Byzantine Consensus with Secure and Intrusion-Tolerant In-Network Ordering

Gabriel Faustino Lima da Rocha[1], Eduardo A. P. Alchieri[1], Giovanni Venâncio[2], Vinicius Fulber-Garcia[2], and Elias P. Duarte Jr.[2]

[1] Universidade de Brasília (UnB)
Brasília - Distrito Federal - Brazil
[2] Universidade Federal do Paraná (UFPR)
Curitiba - Paraná - Brazil

Abstract. Recently proposed consensus protocols make use of the network layer to ensure agreement, while the application layer is still used for termination. Those protocols employ the network layer as a sequencer that delivers ordered messages facilitating agreement. However, tolerating a malicious sequencer is a challenging task. For instance, a malicious sequencer can assign the same sequence number to different messages and send them to different replicas. To avoid this problem, the NeoBFT consensus protocol adds an additional communication step at the replicas. Although this approach mitigates the problem, it negatively impacts system performance by requiring an additional synchronization step among replicas before executing requests. This work proposes NsoBFT (Network Secure Ordered BFT), a consensus protocol that uses a secure message ordering service implemented with USIG (Unique Sequential Identifier Generator), a secure component in the network layer. Thus, no additional synchronization step is necessary for the replicas to execute requests. Experimental results comparing NsoBFT with related work show the advantage of this strategy, in particular confirming that NsoBFT outperforms NeoBFT.

Keywords: Consensus · Distributed algorithms · Intrusion tolerance.

1 Introduction

Distributed consensus and agreement algorithms are the basis for solving several distributed systems problems, they have been successfully used in many different applications and contexts (e.g., [4, 13, 14]). In particular, consensus is basis of state machine replication [21], a comprehensive and widely used approach to build fault-tolerant systems. It has recently also received significant attention in the context of blockchains [30, 18, 11], and several other problems [32, 16, 25]. Consensus is required whenever the 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 [15] and Raft [19], 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) [5, 6] 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 [22].

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. However, the NOPaxos (Network Ordered Paxos) algorithm [17] was the first consensus solution to employ a message sequencer module implemented in the network layer to assign unique sequence numbers to messages. Thus the safety properties are guaranteed at this layer while the application layer is responsible only for termination. The main advantage of that approach is high performance. This is very relevant, as consensus algorithms are well known for being expensive. An empirical evaluation of NOPaxos shows that the algorithm presents a latency that is close to the bare network latency.

NeoBFT[23] is another consensus algorithm that employs the network layer, but it tolerates Byzantine faults and requires secure sequence numbers. The network layer in this case provides an authenticated ordering service. However, this algorithm uses an additional communication step between the replicas before executing a request. That is necessary in order 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, in particular due to the requirement that all messages be digitally signed, as well as the need for additional coordination between replicas at the application layer. Note that in NeoBFT both the network and application layers need to work together for ensuring the safety properties.

In the present work we propose NsoBFT (Network secure ordered BFT), a consensus algorithm that tolerates Byzantine faults while preventing a malicious sequencer from assigning different sequence numbers to the same message. NsoBFT implements a secure sequencer by using a USIG (Unique Sequential Identifier Generator) [29] which constrains the malicious actions that the sequencer can execute. Thus NsoBFT does not require additional coordination in the replicas, *i.e.*, it requires one less communication step in comparison with NeoBFT. The USIG module can be implemented in the network layer by using a co-processor coupled to a switch to perform more complex operations.

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 present 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

	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.

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 and variants, the algorithms were implemented and compared (NOPaxos, NeoBFT, and NsoBFT). Results show that NOPaxos presents superior performance but tolerates only crash faults, while NsoBFT outperforms NeoBFT by requiring one less communication step.

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

2 Background & Related Work: Consensus, from Paxos to NeoBFT

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, and 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 [12]:

- `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:

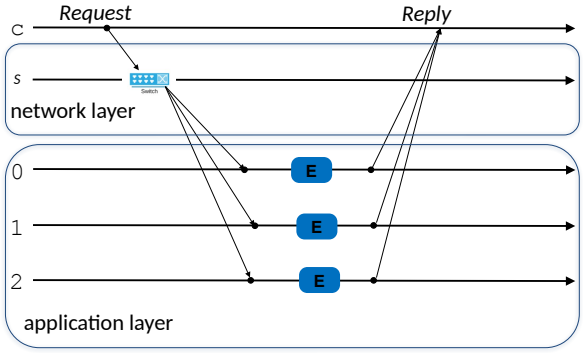
- **Agreement**: if a correct process decides v, then all correct processes eventually decide v.
- **Validity**: a correct process decides v only if v was previously proposed by some process.
- **Termination**: all correct processes eventually decide.

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 properties define the safety requirements of the consensus, and the termination property defines its liveness. It has been proved [10] 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) [9] in which a system is initially asynchronous but after some unknown but finite time instant becomes synchronous respecting time limits both for process execution and message transmission.

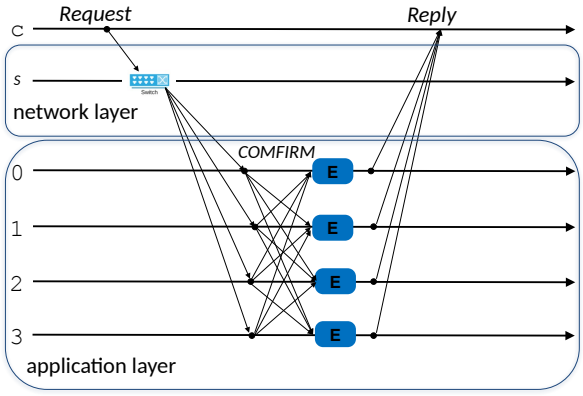
Although several consensus algorithms have been proposed, given the nature of the algorithm introduced in the present work, we start our overview of related work with Paxos [15], which is one of the most important consensus algorithms. Paxos assumes that processes can fail by crashing. Processes assume 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.



(a) NOPaxos.



(b) NeoBFT.

Fig. 1. Normal case execution for NOPaxos and NeoBFT.

Closely related to the algorithm we propose in this paper, NOPaxos (Network Ordered Paxos) [17] 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 1(a)): 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 [23] 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 [5, 6]. 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 [23] 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 1(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 [23], 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, there have been several algorithms to improve aspects of Paxos and PBFT from different perspectives. For instance, MinBFT [29] 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 [20] or to consider systems in which the number of replicas is initially unknown [1]. All of these algorithms order requests by exchanging messages at the application layer.

3 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 [29] 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.

3.1 System Model

NsoBFT assumes a distributed system that consists of an unbounded set of client processes and a set of n server processes, which are also called *replicas*. The Byzantine fault model is assumed, *i.e.*, processes can behave maliciously and not follow protocol specifications. A process is considered correct if it does not fail; otherwise, it is considered faulty. There are at most f faulty replicas out of $n = 3f + 1$ replicas. Processes have unique identifiers, and it is impossible to obtain additional identifiers to launch a Sybil attack [8].

The system is partially synchronous and we assume the GST model [9, 3]. Thus, after an initial period of instability and from a time instant which is also called the Global Stabilization Time (GST) the system becomes and remains synchronous forever. 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. 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. At least one correct sequencer is placed in the network layer (*e.g.,* in a switch) that uses the USIG to assign sequence numbers to client requests.

3.2 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 [29]:

- `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.

3.3 NsoBFT Consensus

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

Normal Execution. A normal execution of BsoBFT follows the message pattern shown in Figure 2. 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 virtual network function [28] that connects all replicas. We note that both technologies have been used to implement consensus algorithms [7, 27, 26].

- Initially, the client multicasts request r to the set of replicas. 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.
- The replicas receive request r already with the sequence number `UI` and verify whether it (UI) is valid using function `verifyUI(PK,UI,r)`, where `PK` is the public key of the service, which should be widely availalbe 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 *log*, and send the response back to the client.

– The client waits for $2f + 1$ identical responses to complete the operation.

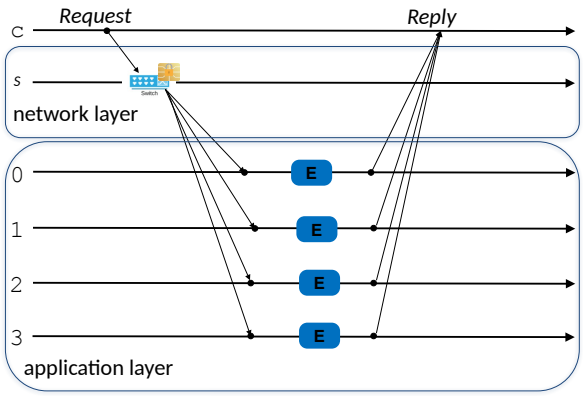


Fig. 2. NsoBFT normal case execution.

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.

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.

– 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 sends a `VIEW-CHANGE` message (if it has not yet been sent). Since all correct replicas will receive those messages, the replica 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 only accept a proposal if their own logs are not ahead of the proposed log. For this, the consensus protocol must define a predicate which is used to check whether proposals can be accepted or not [24]. 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.
- At the end of the consensus execution, all replicas decide on the same log l_d. After that, each replica checks its own log to executes the requests in l_d that it had missed, or rolls back the requests ahead of l_d.

Additional considerations. A malicious sequencer may choose to discard requests from specific clients. As in other implementations (*e.g.*, BFT-SMaRt [2]), 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.

3.4 NsoBFT Variations: MinNsoBFT and MinZyzNsoBFT

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 [29], 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 [29]. 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), 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 [29].

4 Experimental Evaluation

This section reports the 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 study the impact of Byzantine fault tolerance.

4.1 Implementation, Environment and Methodology

All three consensus algorithms that employ the network layer as a sequencer (NOPaxos, NeoBFT, and NsoBFT) were implemented in Java. Experiments were executed on the Emulab [31] testbed. We employed 6 *d430* machines (2.4 GHz E5-2630v3, with 8 cores and 2 threads per core, 64 GB of RAM) connected to a 1 Gbps switch. The NOPaxos was configured with three servers, while NeoBFT and NsoBFT were configured with four servers to tolerate up to one failure. Each server executed on a dedicated machine, while a varying number of clients were deployed on another machine. The sequencer also executed on a dedicated machine.

The software environment comprised the 64-bit Ubuntu 20 operating system and the 64-bit Java Virtual Machine (JVM) version 1.8_431. The Bouncy Castle cryptography library was also used to generate and verify 256-bit signatures. To verify the performance of the different approaches, an "empty" service was implemented, *i.e.*, nothing was processed on the replicas. The number of clients varied, and the throughput was measured on one of the replicas, while the latency was measured on one of the clients. Furthermore, the payload size varied between 0B, 100B, 1kB, and 4kB.

4.2 Experimental Results

In addition to the payload, the messages contained other information, such as the client identifiers and the sequence number. The sizes of messages without signature was the payload size plus 190 bytes. At the same time, the sizes of messages with signatures was equal to the payload size plus 536 bytes. Java serialization (*Serializable* interface) also contributed to those sizes. The average time for the sequencer to sign a packet was approximately 1.81 ms, and for the server replica to check its validity was approximately 0.22 ms, resulting in a total of 2.03 ms of processing overhead in the Byzantine fault-tolerant solutions.

Figure 3 presents the latency and throughput for the three algorithms. As expected, NOPaxos presents the highest throughput and lowest latency. This was expected, as the algorithm only tolerates crash faults and requires fewer replicas. In addition, this protocol does not employ expensive cryptographic primitives. NOPaxos reached a throughput of more than 25 kops/sec (a thousand operations per second) for payload sizes of 0B and 100B. For payloads of size 1kB, the throughput reached approximately 24 kops/sec. Finally, for payloads larger than 4kB, the throughput reached approximately 9 kops/sec. In all cases,

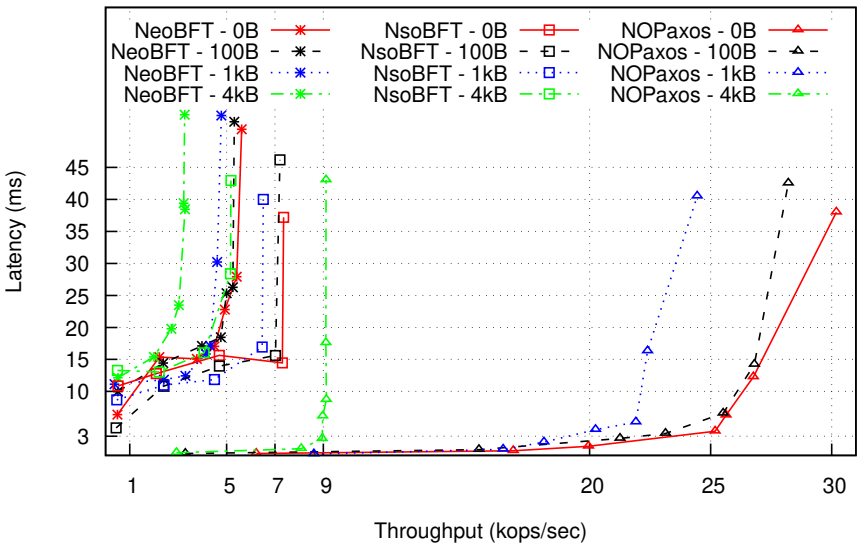


Fig. 3. Latency (95 percentile) *vs.* throughput.

the latency starts relatively low and gradually increases as the system saturates and reaches its peak throughput. From this point on, only the latency increases.

Among the protocols that tolerate Byzantine faults, NsoBFT outperformed NeoBFT because it requires one less communication step. Note that both protocols require the sequencer to sign all messages. NsoBFT presented a throughput that exceeded 7 kops/sec for payloads with 0B and 100B, while NeoBFT presented for the same payload sizes a throughput of slightly more than 5 kops/sec. For 4kB payloads, NsoBFT achieved a throughput of over 5 kops/sec, while the throughput of NeoBFT was roughly 3 kops/sec. As was the case for NOPaxos, the latency remained low up to the system saturation point. However, it is important to note that, in NeoBFT, the system saturated earlier, since this protocol requires one more communication step than NsoBFT.

	NOPaxos				NeoBFT				NsoBFT			
# of clients	0B	100B	1kB	4kB	0B	100B	1kB	4kB	0B	100B	1kB	4kB
1	6.22	5.30	5.60	5.93	0.49	0.51	0.37	0.51	0.51	0.43	0.46	0.48
5	16.85	15.43	16.43	8.08	2.22	2.38	2.41	2.00	2.09	1.99	1.84	1.84
10	19.96	21.25	18.11	8.95	3.77	3.98	3.30	2.72	4.69	4.70	4.49	4.07
25	25.18	23.13	20.25	8.97	4.50	4.76	4.15	3.03	7.29	7.01	6.46	4.57
50	25.63	25.52	21.90	9.12	4.93	5.00	4.32	3.28	7.35	7.20	6.51	5.18
100	26.76	26.81	22.39	9.10	5.62	5.96	4.61	3.23	7.40	7.56	6.46	5.35

Table 2. Throughput (kops/sec) for a varying the number of clients and payload sizes.

Table 2 shows that the throughput varies considerably between the algorithms, especially as the number of clients increases. This confirms that the load does impact the overall system efficiency. In general, the NOPaxos algorithm tends to present higher throughputs with most 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.

	NOPaxos				NeoBFT				NsoBFT			
# of clients	0B	100B	1kB	4kB	0B	100B	1kB	4kB	0B	100B	1kB	4kB
1	0.29	0.30	0.23	0.42	9.39	9.99	11.14	13.13	8.85	8.30	8.68	12.29
5	0.73	0.90	0.94	1.05	15.35	14.44	12.83	15.38	12.79	10.81	10.99	13.03
10	1.42	2.66	2.07	2.60	15.09	17.11	20.43	19.20	15.63	13.95	11.87	16.03
25	3.79	3.43	4.04	6.14	17.04	18.46	25.85	33.50	14.45	15.60	16.94	28.39
50	6.27	6.59	5.20	8.70	32.78	45.34	37.10	48.45	37.19	40.18	40.01	42.96
100	12.26	14.21	16.30	17.58	47.92	47.27	55.23	99,40	47.37	46.08	52.96	84.08

Table 3. 95th percentile of the latency (ms) for a varying number of clients and payload sizes.

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, even with an increase in the number of clients or payload size, since the system has not yet reached its saturation point. 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.

The evaluation results confirm both that tolerating crash faults is cheaper than tolerating Byzantine faults, and make it clear that NsoBFT outperforms NeoBFT when intrusion tolerance is required since it requires one less communication step.

5 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 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. Another alternative is its implementation as virtual network function. A comparison of the cost versus performance of those strategies should reveal insights not only on how efficient Byzantine consensus can be, but also on how those two different technologies (*i.e.*, programmable switches vs. network functions virtualization) compare when applied to a critical application. Other future work includes the implementation and evaluation of MinNsoBFT and MinZyzNsoBFT, the two variants of NsoBFT presented in the paper.

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

References

1. Alchieri, E.A.P., Bessani, A., Greve, F., Fraga, J.d.S.: Knowledge connectivity requirements for solving byzantine consensus with unknown participants. IEEE Transactions on Dependable and Secure Computing **15**(2), 246–259 (2018)
2. Bessani, A., Sousa, J., Alchieri, E.E.P.: State machine replication for the masses with bft-smart. In: International Conference on Dependable Systems and Networks. pp. 355–362. IEEE (2014)
3. Bravo, M., Chockler, G., Gotsman, A.: Making byzantine consensus live. Distributed Computing **35**(6) (2022)
4. Burrows, M.: The chubby lock service for loosely coupled distributed systems. In: The 7th Symposium on Operating Systems Design and Implementation (2006)
5. Castro, M., Liskov, B.: Practical byzantine fault tolerance. In: Symposium on Operating Systems Design and Implementation. pp. 173–186. USENIX (1999)
6. Castro, M., Liskov, B.: Practical Byzantine fault-tolerance and proactive recovery. ACM Transactions on Computer Systems **20**(4), 398–461 (2002)
7. Dang, H.T., Bressana, P., Wang, H., Lee, K.S., Zilberman, N., Weatherspoon, H., Canini, M., Pedone, F., Soulé, R.: P4xos: Consensus as a network service. IEEE/ACM Transactions on Networking **28**(4), 1726–1738 (2020)
8. Douceur, J.R.: The sybil attack. In: International Workshop on Peer-to-Peer Systems. pp. 251–260. Springer (2002)
9. Dwork, C., Lynch, N.A., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of ACM **35**(2), 288–322 (1988)
10. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM **32**(2), 374–382 (1985)
11. Freitas, A.E.S., Rodrigues, L.A., Duarte Jr, E.P.: vcubechain: A scalable permissioned blockchain. Ad Hoc Networks **158**, 103461 (2024)
12. Hadzilacos, V., Toueg, S.: A modular approach to the specification and implementation of fault-tolerant broadcasts. Tech. rep., Department of Computer Science, Cornell University, New York - USA (May 1994)
13. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: {ZooKeeper}: Wait-free coordination for internet-scale systems. In: USENIX Annual Technical Conference (2010)
14. J. C. Corbett, J.D., et al, M.E.: Spanner: Google's globally distributed database. In: The 10th Symposium on Operating Systems Design and Implementation (2012)

15. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems **16**(2), 133–169 (1998)
16. Li, C., Qiu, W., Li, X., Liu, C., Zheng, Z.: A dynamic adaptive framework for practical byzantine fault tolerance consensus protocol in the internet of things. IEEE Transactions on Computers **73**(7), 1669–1682 (2024)
17. Li, J., Michael, E., Sharma, N.K., Szekeres, A., Ports, D.R.: Just say {NO} to paxos overhead: Replacing consensus with network ordering. In: Symposium on Operating Systems Design and Implementation. pp. 467–483. USENIX (2016)
18. Liu, X., Yu, W.: A review of research on blockchain consensus mechanisms and algorithms. In: International Conference on Intelligent Informatics and Biomedical Sciences. vol. 9, pp. 1–10 (2024)
19. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: USENIX annual technical conference (USENIX ATC 14). pp. 305–319 (2014)
20. Saramago, R.Q., Alchieri, E.A., Rezende, T.F., Camargos, L.: On the impossibility of byzantine collision-fast atomic broadcast. In: International Conference on Advanced Information Networking and Applications. pp. 414–421. IEEE (2018)
21. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys **22**(4), 299–319 (1990)
22. Singh, A., Kumar, G., Saha, R., Conti, M., Alazab, M., Thomas, R.: A survey and taxonomy of consensus protocols for blockchains. Journal of Systems Architecture **127**, 102503 (2022)
23. Sun, G., Jiang, M., Khooi, X.Z., Li, Y., Li, J.: NeoBFT: Accelerating byzantine fault tolerance using authenticated in-network ordering. In: ACM Special Interest Group on Data Communications Conference. p. 239–254. ACM, New York, NY, USA (2023)
24. Vassantlal, R., Alchieri, E., Ferreira, B., Bessani, A.: Cobra: Dynamic proactive secret sharing for confidential bft services. In: Symposium on Security and Privacy. pp. 1335–1353. IEEE (2022)
25. Venâncio, G., Fulber-Garcia, V., Flauzino, J., Alchieri, E.A., Duarte, E.P.: Dependable virtual network services: An architecture for fault-and intrusion-tolerant sfcs. In: Conference on NFV and SDN. pp. 1–6. IEEE (2024)
26. Venâncio, G., Turchetti, R.C., Camargo, E.T., Duarte Jr, E.P.: Vnf-consensus: A virtual network function for maintaining a consistent distributed software-defined network control plane. International Journal of Network Management **31**(3) (2021)
27. Venâncio, G., Turchetti, R.C., Duarte, E.P.: Nfv-rbcast: Enabling the network to offer reliable and ordered broadcast services. In: The 9th Latin-American Symposium on Dependable Computing. pp. 1–10. IEEE (2019)
28. Venâncio, G., Turchetti, R.C., Duarte Jr, E.P.: Nfv-coin: unleashing the power of in-network computing with virtualization technologies. Journal of Internet Services and Applications **13**(1), 46–53 (2022)
29. Veronese, G.S., Correia, M., Bessani, A.N., Lung, L.C., Verissimo, P.: Efficient byzantine fault-tolerance. IEEE Transactions on Computers **62**(1), 16–30 (2013)
30. Vukolić, M.: The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In: IFIP WG 11.4 International Workshop Open Problems in Network Security. pp. 112–125. Springer (2015)
31. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An integrated experimental environment for distributed systems and networks. ACM SIGOPS Operating Systems Rev. **36**(SI), 255–270 (2002)
32. Zou, Y., Yang, L., Jing, G., Zhang, R., Xie, Z., Li, H., Yu, D.: A survey of fault tolerant consensus in wireless networks. High-Confidence Computing **4**(2) (2024)