Information Sharing in the Presence of Adversarial Nodes Using Raft

Linir Zamir and Mehrdad Nojoumian

Department of Computer & Electrical Engineering and Computer Science
Florida Atlantic University, Boca Raton, FL 33431, USA
{lzamir2016,mnojoumian}@fau.edu

Abstract. Consensus algorithms are a prominent component in decentralized systems involving multiple unreliable nodes. Since the days Blockchain was first introduced, decentralized systems have been used in many applications such as supply chain management and information sharing paradigms. Raft is a consensus algorithm based on 'Paxos' that is used for replicated state machines when a group of nodes is required to have the same state at any moment. This paper therefore proposes an alternative version of the Raft consensus algorithm to allow sharing information among nodes in a secured fashion while maintaining the security features of the original Raft algorithm even in the presence of adversarial nodes. The proposed model can be implemented to improve cooperation among parties, especially, in resource-constrained platforms where a regular encrypted communication method can heavily affect the system. Our analysis illustrates that the proposed model is practical to be used in many worldwide applications.

Keywords: Consensus, Raft, Information Sharing.

1 Introduction

With the development of information sharing technology and its increasing threats, various secure information sharing techniques have been developed over years. One challenge is to design a closed network system where information can be shared in a secured way under the assumption that data is tamper-proof and secured in the presence of adversaries. Raft consensus algorithm, designed by Ongaro and Ousterhout [14], seems to be one of the initial promising solutions to this problem. The original Raft implementation is an improved Paxos style protocol that is much easier to understand. It offers the same security features as Paxos. The algorithm was designed in the context of replicated state machines in which the algorithm keeps the logs consistent and identical even when a subset of servers are down. This algorithm performs well considering its main objective, i.e., handling servers communications with a client and synchronizing their data. However, to be used for information sharing, it needs to be improved.

The concept of achieving consensus among parties has been studied for many years. It became well-known when Satoshi Nakamoto published the famous Bitcoin paper in 2009 [11] and introduced Blockchain as an infrastructure for the Bitcoin technology; see [13] for details of Bitcoin mining in Blockchain. Achieving consensus among peers on the network was necessary for implementation of Bitcoin. This platform was the first state-of-the art that utilized a consensus algorithm to achieve trust among parties in a decentralized network. Satoshi's idea was based on the one-cpu-one-vote rule, i.e., proof-of-work (PoW) consensus algorithm that has been and still is the most commonly used consensus algorithm for cryptocurrency. Other consensus algorithms were developed ever since, such as proof-of-stake (PoS) [7] and proof-of-authority (PoA) [3], to name a few. Paxos is an example of a fault-tolerant distributed system that is complex to be implemented. However, Raft was introduced as a modern consensus algorithm that is based on the same principles of Paxos but with less complexity.

For this paper, a single server will be called a 'server' and a group of servers will be named a 'cluster.' The paper is organized as follows. The rest of this section briefly reviews information sharing in the original Raft consensus algorithm. Section 2 proposes our newly designed algorithm. Section 3 presents our implementation and illustrates how it operates. Section 4 provides technical analysis, and finally, Section 5 concludes with remarks and future works.

1.1 Our Motivation and Contribution

Secured information sharing among a set of parties has always been a security concern in the research community [8]. With the increasing amount of information that is being shared worldwide as well as the technological capabilities of adversaries, secure information sharing is essential nowadays. Current centralized solutions may provide security for shared data as long as the centralized authority is trusted. This, however, proved to be risky where centralized authorities acted maliciously. The better-known examples are the Facebook scientific experiment [5] and the government surveillance [6]. As such, decentralized solutions and implementations are becoming more common in many domains such as machine learning, supply chain, and information sharing [4, 16, 9, 10].

We therefore propose an improved version of the Raft consensus algorithm that allows secure information sharing among a set of parties in the presence of adversaries. As a proof of concept, we provide an implementation of our algorithm, and then analyze its security and effectiveness.

Drawbacks of Information Sharing Using the Original Raft. We define *information sharing* as the possibility of servers to send secure messages to a central log via a secured and tamper-proof channel in the presence of adversaries. The original Raft algorithm is the first building block to be used for this purpose. However, it has many disadvantages. The first and most important is that Raft can not operate properly when an adversarial node is presented in the cluster. Here, we illustrate a couple of issues when the original Raft is used.

Leader Election: The first stage of the system is to elect a cluster leader. In a malicious environment, a server can self-elect by updating to a new term and sending heartbeat messages to the other servers in the cluster. A malicious leader can also stall the system by initiating election repeatedly, thus preventing the system's availability, i.e., causing a stall election. The servers can also start a new election even if it still receives heartbeat messages from the current leader.

Log Replication: This happens when a new state is presented to the system and the servers eventually are required to arrive at the same state. Replicating the log needs to be secured and immutable as possible, especially when an adversarial server is present. In the Raft algorithm, the leader server can change or drop requests that are sent to him, i.e., violating the integrity of the system.

Another drawback is that the Raft algorithm is a client-server communication with a single client and a cluster of servers, whereas we need the servers to send requests to the cluster without relying on a single authority. This leads to a multi-client-server communication where each node is both a client and a server.

Improved Raft for Information Sharing. We present an alternative method of information sharing in a closed and isolated network with N servers. The communication model is similar to that of the original Raft under the assumption that it is reliable in the presence of an adversary that can cause network lagging and information loss. Therefore, we incorporate the following improvements into the Raft information sharing design:

1. *Byzantine Fault Tolerance:* As stated earlier, it is impractical to use the original Raft as the main consensus algorithm without modifications. The first thing that needs to be taken into account is to allow the system to work under the Byzantine Fault Tolerance (BFT) assumption [15]. This assumption allows the system to work securely in the presence of Byzantine servers that can sabotage the Raft algorithm. Copeland and Zhong [2] provide *Tangaroa* as a solution to this problem.
2. *Client servers:* The Raft algorithm is built as a cluster of servers that communicate to a single client. Whenever the client wishes to update the log, he sends a message to the leader server that handles the log replication for all other servers in the cluster. Non-leader servers cannot send any request to change the data on the log. In our design, every server is able to request a data appendance. This is similar to the state-change in the state machines.
3. *Log modification:* In the original Raft, every server holds a log, where every server keeps the machine states. In our implementation, the log needs to be changed into a data buffer that holds data. The simple state log then turns into an immutable data log. A server can only append information to this log to be shared with other servers in the cluster. For the sake of simplicity, *data log* is used when we refer to the buffer.
4. *Validation:* Data validation can be subjective to the purpose of the network. We provide a design in which every server can share data in the cluster regardless of the meaning of data. The validation process can be added as a second layer to the infrastructure so that servers can be dropped from the cluster when they don't follow the required messages properties.

2 ISRaft: Our New Information Sharing Paradigm

Our improved version of the Raft consensus algorithm, named *ISRaft: Information Sharing Raft*, uses similar fundamentals as the original Raft algorithm, i.e., Leader Election and Data Log Replication. In addition to modifying these features, our new design includes message validation and security in the presence of an adversary. This design of Raft offers the same security properties as the original Raft consensus algorithm that are as follows:

- *Election Safety:* Only a single leader is allowed per cluster.
- *Leader Append-Only:* The elected leader can only append information to the data log. A server can never delete or overwrite any entry.
- *Log Matching:* If there are two data logs on two different servers with the same term and same data, then the entire data logs on the two servers are identical in all entries up to the last term.
- *Leader Completeness:* When a leader commits a message to the log, that message will be present in all future logs of the leader.

There is one more property that the original Raft consensus illustrates, which is the *State-Machine Safety*; however, due to the nature of our design, it is not essential to maintain this safety property as explained later. Compared to the original Raft algorithm, the major change for our design is the ability to have the servers communicate independently and also have them send data to the leader to request for an update of the data logs. This means the *remote procedure call* (RPC) messages that are used in the original Raft as well as the protocol itself must be modified so that they fit into our design.

First, we explain the RPC requests that are used in the modified algorithm. We then review every stage of the information-sharing process to make the whole procedure clear. The Raft algorithm proposed the following two RPCs: *RequestVote* and *AppendEntries*. However, the ISRaft utilizes a modified adaptation and a couple of other RPCs as follows:

- *RequestVote:* This is initiated by candidates in the cluster. This RPC will be sent anytime a server hasn't received any heartbeat message from a leader.
- *AppendEntries:* This is initiated by the leader of the cluster. This RPC is sent as a request to replicate the log on all servers in the cluster. This message is also used as a heartbeat when no log entries are attached to the RPC.
- *AppendEntries-Response:* This is a response that is initiated by any server receiving *AppendEntries* RPC. The RPC contains the data to be appended as well as the signature of the server.
- *RequestAppend:* This is initiated by any server in the cluster to add information to the log.

2.1 Architecture of the ISRaft

In our setting, a server in the cluster can be in one of the three states: *leader, follower,* or *candidate.* A leader is the only authority that can append entries to the log. The other servers can only request the leader to append messages. This prevents the overflowing and message duplication problems. Servers on the cluster have the certified public-key of all other servers. Therefore, when a server sends any RPC request message, he signs it with his own private-key. Servers reject any RPC message that does not include a valid signature. Any *public key cryptosystem* (PKC), such as RSA, can be used in our model. When a leader appends a message from the *RequestAppend* RPC, he adds the public-key of the requesting server to the data log. This way, every data log entry can be traced to the server that has initiated the request.

Similar to the original Raft, our ISRaft algorithm uses terms. These terms are numbered with consecutive integers starting from 1. It is incremented every time a new term is declared by the leader. Every term begins with an *election*, i.e., a process by which the candidate servers send a RequestVote RPC to the cluster. When a candidate receives the majority of votes in the cluster, e.g., $n/2+1$ for a cluster of size n, he declares himself as the new leader and then he increments the term number. The leader can then append the new message to the data log and share it with the servers in the cluster. Any new entry in the data log contains the hash of the previous entry, which can be computed by all servers. To compute a hash value at index i, the server computes the hash of the data at index $i-1$. When two servers agree on a hash value for index i, they verify to make sure they have identical log entries up to index i.

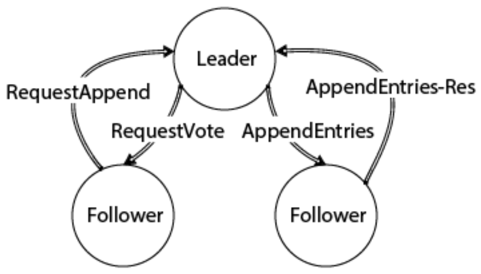


Fig. 1. PRCs that are used in this model.

2.2 Leader Election

The first step of the algorithm is to elect a leader server from all servers in the cluster. All servers start their life as a follower, a state in which the server awaits a heartbeat message from a leader in the cluster. If no message was sent after a predefined period of time, called *election timeout*, the follower changes its state to a candidate. It then begins the election process. The candidate server increments the current term and he sends *RequestVote* RPC concatenate with its own signature to all servers in the cluster. A follower or a candidate server that receives the RPC with a valid signature will send a vote if and only if the following conditions are met:

1. The server is a follower or a candidate.
2. The server has not received any heartbeat message from the current leader.
3. The new term is, at least, the current server's term plus 1.
4. The *RequestVote* is signed with the candidate private-key.

A server that receives the first *RequestVote* RPC for any term will hold the first vote until the end of the *election timeout*. If no other *RequestVote* RPC arrives, it will vote for the first candidate. However, if a second *RequestVote* RPC arrives during that time from a different server, it will immediately vote for the second server. This is to prevent a case where a server starves the system by initiating election constantly.

When a server replies with a vote, it will update its term number and change its state to a follower. The server will then wait for the first heartbeat message from the elected leader, and if none arrives at the expected time, it will become a candidate himself. A candidate wins the election if he received the majority votes from the cluster. The elected leader then changes its state to a leader and sends a heartbeat *AppendEntries* message that includes the signed vote messages that he received from the servers in the cluster. This will be a proof of election and also prevents a self-promoted leader.

While waiting for votes, a candidate may receive an *AppendEntries* RPC from another server claiming to be the leader. If the term in the RPC is at least as large as the candidate's current term and the RPC message contains the signed votes of the majority servers, the candidate recognizes that server as the actual leader and returns to the follower state. Otherwise, the candidate rejects the RPC and continues to remain in the candidate state.

2.3 Log Replication

In our setting, each follower can request an update to the data log from the leader via the *RequestAppend* RPC. This RPC contains the signature of the requesting server and the data itself. The signature guarantees the authenticity of this request, which prevents the malicious servers from forging other server's requests. The server sends the RPC to all other servers in the cluster including the leader. This is to prevent manipulation of data and also to guarantee the leader receives the message in the case a new leader has been elected without the followers' knowledge. The leader then validates the message's signature and begins the data appendance to the data log.

To start, the leader updates its own data log with the new data and increments the term. It will then send *AppendEntries* RPC to all servers in the cluster. This message contains the new data log entry to be appended and all signed votes from the majority of the servers in the cluster. When a server receives the *AppendEntries* RPC, he performs the following tasks:

1. Compares the log entries received from the leader and the requesting server. If the leader itself has requested the update, he skips this procedure.
2. Verifies if the leader's term is at least equal to the requesting server's term.

3. Validates if the leader's votes are legitimate.
4. Verifies if the hash of the RPC message is the expected hash value.

If all these conditions are satisfied, the server appends the message to the log. Otherwise, the server will reject this message, and if no other *AppendEntries* message arrives in the chosen period, the server becomes a candidate. When a server first receives the message from the leader, he will respond with the *AppendEntries* response to all servers in the cluster. A server commits the appended data log only when he receives a similar *AppendEntries* response from the majority of the servers for the same term and data. There might be a case where a server sends the *RequestAppend* RPC to the cluster, including the leader with the data to be appended, but the leader sends the *AppendEntries* RPC without the appended data. This can happen if the leader never receives the RPC from the server, or if the leader is malicious. In both cases, the servers in the cluster will not be able to commit the data log, and as a result, the requesting server will keep sending the *RequestAppend*.

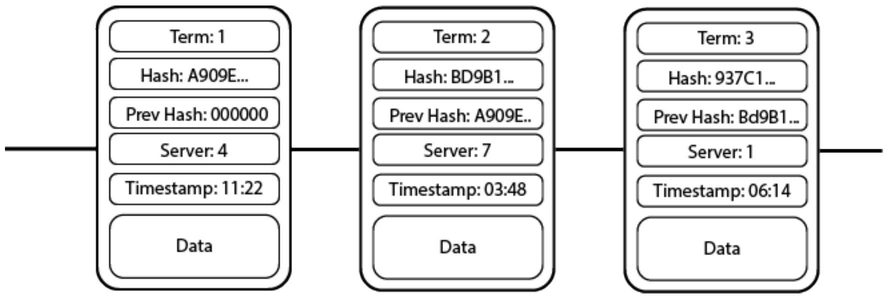


Fig. 2. Data log example.

2.4 Validation

Servers constantly validate their data log with every heartbeat message from the leader. The hash value of the latest data log needs to be in correlation with the server's last data entry. It's also necessary to validate the integrity of the message. This can be done by analyzing the data log itself. Our model doesn't restrict any type of data to be appended to the log, therefore, servers can send any unrestricted data, i.e., our setting doesn't address this type of validation. However, future implementations can have special validation mechanisms to define specific restrictions on the data to be appended.

As an example of validation, we can refer to a cluster that manages relatively complex equations that can be verified by servers. A server can send an equation along with its solution to a cluster. Subsequently, if the majority of parties verify this solution as a correct one, the server then commits it to the data log.

3 Implementation

We implemented the proposed model with C, as a low-level programming language, so that it can be executed on resource-constrained devices that have limited memory and computational power. This implementation also prevents overheads that might be imposed by higher-level programming languages.

The initial program was executed on a Linux environment and then ran on 10 Raspberry Pi's Model 3, all with similar properties, i.e., same clock speed and memory usage. Note that we had to use a programming language that supports the unusual communication properties of RPC when compared with TCP/IP or other protocols. The later model was then implemented on AWS servers, where we had the capability to turn on and off any number of small servers as needed. Some servers in the system were then given the 'malicious' flag, which causes it to act in a non-deterministic way, e.g., spreading false messages, not replying to votes, and clogging the network. In an alternative implementation, we incorporated partitions into the cluster in order to measure and analyze the time it takes for a cluster to elect a new leader and operate when a partition is placed and/or removed. Our implementation is available at: `www.github.com/Morters/Raft_C`.

3.1 Implementation Architecture

As stated earlier, our ISRaft design was developed by C with IPC and RPC libraries. After establishing the commands, it was tested with CuTest on a Linux environment to evaluate its operation. The code is then compiled and ran on a few Raspberry Pi 3-s, and later on multiple Amazon AWS EC2 servers, each operating as a node and communicating with other EC2 servers on a RPC protocol over socket communication. The EC2 servers that we used are t2.nano with 512 MiB of memory and 1 vCPU running at 3.3 GHz. We implemented the RPCs as functions where each RPC can be called by every node on the system. The functions and protocols are organized in the header file *CRaft.h.*Our major goal was to make the minimal changes needed to achieve the same security level as the original Raft. We also intended to make our implementation modular enough so that it can be executed on the majority of devices. Every server was built as a separated AWS EC2 Apache that is identified by a unique IP address and ID.

In our implementation, the servers can send data over socket communication, however, any communication protocol can work. Every server then generates an RSA private-key and public-key pair that is shared with the cluster. Every time a new node enters the cluster, it is required to send its credentials to all servers in the cluster. Accordingly, all servers are required to update their server database. Our implementation does not address a case where there is an error with the credentials or when a server is disconnected from the network since these are out of the scope of our work - although they can be addressed by other mechanisms.

Every server has a *raft* data structure that contains necessary information regarding that specific server. The state includes: `current_state` - current state of the server; `current_term` - integer term number; `voted` - a flag that states whether the server voted for the current term or not; and `datalog` - chain of datalog entries. The datalog size is limited to the storage capability of the server. The log can be stored and accessed on a cloud secured server, however, our design uses an offline information sharing model in a closed environment.

The next phase is to test how our ISRaft addresses malicious behaviors. To validate this, we created specific nodes, called 'malicious, 'which act in a non-deterministic way as explained earlier. For an easier implementation, we used Python to inject these malicious nodes into the cluster randomly. The malicious nodes could have message droppings, datalog mismatching, message duplication, to name a few. For the partitioning, we simply restricted the communication among different servers to simulate a network crash. When needed, the partition could be lifted by reconnecting the servers. Below are our RPC pseudocodes:

Algorithm 1 RequestVote RPC

```
Request
hashed_req_vote /* Vote for the server initiating the RPC */

Response
user_term ← current term number
voted ← 0 /* Flag of the server vote */
if candidate_term < user_term then
    return ERROR
else if voted != 0 and candidate_ID != NULL then
    vote ← Encrypted(candidate_votes)
end if
return SUCCESS
```

Algorithm 2 AppendEntries RPC - Request

```
log_entry /* Log data to be added */
leader_term /* New log term */
signed_votes   /* Cluster votes of the leader. Data is encrypted with the
private-key of the servers and will be decrypted with the public-key */
```

4 Analysis of Our ISRaft

The analysis of our model contains two parts. We initially compare the ISRaft with similar consensus algorithms. Subsequently, we analyze the correctness and efficiency of our model, especially when it's compared with the original Raft algorithm and the similar Tangaroa Raft implementation.

Algorithm 3 AppendEntries RPC - Response

```
n /* The number of servers in the cluster */
user_term /* The latest term of the current server */
prev_log_entry /* The previous log entry */

if leader_term < user_term then
  return FALSE
else if sec_check ! = TRUE then
  /* Check if the signed votes are valid for at least 1/2n+1 for n
  servers in the cluster */
  return FALSE
else if leader.prev_log_entry ! = user.prev_log_entry then
  return FALSE
end if
log.append(log_entry)
if leader_term > current_term then
  current_term ← leader_term
end if
return SUCCESS
```

Algorithm 4 RequestAppend RPC

```
log_entry /* The entry to be appended */
send_encrypt(log_entry) /* Encrypt with private key */
```

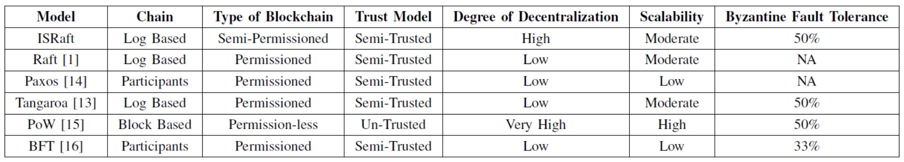
4.1 Comparison

The comparison, shown in Table 3, is partially based on Bamakan et. al. [1] where several consensus algorithms were evaluated with the same criteria. The first comparison aspect is the type of chain that the model will most likely be implemented with, e.g., whether the model is using blocks or log. This can be determined by a number of factors, however, for this comparison we simply take the most commonly used chain method for each of the models. Our ISRaft as well as Tangaroa are based on the original Raft algorithm, therefore, they use a similar *log-based chain* method. Paxos and BFT are based on achieving consensus among parties without having any logging method such as block/log.

The second criteria is the type of Blockchain. Generally, the *permission-less* and the *permissioned* are the two main ways for using Blockchain. Permission-less is also known as public, whereas the other is private. PoW is a permission-less Blockchain where the consensus can operate on an open environment and it is proof-based. The Raft, Paxos, Tangaroa, and BFT are permissioned Blockchain meaning that they operate on a closed environment with some sort of centralization with servers-client communications. Our ISRaft is somewhere in the middle. It was designed to run on an open environment where all nodes operate with the same consensus algorithm and new nodes can join the clusters. The election, however, is voting-based that is similar to most permissioned Blockchains.

Third comparison is the trust model. The network can be *fully-trusted* where the entire network communication is moderated and all nodes can be trusted; *semi-trusted* where part of the network can be moderated and access is granted only to trusted nodes; and finally, *un-trusted* where the network is not moderated. Our ISRaft is categorized as semi-trusted, however, it can be easily modified to be un-trusted if needed. In the un-trusted model, servers can join and leave clusters without approval as long as they provide the appropriate public-key.

The *degree of decentralization* shows how the model is decentralized in comparison to others. When a models' node is dependable on other nodes, the degree of his decentralization is weakened. The original Raft, for example, has low degree of decentralization since different commands can only be initiated by the client to the servers.Our ISRaft is different from all other permissioned algorithms due to the fact that it has a high decentralization degree as the nodes can communicate with each other and achieve a consensus without the intervention of a server node. The *scalability* is how well the model can operate on a larger scale, requiring more computational power and resources. Our ISRaft is not different from the original Raft consensus algorithm form this perspective. The last comparison is the *percentage of byzantine nodes* the model can tolerate. Our ISRaft is an upgrade to the original Raft as the algorithm can tolerate up to 50% of byzantine nodes.



Model	Chain	Type of Blockchain	Trust Model	Degree of Decentralization	Scalability	Byzantine Fault Tolerance
ISRaft	Log Based	Semi-Permissioned	Semi-Trusted	High	Moderate	50%
Raft [1]	Log Based	Permissioned	Semi-Trusted	Low	Moderate	NA
Paxos [14]	Participants	Permissioned	Semi-Trusted	Low	Low	NA
Tangaroa [13]	Log Based	Permissioned	Semi-Trusted	Low	Moderate	50%
PoW [15]	Block Based	Permission-less	Un-Trusted	Very High	High	50%
BFT [16]	Participants	Permissioned	Semi-Trusted	Low	Low	33%

Fig. 3. Comparison of consensus algorithms.

4.2 Efficiency Analysis

Our implementation used EC2 Amazon servers running in a cluster. Each server is an EC2 instance that uses multi-threading as part of its infrastructure. The efficiency tested in this section is the average time it took for a cluster to achieve consensus for the same election timeout value. The comparison of the model is done only with the original Raft and the similar modification of Raft, named Tangaroa [2]. Tangaroa was implemented using similar tools as our ISRaft.

The algorithms were then tested on different similar environments: one with no malicious node active and another with 20% of malicious nodes who can operate in one of three ways: Holding messages, pretending to be different servers in the cluster, or sending random messages.Each malicious node was given a random assignment to either one of these operations.

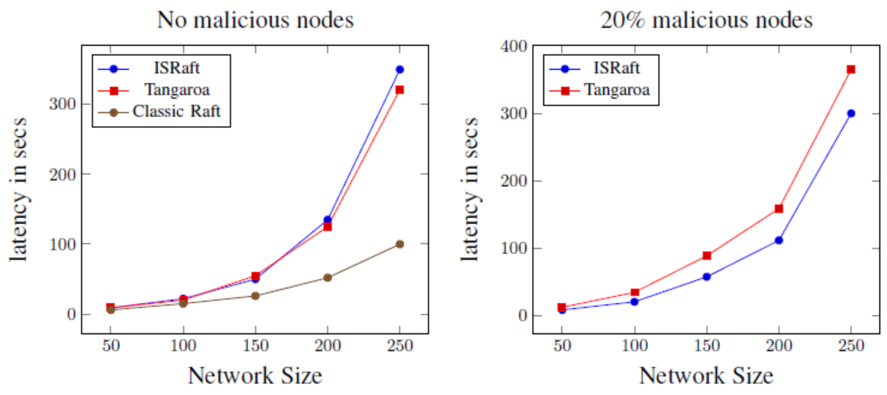


Fig. 4. Algorithms latency with different % of malicious servers.

Figure 4 shows the result of our implementation with respect to the network size (number of nodes) and the total latency of the network for all communications. The latency was calculated by having each node store the time it initiated the command and have it reduced from the end time when the command reached all other nodes. The Raft operates faster than ISRaft and Tangaroa in a network with no malicious modes. This is due to the fact that the servers had to store the entire cluster's keys and many RPCs so that they operate properly, whereas in other models, the nodes only require to store the log and term.

Malicious nodes have a significant effect on the operation of the classic Raft algorithm. The results were always inconclusive as the model kept stalling and messages were not passing through. This is why the original Raft model was not fully tested in this malicious environment. The ISRaft and Tangaroa were both capable of handling 20% of malicious nodes and were given similar results as the one with no malicious nodes.

This analysis illustrates that the original Raft is incapable of handling malicious nodes and it is much faster on a large scale than our ISRaft. However, in real-world applications where malicious nodes exist, the ISRaft operates as expected. The main difference between our ISRaft and the Tangaroa is that the Tangaroa is not as decentralized as our ISRaft and it is required to operate under server-client communication whereas our ISRaft is fully decentralized.

5 Concluding Remarks and Future Works

This paper presents a different way to utilize the Raft consensus algorithm in a secure information sharing environment. This model allows the servers to share information on a shared-log in an adversarial environment.

The servers use public-key signatures with encrypted PRCs while maintaining operative communication. The proposed model can be implemented to improve the cooperation among parties, especially, in resource-constrained infrastructures where a regular encrypted communication method can heavily affect the system. The implementation of the ISRaft consensus algorithm is written in C and tested on multiple servers with a simulator to test the behavior of the cluster in different scenarios. The analysis shows that the proposed model is practical to be used in many real-world applications. As our future work, we are interested in utilizing trust and reputation models [12] to construct a reputation-based consensus scheme.

6 Acknowledgment

Research was sponsored by the Army Research Office and was accomplished under Grant Number W911NF-18-1-0483. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Office or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

References

1. Bamakan, S.M.H., Motavali, A., Bondarti, A.B.: A survey of blockchain consensus algorithms performance evaluation criteria. Expert Systems with Applications p. 113385 (2020)
2. Copeland, C., Zhong, H.: Tangaroa: a byzantine fault tolerant raft (2016)
3. De Angelis, S., Aniello, L., Baldoni, R., Lombardi, F., Margheri, A., Sassone, V.: Pbft vs proof-of-authority: applying the cap theorem to permissioned blockchain. eprints, University of Southampton (2018)
4. De Kruijff, J., Weigand, H.: Understanding the blockchain using enterprise ontology. In: International Conference on Advanced Information Systems Engineering. pp. 29–43. Springer (2017)
5. Deters, F.G., Mehl, M.R.: Does posting facebook status updates increase or decrease loneliness? an online social networking experiment. Social psychological and personality science **4**(5), 579–586 (2013)
6. Dinev, T., Hart, P., Mullen, M.R.: Internet privacy concerns and beliefs about government surveillance–an empirical investigation. The Journal of Strategic Information Systems **17**(3), 214–233 (2008)
7. King, S., Nadal, S.: Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. self-published paper, August **19**, 1 (2012)
8. Li, S., Lin, B.: Accessing information sharing and information quality in supply chain management. Decision support systems **42**(3), 1641–1656 (2006)
9. López-Pintado, O., García-Bañuelos, L., Dumas, M., Weber, I.: Caterpillar: A blockchain-based business process management system. In: BPM (Demos) (2017)
10. Mendling, J., Weber, I., Aalst, W.V.D., Brocke, J.V., Cabanillas, C., Daniel, F., Debois, S., Ciccio, C.D., Dumas, M., Dustdar, S., et al.: Blockchains for business process management-challenges and opportunities. ACM Transactions on Management Information Systems (TMIS) **9**(1), 1–16 (2018)

11. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Tech. rep., Manubot (2019)
12. Nojoumian, M.: Rational trust modeling. In: International Conference on Decision and Game Theory for Security. pp. 418–431. Springer (2018)
13. Nojoumian, M., Golchubian, A., Njilla, L., Kwiat, K., Kamhoua, C.: Incentivizing blockchain miners to avoid dishonest mining strategies by a reputation-based paradigm. In: Computing Conference. pp. 1118–1134. Springer (2018)
14. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: 2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14). pp. 305–319 (2014)
15. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. Journal of the ACM (JACM) **27**(2), 228–234 (1980)
16. Weber, I., Xu, X., Riveret, R., Governatori, G., Ponomarev, A., Mendling, J.: Untrusted business process monitoring and execution using blockchain. In: International Conference on Business Process Management. pp. 329–347. Springer (2016)