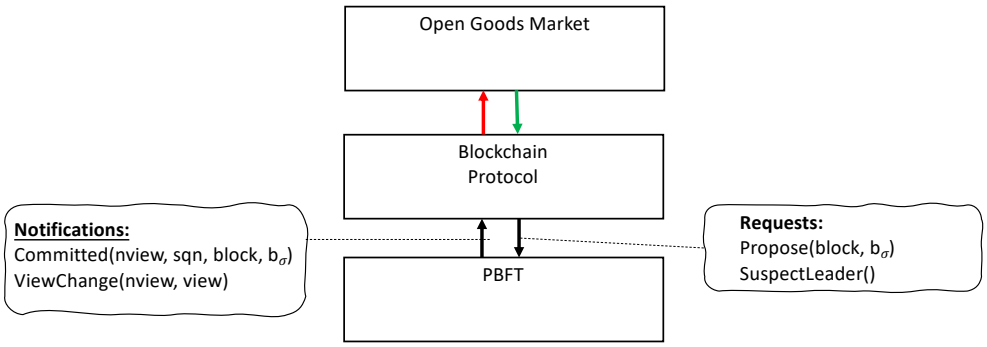

Figure 3: Interface between the Blockchain Protocol and PBFT (revised).

protocols (employed in Babel) where messages should only be exchanged between similar protocols in different machines; and b) this would be very inefficient from the perspective of bandwidth usage, since the non-optimised version would require the block to be sent to each replica by every other replica. To avoid this issue, instead, the COMMITTED notification issued by PBFT to the blockchain protocol only happens within the local process, but, must carry within it enough cryptographic evidence that a large enough number of replicas have agreed on committing that block.
Additional details about this protocol will be provided on future releases of this problem statement, namely some behaviours of the system can be handled by smart contracts registered on and processed by this protocol.

2.3 PBFT

This is the agreement protocol used in our system to allow replicas to agree on which set of operations (i.e., block) to be added to the replicated ledger maintained by the Blockchain protocol. This protocol will be an implementation of the classical PBFT protocol [2], potentially using some optimisations to reduce the size of message exchanged by the protocol. The interface of the protocol to the blockchain protocol is based on events that should be processed asynchronously. There are fundamentally two requests that this protocol can receive from the blockchain protocol above: i) a PROPOSE request that carries a $block$ generated by the blockchain protocol above and the signature of that block (b_sigma), that will serve as the value to be proposed by PBFT on the next non-committed instance of the current view of the protocol; and ii) a SUSPECTLEADER request serves to indicate that the local Blockchain protocol suspects the leader is misbehaving, and that PBFT should start a view change. Each of these requests will lead to triggering notifications to the Blockchain protocol above to indicate to the above protocol the outcome of previous requests. Namely, the COMMITTED notification indicated that a $block$ (with its respective signature b_σ) has been committed on the sequence number sqn of the current view; and the VIEWCHANGE notification that carries a sequence number associated with the new view ($nview$) and the new view of the system ($view$) that is currently employed by PBFT. Notice that the events reported can be changed by students to carry additional information if the groups executing the project consider it necessary.
In terms of messages exchanged by this protocol, the students are free to define their messages as they see fit, following the original specification of the protocol (as discussed in lectures and in [2]), with possible implementation for improving performance. As an additional useful reading for implementing and understanding this protocol, it should be noted that the correctness of PBFT has been discussed in [1].

2.4 Regular Workflow

As a way to make it easier to understand the workflow of client requests within our byzantine tolerant replicated system, Figure 4 captures the life of two operations (which we assume are transactions) that are submitted to the system by two distinct clients to two distinct replicas, one of which is the leader (indicated by the *crown* icon) and another which is a backup replica. The workflow is composed of several interactions whose order is represented

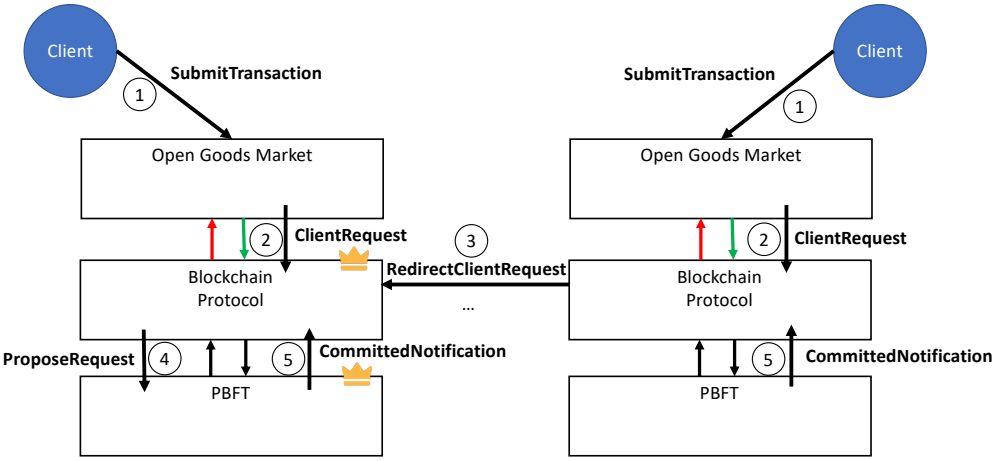

Figure 4: Interface between the Blockchain Protocol and PBFT (revised).

by the numbers within circles (for simplicity we assume some to happen concurrently by using the same number). First, clients issue requests to the application logic of two distinct replicas (1: SUBMITTRANSACTION), these will fundamentally encode some operation the client wants to execute over the system. These client operations are then sent to the Blockchain protocol through a request issued by the Application logic (2: CLIENTRQUEST).
In the case of the leader replica, since it is computing block to be proposed (and eventually added to the ledger), it can process this client request together. However, for a backup replica (replica on the right) this is not true. Therefore the client request information and to be forwarded to the leader replica, which is achieved by sending a message (3: REDIRECTCLIENTREQUEST). Upon receiving this message, the leader replica will add this operation to the block being currently computed (which includes the client request received locally).
Upon closing a block, the leader replica will propose that block to be added to the (replicated) ledger by contacting the local instance of the PBFT protocol using a request for that specific purpose (4: PROPOSEREQUEST). As noted before, only the leader replica executes this step, backup replicas wait for block to be decided by PBFT and committed (and also need to monitor that client operations that they redirect to the leader are eventually committed as part of a block).
When PBFT is able to agree on a value (i.e., commit a block), all Blockchain protocol across all replicas will receive a notification issued by their local instance of PBFT that, among other information, will contain the block itself, and cryptographic proofs that the block was correctly committed by PBFT (5: COMMITTEDNOTIFICATION).
Evidently, the workflow does not terminates at this step, the blockchain protocol will have to process that newly committed block, which will lead to the update of the system state. However, we will present and discuss the details of the Blockchain protocol in a future release of this problem statement (and also in future lectures and labs).

3 Report

This project will require each group to prepare a written report where groups will document the implementation of their solutions, including the presentation of arguments that justify the correctness of their solutions in an asynchronous (more precisely partially synchronous) system under the byzantine fault model as well as an experimental evaluation of the developed system that studies its performance.

3.1 Correctness

When justifying the correctness of the solutions you should be considering the operation of this system in a partially synchronous model under the byzantine fault model. You should also document the attacks that you have considered

in the development of this solution and how these attacks are handled (or mitigated) by your solution. Be as precise as possible in documenting these aspects in the document, and avoid very generic statements.

3.2 Evaluation of Your Solution

You will conduct an experimental evaluation of the developed solutions. To that end groups will have to design a simple client application that generates some workload for replicas of the system. Evaluation is expected to be conducted on the DI and NOVA LINCS research cluster. The main aspects that should be considered in the evaluation of the developed solutions is:

Throughput: Number of client operations that can be handled by the system per time unit. Notice that the throughput should be considered taking into account the number of clients operations registered on the replicated ledger. Naturally, this number might be affected by the number of clients interacting with the system, a factor that should be taken into consideration by students when preparing their experimental evaluation. For instance, it is interesting to study how this performance indicator evolves with increasing numbers of clients interacting with the system (or potentially with the number of replicas or disruptions over the system).

Latency: The time required for client operations to be registered in the replicated ledger. Similar to throughput, latency might be affected by the number of clients interacting with the system, a factor that should be taken into consideration by students when preparing their experimental evaluation. Similar to the previous metric, it might be interesting to study how this performance indicator evolves with increasing numbers of clients interacting with the system (or potentially with the number of replicas or disruptions over the system).

4 Programming Environment

The students will develop their project using the Java language (1.8 minimum). Development will be conducted using a framework developed in the context of the NOVA LINCS laboratory[2] written by Pedro Fouto, Pedro Ákos Costa, João Leitão, named **Babel** [3].
The framework resorts to the Netty framework[3] to support inter-process communication through sockets (although it was designed to hide this from the programmer). The framework will be discussed in the labs, and example protocols will be made available to students. Moreover the top layer that is responsible for injecting load in the system (i.e., propagate messages and receive them) is offered.
The javadoc of the framework can be found here: `https://asc.di.fct.unl.pt/~jleitao/babel/`.
The framework was specifically designed thinking about two complementary goals: i) quick design and implementation of efficient distributed protocols; and ii) teaching distributed algorithms in advanced courses. A significant effort was made to make it such that the code maps closely to protocols descriptions using (modern) pseudocode. The goal is that you can focus on the key aspects of the protocols, their operation, and their correctness, and that you can easily implement and execute such protocols.

5 Operational Aspects and Delivery Rules

Group Formation

Students can form groups of up to three students to conduct the project. Groups composition cannot change across the different phased of the project. While students can do the project alone or in groups of two students this is **highly discouraged**, as the load of the project was designed for three people.

[2] `http://nova-lincs.di.fct.unl.pt`

[3] `https://netty.io`

Since you will be given access to the DI and NOVA LINCS research cluster (see below) to conduct your experiments and extract performance numbers, you must pre-register your group as soon as possible such that you can get credentials to use the cluster. This can be done by sending an e-mail to Alex Davidson (a.davidson@fct.unl.pt) with subject *CSD 22/23 GROUP REGISTRATION*. The e-mail should contain for each member of the group: student number, full name, and institutional e-mail. Even students that plan to do the project alone must register following the procedure above. You will get a reply from the professor (eventually) providing your username and password to access the cluster.

The DI and NOVA LINCS research cluster

The cluster is used for (mostly) research purposes, although some (advanced) courses also use it. The cluster has a limited amount of machines, and hence people might need to compete over computational resources and time. The cluster features a reservation mechanism where you can reserve a set of machines for exclusive access during a period of time. You should not use the cluster for development purposes and only to extract final numbers. Moreover, each group should only reserve at most five machines at a time. Each machine should be able to execute several processes, although you might want to execute different replicas of the system on different machines (and use another one to execute clients). Docker is available in the cluster, and you can freely use it. Each group will be restricted to a limited amount of computational time (where using more machines accounts for a higher consumption of time). Notice that automation is key for success, not only will it allow you to easily repeat experiments, but it might also allow you to run experiments during the night period (where typically usage of the cluster is much more reduced).
The technical specification of the cluster as well as the documentation on how to use it (including changing your group password and making reservations) is online at: `https://cluster.di.fct.unl.pt`. You should read the documentation carefully. The cluster is accessible from anywhere in the world through ssh. Be careful with password management. Never use a weak password to avoid attacks and intrusions on our infrastructure. Incorrect or irresponsible use of the department resources made available to students will be persecuted through disciplinary actions predicted in the School regulations.

Evaluation Criteria

The project delivery includes both the code and a written report that, ideally, should have the format of a short paper. Considering that format, this document refers from this point onward to your final report as simply the *paper*.
This phase of the project will be graded in a scale from 1 to 20 with the following considerations:

- The paper should be at most 10 pages long, including all tables, figures, and references. It should be written with font size no bigger than 10pts and be in two column format. It is highly recommended (for your own sanity) that you use LaTeX to do this. If you do not know LaTeX it is indeed a great time to learn. Videos from an introductory course on LaTeX by João Lourenço shall be provided online at a future date. A LaTeX template for writing the paper shall also be provided.
- The project will be evaluated by the correctness of the implemented solutions, their performance, and the quality of the implementations (in terms of code readability).
- The quality and clearness of the paper can impact the final grade: students with a poorly written report might be penalised on the evaluation, even if they ultimately produce a correct solution.

Deadline

Delivery of project 1 (which includes all phases of this project) is due on May 5^{th} 2023 at 23:59:59.
Further details regarding the delivery rules for the project will be provided at a future date.

References

[1] Miguel Castro, Barbara Liskov, et al. A correctness proof for a practical byzantine-fault-tolerant replication algorithm. Technical report, Technical Memo MIT/LCS/TM-590, MIT Laboratory for Computer Science, 1999.

[2] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.

[3] P. Fouto, P. Costa, N. Preguiça, and J. Leitão. Babel: A framework for developing performant and dependable distributed protocols. In *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*, pages 146–155, Los Alamitos, CA, USA, sep 2022. IEEE Computer Society.