# The Complete Treatise on the Byzantine Generals Problem: Foundations, Solutions, and Applications in Distributed Systems

Soumadeep Ghosh

Kolkata, India

**Abstract**

The Byzantine Generals Problem represents one of the most fundamental challenges in distributed computing, addressing the question of achieving consensus among distributed nodes in the presence of arbitrary failures. This treatise provides a comprehensive examination of the problem's theoretical foundations, mathematical formulations, algorithmic solutions, and practical applications. We explore the impossibility results, lower bounds, and various consensus protocols that have emerged to address Byzantine fault tolerance. The analysis encompasses both synchronous and asynchronous models, examining practical implementations in blockchain systems, distributed databases, and fault-tolerant computing architectures.

The treatise ends with "The End"

## Contents

# 1 Introduction

The Byzantine Generals Problem, first formulated by Lamport, Shostak, and Pease in 1982, stands as a cornerstone problem in distributed systems theory. Named after the Byzantine Empire's military challenges, this problem captures the essence of achieving reliable consensus in distributed systems where some components may fail in arbitrary ways, potentially exhibiting malicious behavior.

Consider a scenario where several divisions of the Byzantine army, each led by a general, surround an enemy city. The generals must coordinate their attack strategy through message passing, but some generals may be traitors who send conflicting information to different loyal generals. The challenge lies in developing a protocol that allows all loyal generals to agree on a common plan of action, despite the presence of traitors.

This seemingly simple military analogy translates directly to fundamental challenges in modern distributed computing systems. In distributed networks, nodes must reach consensus on shared state while tolerating arbitrary failures, including hardware malfunctions, software bugs, network partitions, and even malicious attacks. The Byzantine Generals Problem provides the theoretical framework for understanding and solving these consensus challenges.

The significance of Byzantine fault tolerance extends beyond academic interest. Modern distributed systems, including blockchain networks, distributed databases, cloud computing platforms, and critical infrastructure systems, rely heavily on Byzantine fault-tolerant protocols to maintain consistency and reliability in adversarial environments.

# 2 Problem Formulation and Mathematical Framework

## 2.1 Formal Problem Statement

The Byzantine Generals Problem can be formally stated in terms of a distributed system consisting of $n$ processes, where up to $t$ processes may exhibit Byzantine failures. Each process maintains a local value, and the objective is to reach agreement on a common decision value.

**Definition 2.1** (Byzantine Generals Problem)**.** *Given a system of $n$ processes $P = \{p_1, p_2, \ldots, p_n\}$, where each process $p_i$ has an initial value $v_i \in V$ from some value domain $V$, and at most $t$ processes may exhibit Byzantine failures, design a protocol such that:*

1. ***Agreement**: All non-faulty processes decide on the same value.*
2. ***Validity**: If all non-faulty processes have the same initial value $v$, then $v$ is the only possible decision value.*
3. ***Termination**: Every non-faulty process eventually decides on some value.*

## 2.2 System Models and Assumptions

The analysis of the Byzantine Generals Problem requires careful specification of the underlying system model. We consider several key dimensions that characterize different system environments.

### 2.2.1 Synchrony Model

The synchrony model defines timing assumptions about message delivery and process execution:

**Synchronous Model**: Messages are delivered within a known bounded time $\Delta$, and processes execute in lockstep rounds. This model provides strong timing guarantees that enable timeout-based failure detection.

**Asynchronous Model**: No timing assumptions exist regarding message delivery or process execution speeds. Messages may be delayed arbitrarily, and processes may execute at different rates. This model reflects many real-world distributed systems but presents fundamental impossibility results.

**Partially Synchronous Model**: The system alternates between periods of synchrony and asynchrony, or timing bounds exist but are unknown. This hybrid model captures realistic network conditions while maintaining solvability for consensus problems.

### 2.2.2 Communication Model

The communication infrastructure determines how processes exchange information:

**Point-to-Point Communication**: Processes communicate through direct message passing over reliable or unreliable channels. Authentication mechanisms may prevent message forgery.

**Broadcast Communication**: Processes can broadcast messages to all other processes simultaneously. Different broadcast primitives provide varying delivery guarantees.

### 2.2.3 Failure Model

Byzantine failures represent the most general failure model in distributed systems:

**Definition 2.2** (Byzantine Failure)**.** *A process exhibits a Byzantine failure if it deviates from its specified protocol in any arbitrary manner. Byzantine failures encompass crash failures, omission failures, timing failures, and malicious behavior.*

Byzantine processes may send different messages to different recipients, forge messages from other processes, collude with other Byzantine processes, or remain silent when expected to send messages.

# 3 Impossibility Results and Lower Bounds

Understanding the limitations of Byzantine consensus protocols requires examining fundamental impossibility results and lower bounds that constrain solution approaches.

## 3.1 The $t < n/3$ Lower Bound

The most fundamental result in Byzantine consensus theory establishes the minimum number of processes required to tolerate $t$ Byzantine failures.

**Theorem 3.1** (FLP Lower Bound for Byzantine Consensus)**.** *In a system of $n$ processes where up to $t$ processes may exhibit Byzantine failures, Byzantine consensus is impossible if $t \geq n/3$.*

*Proof Sketch.* The proof proceeds by contradiction. Assume a protocol exists that solves Byzantine consensus with $t \geq n/3$. Consider a system partitioned into three groups of approximately $n/3$ processes each. In the worst case, one group consists entirely of Byzantine processes.

The key insight involves constructing scenarios where loyal processes cannot distinguish between different system configurations. If group $A$ has initial value 0, group $B$ has initial value 1, and group $C$ (Byzantine) sends value 0 to group $A$ and value 1 to group $B$, then groups $A$ and $B$ observe indistinguishable message patterns. This indistinguishability prevents consensus since groups $A$ and $B$ must decide on different values to maintain validity.

The formal proof involves constructing a sequence of indistinguishable executions that lead to different decision values, violating the agreement property. □

This result establishes that any Byzantine consensus protocol requires $n > 3t$ processes to tolerate $t$ Byzantine failures, making it the most resource-intensive fault tolerance approach among distributed consensus protocols.

### 3.2 Asynchronous Impossibility

The Fischer-Lynch-Paterson (FLP) impossibility result extends to Byzantine consensus in asynchronous systems.

**Theorem 3.2** (Asynchronous Byzantine Consensus Impossibility)**.** *In an asynchronous system, no deterministic protocol can solve Byzantine consensus even with a single Byzantine failure, regardless of the number of processes.*

This impossibility result necessitates additional assumptions for practical Byzantine consensus protocols, such as partial synchrony, randomization, or failure detectors.

### 3.3 Communication Complexity Lower Bounds

Byzantine consensus protocols face inherent communication complexity constraints that affect their scalability and performance characteristics.

**Theorem 3.3** (Communication Complexity Lower Bound)**.** *Any Byzantine consensus protocol requires $\Omega(n^2)$ message complexity in the worst case, where $n$ is the number of processes.*

This quadratic lower bound stems from the requirement that each process must communicate with sufficiently many other processes to distinguish Byzantine failures from network partitions or slow processes.

## 4 Classical Solutions and Algorithms

Despite the impossibility results, several important algorithmic approaches solve Byzantine consensus under specific system models and assumptions.

**Algorithm 1** Oral Messages Algorithm OM($m$)

**Input:** Commander $C$ with value $v$, set of lieutenants $L$, parameter $m$
**For each lieutenant** $l \in L$**:**
  If $m = 0$: Lieutenant $l$ uses the value received from commander $C$
  If $m > 0$:
    Commander $C$ sends value $v$ to each lieutenant
    Each lieutenant $l$ acts as commander for OM($m-1$) with remaining lieutenants
    Lieutenant $l$ applies majority function to received values
**return** Decision value based on majority computation

## 4.1 The Oral Messages Algorithm (OM)

The Oral Messages algorithm, proposed by Lamport, Shostak, and Pease, provides the first systematic solution to the Byzantine Generals Problem in synchronous systems.

The OM algorithm requires $m = t$ rounds to tolerate $t$ Byzantine failures, resulting in exponential message complexity $O(n^{t+1})$. While theoretically significant, this exponential complexity limits practical applicability for large values of $t$.

## 4.2 The Signed Messages Algorithm (SM)

The Signed Messages algorithm improves upon OM by leveraging cryptographic signatures to prevent message forgery.

**Algorithm 2** Signed Messages Algorithm SM($m$)

**Initialization:** Each process maintains a set $V$ of signed values
**For** $i = 1$ **to** $m + 1$ **rounds:**
**for** each process $p$ **do**
  Send all newly received signed values to all other processes
  Add valid signed values to set $V$
  Remove values with invalid signatures or excessive signing chains
**end for**
**Decision:** Apply choice function to accumulated signed values in $V$

The SM algorithm achieves polynomial complexity $O(n^2)$ by using signatures to authenticate message origins and prevent Byzantine processes from forging messages from loyal processes.

## 4.3 Practical Byzantine Fault Tolerance (PBFT)

The Practical Byzantine Fault Tolerance (PBFT) algorithm, developed by Castro and Liskov, represents a breakthrough in making Byzantine consensus practical for real-world applications.

PBFT operates in a partially synchronous model and provides both safety and liveness guarantees. The protocol consists of three phases: pre-prepare, prepare, and commit.

#### 4.3.1 PBFT Protocol Phases

**Pre-prepare Phase**: The primary replica broadcasts a pre-prepare message containing the proposed operation and sequence number to all backup replicas.

**Prepare Phase**: Each backup replica validates the pre-prepare message and broadcasts a prepare message to all other replicas. A replica enters the prepared state after receiving $2t$ matching prepare messages from different replicas.

**Commit Phase**: Prepared replicas broadcast commit messages. After receiving $2t + 1$ matching commit messages, a replica commits the operation and sends a reply to the client.

**Theorem 4.1** (PBFT Correctness)**.** *PBFT provides safety and liveness in a partially synchronous system with at most $t < n/3$ Byzantine failures, where $n \geq 3t + 1$ replicas participate in the protocol.*

PBFT achieves $O(n^2)$ message complexity per consensus instance and has been successfully deployed in various distributed systems applications.

# 5 Modern Consensus Protocols

Contemporary distributed systems have driven the development of more efficient and scalable Byzantine consensus protocols that address the limitations of classical algorithms.

## 5.1 Tendermint Consensus

Tendermint provides a Byzantine fault-tolerant consensus engine designed for blockchain applications. The protocol separates consensus from application logic, enabling generic state machine replication.

The Tendermint consensus protocol operates through a sequence of rounds, each consisting of multiple steps:

**Propose Step**: A designated proposer broadcasts a block proposal to all validators.

**Prevote Step**: Validators broadcast prevote messages indicating their preliminary preference for the proposed block.

**Precommit Step**: Upon receiving sufficient prevotes, validators broadcast precommit messages representing their commitment to the proposed block.

**Commit Step**: After gathering sufficient precommit messages, validators finalize the block and advance to the next height.

Tendermint incorporates a view-change mechanism to handle faulty proposers and ensures both safety and liveness properties under partial synchrony assumptions.

## 5.2 HotStuff and Its Variants

HotStuff introduces a three-phase Byzantine consensus protocol that achieves linear message complexity and optimal latency characteristics.

The protocol's key innovation lies in its chained structure, where each phase of consensus for block $n$ simultaneously serves as a phase for block $n+1$. This pipelining approach significantly improves throughput while maintaining safety and liveness guarantees.

**Prepare Phase**: The leader proposes a block and collects prepare votes from replicas.

**Pre-commit Phase**: Upon receiving sufficient prepare votes, the leader broadcasts a pre-commit message and collects pre-commit votes.

**Commit Phase**: With sufficient pre-commit votes, the leader broadcasts a commit message and collects commit votes.

**Decide Phase**: After gathering sufficient commit votes, replicas decide on the proposed block.

HotStuff achieves $O(n)$ message complexity per consensus decision, representing a significant improvement over quadratic protocols like PBFT.

### 5.3 Fast Byzantine Consensus

Recent research has focused on reducing the latency of Byzantine consensus protocols through optimistic execution and speculation mechanisms.

Fast Byzantine Paxos and similar protocols attempt to commit operations in a single round when no failures occur, falling back to slower multi-round protocols when Byzantine behavior is detected. These optimistic protocols trade worst-case complexity for improved common-case performance.

## 6 Applications in Distributed Systems

Byzantine fault tolerance finds applications across numerous domains in modern distributed computing, each presenting unique requirements and constraints.

### 6.1 Blockchain and Cryptocurrency Systems

Blockchain networks represent the most prominent application of Byzantine consensus protocols. These systems require consensus among potentially malicious participants in a permissionless environment.

**Bitcoin's Proof of Work**: Bitcoin employs a probabilistic consensus mechanism based on computational puzzles. While not strictly Byzantine fault-tolerant in the classical sense, proof of work provides consensus guarantees under specific assumptions about computational resources and network synchrony.

**Ethereum's Transition to Proof of Stake**: Ethereum's beacon chain implements a Byzantine fault-tolerant proof of stake protocol that combines economic incentives with cryptographic protocols to achieve consensus among validators.

**Permissioned Blockchain Networks**: Enterprise blockchain platforms like Hyperledger Fabric employ variants of PBFT for consensus among known participants, providing deterministic finality and high throughput.

### 6.2 Distributed Database Systems

Database systems require strong consistency guarantees while tolerating various failure modes, making Byzantine fault tolerance valuable for critical applications.

**Replicated State Machines**: Database replication systems use Byzantine consensus to maintain consistency across geographically distributed replicas, ensuring that all non-faulty replicas observe the same sequence of operations.

**Multi-Master Replication**: Byzantine consensus protocols enable multiple database masters to coordinate updates and resolve conflicts in the presence of Byzantine failures.

### 6.3 Cloud Computing and Distributed Storage

Cloud infrastructure systems leverage Byzantine fault tolerance to provide reliable services despite hardware failures, software bugs, and potential security breaches.

**Distributed File Systems**: Systems like Google File System and Hadoop Distributed File System incorporate Byzantine fault-tolerant protocols for metadata management and consistency maintenance.

**Object Storage Systems**: Cloud object storage platforms use Byzantine consensus for maintaining metadata consistency across distributed storage nodes.

## 7 Performance Analysis and Optimization

Understanding the performance characteristics of Byzantine consensus protocols is crucial for practical deployment and system design decisions.

### 7.1 Latency Analysis

Byzantine consensus protocols exhibit different latency characteristics depending on their design and system model assumptions.

**Synchronous Protocols**: Synchronous algorithms like OM and SM provide predictable latency bounds based on the maximum message delay $\Delta$ and the number of rounds required.

**Asynchronous Protocols**: Partially synchronous protocols like PBFT provide latency bounds only during synchronous periods, with potentially unbounded delay during asynchronous periods.

**Optimistic Protocols**: Fast Byzantine consensus protocols achieve low latency in common cases but may experience higher latency when Byzantine behavior is detected.

### 7.2 Throughput Characteristics

The throughput of Byzantine consensus protocols depends on message complexity, cryptographic operations, and network bandwidth utilization.

Linear protocols like HotStuff achieve higher throughput than quadratic protocols like PBFT by reducing message complexity from $O(n^2)$ to $O(n)$. However, practical throughput also depends on implementation efficiency, cryptographic algorithm choices, and network topology.

### 7.3 Scalability Considerations

Byzantine consensus protocols face inherent scalability challenges due to the $n > 3t$ requirement and communication complexity constraints.

**Sharding Approaches**: Large-scale systems employ sharding techniques to partition the consensus problem across multiple smaller groups, reducing per-shard complexity while maintaining overall system consistency.

**Hierarchical Consensus**: Multi-layer consensus architectures combine fast local consensus with slower global consensus to balance scalability and consistency requirements.

# 8 Security Analysis and Attack Models

Byzantine consensus protocols must maintain security properties against various attack scenarios and adversarial behaviors.

## 8.1 Adversarial Models

Different adversarial models capture varying assumptions about attacker capabilities and coordination:

**Static Adversaries**: The set of Byzantine processes is fixed throughout protocol execution. This model simplifies analysis but may not reflect dynamic attack scenarios.

**Adaptive Adversaries**: Adversaries can corrupt processes dynamically based on observed protocol execution. This stronger model better captures sophisticated attacks but complicates protocol design.

**Mobile Adversaries**: Byzantine failures can move between processes over time, subject to constraints on the total number of concurrent failures.

## 8.2 Attack Vectors

Real-world Byzantine consensus implementations face various attack vectors beyond the theoretical model:

**Network-Level Attacks**: Adversaries may manipulate network infrastructure to delay, drop, or reorder messages between honest processes.

**Cryptographic Attacks**: Weaknesses in cryptographic primitives or implementation vulnerabilities may enable signature forgery or private key compromise.

**Implementation Attacks**: Software bugs, side-channel attacks, or hardware vulnerabilities may allow adversaries to compromise processes beyond the assumed failure model.

## 8.3 Mitigation Strategies

Practical deployments employ various mitigation strategies to enhance security beyond theoretical guarantees:

**Defense in Depth**: Multiple security layers provide redundant protection against different attack vectors.

**Intrusion Detection**: Monitoring systems detect anomalous behavior that may indicate Byzantine failures or attacks.

**Key Management**: Robust key distribution and rotation procedures limit the impact of cryptographic key compromise.

# 9 Implementation Considerations

Translating theoretical Byzantine consensus protocols into practical implementations requires addressing numerous engineering challenges and design decisions.

### 9.1 Message Authentication

Reliable message authentication prevents Byzantine processes from forging messages from honest processes:

**Digital Signatures**: Cryptographic signatures provide strong authentication but introduce computational overhead.

**Message Authentication Codes**: Symmetric authentication codes offer better performance but require pairwise shared secrets.

**Threshold Signatures**: Advanced cryptographic techniques enable efficient verification of messages signed by multiple processes.

### 9.2 Network Communication

Efficient network communication is critical for protocol performance:

**Message Batching**: Aggregating multiple consensus instances reduces per-operation overhead.

**Pipelining**: Overlapping different phases of consensus for consecutive operations improves throughput.

**Compression**: Message compression techniques reduce network bandwidth requirements.

### 9.3 State Management

Consensus protocols must maintain various forms of state information:

**Persistent State**: Critical state information must survive process crashes and restarts.

**Checkpointing**: Periodic state checkpoints enable efficient recovery and garbage collection.

**Log Compaction**: Techniques for safely discarding old log entries prevent unbounded state growth.

## 10 Future Directions and Open Problems

Byzantine consensus remains an active research area with several important open problems and emerging directions.

### 10.1 Scalability Improvements

Despite significant progress, scalability remains a fundamental challenge for Byzantine consensus:

**Sub-linear Protocols**: Developing protocols with better than $O(n^2)$ communication complexity while maintaining Byzantine fault tolerance guarantees.

**Asynchronous Scalability**: Designing scalable Byzantine consensus protocols for asynchronous systems without timing assumptions.

**Cross-Chain Consensus**: Enabling Byzantine consensus across multiple blockchain networks or distributed systems.

### 10.2 Efficiency Optimizations

Improving the practical efficiency of Byzantine consensus protocols through algorithmic and implementation advances:

**Quantum-Resistant Cryptography**: Adapting Byzantine consensus protocols to use post-quantum cryptographic primitives.

**Hardware Acceleration**: Leveraging specialized hardware for cryptographic operations and message processing.

**Machine Learning Integration**: Applying machine learning techniques for failure prediction and protocol optimization.

### 10.3 Novel Application Domains

Emerging application areas present new challenges and requirements for Byzantine consensus:

**Internet of Things**: Adapting Byzantine consensus for resource-constrained IoT devices and networks.

**Edge Computing**: Designing protocols suitable for edge computing environments with variable connectivity and heterogeneous devices.

**Federated Learning**: Applying Byzantine fault tolerance to distributed machine learning systems to handle malicious participants.

## 11 Conclusion

The Byzantine Generals Problem represents one of the most fundamental and enduring challenges in distributed systems theory. From its initial formulation as a theoretical problem to its modern applications in blockchain networks and distributed databases, Byzantine consensus has evolved into a critical component of fault-tolerant distributed computing.

The theoretical foundations established by early impossibility results and lower bounds provide essential constraints that guide practical protocol design. The $n > 3t$ lower bound demonstrates the inherent cost of Byzantine fault tolerance, while the FLP impossibility result highlights the importance of system model assumptions.

Classical algorithms like the Oral Messages and Signed Messages protocols established the theoretical feasibility of Byzantine consensus, while practical systems like PBFT demonstrated the viability of Byzantine fault tolerance for real-world applications. Modern protocols continue to push the boundaries of efficiency and scalability while maintaining strong correctness guarantees.

The widespread adoption of blockchain technology has renewed interest in Byzantine consensus and driven significant advances in protocol design. From the probabilistic consensus of proof-of-work systems to the deterministic finality of proof-of-stake protocols, Byzantine fault tolerance remains central to the security and reliability of decentralized systems.

Looking forward, Byzantine consensus faces ongoing challenges in scalability, efficiency, and applicability to emerging distributed computing paradigms. The fundamental trade-offs between fault tolerance, performance, and resource requirements continue to drive research in more efficient protocols and novel application approaches.

The Byzantine Generals Problem serves not only as a theoretical framework but as a practical guide for building reliable distributed systems in an uncertain and potentially adversarial world. As distributed computing continues to permeate every aspect of modern technology, the principles and protocols developed for Byzantine fault tolerance will remain essential tools for system designers and researchers.

The evolution from impossibility results to practical implementations demonstrates the power of theoretical computer science to provide both fundamental understanding and practical solutions. The Byzantine Generals Problem stands as a testament to the enduring relevance of rigorous theoretical analysis in addressing real-world distributed computing challenges.

# References


[1] L. Lamport, R. Shostak, and M. Pease. *The Byzantine Generals Problem*. ACM Transactions on Programming Languages and Systems, 4(3):382–401, 1982.

[2] M. J. Fischer, N. A. Lynch, and M. S. Paterson. *Impossibility of Distributed Consensus with One Faulty Process*. Journal of the ACM, 32(2):374–382, 1985.

[3] M. Castro and B. Liskov. *Practical Byzantine Fault Tolerance*. Proceedings of the Third Symposium on Operating Systems Design and Implementation, pages 173–186, 1999.

[4] D. Dolev and H. R. Strong. *Authenticated Algorithms for Byzantine Agreement*. SIAM Journal on Computing, 12(4):656–666, 1983.

[5] C. Dwork, N. Lynch, and L. Stockmeyer. *Consensus in the Presence of Partial Synchrony*. Journal of the ACM, 35(2):288–323, 1988.

[6] G. Bracha and S. Toueg. *Asynchronous Consensus and Broadcast Protocols*. Journal of the ACM, 34(4):824–840, 1987.

[7] R. Canetti and T. Rabin. *Fast Asynchronous Byzantine Agreement with Optimal Resilience*. Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing, pages 42–51, 1993.

[8] M. Ben-Or, B. Kelmer, and T. Rabin. *Asynchronous Secure Computations with Optimal Resilience*. Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, pages 183–192, 1994.

[9] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. *Zyzzyva: Speculative Byzantine Fault Tolerance*. ACM Transactions on Computer Systems, 27(4):1–39, 2009.

[10] E. Buchman, J. Kwon, and Z. Milosevic. *The Latest Gossip on BFT Consensus*. arXiv preprint arXiv:1807.04938, 2018.

[11] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. *HotStuff: BFT Consensus with Linearity and Responsiveness*. Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, pages 347–356, 2019.

[12] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. White paper, 2008.

[13] S. King and S. Nadal. *PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake*. White paper, 2012.

[14] L. Lamport. *The Part-Time Parliament*. ACM Transactions on Computer Systems, 16(2):133–169, 1998.

[15] F. B. Schneider. *Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial*. ACM Computing Surveys, 22(4):299–319, 1990.

[16] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and A. Spiegelman. *Solida: A Blockchain Protocol Based on Reconfigurable Byzantine Consensus*. Proceedings of the 21st International Conference on Principles of Distributed Systems, 2017.

[17] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. *The Next 700 BFT Protocols*. ACM Transactions on Computer Systems, 32(4):1–45, 2015.

[18] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. *UpRight Cluster Services*. Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, pages 277–290, 2009.

[19] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo. *Efficient Byzantine Fault-Tolerance*. IEEE Transactions on Computers, 62(1):16–30, 2013.

[20] R. Pass, L. Seeman, and A. Shelat. *Analysis of the Blockchain Protocol in Asynchronous Networks*. Annual International Conference on the Theory and Applications of Cryptographic Techniques, pages 643–673, 2017.

# The End