Technical Report: Raft algorithm for achieving Consensus on Blockchain

Nicola Salvadore - *Drafter*
Università degli Studi di Padova
Padua, Italy
nicola.salvadore@studenti.unipd.it

Alessandro Sgreva - *Drafter*
Università degli Studi di Padova
Padua, Italy
alessandro.sgreva@studenti.unipd.it

Prof. Tullio Vardanega - *Supervisor*
Università degli Studi di Padova
Padua, Italy
tullio.vardanega@math.unipd.it

***Abstract*—Obtaining consensus in a distributed environment is a difficult challenge, but also an extremely important requirement for fault-tolerant systems [2]. In the context of blockchain, in particular, researches about finding the best possible consensus algorithm have been an hot topic in the last years, with some of the most notable outcomes being the Proof-of-Work (PoW) and the Proof-of-Stake (PoS) algorithms [5].**
However, these algorithms are considered to be probabilistic in nature, since they only guarantee eventual consistency with an high degree of probability [5], [6]. Newer applications, like Hyperledger Fabric, have tried a different approach: implementing distributed consensus algorithms, such as Raft, in order to achieve deterministic consensus [7].
In this work, we discuss the results obtained by an under-the-hood analysis of the Hyperledger Fabric implementation of the Raft algorithm. Moreover, we report on scale and stability tests we performed on a Proof-of-Concept instance of the Hyperledger Fabric blockchain. The experiments are aimed to verify the fitness of this algorithm for the specific real use case scenario and its performance in situation of intense traffic, faults or failures.
We also describe the development of the PoC and the tests design, in order to provide a better understanding of our work and the results we obtained.

***Index Terms*—Raft, Hyperledger, distributed consensus, blockchain, fault-tolerance, probabilistic, deterministic**

I. Introduction

The last years have seen a move away from centralized and monolithic computing systems, which are prone to problems related to single points of failure and general unreliability. As such, distributed computing is becoming more popular, also thanks to software as a service (SaaS) platforms [1], which made its usage more streamlined and affordable for large and small businesses.
Of course, there would be no distributed computing without distributed systems to actually perform it. These systems are, in fact, computing environments in which the various components are spread across multiple computers and devices, to form a network. In order to work together, these devices need configurations and protocols, so that their combined effort can prove to be more effective than a single device. However, these systems do not come only with positive features, such as scalability, concurrency and fault tolerance, but also with some problems: like achieving *distributed consensus*.

A. Distributed consensus

At a certain point in their lifecycle, the distributed systems will most likely interact with one or more clients. These clients may communicate with the system, by sending a message and receiving back a reply. In this context, there are four characteristics that a consensus protocol tolerating failures should have [2], [3]:

- *validity*: if a client requests the reading or writing of a value, then it must have been previously proposed during a client-server interaction.
- *agreement*: every correct interaction between a client and the server must agree on the same value;
- *termination*: every correct interaction between a client and the server must terminate in a finite number of steps;
- *integrity*: if all correct interactions between client and server have read or written on the same value, then any other interaction must agree on that value.

Considering a situations in which multiple clients are interacting at the same time with a distributed system of multiple servers, this is where there may be problems in reaching consensus. When responding to the client, the multiple servers may not agree on a single data value, while others may fail or be unreliable for other reasons. As such, it is necessary to develop consensus protocols which are fault tolerant or resilient, by performing communications between servers in order to reach an agreement on a single value.
In general, we can have two possible types of multiple-server systems [2]:

- *symmetric system*: in this context, any server may respond to the client, while all the other servers must synchronize with the server which managed to send the response;
- *asymmetric system*: in this context, only a server elected as *Leader* can respond to the client. All the other servers must synchronize with the Leader.

Algorithms such as Raft are based on asymmetric multi-server systems.

B. Blockchain consensus

Distributed systems are not the only decentralized structures which have to deal with problems related to consensus. Blockchain, for instance, is a technology that takes some of the concepts developed for general distributed systems and applies

them in order to create a distributed decentralized network [4]. In this environment, the interactions that happens between the nodes, consist only of transactions. These transactions need to be approved by the majority of the nodes, before they can be considered valid and inserted into one of the *blocks* that compose the blockchain. These blocks are then stored inside the ledger of the nodes, chained together, and coherently replicated to all the participants of the network. This process allows the creation of a *distributed ledger*.
In order for this network to be able to provide features like: immutability, privacy, security and transparency to the stored data; the participants need to find a common agreement about the present state of the distributed ledger [3]. As such, in order to achieve reliability in the network, there needs to be a defined protocol for reaching consensus among nodes.

C. Probabilistic consensus vs. Deterministic consensus

Many of the most known blockchains (Ethereum, Bitcoin, ...) are public and not permissioned, which means that any node can participate in the consensus process [5]. In these contexts, two of the most famous consensus protocols implemented are:

- *Proof-of-Work*: where computers (nodes/miners) compete against each other to be the first to solve complex computations. This is done, in order to gain the right to provide the next block to the blockchain and gain a monetary reward;
- *Proof-of-Stake*: where the validators, who gain the right to provide the next block, are chosen among some of the most trusted participants in the network.

These protocols have been proven to be effective and to be able to eventually guarantee consistency to the network which implements them, with an high degree of probability. However, they are still defined as *probabilistic* consensus protocols [5] and, as such, they are still vulnerable to so called *ledger fork* [6]. This is a phenomenon in which different participants in the network have a different view of the accepted order of transactions. The cause of this, is the fact that multiple *miners* (nodes building blocks) may propose valid blocks at the same time, with transactions ordered differently from each other. This creates multiple branches of blocks, from which the nodes building new blocks have to chose in order to continue building on the chain. This problem is generally fixed automatically by the blockchain over time, since it is proven [6] that the nodes building new blocks will slowly converge on the one that contains more validated blocks. However, there is still a significant time window in which the blockchain is unstable and not completely consistent, and this is something that makes the network vulnerable to double-spending attacks. In order to prevent this problem from happening, there have been studies and proposals for more deterministic consensus algorithms, where the order of the transactions is indisputably defined for all the members of the network. One example of such implementations is Hyperledger Fabric [7], in which the order of transactions is established by a specific entity called *ordering service*. This makes it so that any block validated by nodes is guaranteed to be final and correct, preventing any occurrence of ledger forks.

D. Main contributions

In this work we report on the analysis conducted on the architecture behind Hyperledger Fabric, with its implementation of the Raft consensus algorithm. In particular, we focused on identifying possible weaknesses and shortcomings in this implementation, by verifying:

- the impact of different system configurations (number of transactions, transactions rate, number of nodes) on the performance of the blockchain, particularly in relation to the scalability of the Raft component;
- the impact of network instability on the performance of the blockchain, mainly related to the traffic and the downtime caused by multiple Leader elections;
- the fitness of the Raft algorithm for the purpose of guaranteeing blockchain consensus.

In order reach this goal, we conducted an "under-the-hood" exploration of a portion of the Hyperledger Fabric codebase, followed by dedicated experiments and measurements on three different instances of the Fabric test network.

E. Structure of the document

Section II gives a thorough description of the Raft algorithm, in a general context. *Section III* presents the general purpose and features of the Hyperledger Fabric blockchain. *Section IV* includes an analysis of the architecture of Hyperledger Fabric, focusing on its implementation of the Raft algorithm for its ordering service. *Section V* describes the design and the implementation of our Proof-of-Concept, consisting of three instances of the Fabric blockchain test network. *Section VI* presents the scale tests we designed for the purpose of evaluating performances and behaviour in situations of high traffic, multiple failures or wider networks. *Section VII* elaborates on the results obtained by the tests, discussing possible weaknesses and shortcomings of the system. Finally, *Section VIII* presents some other works related to the scope of the study and *Section IX* includes some final remarks.

II. The Raft algorithm

In this section we will introduce the main features of the Raft algorithm, with the aim of providing a basic understanding of its role in the real case scenario which is Hyperledger Fabric.

A. Context of application

The Raft algorithm is typically adopted in the context of *state machine replication (SMR)* [8], which is a general method for implementing a fault-tolerant service. This is done by replicating servers (or nodes) and coordinating client interactions with the server replicas.
The state machine can be defined as a combination of:

- a set of *states*;
- a set of *inputs*;
- a set of *outputs*;

- a transition function, described as:

$$input \times state \rightarrow state$$

- an output function, described as:

$$input \times state \rightarrow output$$

- a specific *state* called *start*;

The maximum number of machine replicas that can fail, while still keeping the system operating, is defined by the *crash-fault tolerance*. Moreover, the state machine is required to be *deterministic*, which means that its replicas, starting from the same state and receiving the same input, should return the same output.
In this context, a consensus algorithm is required in order to keep a consistent log of the commands executed.

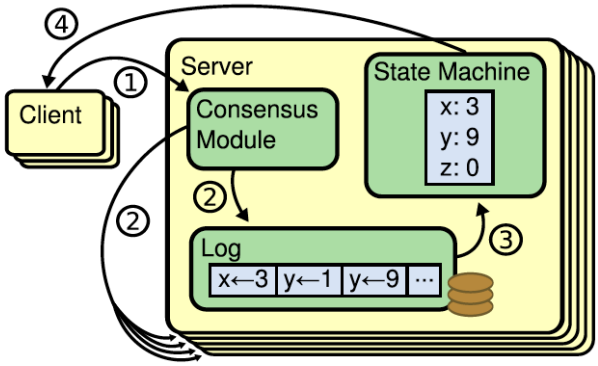

Fig. 1: Replicated state machine interactions.

B. Algorithm overview

The main goal of the Raft algorithm is reaching *consensus* in the context of a distributed system. This means that multiple servers or nodes must agree on some data value that is needed during computation [2], [3]. This is, generally, a primary objective in the design of fault-tolerant distributed systems.
The Raft algorithms defines three main roles, which may be assumed by any node in certain specific situations:

- *Follower*: which is a simple node that composes the replicated state machine (server cluster). This is the default role assigned to nodes when they become part of the distributed system. Follower nodes may only respond to Candidate and Leader nodes;
- *Candidate*: which is a node competing with other Candidate nodes to assume the role of Leader of the distributed system. Nodes may assume this role to request a Leader election, in a situation in which the current Leader node is unresponsive;
- *Leader*: which is unique and the only node that interacts with the client. Any request made to Follower nodes is redirected to the Leader node.

As previously mentioned, these roles are very important for Raft to be implemented for asymmetric multi-server distributed systems.

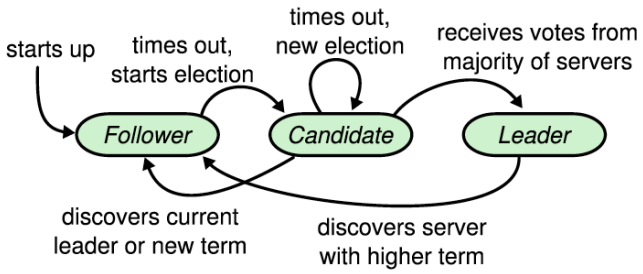

Fig. 2: Representation of the Raft roles.

C. Time management

In order to correctly handle servers states, the Raft algorithm manages the time by dividing it in *terms* of specific length [2]. Each term is uniquely identified by a monotonically increasing integer number called *term number*. This number is stored in each node and attached in node communications.
Each term starts with a Leader election, where the Candidates request votes to other server nodes (Followers) in order to obtain a majority consensus:

- if a Candidate node manages to obtain the majority of the votes, that Candidate becomes the Leader for the current term;
- if no Candidate node manages to obtain the majority of the votes, this situation is called *split vote* and the term will conclude with no elected Leader.

As such, a term can only have one single Leader at any time. Moreover, the term number is also a very important indicator for various system tasks, such as:

- *term number update*: if a node term number is lower than the one of the other nodes in the cluster, it will be updated at the beginning of a new term. The term numbers used for reference are the ones of the Candidates and the one of the Leader, which usually takes the priority;
- *role demotion*: if a Candidate or Leader term number is lower than the other nodes in the cluster, they are demoted to Followers;
- *node communication*: the term number of a node is sent as an attachment to each request they make to other nodes. If a request is made with an outdated term number, it is always discarded.

This makes the term number a crucial factor in time management, for the Raft algorithm.

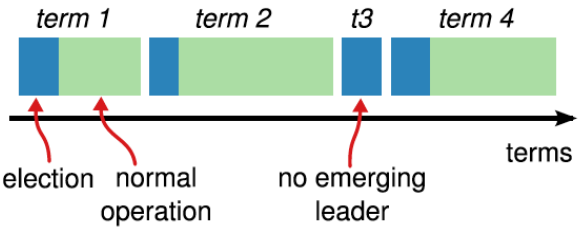

Fig. 3: Representation of terms.

D. Leader election

In order to maintain the authority as Leader of the cluster, the node sends periodic messages to the other Follower nodes. These communications are called *heartbeats* [2].
When a Follower node doesn't receive an heartbeat within a set amount of time, it assumes the Leader is not active anymore and promotes itself to the role of Candidate. It then votes for itself and requests other nodes to vote for him, in order to establish a majority. The result of the election can be one of the following ones:

- the Candidate node receives a vote from the majority of the nodes of the cluster, making it the new Leader. In that moment, the node begins to send heartbeats to notify other nodes of the presence of a new Leader. Candidate nodes receiving the heartbeat will return to the Follower role;
- the Candidate node does not manage to get votes from the majority of the nodes of the cluster. In this case, the election ends with no Leader and the node returns to the Follower role;
- if the term number of the Candidate node who requested the votes is lower than other Candidate nodes in the cluster, the request is immediately refused and the other nodes keep their Candidate role.

This whole process is performed multiple times during the Raft algorithm lifecycle in distributed systems. It ensures that a Leader is always present to serve clients requests and that is condition to do so.

E. Log replication

Each request made by a client is stored by the Leader in the *logs*, which are then replicated in the other Follower nodes [2].
Generally, a log entry contains the following information:

- *command*: which is the instruction specified by the user to be executed;
- *index*: which is the number used to identify the position of the entry in the node log, starting from 1;
- *term number*: which is used to ensure the time of a command specific entry.

The Leader node requests, through a broadcast, that all the Follower nodes synchronize their logs with its own. The Followers shall reply with an acknowledgment, to confirm the completion of such operation, as the Leader will continue to broadcast synchronization requests until all the other nodes have performed it.
A new entry, in this context, is considered *committed* when the majority of the nodes in the cluster has successfully copied it in their logs. At that point, the Leader itself confirms the addition of the entry to its own log, representing the successful outcome of the replication. Following this procedure, it is possible to guarantee that all the previous entry of the log are also committed, otherwise they would not be present inside the log.
After an entry has been committed, the Leader carries out the request present in the entry and responds to the client with its result. In this way, the entries are always executed in the same order they are received and confirmed. If two entries in different logs have the same index and term number, it is guaranteed that they contain the same command and that the logs are identical until the specified index.
In a situation in which the Leader node crashes, the logs may become inconsistent, since the conflicts in the Follower nodes are usually fixed by him. In this case, a new Leader will be elected, which will look for the last coherent Index in the Followers logs and then overwrite every entry beyond that specific index with its own. This helps matching the logs of the Followers with the Leader and restores consistency between them.

F. Safety

In order to maintain consistency on a set of server nodes, the Raft algorithm ensures that the Leader is storing all the entries related to previous terms, committed inside its own log. During a Leader election, the request for a vote that a Candidate sends to other nodes, also contains some information regarding the log of the Candidate. This information includes the term number of said node, which can be evaluated by the receiver, to immediately discard requests from not updated nodes.
In addition to this rule, there are also several others design choices, implemented to avoid consensus malfunctions [2]:

- *leader election safety*: there shall be at most one Leader for each term;
- *log matching safety*: if multiple logs contains an entry with the same Index and Term, then those logs are guaranteed to be identical, till the given index;
- *leader completeness*: the log entries which are committed in a given term shall always appear in the log of the Leader;
- *state machine safety*: if a server applied a particular log entry to its state machine, then no other node in the cluster shall apply a different command in its own log;
- *leader is append-only*: a Leader node can only add commands at the end of its log. No other data altering operation is allowed;
- *follower node crash*: when a Follower node crashes, all the requests sent to the crashed node are ignored. Moreover, the crashed node cannot take part to any Leader election. When the node is restarted, it shall synchronize with the Leader.

These characteristics are considered to be sufficient, in order to ensure correct management of operations, according to the Raft algorithm. Still, there are some limitations to be considered when implementing this algorithm, in particular:

- the presence of a single Leader is indisputably required. As such a system with a large amount of traffic has an high probability of incurring in bottlenecks;
- the Leader election is designed to be able to elect a Leader in at most two terms [2]. How much can the absence of a Leader impact the system during this interval of time?

- the requirement that is the absence of byzantine failures [9] can limit the applicability of the algorithm.

These will be some of the characteristics considered in this work's experimentations, described in a later section.

III. Hyperledger Fabric

In this section we will introduce Hyperledger Fabric, which is a real world case of Raft algorithm implementation for the purpose of reaching consensus on blockchain.

A. Overview

Hyperledger Fabric operates in the branch of proprietary blockchains, which are often implemented to share and maintain data within specific organizations. As a *permissioned* blockchain, in order to become part of the network, an user or organization must request the authorization to join. This procedure ensures, with an high degree of certainty, that all the participants to the network are not malicious. Relying on this core characteristic, we will assume the absence of byzantine faults caused by malicious nodes and refer to nodes system failures as *crashes*.
In this context, organizations or users that take part in the network are called *members* [10]. Each member is responsible for enabling its own *peers*, which shall be adequately configured to work in the network. The peers receive transaction requests from clients, inside their organization, and operate in order to correctly manage such requests.
To be able to initialize transaction requests, the peers are provided with a installed *chaincode*. This code, often referred to as *smart contract*, is dedicated to define how the peer shall manage network transactions [11].
All the nodes keep one or more *distributed ledgers* stored within them, for each channel they are subscribed to, but they may assume different roles [10].

- *Endorser peer*: this type of node, when receiving a transaction request from a client, performs three specific operations:
  1) they validate the transaction, by checking the details of the request and the role of the client;
  2) they run the chaincode, simulating the outcome of the transaction, without immediately updating the ledger;
  3) depending on the results obtained by the two previous operations, the endorser may approve or refuse the transaction.
- *Anchor peer*: this type of node is configured during the setup of *channels*. Channels, in this context, are connections between organizations in the network, that allow communication between them through the use of transactions. The anchor peers are discoverable by other nodes, they receive updates and send them in broadcast to other nodes in the organization.
- *Ordering node*: they are considered to be the central mean of communication in the Hyperledger Fabric network. They are responsible for maintaining coherent ledgers accross the network, they create blocks and send them to the other nodes.

This last type of node, in particular, will be an important focus for later analysis in this paper. In fact, multiple orderer nodes compose an *ordering service*, which is the main entity that implements the Raft algorithm in the Hyperledger Fabric network.

B. The ordering service

Generally public blockchains, such as Ethereum and Bitcoin, are based on probabilistic consensus algorithms, which guarantee eventual consistency with an high degree of probability. However, these blockchains are vulnerable to *divergent ledgers* or *ledger forks*, where the order of the transactions is viewed differently by different participants to the network [6]. In Hyperledger Fabric, it is possible to implement deterministic consensus algorithms that define a specific and unique order for the transaction to be inserted into blocks. This, specifically, is the role of the ordering service and Raft is one of the consensus algorithms that may be configured for its operations [7].
In particular, this service works using the concept of Leader-Follower described by Raft, by electing a Leader between the ordering nodes of a channel. This collection of nodes is described as *consenter set*. The Leader then operates by receiving the requests sent to the ordering service, and replicating them to all the Followers nodes, in order to work as a distributed system.
This system is able to sustain the loss of multiple nodes, even if they are Leader, as long as the majority of the ordering nodes (defined as *quorum*) is still active [7]. As such, it is possible to define the ordering service as *crash-fault tolerant*. This guarantees high availability to the service, which is a very important characteristic for the Hyperledger blockchain to remain active and operational.

C. Ordering nodes role in the transaction flow

The nodes are the most important elements in blockchain networks, since they maintain a ledger that can be queried or updated through the usage of smart contracts and transaction. In Hyperledger Fabric, the ordering nodes play a crucial role in the management of the ledger updates, so that consistency is preserved. More specifically, this updating procedure can be divided in three main steps [7]:

1) *proposal*: during this phase, the client sends a transaction proposal to a subset of peers. They will invoke a smart contract to produce a *ledger update* proposal and then proceed to endorse the result of the operation. Even if the ledger are not immediately updated, at this point, the clients will receive a response, which confirms the reception of the proposal;
2) *ordering & packaging*: during this phase, the client sends some transactions to an ordering service node. These transactions include the positive response previously obtained durint the proposal. At this point, the ordering service nodes work collectively to order the transactions

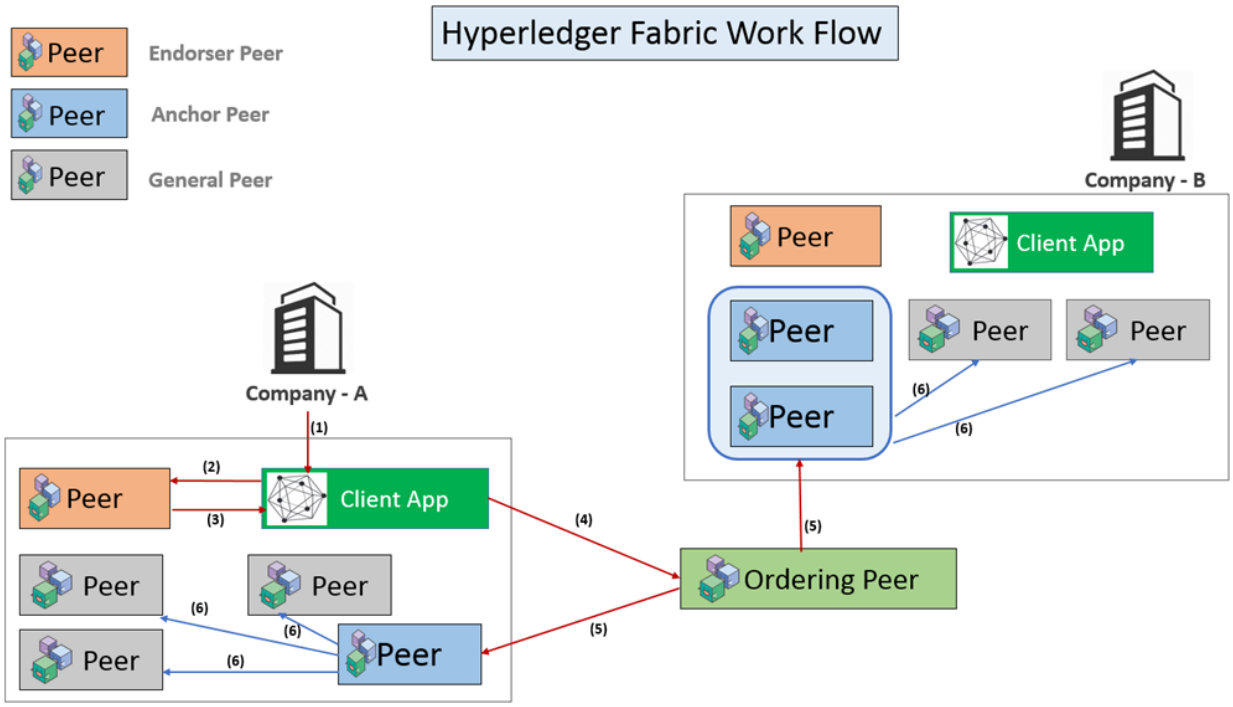

Fig. 4: Hyperledger Fabric general structure.

in a defined sequence, so that they may packaged into blocks;

3) *validation & commit*: during this last phase, the blocks are then stored into the ordering nodes ledgers and distributed to all the nodes which are part of the channel. Even if a node is inactive during the distribution, it will still receive the update once it reconnects to the ordering service, or to other updated nodes.
   Each node then validates the blocks in an independent and deterministic manner, so that consistency is guaranteed.

At the end of this procedure, the ledgers of all the peers connected to the channel are updated with new transactions and blocks, which provide continuity to the blockchain.

IV. Architecture analysis

In this section, we look under the hood of the Hyperledger Fabric implementation of Raft, by analyzing its architecture and reasoning about some of the technical choices. The code, subject of this analysis, is the one present the following link: https://github.com/hyperledger/fabric/.../etcdraft, which includes structures and logic for the Raft consensus mechanism in Fabric.

A. Language of choice

Hyperledger Fabric is one of the first blockchain systems able to run distributed applications written in Go, Java and Node.js, without any systemic dependency on the native cryptocurrency [13]. However, most of the actual Fabric codebase is composed of code written in Go.

Golang (Go) is described as an open source general programming language, with a syntax similar to the C programming language, but a lower barrier to entry.
In term of general approach to the development of a blockchain, there is no declared reason for which the developers decided to choose this language for constructing the architecture of Fabric. Still, studies [14] have reasoned on some of the benefits of using Go with Hyperledger Fabric, such as:

- providing a fast statically typed and compiled language;
- supporting type-safety and dynamic data entry;
- allowing the creation of flexible and modular code;
- providing a multithreading mechanism, which enables distributed computations and simplified network interactions;
- including some important concurrency primitives and channel-based communication for message-based protocols;
- supporting RPC natively, in its `net` module;
- providing an efficient development and testing process, very useful in open-source projects.

The use of this language is also fitting for the development of Fabric consensus mechanisms, particularly Raft, which was the focus of our analysis. Considering that their implementation of this consensus algorithm:

- works with a message-based mechanism, where each transaction is submitted to an orderer node as a message;
- realizes communication between ordering nodes through the use of RPCs;

- contains nodes that should, if possible, not perform blocking operations on their main thread;
- handles nodes status and storage in an atomic manner;
- manages blockchain events through channels (go channels);

we understand how having a language that natively supports these features can be a fitting choice.

B. Raft crash fault tolerance

One of the most important features for a distributed system, such as a blockchain, is the ability to maintain an operational state, even in situations of failure of multiple components. This characteristic is defined as the *crash fault tolerance (CFT)* of the protocol.
The Hyperledger Fabric implementation of Raft, provides a replicated ordering component, where any number of nodes can disconnect at any time, up to 50% of the original cluster size. In fact, even in situations where the Leader of the ordering system disconnects, Raft should be able to elect a new one in at most two terms [2], incurring in only limited downtime. As such, in our tests, we expect to be able to verify that:

- the system is able to work with little or no performance degradation, in contexts of multiple non-Leader nodes crashes;
- the network, despite the loss of cluster Leader nodes, is able to restore its operational state without significant downtime and requests failures;
- the ordering service becomes unavailable when $2/N + 1$ ordering nodes crashes, where N is the size of the original cluster.

C. Fabric consensus components

To better understand the functioning of the Fabric consensus mechanism, we performed an exploration of the actual code used to manage its operations. Below we describe the main components identified during this analysis.

1) Blockcreator: defines a structure where to store the number and the hash of the last created block, so that the next one can be created starting from it. It also provides the function to create a new block, starting from the content provided.

2) Blockpuller: defines a structure to perform the pull of specific blocks from the ledger.

3) Chain: this is one of the most important components of the consensus mechanism. It defines general parameters for the functioning of the ordering service nodes, such as the management of snapshots and timers. In contains functions to start, stop and pause its communications with other nodes, as well as managing some of the main operations, including:

- submission of new transactions;
- managing the status and role of nodes;
- creation of new blocks;
- restore of parts of node ledger through snapshots.

In this context, a *snapshot* is a backup of a (minimal) part of the blockchain ledger. It allows new or reconnecting nodes to perform a faster synchronization with the Leader, than if they were to restart from the empty ledger.
This component makes extensive use go channels for notifying events happening to the chain.

4) Consenter: handles the management of the *consenters*, which are participants to the ordering service (also known as *consenter set*). It provides functionalities for communicating with the node cluster and for the consenter relationship with Fabric channels.

5) Dispatcher: provides functionalities for managing the dispatching of requests, such as requests for transactions or messages between nodes.

6) Disseminator: attaches cluster metadata to messages and requests being sent. In this context, the cluster metadata are information about other nodes connected to the current one (relationships, states, ...).

7) Eviction: periodically checks whether the current node has been evicted from the channel it was connected to and, if confirmed, handles the related closing operations.

8) Membership: manages the node certificates and membership, performing the needed operations in case of configuration changes.

9) Node: represents a node of the chain, and participant to the ordering service. Differently from the Consenter, this components focuses on functions such as:

- starting and running a node;
- handling the elections and related campaigns;
- management of some messages and RPCs.

It also allows for some minor storage-related operations.

10) Storage: defines the specifications related to the storage used by the nodes to store chain related data (ledger, snapshots, ...). In particular, for snapshots, it defines the maximum number of Raft snapshot files to be stored inside the filesystem. The greater the number of snapshots stored, the greater the impact mitigation of possible corrupted snapshots. This, clearly, comes with an higher cost in term of memory.

After this analysis, we had a clearer idea about the mechanisms that govern the ordering service in Hyperledger Fabric. Moreover, this process allowed us to identify critical elements in the design of the Raft implementation, which we could evaluate for later testing. These include:

- after reaching a set threshold of blocks to be managed, the go channel related to the sumbission of new requests is paused. This can cause delays in serving clients requests in situations of intense traffic;

- when a new Leader is elected, it does not immediately begin accepting requests and creating blocks, if other blocks or messages are in flight. This behaviour can add delays in situations with many elections in consecutive terms;
- the orderer node also manages the channels configurations, pausing the chain when a configuration block is being processed. In this context the blockchain becomes unavailable for a significant amount of time;
- configuration changes that cause *quorum loss* (deteriorate crash-fault tolerance) are reported, but not prevented from being processed.

In particular, we focused on the first and second points, which we identified as more closely related to network performances and possible causes of, respectively: ordering service saturation and downtime/unavailability of the ordering service.

V. Proof of Concept (PoC), profiler & testing tool

In the following section, we introduce our Proof-of-Concept, which is based on the Fabric blockchain testnet and acts as system under test (SUT) for our scale tests. Then we also discuss the configuration of a profiler and a testing tool, we used to practically execute our tests.
A repository, containing the code used to structure the these software components can be found at the following link: https://github.com/salvanicola/RCDEXAM.

A. PoC implementation

The main purpose of this PoC was to act as a simple and effective Fabric test network. As such, the development of this piece of software began from the fabric-samples provided by Hyperledger Fabric itself, as introductory guide for that specific blockchain. This repository contains the basic configuration files and some network examples. We chose to start our development from the test-network provided, which is a collection of scripts and configuration files, used to start a small-sized local blockchain.
After downloading and installing the version 2.4 of the Fabric binaries, the test-network sample allowed us to:

1) start an instance of the Hyperledger Fabric network on virtual Docker containers, using the command: `./network.sh up`. This command executes the following steps:
   a) it generates the certificates needed by the network nodes;
   b) it add two peers, with different organizations, to the network;
   c) it add a single orderer, with a dedicated organization, to the network;
2) create a Fabric channel between the two peer organizations, to which the orderer is also joined, using the command `./network.sh createChannel`. This channel acts as a mean of communication for transactions to happen, between the two organization parties;
3) deploy a chaincode, written in Go, on the two organizations peers. This is done using the command: `./network.sh deployCC -ccn basic -ccp <path-to-chaincode>`;
4) request transactions to the network, using adequate preconfigurations and invoking the chaincode through the command: `peer chaincode invoke [...]`.

However, this network configuration, with only two peers and a single orderer was not sufficient for the purpose of evaluating the scalability of the system. As such, we made an effort to manually configure and set up three different networks, with an increasing number of ordering nodes.

B. Technical issues, choices and limitations

The previously described operations could be conducted on a local PC environment, both on Windows and MacOS, with no significant difference in performance. There were, however, compatibility problems between Fabric and the Windows OS, mainly related to the usage of Linux bashes (e.g. Msys/MinGW), which were resolved by dedicated changes to the configuration scripts used.
Additionally, the implementation of a whole blockchain network on a single terminal is quite expensive in term of resources ($\geq$ 2GB of RAM). This was an element that needed to be kept into consideration when scaling it to multiple orderers. As such, we decided to limit the size of any test network ordering service to a maximum of 20 orderers. More specifically, the network sizes chosen for the scale tests were: 5, 10 and 20; since we deemed them to be different enough to detect changes in performance.
In these three network environments, we decided not to modify neither the number of peers, nor the number of organizations. This decision was made to keep the focus on the component which actually implemented the Raft algorithm, which is the ordering service, and its performances. As such, in the following sections when mentioning the *network size*, we will be referring to the size of the ordering service cluster.

C. Configuration of profiler & testing tool

In order to collect informative data about the performances of the network, and particularly of the ordering service, a dedicated profiler was set up. This profiler makes use of the Prometheus monitoring software and of the Grafana web application. Both these pieces of software have been implemented using relative Docker containers and adequately configured to connect with the Fabric blockchain network that was previously launched. More specifically, using the dedicated `prometheus.sh` script, it is possible to launch both applications using the flag `up`, respectively on:

1) `localhost:3000`: Grafana;
2) `localhost:9090`: Prometheus.

Additionally, the Grafana dashboard used for the visualization of data was extensively changed from the basic templates provided, in order to better clearly represent experimentally relevant information. In particular, the data we considered as relevant for monitoring are:

- rate of the Raft consensus proposals, representing the quantitative amount of transactions requested to specific orderers of the ordering service;
- rate of the blocks, streams and requests, indicating the traffic generated by the ordering service computations;
- number of Raft active nodes;
- Raft cluster size;
- the number of the last commited block;
- the identity of the Leader node;
- leader changes experimented by the nodes;
- last blockchain snapshot block number;
- number of cluster communication messages dropped by the network;
- number of broadcast messages processed by the orderers;
- cluster communication messages send time;
- time required by messages to persist Raft data into the storage;
- time required to enqueue transaction requests;
- time elapsed between transaction queuing and commit;
- time required to validate a transaction;
- time required to commit a block on the storage.

This data is provided by Prometheus and obtained by internal metrics defined by Hyperledger Fabric. We also remark that this data was only considered relevant for monitoring purposes and is not necessarily included in the later described scale tests.
In addition to the previously described software, we also made use of Caliper, a blockchain performance benchmarking framework. Using this testing tool, we were able to perform large amount of requests, of different nature, to our PoC network. Moreover, it also allowed the collection of relevant data for determining the level of performance offered by the ordering service, which will be later discussed.

VI. Design and implementation of scale tests

In this section, we describe the logic behind the design of the scale tests, their implementation and expected results.

A. Elements of the testing environment

Following the terminology defined by the Hyperledger Blockchain Performance Metrics White Paper [18], we can describe some of the roles we identified the components of our testing environment.

- *System Under Test (SUT)*: defined as the hardware, software, networks and specific configurations of each, required to run and maintain the blockchain. In our context, the role of the SUT is assigned to the Fabric test network we previously implemented as PoC, also considering its three different versions (with 5, 10 and 20 ordering nodes respectively).
- *Test Harness*: defined as hardware and software used to run the performance evaluation, by injecting workload. In our case, this is represented by the Caliper framework, with the support of an script dedicated to randomly crash nodes, for the purpose of generating faultload;
- *Clients*: this is defined as the identity that can introduce work into the system or invoke its functionalities. The Test Harness is composed by multiple clients with different purposes, more specifically:
  - *Load-generating client*: which submits transactions on behalf of the user to the blockchain network (SUT), generally using an automated test script;
  - *Observing client*: which receives notifications from the SUT or can query the SUT in order to obtain the status of the previously submitted transactions. This is also managed, usually, by an automated script.

  In our testing environment, Caliper generates logical entities called *workers* that fulfill both of these roles, by generating transactions and querying their status at set intervals of time.

B. Performance metrics

Before performing the tests, we identified the most relevant metrics for evaluating the performance of the ordering service we set up in our PoC. In particular, we decided to ignore any measurement regarding reading operations, since they do not trigger transactions, whose elaboration is the main purpose of the ordering service.
The first important metric we considered is the *transaction latency*, which is a network-wide view of the amount of time taken for a transaction's effect to be reflected accross all the network. This measurement includes the time from the point the transaction is submitted to the point when it is available in all the network nodes. This value includes the propagation time and any time required by the consensus mechanism to commit the transaction to the ledger. In the data we collect through Caliper, we consider also statistics over all transactions, with a particular focus on the average values.
Another important metric in this context is the *transaction throughput*, which is the rate at which valid transactions are committed by the blockchain network in a defined period of time. This rate is expressed as transactions per second (TPS), with relation to the network size. Finally, we also consider the *blockchain work* as a function of the transaction throughput and the number of validating consensus nodes. As such, this metric is intended to most completely express the amount of work accomplished by a blockchain network and provide a meaningful characterization of the blockchain performance, with respect to the ordering service. This value is computed as the product of *transaction throughput* with: *log(ordering service size)*.
We evaluated these blockchain-wide metrics to be meaningful also in relation to the performance of the ordering service, which is the main focus of our effort, because of its role in the management of transactions. As such, in a context where the only dynamic component of the testing are the ordering nodes, we assume to be able to relate potential changes in performance directly to the ordering service.

C. Test environments & configurations

As previously mentioned, in order to carry out our tests, we prepared the following three testing environments:

- a test network with 5 ordering nodes and 2 peers owned by 2 different organizations;
- a test network with 10 ordering nodes and 2 peers owned by 2 different organizations;
- a test network with 20 ordering nodes and 2 peers owned by 2 different organizations.

These nodes are all implemented on a local environment through the use of dedicated Docker containers, with no specific resources limitations. Moreover, they are all interested by the transactions and should work together in order to correctly process them.
The Caliper testing tool provides a sample chaincode called Fabcar for benchmarking Fabric type networks. This chaincode is meant to represent the storage in ledger of information related to the production of cars in a factory, including both transactions which create assets in the ledger and transactions that modify already existing assets. Considering the lack of specific requirements, in relation to the nature of the smart contract used for the testing, we decided to deploy and use this chaincode in the testing environments.
Before deciding which parameters to use for the workload in the tests, we wanted to ensure that the values chosen actually allowed for a stable testing lifecycle, without incurring in failures. In order to do that, we performed some preliminary tests on the largest-sized network (20), using a linearly increasing rate of transactions, from 10 TPS to 140 TPS, with a 10 TPS step. At the end of these tests, whose results can be found on Fig. 5, we identified 60 TPS to be the most optimal rate of transactions for our purposes.

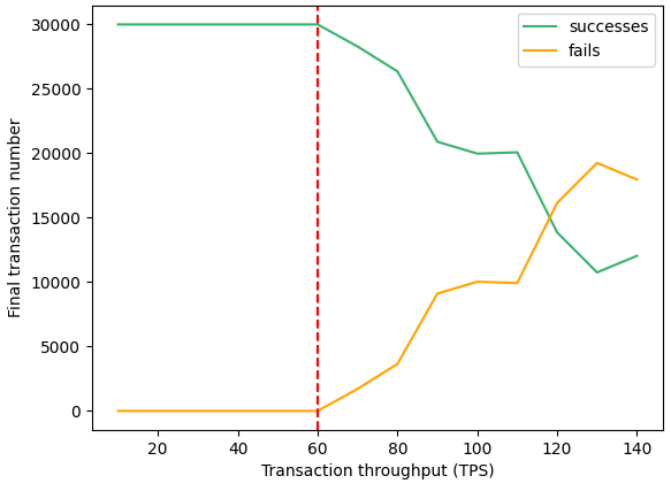

Fig. 5: Network traffic comparison between instances.

We then configured Caliper to perform 15000 asset creation transactions and 15000 asset modification transactions, both at 60 TPS. Moreover, in order to test situations of instability in the network, verify the limits of the crash fault tolerance and trigger Leader elections, we also prepared a script to periodically and randomly pause and restart the Docker containers associated to the ordering nodes.
The goals of these tests are to:

- verify SUT behaviour and performances in standard situations, with different network sizes and comparing the results;
- verify SUT behaviour and performances in situations of network instability, with different network sizes and comparing the results. In particular, we are interested in asserting the impact of the traffic and downtime generated by Leader elections;

We also consider possible additional information we can infer from the tests results, in relation to the fitness and effectiveness of Raft as base for blockchain deterministic consensus algorithms.

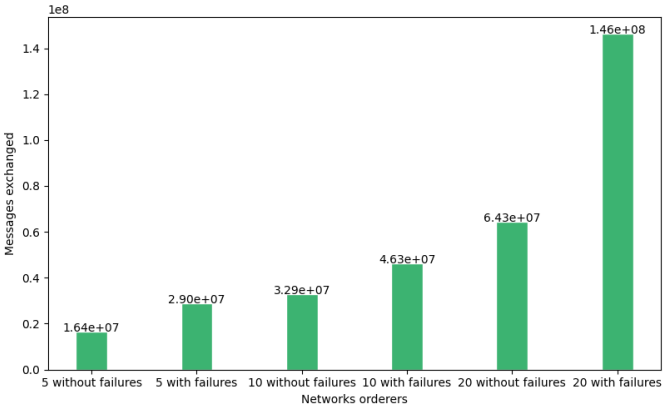

Fig. 8: Network traffic comparison between instances.

VII. Tests results & comments

In this section we present and discuss the results obtained by the previously described scale tests.

A. Results overview

Considering the results obtained by networks in a stable state, as we can see on TABLE I, the transactions encountered no failure during their reception and elaboration. Moreover, no node inadvertently crashed during these tests, and as such no Leader elections were required. Still, it is possible to see in Fig. 8, how the total number of messages sent in the networks increases in a directly proportional way with the size of the network, indicating a larger amount of traffic even in standard network situations. Notice that, when referring to *network traffic* and *messages exchanged*, we only consider those related to the ordering nodes, as they are the most relevant for our tests.
In relation to this, we can see how in unstable network situations, with managed node crashes, a certain amount of transactions actually ends with a failure. This is caused by the occasional crashes of the Leader node and also situations

Ordering service size - Status	Transaction	Success	Failures	Node crashes	Leader changes	Network traffic (# messages)
5 - Stable	Create	15000	0	0	0	1.64e+7
	Change	15000	0			
5 - Unstable	Create	14340	660	41	6	2.90e+7
	Change	13919	1081			
10 - Stable	Create	15000	0	0	0	3.29e+7
	Change	15000	0			
10 - Unstable	Create	14175	825	69	7	4.63e+7
	Change	13875	1125			
20 - Stable	Create	15000	0	0	0	6.43e+7
	Change	15000	0			
20 - Unstable	Create	14104	896	79	5	1.46e+8
	Change	14288	712			

TABLE I: Data obtained by measurements on ordering services of various sizes and states.

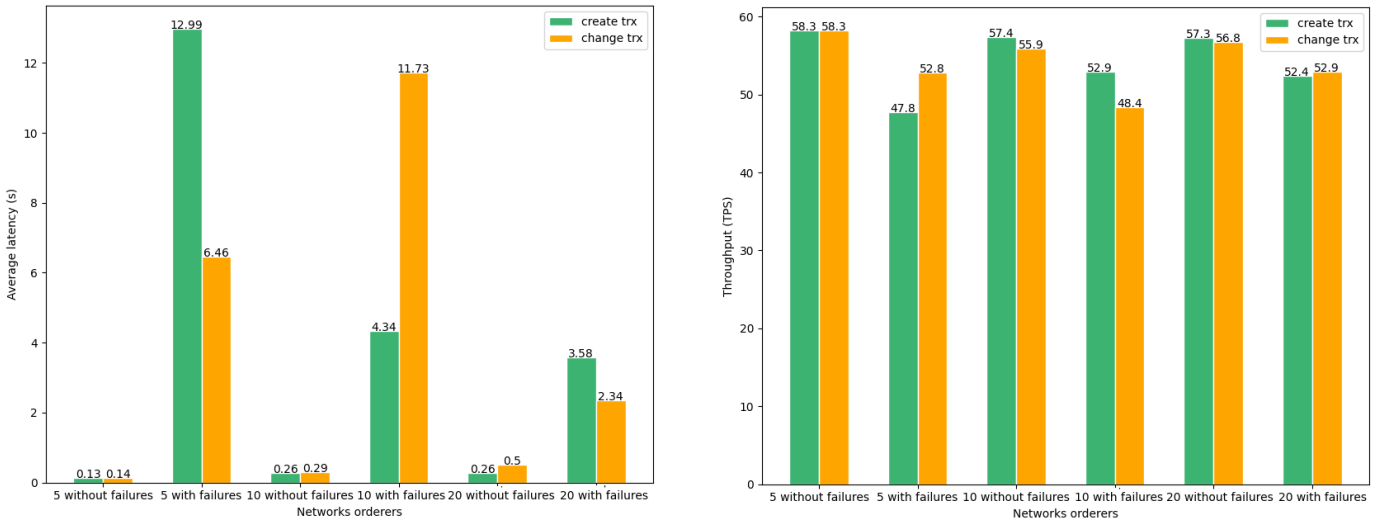

Fig. 6: Avg. transaction latency and throughput comparisons between instances.

in which the number of crashed nodes is higher than the Raft crash fault tolerance ($> 50\%$ of cluster size). In fact, before and during the Leader election, no transaction request can be served by the ordering service and, as such, they are all rejected upon reception. Moreover, we can also see how, in situations of network instability, the number of messages increases not only with respect to smaller-sized networks, but also with respect to stable networks of the same size. This increase is caused by the previously mentioned multiple Leader elections, which require the participation of all the network orderers and consequently generate an high amount of traffic.

In terms of performances, we can see from Fig. 6 how the increasing network size also increases the transaction latency. This is mainly caused by the required update of the nodes ledger, when a block has been successfully created. In fact, since the transaction latency is computed in relation to when its effects have been reflected accross all the network, a larger-sized network is reasonably requiring an higher amount of time to update all nodes. Consequently to this latency increase, the transaction throughput is affected aswell (Fig. 6) with slightly $(0.1 \sim 1.5)$ lower rates of elaborated transactions.

In this regard, the blockchain work is the most significant metric, since it provides us with a clear network performance indicator, in relation to the consensus cluster size. In particular, we can see in Fig. 7 how this value increases with the network size, thanks to the low loss in performance, if compared with the actual size increase.

The network stability also has a very significant impact on the overall performances, as we can see in Fig. 6. In fact, the failures caused by node crashes greatly influenced the transaction throughput, which only considers the successfull transactions performed in the testing time window. Moreover, even focusing on the average transaction latency, we can see how the downtime and traffic generated by the Leader elections and the exceeding of the CFT limits, has caused a

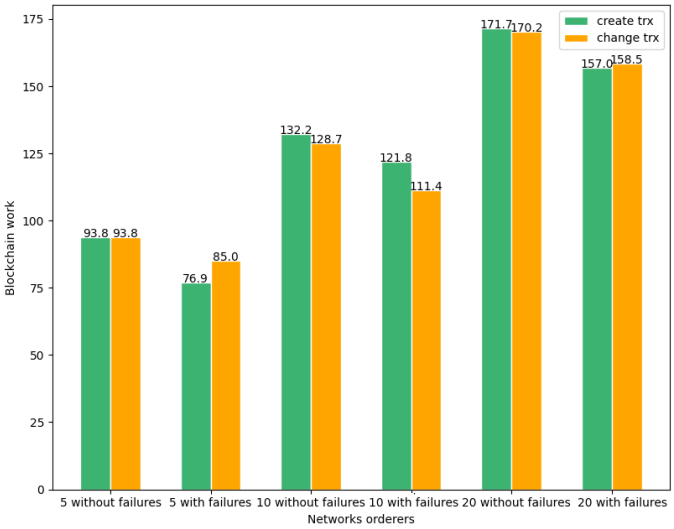

Fig. 7: Blockchain work comparison between instances.

significant increase in the time taken to process transactions. In this regard, the increase in size of the ordering service cluster is shown to increase the resilience of the network to severe performance degradation. As we can see in Fig. 6 with both a decrease in latency and increase in transaction throughput if compared with the smaller-sized networks. This is a consequence of the larger number of nodes required to exceed the limits of the crash fault tolerance.
The blockchain work metric, being directly related to the transaction throughput, also reflects the previously mentioned performance degradation.

B. Results conclusions

Overall, is possible to assert that, in stable network situations, the Hyperledger Fabric implementation of Raft performs rather well also in term of scalability. Considering the larger amount of time required to reflect the transactions effects over the whole network, a loss of just few TPS while increasing the network size from 5 to 20 nodes is reasonable. It seems, in fact, that the larger amount of traffic that can be identified when increasing the network size does not severely impact the actual performances of the system.
On the other hand, in unstable network situations, the system performs significantly worse. The multiple elections required to restore the network to an operational state generate an increasingly larger communication overhead with larger-sized networks. Moreover, the ordering service downtime caused by the Leader elections exacerbates the problem even more, reducing the amount of requests actually carried out by the system. However, as we can see on the 20 ordering nodes network, the previously described losses in performance caused by the additional traffic are counterbalanced by the stability of a larged-sized ordering service. We must, in fact, consider that a wider ordering service ensures that situations in which more than half the nodes crash are less common. As such, we can consider this as a balanced trade-off between scalability and availability.
Still, this degradation of both performance and availability of the Fabric channel, caused by Raft Leader elections, may negatively affect the fitness of the algorithm for the purpose of achieving consensus on the Fabric blockchain. In fact, considering that a single ordering node may take part in multiple Fabric channels at the same time, the communication overhead of a single unstable channel could, in the end, impact the performances of a greater portion of blockchain.
Limiting the membership of an ordering node to a single channel could be a partial solution, to reduce the overall impact of a single channel ordering service instability. Nonetheless, the amount of messages required to manage Leader node elections, which are a paramount feature of the Raft algorithm, is an significant limitation for its implementation in wide blockchain networks.

VIII. Related works

In this section we briefly summarize some of the related works we explored, in the context of applying distributed consensus algorithms to obtain consensus in the blockchain environment.
Dodo Khan, Low Tang Jung, Manzoor Ahmed Hashmani, and Moke Kwai Cheong in Reference [19], performed some scale tests on the Hyperledger Fabric blockchain. While focused on the same network, their work does not make direct references to the Raft algorithm and only provides data about standard network situations. Our work, on the other hand, is aimed towards measuring performances of a Raft-based ordering service, also in situations of unstable network.
Dongyan Huang, Xiaoli Ma and Shengli Zhang in Reference [15], discuss the performance of the Raft consensus algorithm in networks with non-negligible packet loss rate, by focusing on the probability of *network split*. In their work, they define network splits as situations in which half of the nodes are out of the current Leader's control, which is generally caused by meaningful packet loss. This is also a significant difference with respect to our effort, since we performed scalability tests and architectural analysis also in standard network situations or with managed faults.
Md Sadek Ferdous, Mohammad Jabed Morshed Chowdhury, Mohammad A. Hoque and Alan Colman, in Reference [16], provide a general analysis of various blockchains, including their characteristics and consensus algorithms. Hyperledger Fabric is also discussed in this work and, as such, this was quite helpful in providing us with a basic understanding of their blockchain architecture, approach and differences with other existing networks.
Wei Fu, Xuefeng Wei and Shihua Tong, in Reference [17] discuss the Hyperledger Fabric implementation of the Raft algorithm, by also providing some performance testing. Still, this work is mainly focused on proposing a new alternative to the Fabric implementation of Raft, called AdRaft. As such, most of the measurements provided are related to their proposal and not made with the scope of validating the fitness of Raft as blockchain consensus algorithm.

IX. Conclusion & self assessment

Finally, in this section we draw our conclusions with respect to the work we discussed and critically evaluate the whole project experience.

A. Outcome of the work

In this report we presented an analysis we conducted on a real world implementation of the Raft algorithm for achieving blockchain consensus, which is the Hyperledger Fabric ordering service. In our work we managed to identify possible weaknesses and strengths of the implementation, with relation to the expected requirements of the application scope. More specifically, we verified characteristics of the ordering service such as: scalability, availability and performance variation, with both absence and presence of transaction failures.
The results of the tests we performed, indicate that in standard network situations there is a noticeable degradation of performance, with the increase of the ordering service cluster size. However, in instances with multiple node crashes, there is a reasonable trade-off between this performance degradation and an increase in stability, with the increase in size of the ordering service cluster. Moreover, we were also able to correlate the performance degradation to the downtime and traffic caused by Raft Leader elections, which generate a significant communication overhead, especially in the context of wider networks.
We can also concretize the results we obtained in a realistic scenario. In fact, when designing a network and deciding which system configurations to use, there are important elements to consider. If the blockchain (or the single channel) being instantiated relies on low performance hardware and there is a requirement for high transaction throughput, a small-sized network is a sound choice. On the other hand, if the system requires high availability, even at the expenses of a moderate loss in performance, a wider ordering service cluster is the most ideal choice, provided that the network is able to withstand the larger amount of traffic generated.
As such, even considering the losses in performance present when scaling up to 20 ordering nodes, the Raft algorithm is able to provide important positive results in handling situations of network instability.

B. Self assessment

In this project experience we were able to elaborate on the Raft algorithm features, characteristics and limitations, in relation to the real world application that is Hyperledger Fabric. The Raft algorithm limitations are quite clear, and often mentioned as reasons for which real world applications of such algorithm are difficult to realize. Still, with our work we understood that some specific use cases, namely *permissioned blockchains*, can actually be an ideal environment for such endeavour. In fact, the absence of byzantine failures is a strong assumption for the algorithm, and the management of the traffic caused by a Leader-Follower scheme can be difficult. This second point in particular was the focus of our effort, in order to prove that bottlenecks and downtime caused by relying on a single node for most operations could be problematic for the whole system.
As previously discussed on the outcome of the work, we managed to obtain a reasonable understanding of what the impact of the Leader election feature can be on the system, both in positive and negative terms. However, we also understand that our testing was limited by the tools at our disposal. It was unfeasible for us to instantiate a network of more than 20 ordering nodes on a local environment, and as such we were not able to verify the behaviour of the ordering service in wider networks. Moreover, we did not perform tests with multiple channels sharing some of the same nodes, which would have most likely impacted the overall performances of the network. We also noticed that the different sized networks in the unstable scenarios, have a different probabilistic distribution of the crashes of the nodes. As such, the probability of crashes of the Leader is decreasing with the size of the network, which consequently influences the resulting system's performances.
Still, considering the results we managed to obtain, we can assert to have reached our objective of exploring the application of the Raft algorithm to a real-world scenario.

References

[1] "What Are Distributed Systems?" https://www.splunk.com/en_us/data-insider/what-are-distributed-systems.html.
[2] "Raft Consensus Algorithm" https://www.geeksforgeeks.org/raft-consensus-algorithm/.
[3] "Consensus (computer science) - Wikipedia" https://en.wikipedia.org/wiki/Consensus_(computer_science).
[4] "Blockchain - Wikipedia" https://en.wikipedia.org/wiki/Blockchain
[5] Shijie Zhang, Jong-Hyouk Lee "Analysis of the main consensus protocols of blockchain".
[6] Schär, Fabian "Blockchain Forks: A Formal Classification Framework and Persistency Analysis".
[7] "Hyperledger Fabric - The ordering service" https://hyperledger-fabric.readthedocs.io/en/latest/orderer/ordering_service.html.
[8] "State machine replication - Wikipedia" https://en.wikipedia.org/wiki/State_machine_replication.
[9] "Byzantine fault - Wikipedia" https://en.wikipedia.org/wiki/Byzantine_fault
[10] "Hyperledger Fabric (Raft consensus) step-by-step setup" https://medium.com/theta-one-software/hyperledger-fabric-based-proof-of-concept-using-raft-consensus-algorithm-786bcdb98e5a.
[11] "Hyperledger Fabric - Smart Contracts and Chaincode" https://hyperledger-fabric.readthedocs.io/en/release-2.2/smartcontract/smartcontract.html.
[12] "Hyperledger Fabric - Hyperledger Fabric Model" https://hyperledger-fabric.readthedocs.io/en/release-2.2/fabric_model.html.
[13] "Behind the Architecture of Hyperledger Fabric" https://www.ibm.com/blogs/research/2018/02/architecture-hyperledger-fabric/.
[14] "Architecting Enterprise Blockchain Solutions" - Joseph Holbrook https://books.google.it/Architecting_Enterprise_Blockchain_Solutions.
[15] Dongyan Huang, Xiaoli Ma and Shengli Zhang "Performance Analysis of the Raft Consensus Algorithm for Private Blockchains".
[16] Md Sadek Ferdous, Mohammad Jabed Morshed Chowdhury, Mohammad A. Hoque and Alan Colman "Blockchain Consensus Algorithms: A Survey".
[17] Wei Fu, Xuefeng Wei and Shihua Tong "An Improved Blockchain Consensus Algorithm Based on Raft".
[18] "Hyperledger Blockchain Performance Metrics - White Paper" https://www.hyperledger.org/learn/publications/blockchain-performance-metrics.

[19] Dodo Khan, Low Tang Jung, Manzoor Ahmed Hashmani, and Moke Kwai Cheong ”Empirical Performance Analysis of Hyperledger LTS for Small and Medium Enterprises”.
[20] ”Oracle Blockchain Platform adds the Raft Consensus Algorithm” - https://blogs.oracle.com/blockchain/post/oracle-blockchain-platform-adds-the-raft-consensus-algorithm.