Journal of Internet Services and Applications, 2026, 17:1, doi: 10.5753/jisa.2026.7765

Reducing Replica Coordination in BFT Consensus through Dependable and Secure In-Network Message Ordering

Gabriel Faustino Lima da Rocha [**Universidade de Brasília (UnB)** | ***gabriel.faustino@aluno.unb.br***]
Eduardo Alchieri [**University of Brasília (UnB)** | ***alchieri@unb.br***]
Giovanni Venâncio [**Universidade Federal do Paraná** | ***giovanni@inf.ufpr.br***]
Vinicius Fulber-Garcia [**Universidade Federal do Paraná** | ***vinicius@inf.ufpr.br***]
Elias P. Duarte Jr. [**Universidade Federal do Paraná** | ***elias@inf.ufpr.br***]

Department of Computer Science, University of Brasília, Campus Universitário Darcy Ribeiro, Brasília-DF, 70910-900, Brazil.

Received: 10 March 2026 • **Accepted:** 22 July 2026 • **Published:** 25 August 2026

Abstract Programmable networks enable the deployment of in-network ordering services that can improve the performance of consensus protocols. However, existing approaches either tolerate only crash faults or require additional replica coordination to handle Byzantine failures, limiting their performance gains. In this work, we present NsoBFT, a Byzantine fault-tolerant (BFT) consensus protocol that leverages secure in-network ordering through a USIG (Unique Sequential Identifier Generator)-based message ordering service deployed at the network layer. The ordering service is implemented using Network Functions Virtualization (NFV), allowing it to operate as a virtualized network function that provides isolation and flexible integration into modern infrastructures. By combining NFV-based in-network sequencing with a USIG-protected sequencer, NsoBFT eliminates the extra replica confirmation step required by prior BFT in-network protocols, shortening the critical execution path while preserving consensus safety and liveness properties. We implemented NsoBFT and evaluated it against NeoBFT and NOPaxos, two representative in-network consensus protocols. Experimental results show that NsoBFT consistently achieves lower latency and higher throughput than NeoBFT, significantly narrowing the performance gap between crash-tolerant and Byzantine-tolerant in-network consensus.

Keywords: Consensus, Distributed algorithms, Intrusion tolerance.

1 Introduction

Distributed consensus and agreement algorithms are fundamental abstractions for building reliable distributed systems, enabling a set of replicas to agree on a common sequence of operations despite failures. Consensus protocols have been successfully used in many different applications and contexts (e.g., Burrows [2006]; Hunt *et al*. [2010]; J. C. Corbett and et al [2012]). In particular, consensus is the basis of state machine replication [Schneider, 1990], a comprehensive and widely used approach to build fault-tolerant systems. It has recently also received significant attention in the context of blockchains [Vukolić, 2015; Liu and Yu, 2024; Freitas *et al*., 2024], and several other problems [Zou *et al*., 2024; Li *et al*., 2024; Venâncio *et al*., 2024]. Consensus is required whenever the replicas or processes of a distributed system have to agree on a value, given an initial set of proposed values. Consensus properties can be classified into two categories: those that aim to guarantee agreement (*safety*), and those that ensure termination (*liveness*).

Classic consensus algorithms that assume crash faults include Paxos [Lamport, 1998] and Raft [Ongaro and Ousterhout, 2014], among others. Those algorithms cannot deal with arbitrary faulty behavior, often related to intrusion incidents. Intrusion-tolerant consensus algorithms are also called Byzantine Fault-Tolerant (BFT). PBFT (Practical BFT) [Castro and Liskov, 1999, 2002] was the first intrusion tolerant consensus algorithm that was shown to provide feasible implementations. Since then, several variations as well as other BFT consensus algorithms have been proposed [Singh *et al*., 2022].

Consensus is frequently implemented in the application layer considering a message-passing distributed system composed of reliable channels. Such a channel can be implemented with a reliable transport protocol on top of a fair-loss network layer. A well-known limitation of consensus protocols is their high communication overhead, typically requiring multiple communication phases among replicas. Recent advances in programmable networks have enabled a new design space in which part of the consensus logic, particularly message ordering, can be moved into the network layer. Protocols such as NOPaxos (Network Ordered Paxos) [Li *et al*., 2016] demonstrated that in-network ordering can significantly reduce latency and increase throughput in crash fault-tolerant settings. NOPaxos was the first consensus solution to employ a package sequencer module implemented in the network layer to assign unique sequence numbers to packages (messages). Thus the safety properties are guaranteed at this layer while the application layer is responsible only for termination. An empirical evaluation of NOPaxos shows that the algorithm presents a latency that is close to the bare network latency.

Later, NeoBFT [Sun *et al*., 2023] extended this idea to Byzantine environments by employing authenticated in-network ordering. However, NeoBFT still requires additional

replica coordination to prevent equivocation caused by a malicious sequencer, limiting the performance gains achievable through in-network acceleration. The network layer in this case uses digital signatures to provide an authenticated ordering service. However, this algorithm uses an additional communication step between the replicas before executing a request, required to deal with the fact that a malicious sequencer can assign different sequence numbers to the same message. This extra communication step increases the overhead for running consensus. Note that in NeoBFT both the network and application layers need to work together for ensuring the safety properties.

In parallel, the evolution of Network Functions Virtualization (NFV) [Mijumbi *et al*., 2016] has transformed how network services are deployed and managed. NFV enables network functions traditionally implemented in dedicated hardware to be deployed as virtualized software components running on commodity infrastructure. This approach provides flexibility, isolation, scalability, and easier integration into modern cloud and edge environments. Rather than relying on specialized programmable switches or hardware coprocessors, NFV allows sophisticated network-layer services to be instantiated, replicated, and managed dynamically as Virtual Network Functions (VNFs).

In a previous work, we proposed NsoBFT (Network secure ordered BFT) [da Rocha *et al*., 2026], a BFT consensus protocol that integrates a secure message ordering service into the network layer through a USIG (Unique Sequential Identifier Generator) module [Veronese *et al*., 2013] which constrains the malicious actions that the sequencer can execute. The USIG component prevents a malicious sequencer from assigning conflicting sequence numbers to the same package, thereby eliminating the need for the additional replica confirmation phase required in NeoBFT. In this work, besides refining the protocol description, providing a comprehensive correctness discussion, and significantly expanding the experimental evaluation, we leverage NFV to implement the in-network ordering service as a virtual network function. By deploying the sequencer as a VNF, the ordering logic becomes an isolated, manageable, and practically deployable network service.

This design achieves two key objectives. First, it reduces the critical communication path of BFT consensus by removing one coordination step among replicas. Second, it demonstrates that secure in-network ordering can be realized using virtualization technologies, without requiring specialized programmable switch hardware.

NsoBFT and other BFT consensus algorithms, such as PBFT and NeoBFT, require $3f + 1$ replicas to tolerate up to f malicious replicas, in addition to the sequencer. However we also discuss two variants of NsoBFT that require only $2f + 1$ replicas. The first variant is called MinNsoBFT and requires an additional communication step, while the second variant, called MinZyzNsoBFT, does not require any additional steps. In the second variant the client completes an operation when it receives the same response from all replicas. However, if this is not the case the client needs to execute additional actions to synchronize the replicas. Table 1 shows how different consensus algorithms can be compared in terms of the type of faults assumptions, number of replicas required, number of communication steps required, communication complexity, and the requirement of secure components.

In addition to describing and specifying NsoBFT, the algorithms were implemented and compared (NOPaxos, NeoBFT, and NsoBFT). Results show that NOPaxos presents superior performance but tolerates only crash faults, while NsoBFT consistently achieves lower latency and higher throughput than NeoBFT, narrowing the performance gap between crash-tolerant and Byzantine fault-tolerant in-network consensus.

The remainder of this work is organized as follows. Section 2 presents the background, while Section 3 discusses related work. Section 4 presents the NsoBFT algorithm and discusses its variants. The experimental evaluation is presented in Section 5. Finally, Section 6 concludes the paper.

2 Background

This section presents an overview of the background, with consensus in the first subsection, and NFV in the second.

2.1 The Consensus Problem

Consider a distributed system composed of a set of independent processes $\Pi = p_1, p_2, .., p_n$, process p_i is also refereed to as process i. The processes of Π run a *consensus* algorithm to agree on a value of a predefined set of possible values. Initially, processes *propose* values. Eventually *all* correct processes *decide* on a single one of the values proposed. Thus, in order to execute a consensus instance, processes execute two primitives [Hadzilacos and Toueg, 1994]:

- `propose(v)`: a process executes this primitive to propose value v to the set of processes in Π.
- `decide(v)`: a process executes this primitive to notify the interested party (*i.e.*, an application) that v is the decided value.

In order to ensure safety and liveness, and assuming the crash fault model, consensus must satisfy the following properties:

- **Agreement**: if a correct process decides v, then all correct processes eventually decide v.
- **Validity**: a correct process decides v only if v was previously proposed by some process in Π.
- **Termination**: all correct processes eventually decide.

In the Byzantine fault model, the validity property can be strengthened to prevent faulty processes from forcing arbitrary values to be decided. Therefore, a decided value must not only originate from a proposal, but must also satisfy a validity predicate [Cachin *et al*., 2001; Sousa and Bessani, 2012; Vassantlal *et al*., 2022]. This ensures that even in the presence of Byzantine replicas, only admissible values can be selected by the consensus protocol.

The agreement property ensures that all correct processes decide on the same value. Validity relates the decided value to the proposed values. Note that the fault model has a direct impact on validity, and changes in validity can lead to different types of consensus. The agreement and validity

	Paxos	PBFT	NOPaxos	NeoBFT	**NsoBFT**	**MinNsoBFT**	**MinZyzNsoBFT**
Failure	crash	Byzantine	crash	Byzantine	**Byzantine**	**Byzantine**	**Byzantine**
# of replicas[1]	*2f+1*	*3f+1*	*2f+1*	*3f+1*	***3f+1***	***2f+1***	***2f+1***
Comm. steps	4	5	2	3	**2**	**3**	**2**
Comm. Comp.[2]	O(n²)	O(n²)	O(n)	O(n²)	**O(n)**	**O(n²)**	**O(n)**
Secure service	-	-	-	-	**USIG**	**USIG**	**USIG**

Table 1. Comparison of consensus algorithms. The sequencer of the algorithms that employ the network layer to order messages (NoPaxos, NeoBFT, NsoBFT, and their variants) "intercepts" requests and includes the sequence numbers, thus requiring one less communication step.
Notes:
[1] f represents the number of tolerated faults.
[2] n represents the number of replicas in the system.

properties define the safety requirements of the consensus problem, and the termination property defines its liveness. It has been proved [Fischer *et al*., 1985] that it is impossible to solve the consensus problem deterministically in a completely asynchronous distributed system where at least one process can fail by crashing. Therefore, most consensus solutions make some stronger synchrony assumption, e.g. the GST (Global Stabilization Time) [Dwork *et al*., 1988].

2.2 NFV: Virtual Network Functions and Services

Network Functions Virtualization (NFV) has been proposed as a software-based alternative to implement the so-called network middleboxes, which include firewalls, Network Address Translation (NAT) devices, Intrusion Detection Systems (IDS), among others. Middleboxes have traditionally been implemented as specialized hardware, and in that case may represent a significant portion of a network's capital expenditures (CAPital EXpenditures - CAPEX) and operational expenditures (OPerational EXpenditures - OPEX) [Mijumbi *et al*., 2016]. NFV can reduce costs while improving flexibility, simplifying the design, development, and management of network services, and saving physical space and energy. On the other hand, NFV has also opened up the middlebox vendor market, even allowing network functions and services to be provided in an as-a-service model [Boubendir *et al*., 2018].

NFV technology allows middleboxes to be implemented in software as VNFs (Virtual Network Functions) that run on off-the-shelf hardware. While VNFs execute specific network functions, they can be combined to create complete network services, known as SFCs (Service Function Chains) [Halpern and Pignataro, 2015]. An SFC is a composition of multiple VNFs chained in a specific order, through which traffic is steered [Fulber-Garcia *et al*., 2020].

The ETSI has coordinated efforts that led to the NFV-MANO (NFV Management and Orchestration) reference architecture [ETSI Industry Specification Group (ISG) NFV, 2024]. NFV-MANO allows the interoperability of virtual functions and services from different vendors, and includes modules for VNF control and orchestration, including lifecycle management and resource allocation. In addition, NFV-MANO defines communication interfaces and provides abstractions for the resources needed to execute VNFs. Figure 1 shows the NFV-MANO architecture, the NFV Infrastructure (NFVI) and the VNFs themselves.

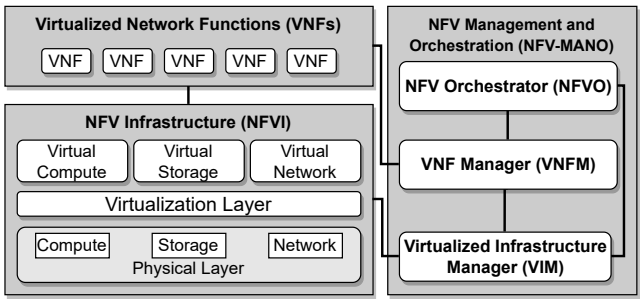


Figure 1. NFV-MANO reference architecture.

The NFVI is the virtualized infrastructure in which the VNFs are instantiated, managed, and executed. The NFVI includes physical storage, network, and compute resources. Physical resources are abstracted into virtual resources through a virtualization layer, which consists of a hypervisor that creates and manages virtualized devices, *i.e.* Virtual Machines (VMs) and containers. This strategy provides isolation so that each VNF can run independently. The VNFs in Figure 1 represent the function instances running on the NFVI.

NFV-MANO itself is divided into three main modules. The first is the NFV Orchestrator (NFVO), which allows the composition of VNFs on SFCs (Service Function Chains), and also manages their lifecycle and resources [Kaur *et al*., 2022]. The second module is the VNF Manager (VNFM), which is responsible for VNF lifecycle management, including instantiation, deletion, configuration, and auto-scaling of VNFs [Venâncio *et al*., 2021a]. To perform its functionalities, the VNFM makes use of the VNF Descriptor (VNFD), a template responsible for specifying for each VNF its requirements in terms of operation and deployment. Finally, the Virtualized Infrastructure Manager (VIM) controls and manages the computing resources of the NFVI. The VIM is primarily responsible for creating, deleting, and reconfiguring virtual devices.

In this work, NFV enables the ordering service to be deployed as an isolated VNF. By hosting the sequencer within the NFV infrastructure, the ordering logic executes in a controlled and logically separated environment, reducing interference from clients and replicas and improving the robustness of the ordering service. This isolation is particularly important in BFT environments, where faulty or malicious components may attempt to disrupt the protocol execution. At the same time, the use of NFV allows the sequencing service to be flexibly deployed over commodity infrastructure without requiring specialized programmable switches or custom hardware. This design combines the benefits of in-network ordering with the

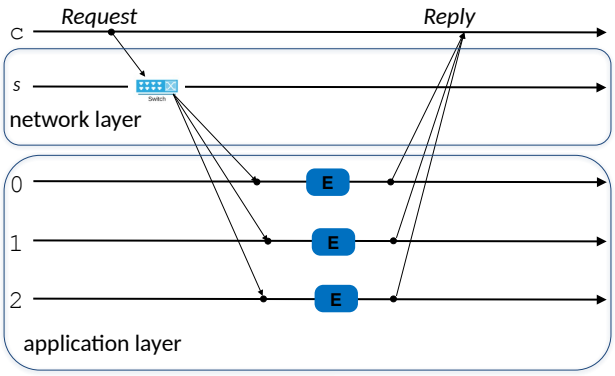
practicality and deployment flexibility of NFV-based infrastructures, enabling a secure and efficient implementation of the proposed consensus protocol.

3 Related Work: Consensus, from Paxos to NeoBFT

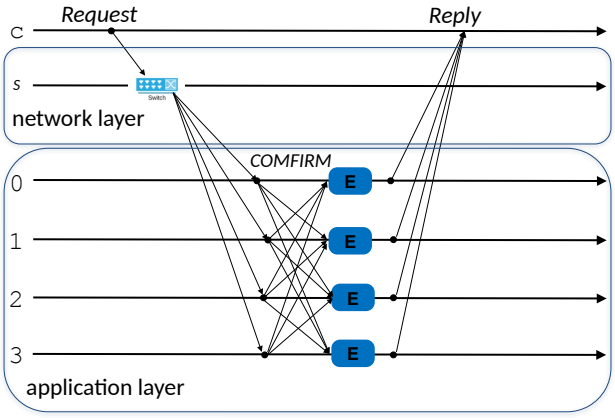
Although several consensus algorithms have been proposed, given the nature of the algorithm introduced in the present work, we start our overview of related work with Paxos [Lamport, 1998], which is one of the most important consensus algorithms. Paxos assumes that processes can fail by crashing. Processes assume three different roles: proposer, acceptor or learner. The algorithm consists of two phases: a preparation phase and a decision phase. Each phase requires two communication steps. As the name implies, proposers make proposals with the aim of having a majority of acceptors ($n/2+1$) accepting a value. When that happens, a decision is taken and is never changed, this guarantees safety. In the first phase proposers send a `prepare` message with a proposal number (frequently called *ballot*) to a majority of acceptors, which can include itself. Proposer i uses increasingly larger proposal numbers, starting with i and each time incrementing with n, thus its proposal numbers are: $i, i+n, i+2n, i+3n, ...$ In this way proposal numbers from different proposers are comparable.

A Paxos acceptor only sends a positive reply to the proposer in Phase 1 if the proposal number is the largest it has received. If the proposer does not get a majority of positive replies, it sends another `prepare` message with a larger proposal number. After getting $n/2 + 1$ positive responses to the `prepare` message, the proposer proceeds to Phase 2 of the algorithm. In Phase 2 the proposer sends an `accept` message to the majority of acceptors, now with the value it wants to get them to accept, plus the proposal number that succeeded in the first phase. Now, an acceptor only sends a positive reply – an accepts a value – if meanwhile it has not received another message (either `prepare` or `accept`) with an even larger proposal number. In that case it returns that number to the proposer, which will try it all again with an even higher proposal number. But there is a final catch: if an acceptor has accepted a value, it must return that value to the proposer with the corresponding proposal number. The proposer now must adopt the value with the highest number as the one it will try acceptors to accept. The idea is that after a majority of acceptors accept a value, that instance of consensus has decided, nothing will change that value, and thus safety is guaranteed.

Closely related to the algorithm we propose in this paper, NOPaxos (Network Ordered Paxos) [Li *et al.*, 2016] was the first consensus protocol to make use of the network layer to improve the performance of consensus. NOPaxos uses a sequencer at the network layer (*e.g.,* implemented in a switch) that assigns sequence numbers to requests, which are then forwarded to the replicas. The replicas deliver and execute requests in the order defined by the sequence numbers and attempt to recover any requests with missing order numbers from the other replicas. During normal execution, requests are executed in only a single round-trip time (Figure 2(a)):



(a) NOPaxos.



(b) NeoBFT.

Figure 2. Normal case execution for NOPaxos and NeoBFT.

a client broadcasts a request r to the replicas through the sequencer; the replicas add r to a log, and a leader replica also executes the operation; all replicas respond to the client with the position p at which r was added in the log, while the leader also sends the response; the client waits for $f + 1$ responses with the same p, including the leader's response, and then gets the final response. The replicas also monitor and replace the sequencer and/or the leader replica in case of failures. As in other algorithms that tolerate only crashes, at least $2f + 1$ replicas are needed to tolerate up to f faulty replicas. Finally, since only the leader replica executes the requests, it is unnecessary to roll back the replicas state. However, if a non-faulty leader replica is replaced due to false suspicions, its requests may not reflect the state of the new leader replica. In this case, the old leader replica updates its state from the new leader instead of performing a rollback.

NeoBFT [Sun *et al.*, 2023] follows a similar approach to that of NOPaxos, but assumes Byzantine faults, *i.e.*, it is intrusion-tolerant. Among the Byzantine fault-tolerant consensus protocols, PBFT (Practical Byzantine Fault Tolerance) is the first to consider practical aspects [Castro and Liskov, 1999, 2002]. PBFT is executed in three phases. PBFT classifies the replicas as primary (which can be seen as a leader) and backups. The primary assigns sequence numbers to proposals, and the backups use these numbers to ensure consistency. A major part of the algorithm is related to tolerating failures of the primary. The backups also monitor the primary replica to detect crashes or malicious behavior.

In order to tolerate malicious failures, in NeoBFT the sequencer signs each request with its assigned sequence number,

resulting in an authenticated ordering service at the network layer. This approach allows a replica to recover lost requests from other replicas by simply checking whether the corresponding signature is valid. Since a malicious sequencer can assign the same sequence number to different requests, this algorithm requires an additional communication step (as shown in Figure 2(b)) to guarantee that at least $2f + 1$ replicas have received the same information. When this occurs, the replicas execute the request, and the response is sent to the client, which waits for $2f + 1$ identical responses to complete its operation.

Thus, NeoBFT requires $3f + 1$ replicas to tolerate up to f malicious failures. Furthermore, in certain cases, it is necessary to insert a gap "request" to ensure system progress. When this occurs, or due to a sequencer change, some replicas may need to roll back their state. Finally, some alternatives for authentication at the network layer have been proposed in the original paper, such as coupling a co-processor capable of generating signatures to the switch or using an HMAC vector implemented directly in the switch.

In terms of not very closely related works, several algorithms have been proposed to improve Paxos and PBFT from different perspectives. For instance, MinBFT [Veronese *et al*., 2013] uses secure components to reduce both the number of replicas and the number of communication steps. Other algorithms have been proposed to address the coexistence of multiple replicas [Saramago *et al*., 2018] or to consider systems in which the number of replicas is initially unknown [Alchieri *et al*., 2018]. All of these algorithms order requests by exchanging messages at the application layer.

To complete with a final definition, consensus algorithms usually keep a *log* as an ordered, durable record of state changes that reflect the state of replicated data. As all replicas keep the replicated log, the algorithm ensures that all servers process the same sequence of operations in the same order, effectively acting as a single replicated state machine.

4 The NsoBFT Consensus Protocol

This section presents the NsoBFT (Network secure ordered BFT) consensus protocol. NsoBFT is a consensus algorithm that, similar to NOPaxos and NeoBFT, employs both the network and application layers in order to provide an efficient implementation of distributed fault-tolerant consensus. Different from the other alternatives, NsoBFT uses the USIG (Unique Sequential Identifier Generator) component [Veronese *et al*., 2013] to securely generate sequence numbers, *i.e.*, a malicious sequencer cannot assign the same sequence number to different messages. Unique sequence numbers ensure that requests are safely ordered. Consequently, NsoBFT eliminates the additional communication step required by NeoBFT and presents message exchange complexity equivalent to that of NOPaxos, which only tolerates crash failures.

4.1 System Model

NsoBFT assumes a standard partially synchronous distributed system model with an unbounded set of client processes and up to f Byzantine faulty replicas among a total of $n = 3f + 1$ replicas or server processes. A Byzantine replica may behave arbitrarily, either by sending conflicting messages, omitting messages, colluding with other faulty replicas, or attempting to disrupt protocol progress. A process is considered correct if it does not fail; otherwise, it is considered faulty. Processes have unique identifiers, and it is impossible to obtain additional identifiers to launch a Sybil attack [Douceur, 2002].

The system is partially synchronous and we assume the GST model [Dwork *et al*., 1988; Bravo *et al*., 2022], meaning that messages may be delayed or reordered arbitrarily, but after some unknown Global Stabilization Time (GST), message delays are bounded and the system becomes and remains synchronous forever. The adversary may control the scheduling of messages before GST but cannot permanently prevent communication among correct processes. We note that most works that solve consensus deterministically require a partially synchronous model to ensure termination.

Processes communicate by sending and receiving messages. Clients multicast their requests to replicas, which send responses back to the client over a point-to-point channel, and communicate with each other as needed, also employing point-to-point channels. Communication channels are secure, in the sense that messages are authenticated and cannot be corrupted. Furthermore, channels are fair-loss, *i.e.*, if a message is repeatedly sent from a correct source process to a correct destination process, it will be eventually delivered. However, the network layer can lose, duplicate, or reorder messages.

The critical security assumption underlying NsoBFT concerns the in-network sequencer. In this work, the sequencer is implemented as a Virtual Network Function (VNF) and incorporates a USIG (Unique Sequential Identifier Generator) module, which is tamper-resistant and capable of producing monotonically increasing, non-equivocating sequence numbers. Specifically, by using the USIG module, the sequencer **(i)** cannot assign the same sequence number to different messages, **(ii)** cannot assign different sequence numbers to the same message, **(iii)** cannot roll back or reuse previously issued sequence numbers, and **(iv)** all sequence numbers are cryptographically bound to the corresponding message.

Most important, the sequencer is not required to be trusted for correctness of ordering decisions beyond the guarantees provided by USIG. Thus, if the sequencer crashes, it stops issuing sequence numbers but does not violate safety. If the sequencer behaves maliciously outside the guarantees enforced by USIG (e.g., by selectively withholding messages), it may impact liveness but cannot compromise agreement or total order among correct replicas. In these cases, NsoBFT executes a view change and the sequencer is replaced. We assume that there is at least one correct sequencer in the network.

We assume that cryptographic primitives (e.g., digital signatures) are secure against forgery and collision. Adversaries are computationally bounded and cannot break standard cryptographic assumptions. This threat model captures realistic adversarial conditions in modern distributed infrastructures while enabling the reduction of replica coordination achieved by NsoBFT.

4.2 USIG: Unique Sequential Identifier Generator

USIG is a secure service that assigns a unique identifier and signs each message. The identifiers are: **(i)** unique (the same identifier is never assigned to two or more messages); **(ii)** monotonic (the identifier assigned to a message is never smaller than the previous one); and **(iii)** sequential (the identifier assigned to a message is always the successor of the previous one). A USIG module must be present in the switch which processes and forwards messages with the identifiers. The interface defined to access the USIG service is as follows [Veronese *et al.*, 2013]:

- `createUI(m)`: receives as parameter a message `m` and returns a `UI` certificate containing the unique identifier (`UI.id`) and the proof (`UI.proof`) that the identifier was created by this component and assigned to `m`. As mentioned above, the identifier is a monotonically increasing counter that increments with each call to this function. The proof is a digital signature that consists of the message hash that is encrypted with asymmetric cryptography. The private key used to generate this signature is securely stored within this component. The proof only requires the corresponding public key to be verified.
- `verifyUI(PK,UI,m)`: verifies whether the unique identifier `UI` is valid for message `m`. This function receives as a parameter the public key `PK`, associated with the respective USIG instance, to verify whether the signature contained in `UI.proof` is valid for `UI.id` and `m`.

Using the USIG component, a sequencer cannot send two different messages with the same identifier to different processes. Thus, each process only needs to store the identifier of the last message received from the sequencer in order to anticipate the expected identifier for the next message. Therefore, the actions of a malicious sequencer are limited to either sending the same message (containing any value) to all processes or sending nothing.

4.3 NsoBFT Consensus

NsoBFT is presented in two parts: the normal execution and the view changing protocol.

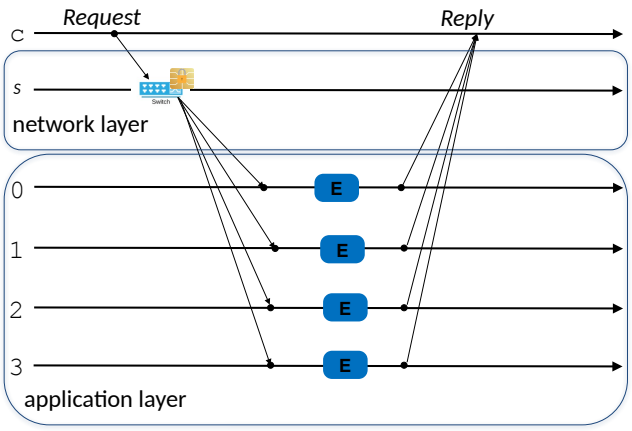


Figure 3. NsoBFT normal case execution.

Normal Execution. A normal execution of NsoBFT follows the message pattern shown in Figure 3. Note that the USIG service at the network layer is instantiated only within the sequencer. The sequencer itself could be placed at a programmable switch or as a VNF [Venâncio *et al.*, 2022] that connects all replicas. We note that both technologies have been used to implement consensus algorithms [Dang *et al.*, 2020; Venâncio *et al.*, 2019, 2021b].

The normal case execution protocol works as follows:

- Initially, the client multicast request r to the set of replicas (Algorithm 1, line 2). However, the request must pass through the sequencer before it reaches the replicas.
- The sequencer uses the USIG to add an identifier (*i.e.*, the sequence number) to r, executing `createUI(r)` to add the sequence number to the message (Algorithm 2, lines 2-3).
- The replicas receive request r already with the sequence number `UI` and verify whether it (UI) is valid using function `verifyUI(PK,UI,r)` (Algorithm 3, line 7), where `PK` is the public key of the USIG service, which should be widely available and authenticated through a reliable and well-known CA (Certification Authority). If the sequence number passes the authentication, the replicas check if `UI.id` is the next number in the sequence to be delivered. Upon both conditions being met, the replicas execute r, update the log, and send the response back to the client (Algorithm 3, lines 13-19).
- The client waits for $2f + 1$ identical responses to complete the operation (Algorithm 1, lines 3-4).

Algorithm 1 NsoBFT Client Protocol

1: **procedure** invoke(r)
2: multicast r to the replicas
3: wait for $2f + 1$ matching replies $reply$
4: **return** $reply$

Algorithm 2 NsoBFT Sequencer Protocol

1: USIG ← starts the USIG service
2: **upon** intercepting request r
3: r.UI ← USIG.`createUI(r)`

In the normal operation, if a replica does not receive a specific request, the sequence number stored in `UI.id` results in a gap in the delivery order. Thus, the replica must retrieve the missing request(s) from other replicas in order to continue its execution. Note that the replicas store the requests in a log. Since the `UI` is unique for each request, no malicious action can be performed during this procedure (Algorithm 3, lines 9-10, 20-26).

Correctness of normal case execution. We now briefly discuss the correctness of NsoBFT during normal execution, showing that the protocol preserves both safety and liveness.

During normal execution, the sequencer assigns a unique sequence number to every client request and generates a corresponding proof. The USIG guarantees that sequence numbers

Algorithm 3 NsoBFT Replica Protocol

```
1: USIG ← starts the USIG service only for verification
2: PK ← USIG service public key
3: nextExpected ← 0

4: upon receipt of r
5: tryDeliver(r)

6: procedure tryDeliver(r)
7: if not USIG.verifyUI(PK,r.UI,r) then
8:     discard message
9: else if r.UI.id > nextExpected then
10:    send ⟨RecRequest, nextExpected⟩ to all replicas
11: end if
12: pendent ← pendent ∪ {r}
13: while ∃r ∈ pendent : r.UI.id == nextExpected do
14:    execute r
15:    append r to the log
16:    pendent ← pendent \ r
17:    send reply to the corresponding client
18:    nextExpected ← nextExpected + 1
19: end while

20: upon receipt of ⟨RecRequest, id⟩ from replica R_j
21: if ∃r ∈ {pendent ∪ log} : r.UI.id = id then
22:    send ⟨RecReply, r⟩ to R_j
23: end if
24:
25: upon receipt of ⟨RecReply, r⟩
26: tryDeliver(r)
```

are monotonically increasing and cannot be forged or reused. Consequently, two different requests cannot be assigned the same sequence number, and the same request cannot be assigned multiple sequence numbers. Upon receiving a request, replicas verify the unique sequence identifier before executing it. Moreover, requests are executed only when the corresponding sequence number matches the next expected value. If a gap is detected, execution is postponed until the missing request is recovered. Consequently, all correct replicas execute the same requests in the same order, ensuring safety.

Notice if the underlying network protocol allows a malicious sequencer to replay a client request and submit it multiple times to the USIG service as if they were distinct requests, different sequence numbers may be generated for the same client message. However, replicas can detect and discard duplicate requests based on their identifiers assigned and authenticated by the clients. In this case, the discarded duplicate creates a gap in the sequence order, that is eventually detected through the normal timeout mechanism, causing replicas to suspect the sequencer and initiate a view change.

Assuming reliable communication among correct processes, which can be implemented using retransmissions on top of fair-loss links, every client request multicast to the system is eventually intercepted by the sequencer and received at all correct replicas. Since correct replicas execute every request in sequence order, each request eventually produces a reply. Furthermore, clients complete operations only after collecting $2f + 1$ matching replies, ensuring that at least one reply originates from a correct replica. Because all correct replicas eventually execute every ordered request, a correct client eventually receives the required quorum of replies and completes its operation, ensuring liveness.

View change. If the sequencer is suspected of being faulty, a new sequencer must be chosen. To this end, it is necessary to determine up to which position in the log the requests should be maintained. The proposed view change protocol keeps in the log all requests for which at least $2f + 1$ responses from the replicas have been sent to the clients (*i.e.*, those requests may have been completed successfully at the clients). On the other hand, requests removed from the log are guaranteed not to have completed at the clients.

To determine which requests must be preserved across view changes, replicas execute a traditional Byzantine consensus protocol. We employ a variant of Byzantine consensus that allows processes to decide only values that satisfy an application-defined validity predicate (P) [Vassantlal *et al.*, 2022]. More specifically, a value v can be decided only if $P(v) = \textit{true}$ for at least $f + 1$ correct replicas. This property is easily enforced by requiring correct replicas to accept a consensus proposal only when the proposed value satisfies the predicate P.

In our protocol, the value proposed during consensus corresponds to the log maintained by the leader replica, which is elected as usual for consensus protocols [Bessani *et al.*, 2014]. A replica accepts a proposed log only if the proposal contains all requests already present in its local log. Consequently, the consensus protocol can only decide logs that extend, rather than discard, the state observed by $2f + 1$ replicas.

Formally, the validity predicate is defined as:

$$P(log_{prop}) \equiv \forall r \in log_{local}, r \in log_{prop}$$

where log_{local} denotes the local log of the replica evaluating the proposal and log_{prop} denotes the log proposed by the leader. Therefore, a proposed log is considered valid only if it contains every request stored in the replica's local log. The view change protocol works as follows:

- When a replica suspects that the sequencer is faulty, that replica sends a `VIEW-CHANGE` message to the other replicas.
- Upon receiving a `VIEW-CHANGE` message, a replica retransmits this message to the other replicas.
- When a replica receives $f + 1$ `VIEW-CHANGE` messages, it also suspects the sequencer is faulty and sends a `VIEW-CHANGE` message to the other replicas (if it has not yet been sent).
- When a replica receives $f + 1$ `VIEW-CHANGE` messages, it starts the following consensus execution to determine up to which point the log should be kept. Consequently, it also decides which log entries should be discarded:
  - A leader replica proposes its log.
  - The other replicas use the previous defined predicate P and only accept a proposal if their own logs are not ahead of the proposed log. This ensures that requests executed and replied from at least $2f + 1$ replicas remain in the decided log.

  - If the proposal is not accepted by at least $2f + 1$ replicas, another replica is chosen to become the leader, and the whole process is repeated.
- By the end of the consensus execution, all replicas decide on the same log l_d. After that, each replica checks its own log to execute the requests in l_d that it had missed, or rolls back the requests ahead of l_d.

Finally, note that the proposed protocol assumes deterministic request execution and requires the application state to support rollback of requests that may be discarded during a view change. NeoBFT also relies on rollback mechanisms to deal with resynchronization caused by gaps in the sequence of delivered requests. Although NOPaxos is described as a rollback-free protocol, this property only holds when leader changes occur correctly. If a correct leader is incorrectly suspected and replaced by a view change, operations executed exclusively under the ousted leader may be lost. In such situations, NOPaxos avoids application-level rollback by requiring the ousted leader to transfer the application state from another replica, effectively replacing rollback with state transfer. Therefore, despite differences in the recovery mechanism, all three protocols must be able to recover from situations in which previously executed operations are no longer part of the agreed system history after a view change.

Correctness of view change protocol. We now briefly discuss the correctness of the view-change protocol.

Firstly, we argue that the traditional consensus protocol to decide the log is started only when at least one correct replica suspects that the sequencer is faulty and that, once started by a correct replica, every correct replica will eventually participate in the same consensus instance. For a correct replica to start the consensus, it must have received $f + 1$ `VIEW-CHANGE` messages. Since at most f replicas can be Byzantine, at least one of these messages must have been sent by correct replicas, i.e., it is necessary that a correct replica suspects the sequencer is faulty. In this case, it is also safe for the replica to suspect that the leader is faulty. Moreover, consider a correct replica that started the consensus. In this case, it received $f+1$ `VIEW-CHANGE` messages and also retransmitted these messages, that will eventually be received at all correct replicas. Consequently, all correct replicas also start the consensus instance to decide the log.

Furthermore, any request that may have completed at a client is guaranteed to remain in the decided log. A client only completes an operation after receiving $2f + 1$ matching replies, which implies that at least $2f + 1$ replicas executed the corresponding request and stored it in their logs. Since a replica only accepts a proposed log if it contains all requests present in its local log (predicate P), any proposal that omits such a request would be rejected by at least $f + 1$ correct replicas. Therefore, the consensus protocol can only decide a log containing all requests that may have completed at clients, ensuring that no externally visible operation is lost during a view change.

Additional considerations. A malicious sequencer may choose to discard requests from specific clients. As in other implementations (*e.g.*, BFT-SMaRt [Bessani *et al*., 2014]), to avoid such a problem, clients must also send requests to the replicas, which will wait to receive its sequence number from the sequencer. A timeout is employed as a bound to the waiting interval. If the timeout expires at some replica, that replica itself must send the request as if it were the client. A new timeout interval is triggered, and if it also expires, the replica suspects the sequencer and proposes a view change.

4.4 A Discussion on NsoBFT Variations

Now we discuss two variations of NsoBFT both of which reduce the required number of replicas from $3f + 1$ to only $2f + 1$ (Table 1). Before the variations are described, it is important to recall that requests that have already completed should not be removed from the log during a view change. The variations must guarantee that this happens with f fewer replicas. The first variation is called MinNsoBFT and the second MinZyzNsoBFT:

- **MinNsoBFT**: This variant works similarly to MinBFT [Veronese *et al*., 2013], requiring an additional communication step before executing a request. Quorums are formed with only $f + 1$ replicas. However, the additional step ensures that messages answered by at least $f + 1$ replicas will remain in the log during a view change.
- **MinZyzNsoBFT**: This variant works similarly to MinZyzzyva [Veronese *et al*., 2013]. In this case, clients must wait for all $2f + 1$ responses to complete an operation. In executions where the network is slow (and responses do not arrive before a timeout expires), or there are malicious replicas, an additional step is necessary: the client sends a message to all replicas and waits for the responses. These messages aim to ensure that at least $f + 1$ replicas execute all requests in order.

Note that, for both variants, the consensus protocol used for view change must tolerate Byzantine faults in a system with only $2f + 1$ replicas. An example of a protocol that can be used is MinBFT [Veronese *et al*., 2013].

Notice that the reduction from $3f + 1$ to $2f + 1$ replicas presents important trade-offs. Both variants introduce additional requirements that move part of the coordination cost to the protocol execution path. While NsoBFT finishes a client operation in a single ordered request-reply exchange, MinNsoBFT introduces an additional communication step among replicas before request execution, whereas MinZyzNsoBFT may require an additional client-assisted recovery phase. It is important to note that both variants still benefit from the secure in-network ordering service. Unlike MinBFT and MinZyzzyva, where ordering is established through replica-to-replica communication, the sequencer equipped with USIG provides a globally consistent request order before requests reach the replicas.

The choice between NsoBFT, MinNsoBFT, and MinZyzNsoBFT depends on the target deployment environment. NsoBFT provides the simplest execution model and the lowest coordination overhead, but requires $3f + 1$ replicas. MinNsoBFT and MinZyzNsoBFT reduce the replica count

to $2f + 1$, which may be attractive in environments where replica deployment costs dominate operational expenses. The price paid for this reduction is additional coordination during normal execution or recovery, depending on the selected variant.

5 Experimental Evaluation

This section presents our experimental evaluation of NsoBFT, with the main goal of showing the performance improvement when compared to NeoBFT. Furthermore, these protocols are also compared to NOPaxos to evaluate the impact of Byzantine fault tolerance. In particular, this experimental evaluation aims to quantify the performance impact of introducing secure in-network ordering and to compare NsoBFT against NeoBFT and NOPaxos, with the aim of answer the following main questions:

- What is the overhead introduced by cryptographic operations required for Byzantine fault tolerance?
- Does eliminating the additional replica coordination step (as done in NsoBFT) significantly improve performance over NeoBFT?
- What is the performance gap between crash-tolerant and Byzantine-tolerant in-network consensus?
- How does performance scale with payload size, number of clients, and the number of tolerated faults?
- How does the proposed NsoBFT protocol behave when the sequencer crashes or behaves maliciously?

5.1 Implementation Details

NOPaxos, NeoBFT, and NsoBFT were all implemented in Python to ensure a consistent experimental comparison across protocols. Cryptographic operations rely on RSA-2048 signatures with SHA-256, implemented using the *python-cryptography* library. When signatures are disabled (as in NOPaxos), the protocol operates without cryptographic overhead.

Clients multicast requests to the replicas using UDP, while replica-to-replica communication relies on TCP to ensure reliable delivery of coordination messages in NeoBFT. The ordering service is implemented as a dedicated VNF, which performs packet processing directly in the network layer. It captures client UDP packets using raw sockets and parses the Ethernet, IPv4, UDP headers and the payload. For each packet addressed to the replicas, it inserts the sequencer metadata into the UDP payload, including a monotonically increasing sequence number and, when required, an RSA-2048 signature computed over the header and payload. After inserting this information, the packet headers are updated (e.g. UDP length and IPv4 checksum) and the packet is forwarded to the replica multicast group. Notice this ensures that all replicas receive an identical ordered message. In Byzantine configurations (NeoBFT and NsoBFT), replicas verify the digital signature before proceeding with protocol execution.

NeoBFT requires an additional confirmation step: replicas forward received ordered packets to othet replicas to collect confirmation messages before delivering the request. These confirmations are exchanged via TCP unicast, increasing coordination overhead and extending the critical path. In contrast, NsoBFT eliminates this additional forwarding phase due to the no-equivocation guarantees provided by USIG. Replicas detect missing sequence numbers by tracking gaps in the ordered stream. Upon detecting a gap, a replica requests retransmission from other replicas. Recovery requests and responses are also performed via TCP unicast.

To isolate consensus overhead from application processing costs, we deployed an "empty" service at each replica. This service executes no application logic beyond acknowledging request execution. Consequently, measured performance differences reflect protocol design decisions, replica coordination patterns, network communication, and cryptographic overhead rather than application-specific computation.

5.2 Experimental Setup and Methodology

All experiments were conducted on the Emulab testbed [White *et al*., 2002]. We used *d430* machines equipped with Intel Xeon E5-2630v3 processors (2.4 GHz), 8 cores with 2 hardware threads per core, and 64 GB of RAM. The machines were interconnected through a dedicated 1 Gbps Ethernet switch. The number of machines varied according to the configuration under evaluation.

Each replica executed on a dedicated physical machine to eliminate resource contention. Clients were deployed on a separate machine, with the number of client threads adjusted to stress the system and drive it to saturation. The sequencer was deployed on its own dedicated machine and operated as a bridge between clients and replicas, implementing the NFV-based in-network ordering service. All nodes ran 64-bit Ubuntu 20.04 LTS. Figure 4 illustrates the experimental architecture.

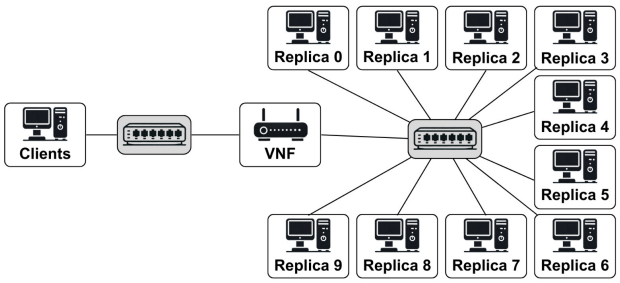


Figure 4. Experimental setup.

We evaluated four payload sizes: 0B, 100B, 512B, and 1kB, allowing us to assess both coordination-dominated and bandwidth-sensitive scenarios. The number of concurrent clients was progressively increased to identify peak throughput and saturation behavior.

Each client issued 10,000 requests during each experimental run. Throughput is reported in thousands of operations per second (kops/sec) and was measured at the replicas to capture the processing capacity of the system. Throughput measurements were collected every 1,000 executed operations, and the reported values correspond to the average and standard deviation across the collected samples. Latency was

measured at the clients as the end-to-end delay between request submission and reply reception. For latency, we report the average, standard deviation, and the 99th percentile. To mitigate the impact of transient fluctuations, the 5% highest and 5% lowest of latency samples were removed prior to computing the average and standard deviation.

To evaluate scalability and fault tolerance overhead, we varied the number of tolerated faults f, adjusting the total number of replicas accordingly. This allowed us to analyze how coordination costs grow with system size and to compare the scalability properties of NsoBFT, NeoBFT, and NOPaxos under increasing resilience requirements.

5.3 Experimental Results

This section presents a detailed evaluation of the proposed approach. We first analyze the cryptographic overhead introduced by Byzantine fault tolerance. We then compare the performance of NsoBFT, NeoBFT, and NOPaxos in terms of throughput and latency under varying payload sizes and client loads. We also evaluate scalability by examining the impact of increasing the number of replicas (i.e., the fault threshold f) on system performance. Finally, we access the impact of sequencer failures in the NsoBFT protocol.

Cryptographic Overhead. In addition to the application payload, each message begins with a fixed 20-byte header structure placed at the start of the message. This header contains an 8-byte sequence number, a 4-byte replica identifier and another 8-bytes indicating the message type (e.g., a new message, a request for missing packet from other replicas, or quorum confirmation in the last step of NeoBFT).

In the NOPaxos protocol, which does not rely on signatures, the total message size corresponds to the payload plus these 20 bytes of protocol metadata. When signatures are enabled (NeoBFT and NsoBFT), a 256-byte RSA-2048 signature is inserted immediately after the header, resulting in a total message size equal to the payload plus 276 bytes. The signature is computed over the header and payload fields, introducing additional cryptographic processing overhead. On average, the sequencer required 0.6486 milliseconds to generate each signature, while replicas spent a fraction of that time, 0.0233 milliseconds, verifying it, which noticeably increases processing cost in Byzantine configurations.

Micro-benckmarks. We begin by presenting the results of micro-benchmarks commonly used to evaluate consensus protocols. These benchmarks employ an "empty" service to isolate protocol overhead, measuring throughput at the replicas and latency at the clients.

Figure 5 shows the average latency *vs.* throughput curves for the three algorithms, considering a system configured to tolerate a single failure and different payload sizes. As expected, NOPaxos achieves the highest throughput and lowest latency, since it tolerates only crash faults, requires fewer replicas, and does not rely on expensive cryptographic operations. For all payloads, NOPaxos exceeds 5 kops/sec, reaching approximately 6 kops/sec for 0B and 100B payloads. In all cases, the latency remains low under moderate load

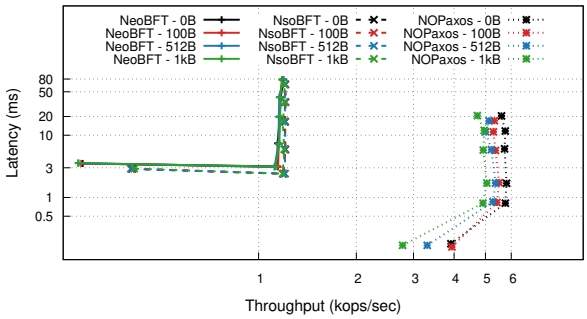


Figure 5. Latency *vs.* throughput, $f = 1$.

and increases sharply as the system approaches saturation and peak throughput, after which additional load impacts only latency. Among the Byzantine fault-tolerant protocols, NsoBFT showed slightly superior performance to NeoBFT because it requires one less communication step. Note that both protocols require the sequencer to sign all messages.

Table 2 shows that the throughput varies considerably between the algorithms. In general, the NOPaxos algorithm presented higher throughputs in all combinations of clients and payloads. This is due to NOPaxos not requiring cryptography and its smaller number of replicas. Likewise, NsoBFT outperforms NeoBFT because it requires one less communication step.

Table 3 shows the latency varies according to the number of clients and the payload size. For the NOPaxos algorithm, latency remains relatively low compared to the other algorithms. For NeoBFT, however, latency increases considerably, especially when the number of clients is high, and the payload is larger. NsoBFT also presents high latencies under similar loads, but generally remains lower than NeoBFT.

Another important observation from tables 2 and 3 is that the payload size has a relatively small effect on performance. For all evaluated protocols, throughput and latency remain largely stable when the payload increases from 0B to 1kB. This behavior can be explained by the fact that client requests are disseminated using UDP multicast (Section 5.1), and even for the largest evaluated payload (1kB), the resulting packets remain below the Ethernet MTU of 1500 bytes, avoiding IP fragmentation. Consequently, increasing the payload size does not introduce additional packet transmissions or reassembly overheads, leading to only minor performance variations. As a result, the number of clients and the protocol coordination pattern have a much stronger influence on throughput and latency than the payload size itself. Furthermore, the performance advantage of NsoBFT over NeoBFT remains consistent across all payload sizes, indicating that the gains stem from the elimination of an additional replica coordination step.

Fault-scalability. Our next experiment evaluates the impact of the number of replicas on system throughput and latency, with 10 clients issuing requests carrying payloads of 1kB bytes. For these experiments, the number of tolerated failures (f) ranged from 1 to 3 failures.

Figure 6 presents the throughput as the number of repli-

Protocol	# of clients	0B	100B	512B	1kB
NoPaxos	1	3.91 ± 0.06	3.94 ± 0.12	3.30 ± 0.20	2.77 ± 0.47
	5	5.75 ± 0.50	5.42 ± 0.40	5.26 ± 0.22	4.91 ± 0.31
	10	5.80 ± 0.15	5.50 ± 0.17	5.36 ± 0.12	5.04 ± 0.20
	25	5.72 ± 0.27	5.36 ± 0.30	5.21 ± 0.33	4.92 ± 0.30
	50	5.75 ± 0.09	5.28 ± 0.09	5.00 ± 0.08	4.95 ± 0.15
	100	5.60 ± 0.19	5.32 ± 0.29	5.12 ± 0.09	4.71 ± 0.32
NeoBFT	1	0.28 ± 0.0006	0.28 ± 0.0007	0.27 ± 0.0006	0.27 ± 0.0007
	5	1.14 ± 0.020	1.14 ± 0.025	1.11 ± 0.011	1.12 ± 0.084
	10	1.14 ± 0.050	1.15 ± 0.048	1.15 ± 0.025	1.15 ± 0.063
	25	1.16 ± 0.043	1.15 ± 0.058	1.15 ± 0.081	1.16 ± 0.021
	50	1.16 ± 0.042	1.16 ± 0.083	1.15 ± 0.085	1.16 ± 0.079
	100	1.19 ± 0.057	1.18 ± 0.079	1.19 ± 0.008	1.17 ± 0.061
NsoBFT	1	0.40 ± 0.0006	0.40 ± 0.0007	0.40 ± 0.0006	0.41 ± 0.0006
	5	1.18 ± 0.084	1.19 ± 0.116	1.20 ± 0.956	1.18 ± 0.690
	10	1.20 ± 0.068	1.20 ± 0.006	1.20 ± 0.065	1.19 ± 0.010
	25	1.20 ± 0.032	1.20 ± 0.046	1.20 ± 0.045	1.18 ± 0.031
	50	1.20 ± 0.022	1.21 ± 0.025	1.20 ± 0.002	1.20 ± 0.022
	100	1.20 ± 0.002	1.20 ± 0.037	1.20 ± 0.026	1.20 ± 0.019

Table 2. Average throughput and standard deviation in kops/sec for different numbers of clients and payload sizes ($f = 1$).

		0B		100B		512B		1kB	
Protocol	# of clients	99th	avg $\pm$ sd	99th	avg $\pm$ sd	99th	avg $\pm$ sd	99th	avg $\pm$ sd
NoPaxos	1	0.35	0.18 ± 0.05	0.35	0.16 ± 0.05	0.36	0.17 ± 0.05	0.39	0.17 ± 0.05
	5	1.86	0.81 ± 0.24	1.84	0.83 ± 0.26	2.03	0.85 ± 0.27	1.94	0.81 ± 0.22
	10	4.05	1.69 ± 0.51	3.62	1.73 ± 0.55	3.64	1.70 ± 0.51	3.62	1.70 ± 0.50
	25	16.32	6.03 ± 3.34	16.84	5.69 ± 2.62	16.23	5.87 ± 2.96	15.56	5.77 ± 2.71
	50	32.57	11.69 ± 5.39	33.55	11.33 ± 5.06	33.58	11.33 ± 5.07	33.80	11.93 ± 5.61
	100	48.16	20.45 ± 7.72	47.83	17.14 ± 5.27	48.28	17.08 ± 5.17	47.75	20.65 ± 7.54
NeoBFT	1	4.69	3.52 ± 0.90	4.70	3.52 ± 0.76	4.78	3.57 ± 0.75	4.79	3.56 ± 0.73
	5	5.24	3.14 ± 0.19	5.23	3.10 ± 0.17	9.12	3.09 ± 0.17	6.43	3.14 ± 0.16
	10	9.81	7.42 ± 0.24	10.63	7.26 ± 0.25	9.84	7.29 ± 0.24	10.21	7.26 ± 0.25
	25	23.72	19.87 ± 0.24	23.65	19.75 ± 0.24	24.63	19.86 ± 0.27	24.79	19.71 ± 0.23
	50	46.86	40.99 ± 0.32	45.85	40.91 ± 0.33	47.83	40.84 ± 0.35	48.85	41.19 ± 0.33
	100	88.27	78.56 ± 0.41	88.41	78.34 ± 0.48	87.48	78.74 ± 0.43	88.13	78.63 ± 0.44
NsoBFT	1	3.39	2.88 ± 0.90	3.33	2.88 ± 0.76	3.34	2.89 ± 0.75	3.26	2.90 ± 0.73
	5	5.18	2.40 ± 0.21	5.14	2.42 ± 0.23	5.17	2.41 ± 0.21	4.80	2.41 ± 0.20
	10	9.15	5.92 ± 0.23	8.81	5.91 ± 0.24	8.83	5.94 ± 0.24	9.70	5.94 ± 0.24
	25	23.43	16.49 ± 0.20	23.16	16.50 ± 0.20	23.14	16.54 ± 0.20	23.60	16.58 ± 0.20
	50	42.38	34.11 ± 0.27	45.86	34.13 ± 0.27	46.30	34.22 ± 0.26	42.57	34.29 ± 0.26
	100	78.55	65.84 ± 0.27	82.88	65.87 ± 0.29	78.98	66.03 ± 0.29	79.30	66.16 ± 0.29

Table 3. 99th percentile, average (avg) and standard deviation (sd) of latency (ms) for different numbers of clients and payload sizes ($f = 1$).

cas grows (i.e., as f increases). As expected, throughput decreases for all protocols as the system size expands, since tolerating more faults requires additional replicas, leading to higher communication and processing overhead. Nevertheless, NsoBFT consistently outperforms NeoBFT across all configurations. The performance gap becomes more evident as f increases, demonstrating that eliminating the extra coordination step in NsoBFT improves scalability. In contrast, NOPaxos maintains higher throughput overall due to its crash-only fault model and lower replica requirement.

Figure 7 shows the corresponding latency results. Latency increases with the number of replicas for all protocols, reflecting the additional communication paths and signature verification costs. However, NsoBFT exhibits consistently lower latency than NeoBFT, particularly at higher values of f, confirming that reducing replica coordination shortens the critical execution path. As expected, NOPaxos achieves the lowest latency, since it does not incur Byzantine-specific cryptographic overhead.

In general, the results indicate that NsoBFT scales more efficiently than NeoBFT as the fault threshold f increases, narrowing the performance gap between crash-tolerant and Byzantine-tolerant in-network consensus. The performance degradation observed in NeoBFT as the number of replicas

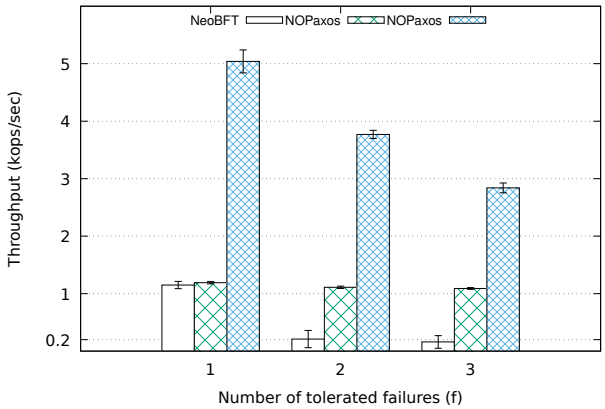


Figure 6. Throughput for payloads of 1k bytes, 10 clients, and different number of tolerated failures (f).

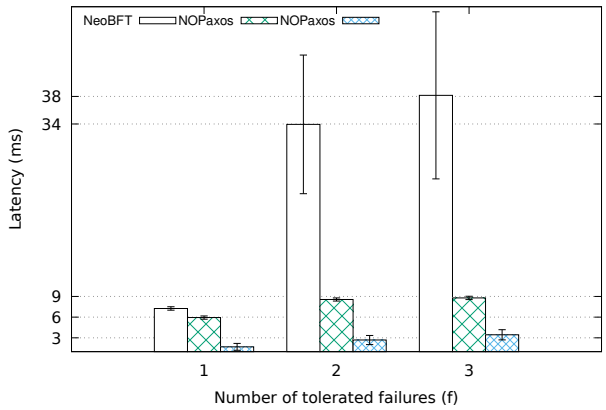


Figure 7. Latency for payloads of 1k bytes, 10 clients, and different number of tolerated failures (f).

increases is not solely due to the presence of an additional communication step, but also to how the cost of this step scales with the system size. In NeoBFT, each request must be forwarded among replicas so that confirmation messages can be collected before execution. As the number of replicas grows, the number of confirmations that must be transmitted, received, and processed also increases. Notice the performance difference between configurations with 4, 7, and 10 replicas is not necessarily linear. Moving from 4 to 7 replicas increases the size of the confirmation quorum and the number of replica-to-replica interactions required in the additional coordination phase. Moving from 7 to 10 replicas further increases these interactions, but the relative impact may differ because the system is already operating in a regime where communication and cryptographic processing dominate the execution cost.

Another interesting observation from these experiments is that the performance of NsoBFT is affected much less by the increase in the number of replicas than NeoBFT. This does not mean that NsoBFT is completely independent of the system size. Rather, the main costs in NsoBFT remain concentrated in the sequencer operations, signature generation and verification, whose costs do not grow with the number of replicas but dominate the overall execution costs in the evaluated configurations. In particular, client requests are disseminated using UDP multicast (Section 5.1). Consequently, the sequencer transmits a single ordered packet regardless the number of replicas in the system. The small performance degradation observed in NsoBFT is mainly associated with the larger number of replies that clients must collect before completing an operation. As the number of tolerated faults increases, the system requires more replicas and, consequently, larger quorums.

Finally, note that the relative performance degradation caused by increasing the number of replicas tends to be more pronounced in NOPaxos than in NsoBFT. Since NOPaxos does not incur the computational overhead of signature generation and verification, communication-related costs represent a larger fraction of its execution time. Consequently, the increase in reply traffic resulting from additional replicas becomes more noticeable. Although clients only wait for the first reply, every replica still executes the request and transmits a response, increasing both network utilization and processing overhead. In contrast, in NsoBFT these communication costs are partially masked by the cryptographic processing costs that dominate the execution time in the evaluated configurations.

Behavior under failures. Our final experiment evaluates the behavior of NsoBFT under sequencer failures. Figures 8 and 9 illustrate the evolution of throughput and latency over time, reporting average values measured every second during a 300-second execution. The experiment was performed with 10 clients issuing requests carrying 100B payloads in a system configured with four replicas ($f = 1$). The deployment included three sequencers, allowing the system to perform successive view changes and activate backup sequencers whenever the current sequencer failed.

As the clients started their execution, the system's throughput stabilizes at approximately 1.20 kops/sec with a latency of approximately 6 milliseconds. We consider two failure scenarios. In the first scenario, at second 100, the sequencer crashes and stops assigning sequence numbers to client requests. When replicas receive requests that should contain a valid sequence number but arrive without one, they immediately suspect a sequencer failure and start a view-change procedure to replace the faulty sequencer. This process causes a temporary drop in throughput and a corresponding increase in latency while the new sequencer is elected. However, the protocol quickly recovers and the system promptly resumes stable operation under the new sequencer.

In the second failure scenario, at second 200, the sequencer behaves maliciously by creating gaps in the sequence order, skipping some sequence numbers. This behavior can occur if the sequencer invokes the USIG multiple times for the same request, artificially advancing the sequence counter without forwarding the corresponding ordered requests to the replicas. Notice that even if the sequencer forwards some correctly ordered requests to a subset of replicas, the skipped sequence numbers will eventually create a gap in the sequence order when replicas wait for the next expected sequence number. The same behavior is observed if the sequencer selectively avoids proposing an order for requests issued by a specific client. As a result, eventually a timeout expires (20 seconds) at the replicas that suspect malicious behavior from the sequencer and initiate a view change. Between the occurrence of the fault and the election of a new sequencer, no sequencer is available to assign new sequence numbers, effectively causing a temporary pause in request processing. Once a new sequencer is elected, normal operation resumes and the system quickly recovers its pre-failure performance levels.

6 Conclusion

Consensus is a central abstraction for building reliable distributed systems, especially in environments prone to intrusions and failures. In this work we introduced the NsoBFT algorithm that employs the network layer with an USIG module to improve the performance of Byzantine consensus. Secure and unique sequence identifiers are used to mitigate the impact of malicious sequencers. The experimental evaluation provided insights into the trade-offs between performance and resilience, as crash- and intrusion-tolerant algorithms

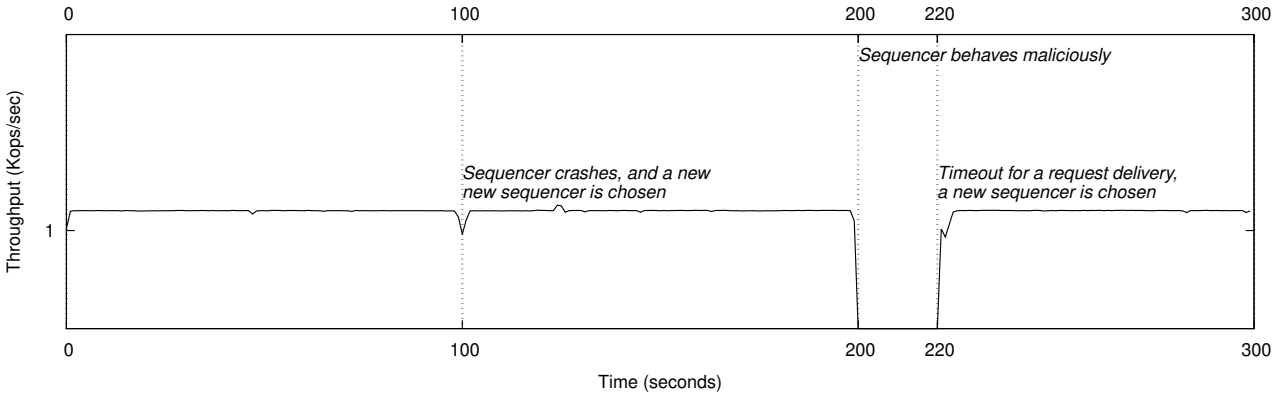


Figure 8. Throughput evolution across time and events, for payloads of 100B, 10 clients and $f = 1$.

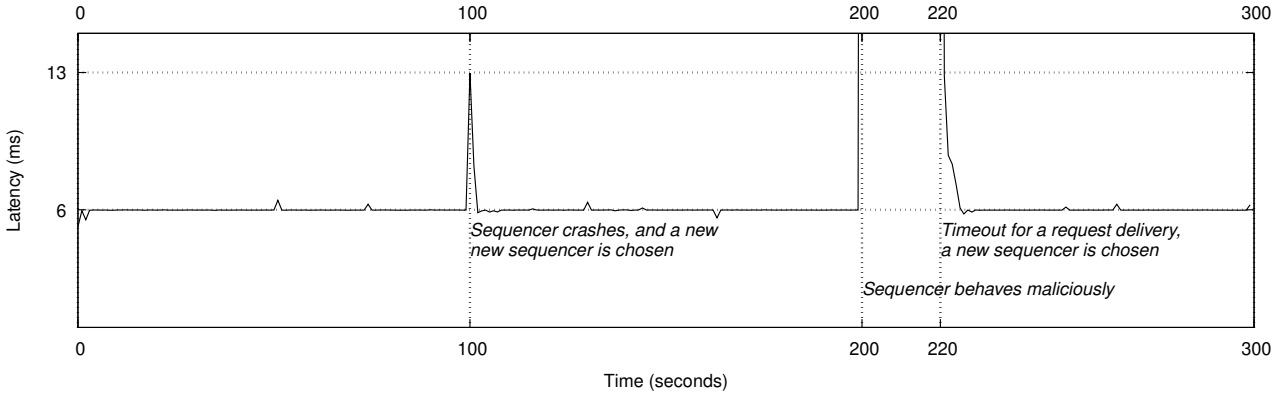


Figure 9. Latency evolution across time and events, for payloads of 100B, 10 clients and $f = 1$.

were compared. Results also confirm that NsoBFT, on the other hand, proved to be an efficient algorithm, making it an attractive option for critical applications that require high availability and data integrity.

Future work involves the implementation and evaluation of the proposed sequencer in programmable switches. Other future work includes the implementation and evaluation of MinNsoBFT and MinZyzNsoBFT, the two variants of NsoBFT presented in the paper.

Declarations

Acknowledgements

This journal version substantially extends our previous conference paper published at LADC 2025 da Rocha *et al*. [2026].

Funding

This work was partially supported by the Brazilian Research Council (CNPq – Conselho Nacional de Desenvolvimento Científico e Tecnológico) grants 305108/2025-5 and 406993/2025-4.

Authors' Contributions

All the authors developed the work as a whole in a collaborative effort.

Competing interests

The authors declare that they have no conflicts of interest.

Availability of data and materials

The implementation of the protocols developed in this study is available in the following repository: `https://github.com/Faustino27/VNF_NsoBFT`

References

Alchieri, E. A. P., Bessani, A., Greve, F., and Fraga, J. d. S. (2018). Knowledge connectivity requirements for solving byzantine consensus with unknown participants. *IEEE Transactions on Dependable and Secure Computing*, 15(2):246–259. DOI: 10.1109/tdsc.2016.2548460.

Bessani, A., Sousa, J., and Alchieri, E. E. P. (2014). State machine replication for the masses with bft-smart. In *International Conference on Dependable Systems and Networks*, pages 355–362. IEEE. DOI: 10.1109/dsn.2014.43.

Boubendir, A., Bertin, E., and Simoni, N. (2018). Flexibility and dynamicity for open network-as-a-service: From vnf and architecture modeling to deployment. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–6. IEEE. DOI: 10.1109/noms.2018.8406135.

Bravo, M., Chockler, G., and Gotsman, A. (2022). Making

byzantine consensus live. *Distributed Computing*, 35(6). DOI: 10.1007/s00446-022-00432-y.

Burrows, M. (2006). The chubby lock service for loosely coupled distributed systems. In *The 7th Symposium on Operating Systems Design and Implementation*. DOI: 10.5555/1298455.1298487.

Cachin, C., Kursawe, K., Petzold, F., and Shoup, V. (2001). Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer. DOI: 10.1007/3-540-44647-8$_3$1.

Castro, M. and Liskov, B. (1999). Practical byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation*, pages 173–186. USENIX. DOI: 10.5555/296806.296824.

Castro, M. and Liskov, B. (2002). Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461. DOI: 10.1145/571637.571640.

da Rocha, G. F. L., Alchieri, E. A. P., Venâncio, G., Fulber-Garcia, V., and Duarte Jr., E. P. (2026). Byzantine consensus with secure and intrusion-tolerant in-network ordering. In Rodrigues, L. A. and Oliveira, R., editors, *Latin American Dependable and Secure Computing (LADC)*, pages 148–162, Cham. Springer Nature Switzerland. DOI: 10.1007/978-3-032-11539-3$_9$.

Dang, H. T., Bressana, P., Wang, H., Lee, K. S., Zilberman, N., Weatherspoon, H., Canini, M., Pedone, F., and Soulé, R. (2020). P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking*, 28(4):1726–1738. DOI: 10.1109/tnet.2020.2992106.

Douceur, J. R. (2002). The sybil attack. In *International Workshop on Peer-to-Peer Systems*, pages 251–260. Springer. DOI: 10.1007/3-540-45748-8$_2$4.

Dwork, C., Lynch, N. A., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–322. DOI: 10.1145/42282.42283.

ETSI Industry Specification Group (ISG) NFV (2024). Network functions virtualisation (nfv) release 4; management and orchestration; architectural framework specification. Group Specification GS NFV 006 v4.5.1, European Telecommunications Standards Institute (ETSI). Available at:`https://www.etsi.org/deliver/etsi_gs/NFV/001_099/006/04.05.01_60/gs_NFV006v040501p.pdf`.

Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382. DOI: 10.1145/3149.214121.

Freitas, A. E. S., Rodrigues, L. A., and Duarte Jr, E. P. (2024). vcubechain: A scalable permissioned blockchain. *Ad Hoc Networks*, 158:103461. DOI: 10.1016/j.adhoc.2024.103461.

Fulber-Garcia, V., Duarte Jr, E. P., Huff, A., and dos Santos, C. R. (2020). Network service topology: Formalization, taxonomy and the custom specification model. *Computer Networks*, 178:107337. DOI: 10.1016/j.comnet.2020.107337.

Hadzilacos, V. and Toueg, S. (1994). A modular approach to the specification and implementation of fault-tolerant broadcasts. Technical report, Department of Computer Science, Cornell University, New York - USA. Available at:`https://people.cs.umass.edu/~arun/cs677/reading/HT.pdf`.

Halpern, J. M. and Pignataro, C. (2015). Service Function Chaining (SFC) Architecture. (7665). DOI: 10.17487/RFC7665.

Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. (2010). {ZooKeeper}: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*. Available at:`https://www.usenix.org/conference/usenix-atc-10/zookeeper-wait-free-coordination-internet-scale-systems`.

J. C. Corbett, J. D. and et al, M. E. (2012). Spanner: Google's globally distributed database. In *The 10th Symposium on Operating Systems Design and Implementation*. DOI: 10.1145/2491245.

Kaur, K., Mangat, V., and Kumar, K. (2022). A review on virtualized infrastructure managers with management and orchestration features in nfv architecture. *Computer Networks*, 217:109281. DOI: 10.1016/j.comnet.2022.109281.

Lamport, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169. DOI: 10.1145/279227.279229.

Li, C., Qiu, W., Li, X., Liu, C., and Zheng, Z. (2024). A dynamic adaptive framework for practical byzantine fault tolerance consensus protocol in the internet of things. *IEEE Transactions on Computers*, 73(7):1669–1682. DOI: 10.1109/tc.2024.3377921.

Li, J., Michael, E., Sharma, N. K., Szekeres, A., and Ports, D. R. (2016). Just say {NO} to paxos overhead: Replacing consensus with network ordering. In *Symposium on Operating Systems Design and Implementation*, pages 467–483. USENIX. DOI: 10.5555/3026877.3026914.

Liu, X. and Yu, W. (2024). A review of research on blockchain consensus mechanisms and algorithms. In *International Conference on Intelligent Informatics and Biomedical Sciences*, volume 9, pages 1–10. DOI: 10.1109/iciibms62405.2024.10792685.

Mijumbi, R., Serrat, J., Gorricho, J.-L., Bouten, N., De Turck, F., and Boutaba, R. (2016). Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys & Tutorials*, 18(1):236–262. DOI: 10.1109/COMST.2015.2477041.

Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *USENIX annual technical conference (USENIX ATC 14)*, pages 305–319. Available at:`https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro`.

Saramago, R. Q., Alchieri, E. A., Rezende, T. F., and Camargos, L. (2018). On the impossibility of byzantine collision-fast atomic broadcast. In *International Conference on Advanced Information Networking and Applications*, pages 414–421. IEEE. DOI: 10.1109/aina.2018.00069.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319. DOI: 10.1145/98163.98167.

Singh, A., Kumar, G., Saha, R., Conti, M., Alazab, M., and Thomas, R. (2022). A survey and taxonomy of consensus protocols for blockchains. *Journal of Systems Architecture*, 127:102503. DOI: 10.1016/j.sysarc.2022.102503.

Sousa, J. and Bessani, A. (2012). From byzantine consensus to bft state machine replication: A latency-optimal transformation. In *Proceedings of the 9th European Dependable Computing Conference (EDCC)*, pages 37–48. IEEE. DOI: 10.1109/EDCC.2012.32.

Sun, G., Jiang, M., Khooi, X. Z., Li, Y., and Li, J. (2023). NeoBFT: Accelerating byzantine fault tolerance using authenticated in-network ordering. In *ACM Special Interest Group on Data Communications Conference*, page 239–254, New York, NY, USA. ACM. DOI: 10.1145/3603269.3604874.

Vassantlal, R., Alchieri, E., Ferreira, B., and Bessani, A. (2022). Cobra: Dynamic proactive secret sharing for confidential bft services. In *Symposium on Security and Privacy*, pages 1335–1353. IEEE. DOI: 10.1109/sp46214.2022.9833658.

Venâncio, G., Fulber-Garcia, V., Flauzino, J., Alchieri, E. A., and Duarte, E. P. (2024). Dependable virtual network services: An architecture for fault-and intrusion-tolerant sfcs. In *Conference on NFV and SDN*, pages 1–6. IEEE. DOI: 10.1109/nfv-sdn61811.2024.10807480.

Venâncio, G., Garcia, V. F., da Cruz Marcuzzo, L., Tavares, T. N., Franco, M. F., Bondan, L., Schaeffer-Filho, A. E., Paula dos Santos, C. R., Granville, L. Z., and P. Duarte Jr, E. (2021a). Beyond vnfm: Filling the gaps of the etsi vnf manager to fully support vnf life cycle operations. *International Journal of Network Management*, 31(5):e2068. DOI: 10.1002/nem.2068.

Venâncio, G., Turchetti, R. C., Camargo, E. T., and Duarte Jr, E. P. (2021b). Vnf-consensus: A virtual network function for maintaining a consistent distributed software-defined network control plane. *International Journal of Network Management*, 31(3). DOI: 10.1002/nem.2124.

Venâncio, G., Turchetti, R. C., and Duarte, E. P. (2019). Nfv-rbcast: Enabling the network to offer reliable and ordered broadcast services. In *The 9th Latin-American Symposium on Dependable Computing*, pages 1–10. IEEE. DOI: 10.1109/ladc48089.2019.8995681.

Venâncio, G., Turchetti, R. C., and Duarte Jr, E. P. (2022). Nfv-coin: unleashing the power of in-network computing with virtualization technologies. *Journal of Internet Services and Applications*, 13(1):46–53. DOI: 10.5753/jisa.2022.2342.

Veronese, G. S., Correia, M., Bessani, A. N., Lung, L. C., and Verissimo, P. (2013). Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30. DOI: 10.1109/tc.2011.221.

Vukolić, M. (2015). The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In *IFIP WG 11.4 International Workshop Open Problems in Network Security*, pages 112–125. Springer. DOI: 10.1007/978-3-319-39028-4_9.

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. (2002). An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Rev.*, 36(SI):255–270. DOI: 10.1145/844128.844152.

Zou, Y., Yang, L., Jing, G., Zhang, R., Xie, Z., Li, H., and Yu, D. (2024). A survey of fault tolerant consensus in wireless networks. *High-Confidence Computing*, 4(2). DOI: 10.1016/j.hcc.2024.100202.