UNIVERSITY OF PISA

AND

SANT' ANNA

SCHOOL OF ADVANCED STUDIES

MASTER DEGREE IN

COMPUTER SCIENCE AND NETWORKING

# Discovering and Securely Storing a Network Topology

Candidate

**Francesco Balzano**

Supervisor

**Fabrizio Baiardi**

Academic Year

2017/18

# Abstract

We present an automated tool to discover the topology of a network and to securely store it in a Blockchain. Our tool consists of two modules. The first one implements the network topology inference algorithm. This algorithm that has been defined according to state-of-the-art techniques can infer a network topology even in cases where only partial information is available. The second module is the Blockchain related one. It adopts a fast, energy-efficient consensus algorithm and is tailored to store topology information. The two components are independent entities and experimental results confirm that their integration results in an accurate network topology reconstruction which is resilient to tampering attempts.

# Contents

# 1. Introduction

The number of devices connected to IP networks is continuously growing and will reach 28.5 billion in the near 2022 according to a Cisco forecast.

This increase in the number of networked devices is due both to the increasing number of people having access to the Internet and to the rise of machine-to-machine (M2M) communications.

M2M communications refer to direct communications between devices that do not involve the presence of human beings. It is foreseen that M2M connections will be more than half of the global connected devices and connections by 2022 [12].

As a consequence, IP networks are becoming much more widespread and complex. This influences both the Internet, where the Internet Service Providers (ISP) face the challenges deriving from always increasing bandwidth requests, and the Local Area Networks (LAN), that should accommodate a larger number of devices usually involving M2M communications too.

Keeping into account this scenario, we have reviewed the current research on network topology inference to implement an automated tool that faces such complexity by providing a map of the network.

Our tool is based on state-of-the-art techniques to infer the network topology even in cases where only partial information is available and it can detect changes in the topology, i.e. the addition or the removal of a node.

The maintenance of an accurate knowledge of the network topology is mandatory to implement several critical network management tasks such as network diagnosis and resource management.

Our tool periodically checks the reachability of network nodes, thus enabling the discovery of faulty nodes and network partitions in a timely fashion. It also captures and analyzes traffic, to support the discovery of new nodes and edges. Finally, it returns a graphical representation of the inferred network topology, thus helping network administrators to decide, for instance, where to add new routers and if the current hardware is correctly configured.

Although researchers usually deploy their network topology inference tools to target the Internet, we have chosen to target LANs.
As we already said, LANs are becoming complex networks that connect several IP devices. Communications inside LANs involve both users, which are connected via PCs and smartphones, and machines. Examples of M2M communications in a LAN are SCADA (Supervisory Control and Data Acquisition) systems, that is control systems used to monitor and operate industrial processes, access control systems, to enforce physical security, surveillance applications using IP cameras and fire detection systems, which use the IP protocol to notify a remote control room of a fire alarm.
The deployment of our network discovery tool in this scenario may provide benefits for both safety and security.
Indeed, we have made one step ahead with respect to pure network topology inference by implementing a blockchain to store the topology information the tool returns.
Then, we simulated some attacks to the topology stored into the blockchain to show that, under proper assumptions, the blockchain can resist to the tampering attempts carried out by malicious nodes.
To the best of our knowledge, the use of a blockchain to securely store the topology information returned by an automated network discovery tool is a topic not yet investigated in literature.

## 1.1 Thesis Goals

This thesis implements and evaluates a tool that consists of two cooperating but independent modules.
The first module implements the network topology inference algorithm. We set up a collection of virtual test networks modelling common use cases in a LAN scenario and ran the algorithm on these networks. The algorithm performance is assessed by comparing the original topologies with the inferred topologies.
The second module implements the blockchain, that securely stores the topology the algorithm returns. The information on a system topology is critical because it describes the structure of the network. Furthermore, its analysis supports the detection of faulty nodes,

network partitions and malicious nodes. In order to make the topology information tamper-proof we store it in a blockchain. Then, we simulate some attacks to the blockchain by malicious nodes and assess the results of such attacks.

## 1.2 Thesis Structure

The thesis is organized as follows.

**Chapter 2** presents a survey of the current state-of-the-art techniques in the fields of network topology inference and blockchain.
We characterize network topology inference by describing the various levels where the inference may be carried out and the metrics generally used to assess the inferred topologies. We describe some existing algorithms and tools to infer the topology at router level with particular emphasis on iTop [27], that is the basis to implement our tool.
Then, we describe the main features of a blockchain by looking at the Bitcoin's blockchain, which was the first one ever implemented [35]. Since the Bitcoin consensus protocol is unsatisfying for our purposes, we present the Ripple Protocol Consensus Algorithm (RPCA), the consensus protocol powering the Ripple distributed ledger [37]. Finally we introduce Algorand, an emerging solution which should guarantee fast agreement among participants and network partition resilience [10, 11, 22].

**Chapter 3** describes the implementation of the network topology inference module. The implemented algorithm is based on iTop.

**Chapter 4** describes the implementation of the blockchain module. The implemented blockchain is based on Ripple.

**Chapter 5** discusses the simulations carried out for both the network topology inference algorithm and the blockchain.

**Chapter 6** reports the conclusions of this thesis and outlines further developments.

# 2. Related Works

This chapter reviews state-of-the-art techniques for both network topology inference and Blockchain.

## 2.1 Network Topology Inference

Network topology inference is a discipline that has been developed in parallel with the growth of Internet. As the Internet was becoming more and more big and complex, it emerged the need for an accurate spatial analysis of it to manage such complexity. Network planning, optimal routing algorithms, failure detection measures are only a small subset of the applications that could benefit from such analysis. Whichever is the case, the starting point is to be able to access information on the network topology and, in the best case, a complete description of the topology. Since this is seldom possible, because network operators are not willing to divulgate the topology underlying their Autonomous Systems(AS), researchers have started to develop techniques to infer, model and generate the Internet topology at different levels of granularity.

Although the developed network inference tool is devised to work in local area network (LAN) settings, this paragraph will regard the Internet too. The reason is that network topology inference techniques presented in literature usually target the Internet scenario. Hence, to offer a full view of the landscape we will review the selected works in their entirety, although our focus will be on the subsets useful for the developed tool and its context.

### 2.1.1 Topology Characterization

The metrics used to characterize and to compare topologies are taken from graph theory.

The real topology is usually referred to as *Ground Truth* (GT) topology. It is represented by the graph $G_{GT} = (V_{GT}, E_{GT})$, where $V_{GT}$ is the set of nodes and $E_{GT}$ is the set of edges, which may be either directed or undirected. The metrics most commonly used to compare different topologies are the following:

**Node Degree:** The degree of a node is the number of edges connected to the node.

**Average Degree:** The average node degree gives information about the connectivity of a network. Higher values indicate that the network is more connected and thus more robust. Let us consider an undirected graph with $n$ nodes and $m$ links: the average node degree is defined as $\bar{k} = 2m / n$.

**Degree Distribution:** The degree distribution $P(k)$ of a network is the fraction of nodes in the network with degree $k$. Thus if there are $n$ nodes in total in a network and $n_k$ of them have degree $k$, we have $P(k) = n_k / n$. The degree distribution provides a more detailed view of the network with respect to the average node degree.

**Joint Degree Distribution:** The joint degree distribution (JDD) is the probability that a randomly selected edge connects $k_1$- and $k_2$- degree nodes: $P(k_1, k2) = m(k_1, k_2) / m$ , where $m(k_1, k_2)$ is the total number of edges connecting nodes of degree $k_1$ and $k_2$. JDD contains more information about connectivity in a graph than degree distribution. Indeed, it provides information about the neighborhoods of a node because from $P(k_1, k_2)$ it is always possible to deduce the average degree $\bar{k}$ and the degree distribution $P(k)$.

**Clustering:** clustering is a measure of the degree graph nodes tend to cluster together. It is possible to give a global definition, if we look at the entire graph, or a local definition, if we look at the single nodes inside the graph. If $\overline{m}_{nn}(k)$ is the average number of links between the neighbors of k-degree nodes, local clustering is the ratio of this number to the maximum possible number of such links: C(k) = $2\overline{m}_{nn}(k\ ) / k(k - 1)$. Instead, the global clustering coefficient is based on triplets of nodes. A triplet consists of three nodes that are connected by either two (open triplet) or three (closed triplet) undirected ties. The global clustering coefficient is the number of closed triplets over the total number of triplets (both open and closed) in the graph.

**Coreness:** The k-core of a graph is a maximal subgraph where each node has at least degree $k$.

**Shortest Path Length:** The shortest path length distribution is the distribution of the probability that two nodes are at minimum distance *x* hops from one another.

**Betweenness:** Betweenness is a measure of the centrality of a node inside a graph. It is given by the number of shortest paths passing through a node.

**Spectrum:** The spectrum of a graph is the set of eigenvalues of the associated adjacency matrix.

### 2.1.2 Network Topology Levels

The most detailed network topology taxonomy is due to Donnet and Friedman [16]. They identify three different levels at which the network topology may be described: the *link layer topology*, the *network layer topology* (also referred as the Internet topology) and the *overlay topology*.

The link layer topology deals with the layer 2 of the ISO/OSI stack and describes how data link layer devices, bridges and switches are interconnected.

The network layer topology deals with the layer 3 of the ISO/OSI stack and is further divided into four sub layers:

- The *IP interface level* defines a network whose nodes represent the IP interfaces of routers and hosts.
- The *router level* defines a network where nodes represent single routers. This network graph is computed by aggregating the IP interfaces belonging to the same router via techniques called *alias resolution*.
- The *point-of-presence level* is computed by aggregating into a single node the routers that are geographically co-located.
- The *Autonomous System (AS) level* defines a network where nodes represent the ASes while links represent business relationships between ASes. Network information is gathered from inter-domain routing information and address databases.

The overlay topology level describes the topology of an overlay network, such as a peer-to-peer system. Topology inference at this level is protocol-dependent.

The focus of the present work is on the network layer topology. In particular, we follow the most common convention, embraced by Haddadi *et al.*, that classifies the network layer in just two categories: router level and AS level [26].

### The router level

At the router level the network is modeled as a graph where the nodes represent the routers (or the end hosts) and the edges represent physical connections between routers. At this level the main tool used to discover the network topology is *traceroute*. Traceroute is a networking tool created by Van Jacobson in 1989 to discover the path that a packet takes inside the network. The topology inferred with traceroute could be different from the real topology for a multiplicity of reasons that we describe in section 2.1.3.

Traceroute can infer a network topology where the nodes represent the network interfaces of the routers. In order to obtain a topology in which each node represents a router, alias resolution techniques are applied to the inferred topology. In other words, we consider alias resolution techniques as an optimization step to be applied after the network topology inference to produce a more accurate router level topology.

### The AS level

An Autonomous System (AS) is either a single network or a collection of networks administered by a single entity, typically an ISP (Internet Service Provider) or a very large organization. Each AS is identified by a unique 16-bit number assigned by the internet assigned numbers authority (IANA).

At the AS level, the network is modeled as a graph where the nodes represent the ASes and the edges business relationships between pairs of ASes. In other words, the edges connect pairs of ASes that can exchange traffic with each other.

There are two main sources of information to infer the AS level topology: internet registries and BGP routing information [41].

Network topology inference at AS level is out of the scope of the present work. In the following, we will focus on the router level.

## 2.1.3 Network Topology Inference at Router Level

There are basically two methods to infer the network topology at this level that use, respectively, *network tomography* techniques or the *traceroute* tool.

**Network tomography**

Network tomography is the inference of the internal behavior and topology of a network based on end-to-end network measurements [8,14]. There are various fields of application for network tomography, included network topology inference. Additive metrics such as delay and packet loss are collected using monitors at the edge of the network. Usually, a single sender is employed, and the metrics are collected by sending either unicast or multicast packets to a set of receivers [17,18]. Then, the monotonicity of the aforementioned metrics is exploited to infer the logical network topology.

The advantage of this method on traceroute-based methods is that it is not affected by internal, non-cooperative routers. The drawback is that it only infers a logical topology that can be very different from the physical one. Indeed, each vertex of the logical topology represents a physical network device where traffic branching occurs, that is, where two or more source–destination paths diverge. This means that if a packet traverses a linear chain of *k* routers before branching, those *k* routers will be represented as a single, logical node in the inferred topology.

We believe that network tomography is not yet a valid alternative to traceroute-based methods for network topology inference. Indeed we are interested in inferring a topology that is as close as possible to the real topology and, at the time of writing, this goal is better achieved with traceroute-based methods.

**Traceroute-based methods**

Traceroute is a tool used to trace the sequence of nodes that a packet encounters on its way up to a final host. It allows the sender to reconstruct the route taken by the packet to reach the destination. The idea underlying all traceroute-based methods is to collect enough traces by sending traceroute probes to a wide set of receivers. Then, the collected traces are combined to infer the network topology.

There are three variants of traceroute, respectively based on ICMP, UDP or TCP packets.

The classical version of traceroute uses Internet Control Message Protocol (ICMP) packets. The sender sends ICMP *Echo Request* packets with increasing value of the *Time To Live* (TTL) field, starting from one, towards the destination. According to the IP protocol, a router decrements by one the TTL field of the IP header before forwarding any packet. When a packet with TTL equal to one reaches a host, the host discards that packet and sends an ICMP *Time Exceeded* packet back to the sender. The traceroute tool uses the IP source address of these returning packets to reconstruct the list of hosts traversed by the packet.

Since some hosts may be configured not to answer to certain types of ICMP packets, or firewalls may filter out ICMP packets, TCP and UDP variants have been devised to overcome these limitations. UDP traceroute sends UDP packets to an invalid destination port, while TCP traceroute sends TCP *SYN* packets to a well-known port, like the port 80 of web servers.

In Linux, the traceroute utility is called *traceroute* and it sends by default UDP packets to elicit ICMP *Time Exceeded* responses. The flag *–M* signals the tool to use ICMP (-*M I* ) or TCP (-*M T* ) packets. In Windows, the traceroute utility is called *tracert* and can only send ICMP probes.

We discuss now the main limitations of traceroute-based methods.

The first limitation deals with **aliases**. Since routers have more than one network interface card (NIC), and thus more IP addresses, the same router may participate in trace collection with different IP addresses, according to the interface that receives the packet. The trace collection phase is not affected negatively by the presence of multiple interfaces per host. The problem arises when the collected traces are merged to infer the actual topology. Without further information there is no way to tell if there are subsets of IP addresses that belong to the same router, and thus should be mapped to the same node in the inferred topology. So a 1-to-1 relationship between IP addresses and nodes is established, which results in the presence of false positive nodes in the inferred topology. To overcome this problem, **alias resolution** techniques have been devised that try to identify the interfaces belonging to the same router.

As an example, we present a widely known alias resolution technique called *address based method* [24]. This method assumes that a router returns an ICMP error message to the probing host through the interface that belongs to the shortest path towards the host. For this reason the host simply sends a UDP probe with a high port number to a list of target IP interfaces, causing the corresponding routers to answer with an ICMP *Port Unreachable* message. If two or more interfaces answer with the same IP source address, then they belong to the same router. The drawback of this method is that it does not work with routers that do not generate ICMP messages. This technique has been implemented in many tools, such as iffinder [28] and Mercator [24].

Several other alias resolution techniques exist that exploit, for example, the ID field of the IP header (*IP identification based methods*), the similarities in router host names (*DNS based methods*) and the Record Route IP option [25].

In our work we haven’t considered alias resolution techniques because we deal with LANs and so we can take advantage of network owner’s cooperation to resolve aliases.

The second limitation deals with **non-cooperative routers**, that is routers which do not take part, fully or in part, to the trace collection process. A non-cooperative router is *anonymous* if it does not answer to the source when it is the destination of the probe packet but it forwards probe packets to the next hop. A non-cooperative router is *blocking* if it drops all the traceroute traffic, never forwarding packets to the next hop or sending responses back to the source. On the other hand, a router that correctly participates in the traceroute operations is *responding*. There are several reasons why non-cooperative routers may appear along the probed path: router configurations, privacy policies and firewalls are some examples. The impact of non-cooperative routers on traceroute-based methods can be severe, preventing them to infer an accurate topology in real scenarios. Anyway, there are methods that allow to overcome, at least partially, this limitation. In the implementation of our tool, we have looked at those methods.

Finally there are some limitations inherent to traceroute itself. For instance, the traceroute probes may be unable to see links that are used as backup links. The problem is even more severe in case of load-balancing routers. Here, the packets sent to trace the path towards a destination can take different routes, resulting in a wrong inferred path where some existing links are not discovered and some nonexistent ones are added. Figure 1 from [2] shows such a case. To overcome this problem B. Augustin et al. have developed *Paris Traceroute*, a new kind of traceroute that controls the probe packet header fields so that all the probes towards a destination follow the same path in the presence of per-flow load balancing [3].

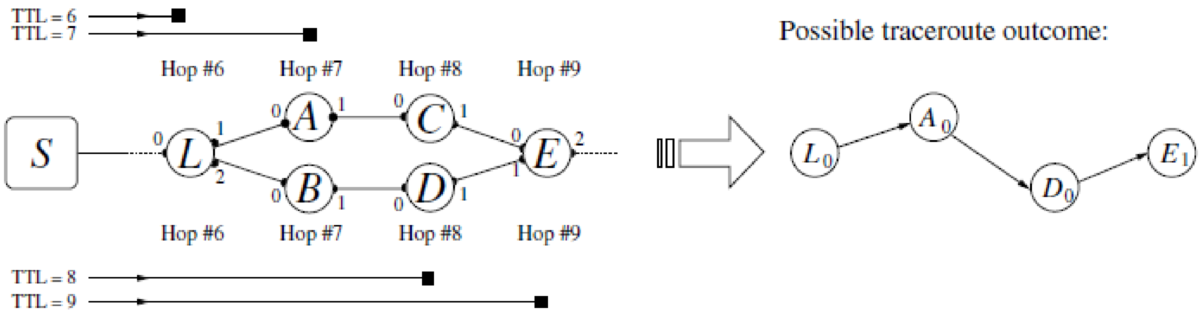


**Figure 1: The inferred topology contains both false positive and false negative edges.**

Lakhina et al. state that traceroute sampling introduces a bias with respect to the real topology. In particular, they show that the inferred node degree distribution computed for a network with Poissonian degree distribution exhibits power-law properties after traceroute sampling [30].

A successive study of Achlioptas et al. [1] confirms this result.

Marchetta *et al.* assert that traceroute may generate misleading information in the inferred paths in terms of traversed devices. In particular, they show that traceroute could poorly estimate the number of equal-cost routes to the destination, the presence of suboptimal routing and the routing stability [34].

### 2.1.4 Examples of Traceroute-based Methods

Although traceroute suffers from the above limitations, it remains the best tool to perform network topology inference at router level. Below we list some of the major, state-of-the-art network topology inference tools that use traceroute. Particular emphasis is posed on *iTop*, which we believe has the best features for a LAN scenario. Our automated network topology inference tool is indeed heavily based on *iTop*.

**Scamper** is a tool that actively probes the Internet in order to analyze topology and performance [33]. It supports the well-known ping and traceroute techniques, as well as Paris traceroute. It is a distributed system that includes a number of remote monitors located all over the world and a central host that controls them. Traceroute probes are sent in parallel to the destinations. Both IPv4 and IPv6 probing is supported. It is maintained by the Center for Applied Internet Data Analysis (CAIDA) and the collected datasets are available to the community. It implements four different alias resolution techniques. Non-cooperative routers are not taken into account. Scamper is the successor of the former mapping system *Skitter* [28].

**Merging Nodes** (MN) is a traceroute-based method to infer network topology in presence of anonymous routers. The collected traces are organized into an *induced topology* with duplicated components with respect to the actual topology. The duplicated components are due to non-cooperative routers in the real topology. Merging conditions are then computed and applied to the components of the induced topology. In particular, the nodes belonging to the induced topology are grouped into disjoint *equivalent classes*. Finally, all the nodes in the same class are merged together and the inferred topology is derived. MN does not take into account the presence of blocking routers and does not provide alias resolution techniques [40].

**Isomap** is a traceroute-based approach that considers both anonymous and blocking routers. It builds an *initial topology* that contains one virtual router for each anonymous router found in the traces. When a blocking router is found no virtual router is added to the initial topology. Instead, it adds a link between the two last responding routers on the opposite sides of the unobserved section. This behavior can lead to underestimate the real topology, because routers in the middle may be hidden by the blocking routers at the edge of the unobserved section. Then, Isomap merges the nodes in the initial topology. The first phase merges into the same one all the nodes with the same neighbors, then a metric of distance is used to perform a further merge of all the nodes at a distance lower than a given threshold. Isomap does not provide any alias resolution technique [29].

**iTop** is a network topology inference strategy that considers both anonymous and blocking routers. Some monitors are put at the edge of the network to be inferred. Then, each monitor sends traceroute probes to a set of destinations, and the collected traces are sent to a central server called Network Operation Center (NOC). The NOC applies the algorithms in the paper to produce the inferred topology.
Holbert *et al.* [27] make the following assumptions. They call *Ground Truth* the real topology, and they model it as an undirected graph $G_{GT} = (V_{GT}, E_{GT})$, where $V_{GT}$ is the set of nodes and $E_{GT}$ is the set of edges. The monitors are a subset of $V_{GT}$, but they are considered external to the network. The authors assume the routing algorithm uses shortest paths and that the same path is used between any pair of monitors. They also assume that it is possible to determine the hop distance between any pair of hosts. This can be implemented by looking at the value of the TTL field in the header of IP packets that blocking routers do not discard. Such packets are not traceroute packets, but packets exchanged by cooperative routers on open ports. As an example, an HTTP request to a web server inside a demilitarized zone (DMZ) cannot be filtered out, or the web server will be useless.
Finally the authors assume that the traces are processed with some alias resolution technique at the NOC, before actually starting the inference process.

iTop operates in three phases. In the first phase, it combines the collected traces to build a *virtual topology*, $G_{VT} = (V_{VT}, E_{VT})$, that may include several duplicated components with respect to the ground truth topology. These duplicated components are due to non-cooperative routers. Indeed, when we find a non-cooperative router we can only add a placeholder, a *virtual router*, to the virtual topology, because we cannot infer whether we already found such a router in some previous trace. The problem arises because we own no information about the identity of non-cooperative routers. Instead, when cooperative routers are found in the traces we add a virtual router to the virtual topology only if such a router wasn't already added. Now, no discrimination problem arises, because we know the identity of the router.

The phases two and three try to remove the duplicated components (both edges and nodes) from the virtual topology, creating a *merge topology* $G_{MT}$ that should be as close as possible to the real topology. To this purpose the second phase computes the *merge options* for each link of the virtual topology. These options point out pairs of links in the virtual topology that can be merged together without impairing the structure of the inferred graph with respect to the real topology. The third phase actually merges the proper links of the virtual topology and it produces the merge topology.

*Phase I: Virtual Topology Construction*

The authors assume the availability of a set of monitors at the edge of the network. Firstly, each pair of monitors $m_1$ and $m_2$ compute their mutual *hop distance* $d(m_1, m_2)$, then each monitor execute traceroute towards the other monitors. At the end, the set of all the gathered traces will be sent from each monitor to the NOC. Each trace corresponds to a path in the virtual topology. The NOC parses all the traces and, for each trace, it adds a path to the virtual topology. When all the traces have been processed, the virtual topology is complete. In this section the authors discuss how to parse the distinct kinds of traces that can be found. The traces are parsed according to the the type of the routers along the path. The authors define five router types:

- *Responding* routers correctly take part to the traceroute process.
- *Anonymous* routers do not reply when the packet is addressed to them, but do forward the packets if they are not the final destination of the probe.
- *Blocking* routers drop all the traceroute packets they receive.
- *Non-Cooperative* routers may be either anonymous or blocking.
- *Hidden* routers may be anonymous, blocking or responding.

Non-Cooperative and Hidden routers are introduced because the lack of information may prevent the categorization of the routers if only the first three classes are used.

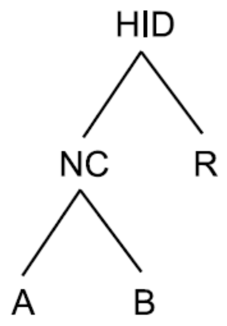


**Figure 2: Router type hierarchy.**

As far as concerns the traces, three different cases should be handled because the path between the monitors may include:

1. Responding routers only.
2. At least one anonymous router.
3. At least one blocking router.

Let us assume that monitors $m_1$ and $m_2$ are used to collect traces, and that only responding routers appear in the gathered traces. Here the path extracted from the traces will be something like $m_1$, $v_1$, $v_2$, … , $v_{n-1}$, $m_2$. Since all the routers are responding, it will suffice to add all the links and the nodes comprising the path to the virtual topology. These nodes are marked as 'responding' in the virtual topology.

If some anonymous router is present in the traces collected between $m_1$ and $m_2$, we will find in the traces a row full of '*' for each anonymous router. Anyway, the destination correctly answers to the traceroute probe, and so do all the other responding routers. Here a virtual router marked as 'responding' is added for each responding router, and a virtual router marked as 'anonymous' is added for each row full of '*' in the traces. Links are added between virtual routers to match the sequence extracted from the traces.

If the path between $m_1$ and $m_2$ includes at least one blocking router, then $m_1$ will not receive any answer from $m_2$ and vice versa. Only the routers before the blocking one will respond to the probes: all the following rows will be full of '*', meaning that no response was received. By merging the traces collected from $m_1$ and $m_2$ we produce the following trace:

$m_1, x_1, \ldots, x_i, \ldots, x_{n-j}, \ldots, x_{n-1}, m_2$, where $x_i$ and $x_{n-j}$ are the last known routers seen by $m_1$ and $m_2$ respectively. The trace fragments $m_1, x_1, \ldots, x_i$ and $x_{n-j}, \ldots, x_{n-1}, m_2$ are handled as described above, and so responding and anonymous routers are added accordingly. The fragment between the last observed routers $x_i, \ldots, x_{n-j}$ deserves instead special care. By using the hop distance, the NOC knows that, in general, there are $n-j-i-1$ unobserved routers between $x_i$ and $x_{n-j}$. If there is only one unobserved router in the middle, that is if $n-j-i-1 = 1$, then a virtual router, marked as 'blocking', will be added to the virtual topology, together with the links towards $x_i$ and $x_{n-j}$ . If, instead, there are more than one unobserved router in the middle, namely $n-j-i-1 > 1$, we mark as 'non-cooperative' the routers attached to $x_i$ and $x_{n-j}$ and we add them to the virtual topology. Indeed, we only know that such routers are not responding, but not whether they are anonymous or blocking. If there are other routers between the two non-cooperative ones, that is $n-j-i-1 > 2$, then we mark such routers as 'hidden' (because we do not know whether they are anonymous, blocking or responding) and add them to the virtual topology.

As an example, Figure 3 from [27] compares a ground truth topology to the virtual topology returned by the steps discussed above. Nodes A, E, F, G and I acts as monitors and the traces are collected by the following pair of monitors: A-E, A-F, A-G, A-I, E-F, E-G and G-I. The virtual topology overestimates the real topology because of non-cooperative routers.

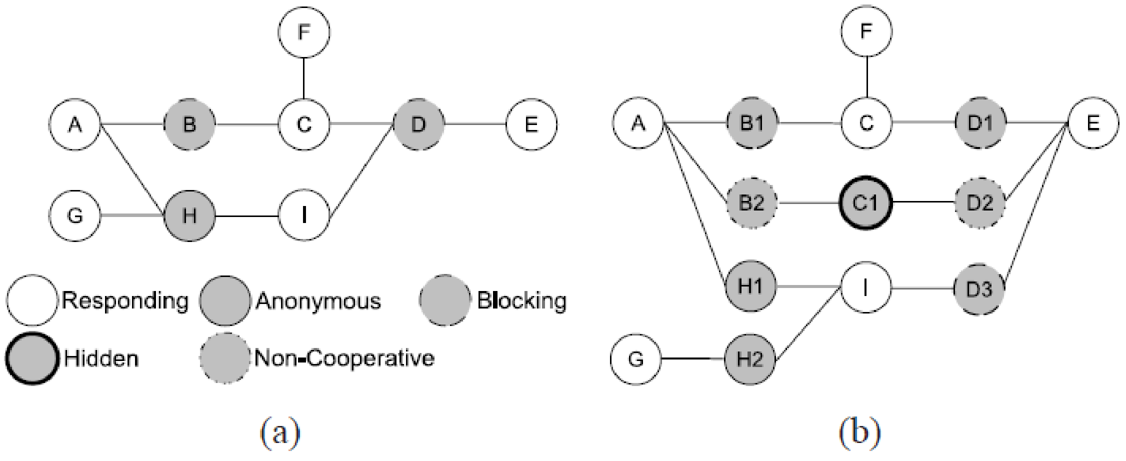


**Figure 3: GT (a) and virtual topology derived from traces (b).**

*Phase 2: Merge Options*

Let us denote by $G_{MT} = (V_{MT}, E_{MT})$ the merge topology. In order to infer $G_{MT}$ from $G_{VT}$, iTop identifies the valid merge options for each link $e_i$ in $E_{MT}$. Let $M_i$ denote the set of valid merge options for each link $e_i$. Initially, $M_i = \emptyset$ for each $e_i$ in $E_{MT}$. The merging option phase analyzes all the link pairs in the virtual topology and applies them three conditions in sequence. If all the three conditions hold, the merge option is valid for the pair, otherwise the pair is not eligible for merging. The three conditions are:

1. *Trace Preservation.* A merge option between two links satisfies this condition if the two links do not appear together in any path.
2. *Distance Preservation.* A merge option between two links satisfies this condition if their merging does not reduce the hop distance between any pair of monitors.
3. *Link Endpoint Compatibility.* A merge option between two links is valid if their endpoints can be combined without violating the hierarchy in Figure 2. The basic idea is that a merge can only increase the specialization of the involved nodes. Holbert et al. have compiled a table, reported in Table I, that shows the types of endpoints that compatible links can have. If two links have incompatible endpoints than the corresponding entry is marked as '-', otherwise the entry indicates the type of the resulting endpoints. If the matching endpoints of two links are responding routers, than the responding router must be the same in both links.

| | R-R | R-A | R-B | R-NC | A-A | A-HID | NC-NC | NC-HID | HID-HID | A-NC | B-NC | A-B |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R-R | - | - | - | - | - | - | - | - | R-R | - | - | - |
| R-A | - | **R-A** | - | **R-A** | - | R-A | - | R-A | R-A | - | - | - |
| R-B | - | - | **R-B** | **R-B** | - | - | - | R-B | R-B | - | - | - |
| R-NC | - | **R-A** | **R-B** | **R-NC** | - | - | - | R-NC | R-NC | - | - | - |
| A-A | - | - | - | - | A-A | A-A | A-A | A-A | A-A | A-A | - | - |
| A-HID | - | R-A | - | - | A-A | A-HID | A-NC | A-HID | A-HID | A-NC | A-B | A-B |
| NC-NC | - | - | - | - | A-A | A-NC | NC-NC | NC-NC | NC-NC | A-NC | B-NC | A-B |
| NC-HID | - | - | - | - | A-A | A-HID | NC-NC | NC-HID | NC-HID | A-NC | B-NC | A-B |
| HID-HID | R-R | R-A | R-B | R-NC | A-A | A-HID | NC-NC | NC-HID | HID-HID | A-NC | B-NC | A-B |
| A-NC | - | - | - | - | A-A | A-NC | A-NC | A-NC | A-NC | A-NC | A-B | A-B |
| NR-NC | - | - | - | - | - | A-B | B-NC | B-NC | B-NC | A-B | B-NC | A-B |
| A-B | - | - | - | - | - | A-B | A-B | A-B | A-B | A-B | A-B | A-B |

**Table 1: compatible endpoint classes and resulting classes after merging.**

*Phase 3: Merging Links*

The third phase merges the links in the virtual topology $G_{VT}$ to derive the merge topology $G_{MT}$. Initially $G_{VT} = G_{MT}$, then $G_{MT}$ is reduced by iteratively merging pair of links with valid merge options. Each merge combines two links in $E_{MT}$, overlapping their endpoints and thus reducing the number of network components accordingly. When no merge option remains, the merging process is concluded and the resulting merge topology is the final one.

Algorithm 1 shows the pseudo code of the merging phase.

**Algorithm 1: iTop Merging Phase**

```
   Input:   Initial iTop topology $G_{MT} = (V_{VT}, E_{VT})$,
            Merge options $M_i$ for each $e_i \in E_{MT}$
   Output:  Merged iTop topology $G_{MT} = (V_{MT}, E_{MT})$

1  while $\exists e_i \in E_{MT}$ s.t. $M_i \neq \emptyset$ do
2      $e_i = \text{argmin}_{e_j \in E_{MT}} |M_i|$
3      $e_j = \text{argmin}_{e_j \in M_i} |M_j|$
4      if $C(e_i, e_j)$ == true then
5          merge($e_i, e_j$)
6      else
7          $M_i = M_i \setminus \{e_j\}$
8          $M_j = M_j \setminus \{e_i\}$
9  return $G_{MT} = (V_{MT}, E_{MT})$
```

The algorithm selects all the link pairs with valid merge options (line 1), makes a compatibility check (line 4) and merges the two links if the check was successful, otherwise updates the valid merge options of each link by removing the other link, which is no more a valid option (lines 7 and 8). First the link $e_i$ with fewest merge options is taken (line 2), and then the link $e_j$ with fewest merge options among the valid options for $e_i$ is taken. The idea is that you can increase the probability of finding two links that can be merged together by taking two links with few merge options.

The compatibility check in the function $C(e_i, e_j)$ verifies if the endpoints of the two links are compatible according to Table I. The reason for this further check is that some previous merge could have changed the endpoint type of some link, and so some this may invalidate some merge options previously computed.

The function *merge($e_i$, $e_j$)* merges $e_j$ into $e_i$. Then, the endpoints of $e_i$ are changed according to Table I. The endpoint classes of $e_i$ and $e_j$ are matched to the first row and to the first column of the table, respectively. Then, all the paths in the merge topology that include $e_j$ are updated: $e_i$ replaces $e_j$ and the set $M_i$ is updated so that $M_i = M_i \cap M_j$. All the links having a merge option with both $e_i$ and $e_j$ preserve their merge option with $e_i$ only. The merge option is removed from all the links having a merge option with either $e_i$ or $e_j$. This implies that $e_i$ can be further merged only with those links that had a merge option with both $e_i$ and $e_j$. Finally $e_j$ is removed from the merge topology.

*Topology Evaluation*

Holbert et al. have evaluated the topologies inferred by iTop, Isomap and Merging Nodes by simulating both realistic and random networks. Two scenarios are evaluated: in the former both anonymous and blocking routers are present, in the latter only anonymous routers are present. The inferred topologies are compared to the real ones by applying metrics such as the number of nodes or links, the cumulative distribution of node degree, the betweenness centrality and the node degree distribution. For a detailed analysis of the results please refer to [27]. Here we only note that in the given scenarios, iTop outperforms the other approaches, returning results that are within 5% of the ground truth topology with regard to all the considered metrics.

## 2.2 Blockchain

A blockchain is a data structure that stores a secured, agreed-upon and shared ledger without requiring a trusted, centralized authority.
The blockchain technology is considered a disruptive one that could revolutionize all the fields requiring trust among parties. The basic point is that for the first time trust can be achieved in a distributed way, rather than in a centralized one. This has several implications because by removing the centralization point we remove a bottleneck, thus fostering services with unprecedented levels of scalability. It also means more trust, since the ledger is not stored in a single location but is instead shared and verifiable by all the participants. Finally, by removing the need for a central authority, like a financial institution, we cut the

costs of the offered services. The interest is rising not only from academia, but from industries and governments too. Industries are usually looking at blockchain as an enabling technology to provide smarter services at lower costs. As an example Siemens, a German multinational active in the fields of electrification, automation and digitalization, is experimenting in Brooklyn a microgrid where independent consumers produce energy from their own photovoltaic systems and then trade such energy exploiting a blockchain-powered platform, thus removing the need for any intermediary [38]. Governments are instead looking at blockchain to deliver digital public services with the highest standards of security and privacy. As an example, 27 States have signed the "European Blockchain Partnership" to cooperate in the establishment of a European Blockchain Services Infrastructure (EBSI) [20].

This section describes the Bitcoin blockchain, which was the first blockchain ever implemented. Since the Bitcoin consensus protocol is inefficient we present Ripple, a distributed ledger technology that trades some of the Bitcoin generality to provide a much faster consensus algorithm. Finally, we introduce Algorand, a novel blockchain that aims to guarantee fast agreement among participants and network partition resilience.

### 2.2.1 The Bitcoin Blockchain

A lot of different blockchains have been proposed and implemented in the last 10 years. Although each one has its own distinguishing features, the basic ideas underlying blockchain and distributed ledger technology are similar and date back to 2008, when someone under the pseudonym of "Satoshi Nakamoto" released the Bitcoin protocol.

**Bitcoin**

The first Blockchain was embodied in Bitcoin, an electronic paying system that allows users to securely exchange electronic cash (bitcoins) without relying on a centralized financial institution [35]. Bitcoin allows users to exchange electronic cash by broadcasting *transactions* over an unstructured peer-to-peer network where anyone can leave and join at any time. A transaction is basically an announcement that the sender makes to the entire network to tell that it has transferred some amount of money to one or more receivers. The Bitcoin protocol was the first one to solve the problems inherent to a completely decentralized paying system, most notably the *double-spending* problem, that happens when a user manages to spend twice the same money. This problem is peculiar of payment systems that lack a centralization point. Indeed, in traditional payment systems a trusted central authority collects, processes and validates every financial transaction in the network. Double spending can never happen because if a user tries to spend the same money in two different transactions the central authority will validate only the first and reject the second one, because it discovers that such funds have already been spent. This is possible because the central authority collects every transaction in the network, and the receiver of the transaction waits for the central authority to confirm that the payment was successfully carried out. At the opposite, in decentralized payment systems such as Bitcoin there is no central authority and thus no node in the network is guaranteed to receive and process all the transactions. This make the risk of double spend concrete: how is it possible to guarantee that double spend does not happen, if the validation of transactions is made by nodes that, in general, do not know the entire history of transactions? An attacker could create two transactions that spend the same money and then send one transaction to a validating node and the second transaction to another validating node. If the validating nodes do not know both transactions, the attacker reaches its goal, because it manages to have both transactions validated although its funds would allow only one transaction. Later the two validating nodes could receive the other transaction, but it would be too late because the attacker could have already convinced the two different receivers of the validity of its transactions.

This problem prevented the adoption of decentralized paying systems until Bitcoin solved it. The protocol to solve the double spending problem in a decentralized peer-to-peer network is based on the blockchain.

### Addresses

Bitcoin makes use of cryptography. The user must choose as private key a 256-bit number. This number should be picked randomly and there are some (loose) limitations on its value, because it must be a valid Elliptic Curve Digital Signature Algorithm (ECDSA) private key. Indeed the ECDSA is applied to the private key to generate the associated public key. While the private key should be kept secret, the public key is divulgated to all the nodes belonging to the Bitcoin network. A user uses its private key to sign its own transactions. The public key is instead used by validating nodes to check the authenticity of the signature. Public keys are also used to generate the Bitcoin addresses, that is the addresses that nodes use to receive payments in the Bitcoin network. A chain of SHA-256 and RIPEMD-160 hashing are made on the public key to generate the Bitcoin address, which is represented in Base58.

**Transactions**

A transaction represents a transfer of funds between one or more payers and one or more payees [39]. There is no notion of account in Bitcoin. In order to transfer funds a user has to demonstrate that it owns the funds that it wishes to transfer. This demonstration is not done by looking at its account balance, since accounts do not exist, but showing that in the past it was the payee of some transactions and that the funds that it received via those transactions have not yet been spent. A transaction is basically characterized by the following elements:

- **Input**: the payer indicates as input a list of transactions for which he was the payee, and whose funds is now willing to transfer to one or more payees.
- **Output**: the output of a transaction indicates the addresses of the payees and the amount to be transferred to each payee.
- **Fee** (Optional): An amount of funds is paid to the *miner*, that is the node, belonging to the Bitcoin network, that will add the transaction to the distributed ledger. Paying a bigger fee means assigning a higher priority to the transaction.
- **Public Key:** for each output, a transaction also indicates the public key of the receiver as a part of a script.
- **Script:** for each input, a transaction also indicates a script to specify a set of arbitrary conditions that must be met in order to transfer funds. In the most common case, the script contains a signature that is compared against the linked output transaction. If the public key specified in the output transaction matches the signature, then the payer has proved that it is the owner of the linked transaction.

We have reviewed the basic components of a transaction in the Bitcoin network and outlined how, using scripts and asymmetric encryption, the Bitcoin system can automatically verify if a payer really owns the funds that it claims. Anyway, we have not yet discussed of how Bitcoin has solved the double spending problem.
The adopted mechanism is called *Proof of work* and, though no formal proof exists of its correctness, it has actually worked in the Bitcoin network for 10 years.

**Proof of work**

As we said, transactions have to be validated before they can be appended to the ledger. This is done by special nodes called *miners.* Since miners could be malicious, some mechanism has to be devised to prevent that fraudulent transactions are appended to the ledger. This mechanism is called Proof of Work (POW). Firstly, miners take a set of transactions and group them to form a block. Then, they start the POW, which is essentially a challenge where the first one that guesses the correct value has the right to add the mined block to the blockchain. The POW involves scanning for a value that when the resulting block is hashed, such as with SHA-256, it results in a hash value that begins with a certain number of zero bits. On the average, this requires a work exponential in the number of zero bits and can be verified by executing a single hash.
When a miner solves the POW, it broadcasts the block on the network. The other miners check both the POW solution and the transactions inserted in the block and, if everything is correct, append the block to the previous block in their local copy of the blockchain, and start working on a new POW. In this way the blockchain, or ledger, is built one block after the other, and each miner maintains a local copy of it. It should be noted that a block is made of two parts:

- a header, containing a timestamp, a nonce (the POW solution) and two hashes;
- a payload, containing the actual transactions.

A blockchain is a tamper proof data structure because of the use of hash pointers. A hash pointer is a pointer to where some information is stored together with a cryptographic hash of that information. The block headers use hash pointers. When a new block is to be added to the blockchain, a hash of the previous block is computed and stored in the last appended block. Since SHA-256 is used, finding a collision is unfeasible, so every change in the previous block would be detected because it would led to a change in the computed hash value. Furthermore, since hash pointers are concatenated, any change in any previous block would be discovered, because all the hashes from the tampered block to the last appended block would be different from those previously computed.

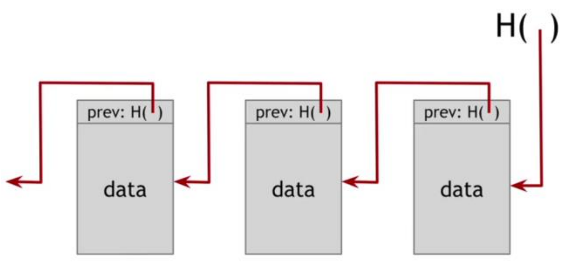


**Figure 5a: Hash pointers in a blockchain.**

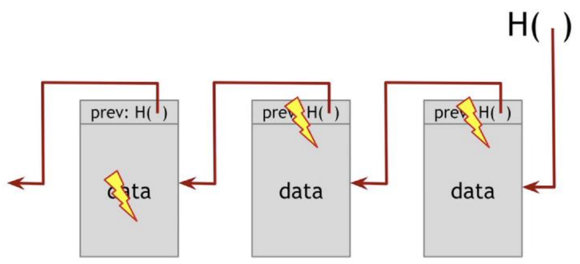


**Figure 5b: tampering the data in one block causes the change of the hash pointer value for all subsequent blocks.**

After describing POW, we can tell why it has been adopted and its drawbacks.

There are two main reasons for the adoption of the POW in the blockchain building process. First, it makes attacks unfeasible until honest nodes own at least 51% of the total computing power in the Bitcoin network.

Second, it is a way of building the distributed ledger that works in practice, and indeed the consensus on the state of the distributed ledger has always been reached from Bitcoin inception.

Bitcoin is a system based on incentives that rewards a miner every time it mines a new block. According to the Bitcoin protocol, miners should always work on the longest chain of blocks. Anyway, sometimes forks may happen so that two blocks in the blockchain have the same father. When there is a fork and the two forked branches have equal length, a miner is to choose one of the two branches for mining. When the next block is mined one branch will become longer, and all the (honest) miners will start mining on that branch. As a result, the miner that discovered the block that now has become orphaned wasted cycles of CPU, because it was working on a branch that has been abandoned. This implies that it will never receive the reward for mining the block.

Actually, the orphaned block could not be lost if the miner that created it can solve a series of consecutive POWs such that the abandoned chain becomes longer than the other one.

This is the behavior of a malicious miner. The miner could spend its money in a transaction, wait that some miner validates that transaction by inserting it in the newly mined block and get the service for which it paid for. At this point, it could try to fork the blockchain to create a new branch whose father is the last block that does not include that transaction. If it manages to create a branch longer than the existing one, then its transaction will not be considered validated by the Bitcoin nodes, because it does not belong to the longest chain. So the malicious miner would obtain a service for something that it did not pay, and those money could be spent again, thus carrying out a double spending attack. Anyway, if the computing power of the malicious miner is less than 50% of the total computing power of the network, the probability that it solves $k$ consecutive POWs decreases exponentially with $k$ [35]. Because in Bitcoin a transaction is considered confirmed only when at least 5 blocks follow it in the longest chain, and the expected computing power of all malicious miners in the Bitcoin network should be far away from 50% of the total, the creation of a new longest branch by solving six consecutive POWs is an event with negligible probability. Thus the Bitcoin blockchain is inherently secure and the consensus on the shared ledger’s transactions is eventually reached looking at the longest chain.

The main drawback of using a POW as a way of reaching consensus is that it is CPU greedy by nature. The majority of miners simply waste CPU cycles looking for the right nonce that solves the POW puzzle. This is the reason why miners usually join pools: to share the revenues when a miner of the pool solves the POW. Anyway, this activity is only justified by the high values of bitcoins, which allow miners to cover the costs for the electricity needed for mining. Since we are interested in blockchain for its security features, and our field of application targets LAN and not the Internet, we could tolerate to make some assumptions about the peer-to-peer network. These assumptions allow us to avoid the use of POW as a way of reaching consensus. Instead, we will look at the Ripple system [37].

Ripple allows the creation of a distributed ledger that has all the main features of Bitcoin’s blockchain. The consensus on the ledger state is reached by exploiting a distributed consensus algorithm that is much faster than POW and much more efficient. As an example, Bitcoin adds new transactions to the ledger about every 10 minutes, Ripple every few seconds. The analysis in [32] shows that running a Ripple server has the same energy needs of running an email server.

### 2.2.2 The Ripple Ledger

In Ripple the authors talk about ‘ledger’ rather than ‘blockchain’, but the substance is the same: a tamper-proof, append-only, distributed data structure where the transactions are inserted after they are validated through a consensus process.

Ripple uses the *Ripple Protocol Consensus Algorithm* (RPCA), a generic consensus algorithm that does not require fully synchronous communications among nodes and achieves consensus via collectively trusted subnetworks.

This section highlights the main features of RPCA and then sketches the proofs that guarantee its correctness.

**How RPCA works**

RPCA proceeds in rounds. Rounds are continually repeated: at the end of each round a new ledger has been validated. Each round consists of two phases:

- **Deliberation:** the network participants agree on the transactions to apply to a prior ledger, according to the positions of their chosen peers.
- **Validation:** the network participants agree on which ledger was generated at the end of the current round, according to the ledgers received by the chosen peers.

Before discussing phases, we define the main concepts of RPCA:

- **Ledger:** the shared distributed state. Each ledger has a unique ID and a sequence number. The former uniquely identifies the ledger contents, the latter describes the ledger position in the chain of the approved ledgers. We call *last-closed* ledger the last one that has been validated by RPCA and that will serve as basis to build the next ledger.
- **Transaction:** a transaction is an instruction for an atomic change in the ledger state.

- **Transaction Set:** a transaction set is a set of transactions under consideration by the consensus process.
- **Node:** a node is one of the actors running the RPCA.
- **Unique Node List (UNL):** each node must choose a trusted subset of the nodes in the network. They are the *peers* of the node, and form its UNL. The node will listen at the opinion of its peers to reach consensus on the next ledger, so it should be careful in choosing only honest nodes.
- **Position:** a position is the node current belief of the transactions to be included in the next ledger.
- **Proposal:** a proposal is a position revealed to the network. Every time a node changes its mind on the transactions to be included in the next ledger, it updates its position and shares a new proposal on the network.
- **Dispute:** a dispute is a transaction for which the node is not in accordance with one or more of its peers.

A node of the Ripple network may be either a *proposing* node or an *observing* node. In the former case, the node actually takes part to the consensus process and it sends its proposals to the peers that are listening for its opinion. Instead, an observer node is a passive one, it never shares its proposals and it only updates its position according to the proposals received from the majority of its peers.
We describe now the steps of a proposing node. Such a node will proceed in rounds. At the end of each round, it will decide whether the consensus was reached or not and, if it is reached, the state of the distributed ledger. Each round includes the two phases "Deliberation" and "Validation". In turn, the deliberation one includes the two sub phases "Open" and "Establish"; the validation phase is also called "Accept" phase.
Let us look at these phases in more detail.

## Open Phase

In this phase the node simply collects from the network the transactions to be inserted into the open ledger. The duration of this phase is a tradeoff between latency and throughput: a shorter duration means that the received transactions will be validated earlier, but also that a consensus round will validate fewer transactions. The node switches to the "Establish" phase when one of the following conditions are satisfied:

- There are some transactions in the open ledger and more than `LEDGER_MIN_CLOSE` time is elapsed.
- There are no transactions in the open ledger and more than `LEDGER_MAX_CLOSE` time is elapsed, with `LEDGER_MAX_CLOSE > LEDGER_MIN_CLOSE`.
- More than fifty percent of the peers of this node have switched to the "Establish" phase. This condition prevents this node to fall behind.

## Establish Phase

In this phase the node exchanges proposals with its peers in an attempt to reach consensus on the transactions to be included in the next ledger. Each proposal carries at least the ID of the last validated ledger and a transaction set with the transactions to be included in the next ledger. Firstly the node checks that its last validated ledger matches the last validated ledger of its peers, otherwise it asks that ledger to its peers. Then, it inspects its peers' proposals to compare them with its own position. A position contains the transactions the node believes should be included in the next ledger. For each transaction in its position, the node counts how many peer proposals include such transaction. If this count is larger than a given threshold, the transaction remains in its position, otherwise it is dropped. Then the node scans each transaction in its peers' proposals, adding to its position those transactions that are supported by a number of peers larger than a given threshold. This threshold is not fixed but increases with time. The idea is that a larger time in the "Establish" phase is due to a lack of agreement on the transactions to be included in the next ledger. By making the threshold for including one transaction more and more severe, only the few transactions that are supported by a large majority of peers remains, and this simplifies the achievement of a consensus. Figure 6 from [13] shows the threshold values.

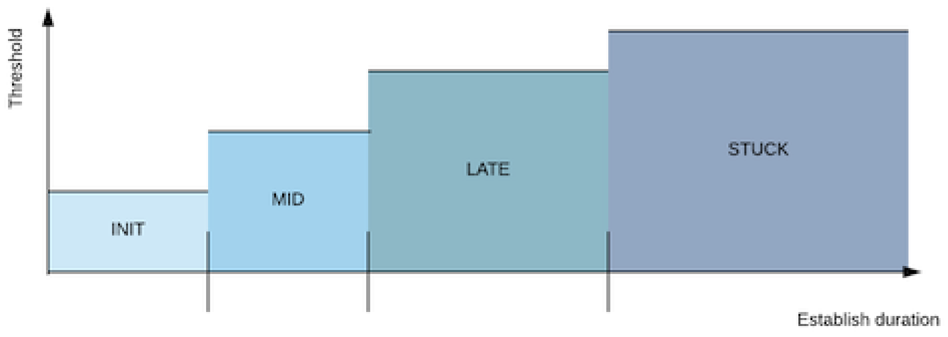


**Figure 6: The threshold for including a transaction in the current position increases with the duration of the Establish phase. Namely, four regions are defined: "Init", "Mid", "Late" and "Stuck".**

In order to reduce the amount of messages in the network, a node sends an updated proposal to its peers only when it changes its position.

A node passes to the “Accept” phase when the following three conditions are verified:

- More than `LEDGER_MIN_CONSENSUS` time is elapsed in the establish phase.
- At least 75% of the previous round proposers have proposed or more than `LEDGER_MAX_CONSENSUS` time is elapsed, with `LEDGER_MAX_CONSENSUS > LEDGER_MIN_CONSENSUS`
- At least `QUORUM` peers share its position, meaning that the transactions in the respective positions are exactly the same.

### Accept Phase

In this phase the node applies the transactions resulting from the “Establish” phase to the prior ledger to generate the new one. Then it shares the new validated ledger with its peers: if at least a `QUORUM` of its peers agree on the resulting ledger, than the node declares the consensus reached and passes to the “Open” phase of the next round. The transactions that were not validated are not dropped, but the node will try to validate them in the next round. A transaction is dropped only when it fails to be validated after a given number of consecutive trials. Ripple suggests to share only a hash of the ledger, in order to reduce the load on the network and to speed up the consensus process. The full ledger will be sent only if some peers explicitly request it. This may happen if, for some reason, a node realizes that it has validated a ledger which differs from the one of its peers, and so it decides to directly ask them the new ledger.

**Why RPCA works**

RPCA is a Byzantine fault-tolerant agreement protocol capable of reaching consensus by using collectively trusted subnetworks within the whole network.
The authors of [9] prove that RPCA has two key properties: safety and liveness. Safety means that the different nodes running RPCA will eventually agree on the distributed ledger state; liveness means that the network cannot get into a state in which some honest nodes can never fully validate a new ledger.
A complete analysis of the mathematical properties of RPCA is out of the scope of this thesis, and we refer to [9]. We will instead describe the network model that RPCA assumes and outline the main proofs of the safety and liveness properties of RPCA, with particular emphasis on the assumptions under which such properties hold.

**Definitions**

Let $P_i$ be a node in the network with unique identifier $i$. Each node $P_i$ chooses a unique node list $\mathrm{UNL}_i$ made of arbitrary nodes.
A node that is not crashed and correctly takes part to the consensus protocol is said to be *honest*; a node that does not behave according to the consensus protocol is said to be *byzantine*.
For any $P_i$ we denote $n_i = |\mathrm{UNL}_i|$ and define quorum $q_i$ a parameter that roughly specifies the minimum number of peers in $\mathrm{UNL}_i$ that should agree on some decision for $P_i$ to share such decision. The authors of [9] suggest $q_i = \lceil 0.8\, n_i \rceil$.
Finally, we define $t_i$ the maximum number of byzantine nodes in $\mathrm{UNL}_i$. We assume
$t_i \leq n_i - q_i$.

**Safety**

The authors show that RPCA achieves safety by proving properties that provide increasing benefits but that also require more and more severe limitations on the UNL of any pair of nodes in the network.

They start assuming "Byzantine accountability" of the network. This means that a byzantine node cannot convince two different nodes to validate different ledgers in the same consensus round. Since ledgers are broadcasted in the network with a gossip protocol, the two nodes will receive both contradictory ledgers in time to realize that the emitting node is cheating, and thus they will ignore such ledgers.

**Proposition 1.** Assuming Byzantine accountability, two honest nodes $P_i$ and $P_j$ cannot fully validate different ledgers with the same sequence number iff

$$\left|\mathrm{UNL}_i \cap \mathrm{UNL}_j\right| > n_i - q_i + n_j - q_j$$

Assuming as quorum $q_i = \lceil 0.8\, n_i \rceil$, this means that every pair of nodes need an overlap of 41% the maximum size of their respective UNLs .

From now on the authors do not assume Byzantine accountability.

Let us define some other quantities that will be useful for the next propositions.

For any pair of nodes $P_i$ and $P_j$, let $\mathrm{O}_{i,j} = \left|\mathrm{UNL}_i \cap \mathrm{UNL}_j\right|$ and let $t_{i,j} = \min\{t_i, t_j, \mathrm{O}_{i,j}\}$.

$t_{i,j}$ is the fault tolerance, i.e. the largest number of tolerated byzantine faults in $\mathrm{UNL}_i \cap \mathrm{UNL}_j$. We assume that there are at most $t_i$ byzantine nodes in $\mathrm{UNL}_i$ and $t_j$ byzantine nodes in $\mathrm{UNL}_j$.

Furthermore, let us call $seq(L)$ the sequence number of ledger $L$.

**Proposition 2.** If $P_i$ fully validates some ledger L with seq(L) = s then $P_j$ cannot fully validate any contradictory ledger with the same sequence number $s$ iff

$$\mathrm{O}_{i,j} > (n_i - q_i) + (n_j - q_j) + t_{i,j}$$

Assuming 80% quorums and 20% fault tolerance, proposition 2 requires at least 61% of overlaps among the UNL of all nodes.

Anyway, proposition 2 is still not sufficient to provide full safety. Indeed, it is still possible for two nodes to exit from the deliberation phase and enter the validation phase, but then they may be unable to validate that ledger.
The last property that the authors prove is that the chain of ledgers approved by RPCA does not fork. This is the most important property, because if distinct nodes validate ledgers in distinct chains then the consensus cannot be reached. More precisely, the authors prove that if any node fully validates a ledger $L$, then no node can fully validate a ledger $L'$ such that $seq(L') \geq seq(L)$ and $L'$ is not a descendant of $L$.
This derives from the following theorem.

**Theorem 1.** RPCA guarantees fork safety if $O_{i,j} > n_j/2 + n_i - q_i + t_{i,j}$ for every pair of nodes $P_i$ and $P_j$.

Keeping the assumptions of 80% quorums and 20% fault tolerance, theorem 1 requires an overlap larger than 90% between the UNLs of any pair of network nodes.
This is a narrow margin and indeed the authors suggest that each node should adopt a UNL made of the same five trusted nodes, possibly adding a sixth node at will to include some variation in the UNLs.

**Liveness**

We have shown that the validated ledgers are consistent with one another. Now we would like to demonstrate that the consensus algorithm will actually build such ledgers. Unfortunately, there is no way to provide such a guarantee in the general case. Indeed, the FLP theorem states that it is not possible to guarantee forward progress in a fully asynchronous network with one faulty node [21].
The authors assume that no nodes are faulty and that messages in the network are delivered within some maximum delay bound. They define "leaf" a node of the network whose UNL includes also the sixth extra node. In this setting, they prove the following theorem:

**Theorem 2.** Suppose for all nodes the UNL quorum is set to $n - \lfloor (n - 1)/k \rfloor$ for some integer $k$. XRP LCP cannot get stuck in a network consisting of a single agreed-upon UNL $X$ of size at least $k$ along with an arbitrary number of leaf validators.

Theorem 2 proves that in the restricted case where no nodes are faulty and network messages are delivered with bounded delay, if all the nodes adopt the same UNL with the possible extension of an extra node, RPCA is guaranteed to advance in the validation of new ledgers.
Forward progress cannot be guaranteed in the general case of fully asynchronous networks where some node may be byzantine.

### 2.2.3 Algorand

Algorand is a new cryptocurrency that, according to its authors, confirms transactions in less than one minute, scales well up to 500.000 users and recovers quickly from network partitions [11, 22].
This is due to the adoption of a byzantine agreement protocol called $BA\star$, that allows Algorand to rapidly reach consensus on the transactions to be included in each block. Furthermore, under proper assumptions, there is a negligible probability of a fork in the blockchain.
$BA\star$ satisfies the following properties:

- **Safety:** all users agree on the same transactions.
- **Liveness:** new transactions are added to the blockchain.

Anyway, $BA\star$ guarantees safety and liveness if, respectively, weak synchrony and strong synchrony assumption holds.
Weak synchrony implies that the network may be asynchronous for a long, but bounded, period of time. After this period, the network is again strongly synchronous for at least the time needed by Algorand to ensure safety.
Strong synchrony assumes that at least 95% of the honest users can send messages that will be received by other honest users within a maximum delay bound.
To quickly reach consensus on a new set of transactions, only a small subset of the Algorand users, the *committee*, iteratively runs the $BA\star$ protocol.

At the end of each correct execution of *BA*★, the committee members deliberate a new block and add it to the blockchain. The block is propagated to the other Algorand users via a gossip protocol.

*BA*★ consists of two phases which, in turn, include many steps.

The first phase is the **reduction** one. In this phase, *BA*★ takes as input several candidate blocks and reduces them to only two blocks, one of which is the empty block.

In the second phase, *BA*★ reaches the agreement on either the proposed nonempty block or the empty block.

The trust on a restricted set of nodes to carry out the consensus protocol is similar to what is done in Ripple. Anyway, the authors of Algorand recognize this is vulnerability because attackers may carry out denial-of-service (DoS) attacks against the trusted nodes to impair the consensus process.

Their solution is that the members of the committee may change. In particular, at any time a user can check if it is a member of the committee by executing a *Verifiable Random Function* (VRF). This function takes as argument the node's private key and a seed, stored in the last blockchain block, and returns a string proving if the user is a member of the current committee. This function is computed independently by each node and the result is kept secret, so that an attacker cannot know in advance which are the members of the committee.

When a committee member expresses its vote for a block, it appends the computed proof to its message. This way, each node can verify that it is a member of the current committee. Now, malicious nodes could implement a DoS attack against a committee member because even these nodes know that it is a member. Anyway, from the protocol point of view this attack would be irrelevant, because each committee member is required to speak just once, and after it has sent its message it no longer belongs to the committee. In the next step, another user will replace it in the committee. This is possible because *BA*★ avoids any private state, except from the private key hold by each user.

Furthermore, the choice of the committee members is weighted according to the amount of money users have in their account. This prevents Sybil attacks on the choice of the committee members. Algorand avoids forks and double-spending as long as 2/3 of the money is owned by honest users.

# 3. Network Topology Inference

This chapter describes the choices to implement the network topology inference algorithm. The theoretical foundations are described in section 2.1 [27]. The algorithm is implemented in Python using a procedural style. There are two main actors that will make use of the algorithm. The first are the monitors. Monitors are network nodes that do not belong to the topology we wish to infer. In literature, monitors are allocated at the edge of the network and the goal of each monitor is to collect traces towards each other monitor. After they gather such traces, they will send them to a central Network Operating Center (NOC) for processing. The NOC is responsible for actually running the algorithm and infer the network topology. We will assume a slightly different approach with respect to what is usually done in literature. We still map monitors at the edge of the network, not being part of it, but they also have the capability to infer the network topology based on the gathered traces, that is they also act as local NOCs. After inferring a topology, a monitor sends it to the nodes implementing the blockchain. These nodes receive the topologies from the various nodes, reach a consensus on the edges of the received topologies and, at the end of each consensus round, publish a ledger that contains the inferred topology. So, the first difference with respect to literature is that in our approach there are several NOCs that reconstruct local topologies that may be different from one another. Then, blockchain nodes actually merge these topologies to produce a single topology at the end of each consensus round. The second difference is that we devise the presence of sensors, which are nodes placed inside the network. While monitors provide a first snapshot of the inferred topology, sensors begin their work after monitors have finished their execution and focus on detecting changes in the network, i.e. the join and leave of nodes. When a sensor detects that something changed in the network, it runs the network topology inference algorithm and sends the new inferred topology to the Blockchain nodes, which will validate the new information and produce a new ledger containing the updated inferred topology.

## 3.1 The Network Topology Inference Algorithm

Let us consider a node running the network topology inference algorithm. It may be either a monitor or a sensor. The pseudocode of this algorithm, that we will call "iTop" as the authors of [27], is the following:

```
Algorithm 2: iTop
Input:   A placement of monitors/sensors in the network
Output:  The inferred topology
1 compute_distances()
2 store_traces()
3 create_virtual_topology()
4 compute_merge_options()
5 create_merge_topology()
6 return save_topology()
```

We will now describe the functions inside the algorithm.

**compute_distances( )** computes the hop-distance between this node and any other node in a list of target nodes. If the node running iTop is a monitor then the target nodes are the other monitors. If, instead, the running node is a sensor the list of target nodes is made of network nodes. In our simulations, distances are collected by pinging the target hosts, exploiting a network that does not contains yet firewalls or blocking routers. This simplification reflects the assumption made in [27], according which it is always possible to derive the hop distance between any pair of nodes by exploiting the TTL field of IP packets that are not blocked by intermediary routers.

**store_traces( )** gathers the traces from this node towards the target nodes and stores them on disk. Traces are gathered through the traceroute utility. At this point, no assumption is made on the network: if anonymous or blocking routers are present, the resulting traces will contain rows full of asterisks, meaning that no response was received for a probe packet with a given TTL. As an example, Figure 7 shows the trace collected with traceroute when an anonymous router precedes a responding one along the path. The row corresponding to the anonymous router is made of three asterisks.

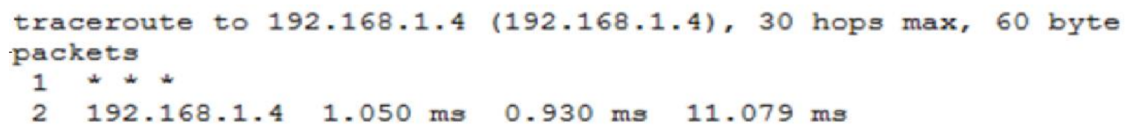

```
traceroute to 192.168.1.4 (192.168.1.4), 30 hops max, 60 byte
packets
 1  * * *
 2  192.168.1.4  1.050 ms  0.930 ms  11.079 ms
```

**Figure 7: trace collected with Traceroute.**

**create_virtual_topology( )** builds the virtual topology as described in section 2.1.4. The function loads the traces previously stored on disk and scans them one line after the other. In our implementation, we allow each monitor to use the traces collected by the other monitors too. The pseudocode is reported in Algorithm 3. Implementation details may be found in the actual code in the public repository of the thesis [15], here we provide a higher level description of the main implementation choices.

There are two important data structures, **topo** and **paths**. topo is a dictionary to store the virtual topology under construction. It takes the name of the network node as key and a pair telling the type of the node (responding, anonymous, etc …) and the list of its neighbors as value. paths is a dictionary containing the sequence of nodes encountered from each source of traceroute to all the final destinations. This data structure will be useful in a successive phase but it is convenient to create it while scanning the traces.

**Algorithm 3: Virtual Topology Construction**

```
Input:   traces = list of gathered traces
         distances = matrix of distances between nodes
Output:  topo = virtual topology
         paths = paths in the virtual topology
function create_virtual_topology():
    for each trace in traces do
        src = get_source(trace)
        dst = get_destination(trace)
        if get_answer_from_destination(trace) then
            rr = get_routers(trace)
            add_routers(topo, rr)
            add_path(paths, rr)
        else
            dist = distances[src][dst]
            rrs = get_routers(src, dst, traces)
            rrd = get_routers(dst, src, traces)
            nrr = non_responding_routers(traces, src, dst, dist)
            add_routers(topo, rrs ∪ rrd ∪ nrr)
            add_path(paths, rrs ∪ nrr ∪ reversed(rrd))
return (topo, paths)
```

The algorithm scans all the gathered traces. From each trace it extracts the identity of the node that collected that trace (the source) and the identity of the final destination node (lines 3 and 4). Then it checks if the final destination could answer the probe (line 5). If this is the case, then only responding and anonymous routers may be present along the path. The function *get_routers()* parses the trace and returns these routers, which are added to the virtual topology and to the encountered paths (lines 7 and 8). If, instead, the traceroute final destination could not answer to traceroute, at least one blocking router was encountered along the path. In order to reconstruct the path at best, we assume that also the trace collected in the reverse direction, that is from *dst* to *src*, is available. Otherwise the function *get_routers(dst, src, traces)* will return an empty list and we will reconstruct only the fragment of path from *src* to the last node that did not block the packets. Instead, if also the other trace is available, the algorithm reconstructs the fragment of path from *src* to the block (line 11), the unobserved routers in the blocked portion of network (line 12) and the fragment of path from the block to *dest* (line 13). The function *non_responding_routers( )* makes a first inference on the number and type of non-responding routers present in the unobserved section by exploiting the hop distance *dist* between *src* and *dst* as described in section 2.1.4.

**compute_merge_options( )** computes, for each link $e_i$ of the virtual topology, the set of links of the virtual topology that are valid merge options for $e_i$. Algorithm 4 shows its pseudocode.

```
Algorithm 4: Merge Options Computation
Input:   traces = list of gathered traces
         paths = paths in the virtual topology
         topo = virtual topology
Output:  M = merge options table
         C = endpoint compatibility table
function compute_merge_options():
    EP = edges_in_paths(paths)
    M = init_merge_options_table(topo)
    trace_preservation(M, EP)
    distance_preservation(topo, paths, M)
    C = compatibility_table()
    endpoint_compatibility(M, C, topo)
    return (M, C)
```

The function *edges_in_paths( )* returns a table having as key each virtual topology edge and as value the list of paths where the edge appears (line 2). The merge option table is initialized in line 3. Initially, each edge of the topology has all the other topology edges as valid merge options. Then, both tables are used to impose the trace preservation condition (line 4). Lines 5-7 apply at first the distance preservation and then the endpoint compatibility conditions. Finally, the definitive merge option table is returned, together with the endpoint compatibility table.

**create_merge_topology()** applies algorithm 1 of section 2.1.4 to compute the merge topology starting from the virtual topology. Basically, all the edges with a non-empty set of merge options are selected, starting from the ones with few merge options. Then, the merge options of each selected edge are sorted in increasing order with respect to their own merge options. The selected edge is compared against those belonging to this sorted set. As soon as we find an edge in the set which is compatible with the selected one, we carry out the merge. Then we update the topology and the merge options table accordingly. We go on until the merge options set of all the edges becomes empty. Finally *create_merge_topology()* returns the merge topology, which is now the final one returned by iTop.

We would like to note that in the merge process special care has to be taken because after each merge, a link is collapsed into another link. This means that either one or two nodes are collapsed into the respective endpoints of the other link. This can have implications even on distinct links that are not involved in the merging phase. As an example, Figure 8 shows an example where the edge $e_j$ is going to be collapsed into $e_i$. Router R3 will be collapsed into R1.

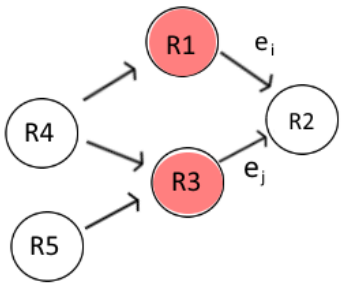


**Figure 8: Edge $e_j$ is going to be collapsed onto $e_i$. The figure highlights the nodes that will be collapsed.**

Figure 9 represents the situation after merging $e_j$ into $e_i$: since the merging has deleted the router R3, the outgoing edges from R4 and R5 have become invalid.

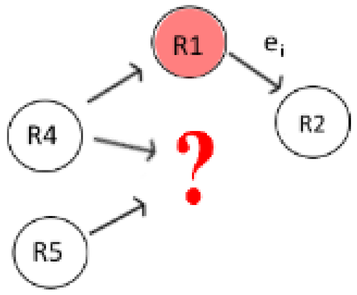


**Figure 9: $e_j$ has been merged into $e_i$.**

R5 should now point R1, because the node R5 was pointing to has been collapsed into R1. Instead, the invalid pointer from R4 should be deleted. This is an example of the situations, neglected in the paper, we have to manage to avoid inconsistency in the final topology.

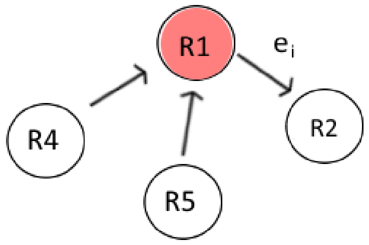
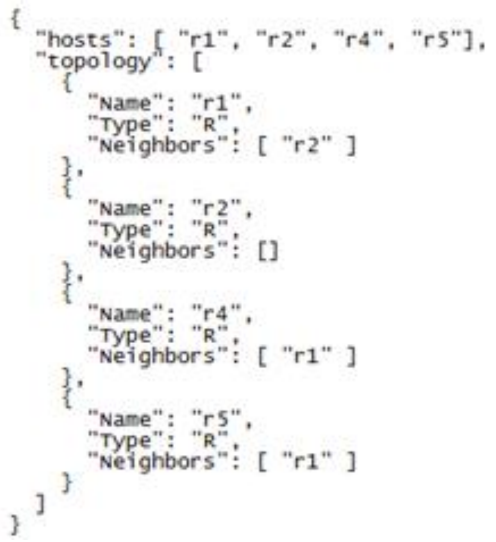


**Figure 10: resulting graph after the corrections**

**save_topology()** stores the topology on disk, in JSON format. The JSON file lists the nodes belonging to the topology and then the type and the neighbors of each node in the topology. Figure 11 shows a sample of the topology in Figure 10. In this case all the nodes are responding, thus their type is represented as “R”.



```
{
  "hosts": [ "r1", "r2", "r4", "r5"],
  "topology": [
    {
      "Name": "r1",
      "Type": "R",
      "Neighbors": [ "r2" ]
    },
    {
      "Name": "r2",
      "Type": "R",
      "Neighbors": []
    },
    {
      "Name": "r4",
      "Type": "R",
      "Neighbors": [ "r1" ]
    },
    {
      "Name": "r5",
      "Type": "R",
      "Neighbors": [ "r1" ]
    }
  ]
}
```

**Figure 11: sample topology returned by iTop in JSON format.**

## 3.2 The Monitors

Monitors are special nodes at the edge of the network. Each monitor basically runs the iTop algorithm using as target nodes the other known monitors.
Monitors are the first component of our framework because they are the first ones that run the network topology inference algorithm to produce a first snapshot of the inferred topology. After executing the iTop algorithm, the monitor organizes the inferred topology in suitable transactions and sends them to the blockchain nodes to be included in the ledger.
Monitors and sensors are logically separated entities but in a real scenario they could be deployed on the same physical nodes.

## 3.3 The Sensors

Sensors are nodes typically placed in the middle of the network to track network changes like the additions or the removals of nodes and edges.
Sensors may have both active and passive capabilities, which we found convenient to pack inside the Python class *sensor*. Indeed, a sensor is a multithreaded object that concurrently captures traffic looking for unknown nodes (passive capability) and pings known nodes to

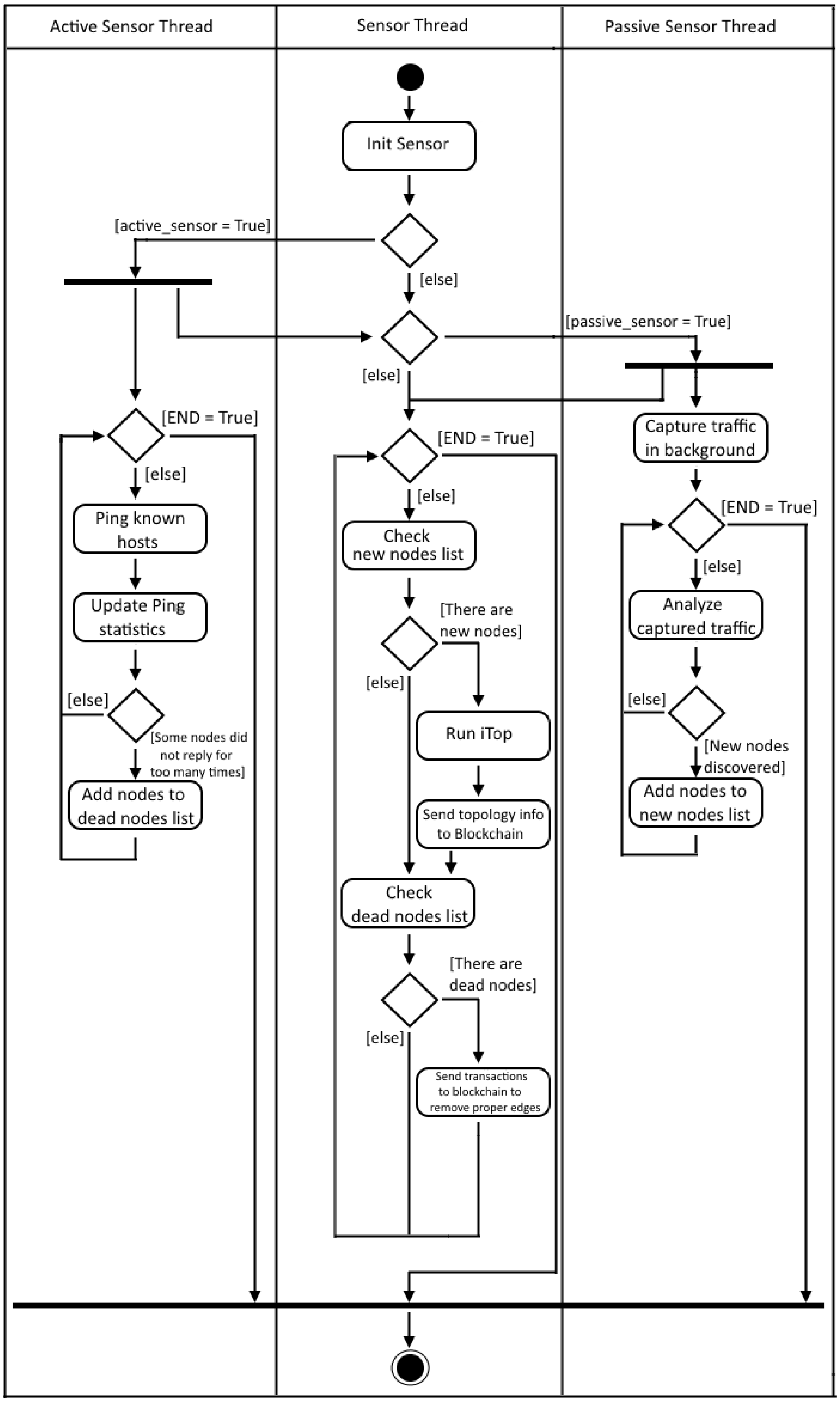


**Figure 12: Activity diagram of a sensor.**

check whether they are still reachable (active capability). The activity diagram in figure 12 shows the behavior and the interactions of a sensor.

When a sensor is created, it performs some initialization tasks. As an example, a sensor is also a client of the blockchain, to which it sends transactions, so it will have to initialize an object of type *client_node* (see chapter 4) which is responsible for the interactions with the blockchain nodes.

Although the sensor is only one, we have depicted three lanes to account for the different capabilities of a sensor: active, passive and management. Furthermore, each of the three lanes corresponds to a separate thread of execution. For all the threads, the stop condition is the same shared Boolean variable. A sensor ends when all its threads have terminated the execution.

*Sensor* is the main thread. According to its initialization parameters, it decides whether the active and the passive threads have to be spawned. Until the stop condition does not hold, it loops checking the state of two shared lists. The new nodes list is shared with the passive sensor thread. This list is not empty anytime the passive sensor thread has discovered some unknown hosts. If the list is not empty, the sensor has to discover how the topology has changed from its last inference, so it runs the network topology inference algorithm (iTop). Since a sensor does not consider the presence of monitors, it sends the probes towards the discovered network nodes, possibly including other sensors, in order to collect the traces. So the trace collection process could be slow if the network includes a large number of nodes. Furthermore, it could be the case that most network components are unchanged. What we would like to do is to find the smallest set of network nodes such that we cover all the network elements just by collecting the traces towards the nodes in the set. Unfortunately, this problem can be modeled as a minimum set cover problem, a well-known NP-hard problem [19]. Anyway, we believe it is convenient to run a fast version of iTop when a few changes of the topology have occurred, although this version could be less accurate than the original version, so we make the following assumptions:

- The fast version of iTop, which differs from the original one because it does not use all the available network nodes in the trace collection phase, is optional and, if used, it alternates with the original version.

- The network elements we want to cover are the network nodes. We make this simplifying assumption, that neglects new edges, because the edges could be much more than the nodes and we are interested in a fast algorithm. Furthermore, we accept that, if some edge is not discovered, it will be discovered in the next iTop execution because it will use all the network nodes to collect the traces.
- We adopt the greedy algorithm in [19]. Basically, we consider the sensor as root and we compute the shortest paths towards each node of the old, known topology. If there is only one shortest path from the sensor towards a node, it means that a probe packet will encounter all the nodes along the path. If, instead, there are distinct shortest paths from the sensor towards a node, a probe packet will meet for sure the nodes in the intersection of the shortest paths. Indeed, we assume that routing is based on shortest paths and in the latter case we cannot know a priori which path the probe packet is going to follow. Then, we associate to each network node a set with the nodes that a probe packet from the sensor to that node will surely meet along the path.

  The greedy algorithm simply sorts the nodes in decreasing order with respect to the cardinality of such sets, and then picks them one after the other until the union of the sets of the picked nodes covers all the network nodes. The picked nodes, together with the new nodes, are the target nodes used as recipient of the trace collection process. Then, iTop is run and the new topology is inferred. If the new topology contains all the network nodes of the old topology we consider it satisfying, otherwise another execution of iTop, using all the network nodes, is triggered.

After the sensor has inferred the network topology, it sends it to the blockchain nodes. Then, it checks the dead nodes list it shares with the active sensor thread. This list is not empty only if the active thread has detected dead nodes in the network. A dead node is a network node that did not reply to ping for a given number of consecutive times. If the sensor detects a dead node, it sends special transactions to the blockchain nodes telling them to remove from the ledger all the edges where such node appears.

The passive sensor thread loops until the stop condition does not hold. It captures and analyzes traffic exploiting the *tcpdump* tool. It works at network level, looking at the source address of the IP packet. Every time it discovers an unknown IP, it adds such IP to the new nodes list.

The active sensor thread loops until the stop condition does not hold. It checks the liveness of known network nodes with the *PING* utility. If a node does not reply to the ICMP packets issued by PING for *max* consecutive times, where *max* is an arbitrary integer parameter, the thread declares dead such node and adds it to the dead nodes list.

# 4. Blockchain

The second main component after the network topology inference algorithm is the blockchain, that is used to securely store the previously inferred topology in a distributed ledger. We have decided to implement our own blockchain which is based on Ripple and is specifically tailored to store topology information. Anyway, the network topology inference algorithm and the blockchain are two independent components. Indeed, the algorithm stores the inferred topology in JSON and it can be used as a stand-alone tool. The implemented blockchain could be used, with small modifications, to store arbitrary data or, on the contrary, any off-the-shelf blockchain could store the topology information.

We say that topology information is securely stored into the ledger because our blockchain is resilient to attacks that may occur both before and after the validation of each ledger.

A malicious node could try to insert fraudulent transactions inside the blockchain before a ledger gets validated. In chapter 5 we experimentally show that if the number of malicious nodes in the UNLs is below a given threshold, such transactions will not appear in the validated ledgers.

A malicious node could also try to tamper a validated ledger by inserting fraudulent transactions or removing honest transactions from it. Also this attack is going to fail, because each validated ledger has a unique id which is obtained by hashing, with SHA-256, the concatenation of its sequence number and its validated transactions. So changing any transaction or the sequence number of the ledger would result in a ledger id which differs from the one previously computed. This supports the detection of the tampering attempt.

This chapter describes the architecture of our blockchain. We show the classes that comprise it and the interactions that happen among the blockchain components to realize a common use case.

## 4.1 The Blockchain Architecture

The blockchain mainly includes two kinds of nodes:

- The *client* nodes send the transactions to be inserted into the ledger to the server nodes.
- The *server* nodes validate the received transactions and publish a new ledger every time they reach the consensus with the server nodes in their own UNL.

Figure 13 shows this architecture. Client nodes are depicted in pink, server nodes in green. The black dotted arrows represent the transactions sent by a client; the green dotted arrows outbound from a server node point to the nodes comprising its UNL. Each server node applies the validated transactions to its local copy of the ledger.

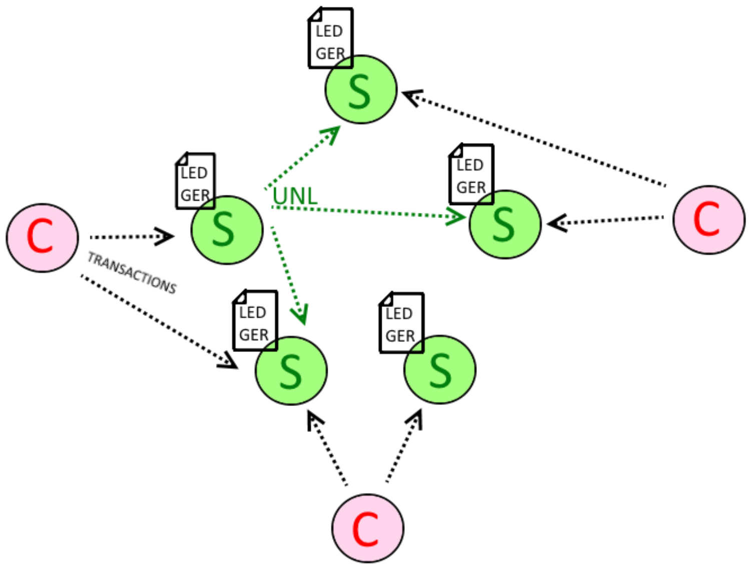


**Figure 13: Blockchain architecture.**

## 4.2 The Blockchain Classes

In this paragraph we review the classes comprising the blockchain package, whose interactions are reported in the UML class diagram in Figure 14. Also in this case we found convenient to organize the code in an object-oriented style. For the sake of clarity we do not report methods in Figure 14.

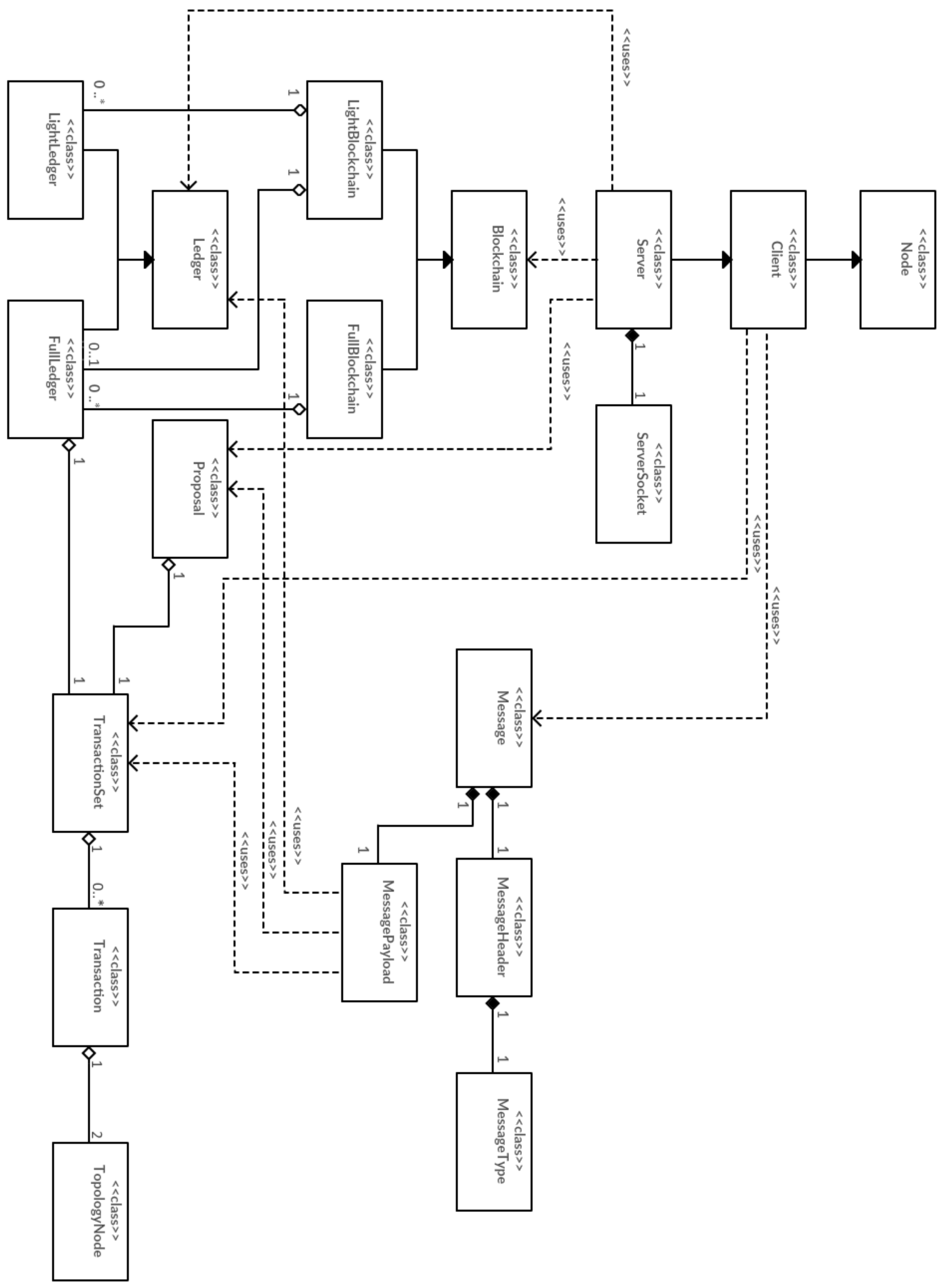


Figure 14: Blockchain UML class diagram.

**Node.** Node is the base class for the Blockchain nodes *Client* and *Server*. It has methods and attributes that its derived classes need. As an example, it specifies the IP address and TCP port the node uses for communications. It also uses the *rsa* module [36] to generate a random pair of private and public keys. Finally it provides methods to sign messages and to verify other nodes' signatures.

**Client.** The Client class models a client of the blockchain, that is a node that sends transactions to the Blockchain server nodes for validation, but does not take part to the consensus process. A client is parametrized via a configuration file that specifies the IP addresses and TCP ports of its validators, the server nodes that should validate its transactions. According to our protocol, server nodes only accept signed messages from already known clients. This is the reason of the method *ask_client_registration()* a client uses to send its id and its public key to its validators. This enables a validator to check the authenticity of the signature in each message from its clients. A client has the capability to parse the topology (stored in JSON format), create a Transaction for each edge in the topology, pack the Transactions into a TransactionSet and insert the TransactionSet in a signed Message, and finally send the Message to its validators.

**Server.** The Server class models a blockchain node that actually takes part to the consensus process. It inherits from Client because it is a "full" node that can both run the consensus algorithm and send transactions to the blockchain. It is parametrized via a configuration file that specifies the nodes in its UNL (its peers) and provides the values for the parameters used in the consensus algorithm. These parameters are discussed in section 2.2.2 and regulate the amount of time spent in each phase and the quorum value. A server node asks a registration to the servers in its UNL by sending them its id and its public key. If they accept the registration request, they will add the server to the list of their *observers* and will notify it every time they deliberate a new proposal or a new ledger. The server will use the proposals and the ledgers received by its peers to run the consensus algorithm and update its position as described in section 2.2.2.
The class Server spawns a thread that runs an instance of the ServerSocket class, that handles the communications with the other nodes.

The main thread instead loops to iterate the three phases of the consensus algorithm. In the *Open* phase, the node is simply waiting to receive transactions from its clients. In the *Establish* phase, it sends proposals to its observers and receives proposals from its peers, properly updating its position until it agrees with the quorum of its peers on the transactions to include in the ledger. In the *Accept* phase, the node applies the validated transactions to its previous ledger to generate the new last closed ledger. The transactions are applied deterministically, meaning that every honest node that applies the same set of transactions to the same old ledger will produce the same new ledger. Finally, the node sends its last closed ledger to its observers and receives the last closed ledger from its peers. If at least the quorum of its peers propose its same ledger, the consensus is reached on that ledger. Otherwise, consensus is not reached and in the next round of consensus the node will try to validate also the unapproved transactions received in the Open phase of the previous round. Anyway, a transaction is discarded if it fails to be validated after a given number of consecutive rounds.
The Server class can also export in graphic format the topology in the ledger. In order to do this, it uses the Graph Tool library [23].

**ServerSocket.** ServerSocket is the Server component to manage communications. It uses TCP sockets. A ServerSocket instance waits for incoming connections and spawns a new thread to handle each received request. These threads process the messages asynchronously with respect to the main Server thread.

**Message.** Message is the wrapper class for the messages exchanged among the blockchain nodes. It is made of a MessageHeader, that specifies information about the sender and the type of carried message, and a MessagePayload, that carries the actual payload of the message.

**MessageHeader.** MessageHeader is the header of a Message. It carries the id of the sender together with its signature. This allows any receiver that knowns the sender’s public key to verify the signature authenticity. A MessageHeader also specifies the type of the associated message payload. Finally, since the message size could be quite large, the sender has the capability to send several small messages that the receiver will use to reconstruct the original message. In order to do this, each MessageHeader also carries an id and a boolean flag.

**MessageType.** This class enumerates the possible types of a message.

**MessagePayload.** This class is used to carry the payload of the message. The sender node must ensure that the MessageType declared in the header matches the actual type of the payload.

**TopologyNode.** A TopologyNode represents a node of the inferred topology. It has a name and a type. The type can be one among ‘R’, ‘A’, ‘B’, ‘NC’, ‘H’ and ‘P’, which are abbreviations for responding, anonymous, blocking, non-cooperative, hidden or pattern. The type ‘P’ is a special type for nodes arbitrarily inserted into the inferred topology to make the inferred edges fit a given pattern. For instance, a sensor could be told that it is making inference in a branch of a tree-structured network, but that the root of the tree is not going to answer to its probes. If we know the root, we could make the sensor add special transactions where the root node has a type ‘P’. In this way, the inferred branch is stuck to the root and it does not appear as a separate component with respect to the main tree. In chapter 5 we will show a use case for this pattern.

**Transaction.** A Transaction represents an oriented edge of the inferred topology. It contains two TopologyNodes and a type that tells if such edge must be added or removed from the topology in the ledger. It has a unique id that is derived by hashing the names of the TopologyNodes together with its type.

**TransactionSet.** A TransactionSet is a wrapper for a set of transactions. It has an id which is produced by hashing this set of transactions. Two TransactionSets are compared by looking at their content: they are declared the same if they carry the same transactions. The id can’t be used to make a fast comparison between two TransactionSets because adding the same transactions to two distinct TransactionSets in a different order would produce two different ids.

**Proposal.** The Proposal class models the proposals sent by the blockchain server nodes in the Establish phase. A Proposal is uniquely identified by a tuple made of the id of the proposing node, the round (a round equal to *n* means that this is the $n^{th}$ proposal sent by the issuing node during the same Establish phase), the TransactionSet carrying the proposed transactions and the id of the ledger to which the proposal should be applied.

**Ledger.** The Ledger class is the interface for the ledger. A Ledger has a sequence number that represents the position of this ledger in the chain of validated ledgers and an id, produced by hashing together the sequence number and the id of the validated TransactionSet carried by this ledger.

**LightLedger.** This class inherits from the Ledger base class. It is called light because actually it does not contain the validated TransactionSet, but only its id.

**FullLedger.** This class inherits from the Ledger base class. It is called full because it carries all the transactions included in the validated TransactionSet, not only its id.

**Blockchain.** The Blockchain class is the interface for the blockchain. The blockchain has been implemented as a linked list of ledgers. The last ledger of the list is the last closed ledger and represents the last state of the topology on which the blockchain nodes have reached consensus. To retrieve the stored topology it suffices to access the last ledger, which stores the more recent snapshot of the topology.

**LightBlockchain.** It is convenient to store ledgers than contain the whole set of agreed transactions as this avoids to navigate the blockchain to retrieve the stored topology. It is sufficient to look at the last validated ledger. So, retrieving the stored topology is fast but the drawback is that maintaining all the full ledgers may take a lot of space. For this reason, we make it possible to use a light blockchain that only maintains one FullLedger, the last validated ledger. All the previous ledgers are LightLedger. This means that at any time the LightBlockchain stores the transactions of the last validated ledger together with the id of all the previously approved ledgers. This kind of blockchain may be used by nodes that are not interested in keeping the full history of the approved ledgers.

**Full Blockchain.** This class implements a blockchain that stores the whole sequence of FullLedgers, from the inception to the last validated ledger.

## 4.3 A Blockchain Use Case

Figure 15 shows the sequence diagram for the most common use case of the blockchain, that is a client sending transactions to the server nodes for validation. In order to make the diagram more readable, we consider a blockchain made of just two nodes. We also omit the representation of the internal processes not involved in the interactions among classes.
We suppose that the servers already trust each other and that a new client arrives willing to insert transactions into the ledger. Each Server instantiates a ServerSocket to manage the incoming requests. When the client joins the network, it invokes the method *ask_client_registration()* to send its id and its public key to the servers. This method spawns two separate threads to make the two requests. This is the reason why the communications are represented as asynchronous.
In general, all the communications are asynchronous, because every time a node sends a message it instantiates a new thread.
The ServerSocket receives incoming messages and spawns a new thread to manage each request. Anyway ServerSocket does not know how to manage the received messages and so the thread runs the Server method *handle_message()* that performs the proper handling of the message based on its type. The dotted lines show that a response is returned to the client. If the registration is successful, the client can send its transactions to the servers. Then the servers exchange their proposals and, assuming that one exchange is sufficient to agree on the transactions, each server applies the transactions on the previous ledger (if any) and generates the new ledger. Finally the two servers exchange the ledgers and if they are equal it means that the consensus has been reached.

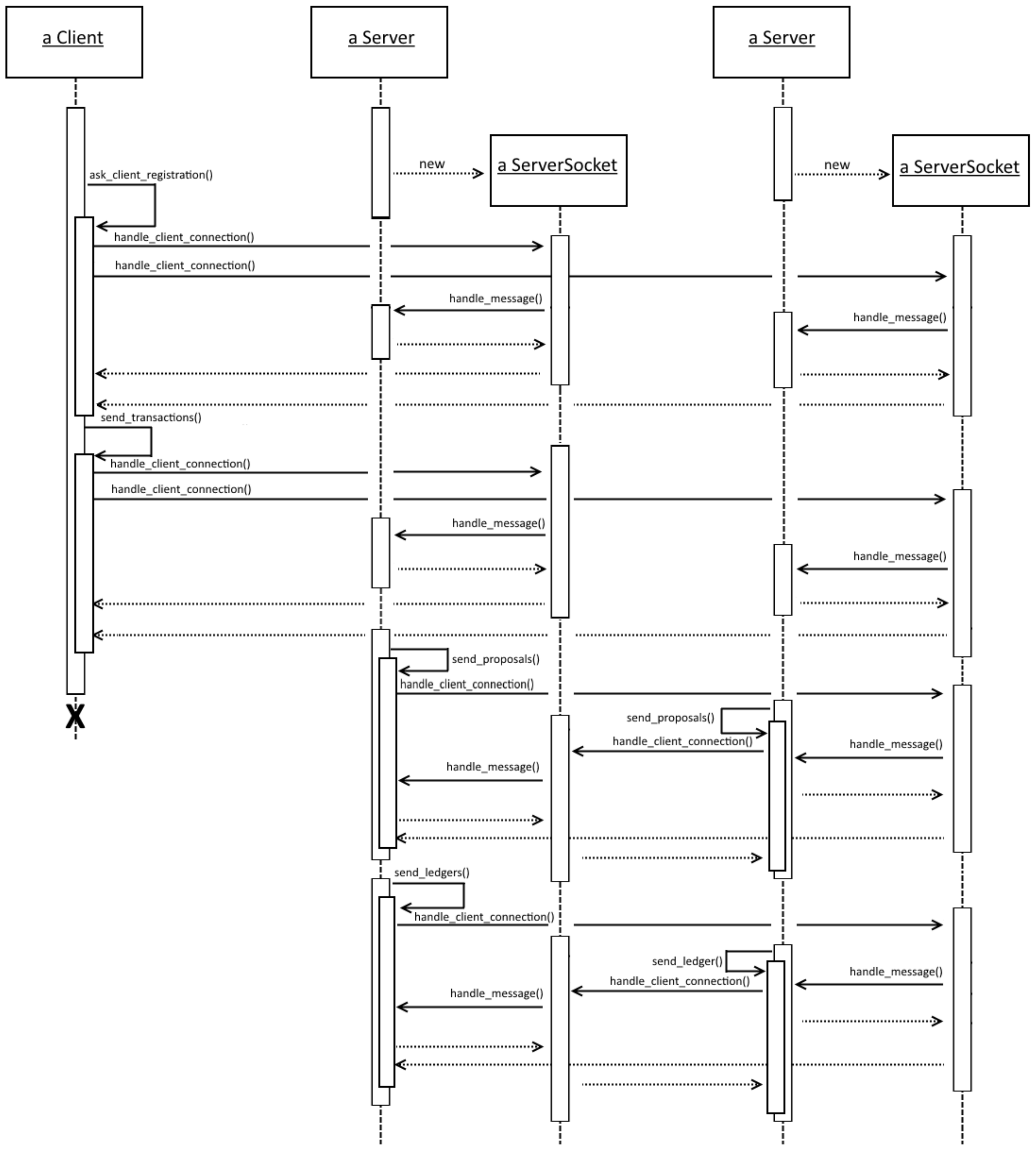


**Figure 15: Sequence diagram for a simple consensus round involving one Client and two Servers.**

# 5. Experimental Results

In this chapter we discuss the experiments carried out for both the network topology inference algorithm and the blockchain.
The metric used to assess the algorithm is the accuracy of the inferred topologies.
The metric used to assess the blockchain is the time to reach consensus, even in presence of malicious nodes.
In both cases we define experiment a single run of a test.
We define simulation a repeated and independent execution of the experiments with the same parameters.
Each simulation runs 20 experiments with the same parameters. Then some metrics are computed individually on each experiment. The metrics of each simulation are computed as the average of the metrics computed in the associated experiments.

## 5.1 Network Topology Inference Algorithm

In order to test the network topology inference algorithm we needed a test network.
Among the many network emulators we chose Mininet **[**31**]**.
**Mininet** is a network emulator that creates a network of virtual hosts, switches, controllers, and links. Mininet hosts are capable to run standard Linux software and its switches support the OpenFlow protocol. Mininet uses process-based virtualization to run many hosts and switches on a single Linux kernel. Mininet is based on Linux *network namespaces*, a lightweight virtualization feature that provides individual processes with separate network interfaces, routing tables, and ARP tables. It can create kernel or user-space OpenFlow switches, controllers to control the switches, and hosts to communicate over the simulated network. Switches and hosts are connected by using virtual Ethernet pairs.
One of the reasons of our choice it is that it can run standard Linux software on its virtual hosts. This means that our code, tested on Mininet virtual hosts, could be deployed on physical Linux hosts without modifications.

Another reason is that by exploiting Linux network namespaces Mininet allows the creation of complex networks on a single Linux machine. The creation of a virtual network involves the use of the Mininet Python API to setup the network elements (hosts, switches, routers and the proper links) and then the use of standard Linux commands on virtual hosts and routers to setup the communications. As an example, we used the *ip route* utility to set up static routes on each network node and the *iptables* command to simulate firewalls and non-responding routers inside the virtual network.

### 5.1.1 Workflow

All the experiments were developed according to the following workflow. The metrics are evaluated on the last ledger hold by the blockchain nodes at the end of each experiment.

- The blockchain server nodes are started on the same machine where the test network is setup.
- The Mininet network emulator is used to set up the test network. Static routes are defined on each node so that communications are possible between any pair of network hosts.
- Non-responding routers and firewalls may be inserted into the network.
- The test topology is complete. Monitors may be inserted at the edge of the topology, running the network topology inference algorithm to reconstruct a first snapshot of the topology. In such a case, the inferred topology is sent to the blockchain nodes. In the following tests the sensors also execute the monitors’ functionality.
- Sensors are deployed into the network.
- Traffic is generated among the network hosts by using Mininet API. The sensors capture this traffic and use it to figure out the network nodes.
- Each sensor runs the network topology inference algorithm using as target the discovered nodes and sends the inferred topology to the blockchain nodes.
- Each blockchain node receives the inferred topology from each sensor. The blockchain nodes carry out the consensus algorithm and at the end they publish a ledger with the validated transactions. After some time the experiment ends and a

blockchain node is chosen to export a graphical representation of the last validated ledger. If all the received transactions were validated, the ledger contains a topology which is the union of the topologies received from all the sensors. This topology is compared against the original network topology and some metrics are computed.

### 5.1.2 Simulations

We have decided to set up test networks that are as close as possible to real networks. In order to do this, we have configured each virtual host and router by hand using tools that, as we already said, work without modifications on physical hosts too. This creates networks that model common use cases in real LAN scenarios and that are not very big in size. For this reason, we have been able to compare the original and the inferred topologies by looking at the individual network nodes. So we did not evaluate the aggregate metrics reported in section 2.1.1 but we concentrated on the individual nodes. We also neglect the results concerning edges because the presence or absence of edges is a consequence of the presence or absence of their endpoints, that is of nodes, and indeed we recognized that usually the same results were achieved by considering nodes or edges.

Let us define the metrics involved in the experiments:

- **True Positive (TP).** A node belonging to the original topology appears in the inferred topology.
- **True Negative (TN).** A node that does not belong to the original topology is not represented in the inferred topology.
- **False Positive (FP).** A node that appears in the inferred topology but that does not belong to the original topology.
- **False Negative (FN).** A node that does not appear in the inferred topology but that belongs to the original topology.
- **Precision (P).** The precision is a metric that tells us how much the algorithm has been precise in placing nodes in the inferred topology with respect to the nodes in the real topology. It is defined as $P = |TP| \,/\, |TP + FP|$. This metric alone is not sufficient to characterize the reconstructed topology because if we have a network

made of 100 nodes and the algorithm infers a network with only one true positive node, the resulting precision would be $1 / 1 + 0$ = 100%. So in this case the algorithm is very precise although the reconstruction is not good.

- **Recall (R).** The recall is a measure of completeness that complements the precision. It gives us an indication about how many nodes of the original topology the algorithm misses. It is defined as $R = |TP| / |TP + FN|$. In the previous example the recall would be $1 / 1 + 99$ = 1%, thus telling us that the algorithm missed almost all the nodes.
- **F1-Measure ($F_1$).** The F1-measure summarizes the results of a test by keeping into account both precision and recall. The F1-measure is the harmonic average of precision and recall. It reaches its best value at 1 (perfect precision and recall) and worst at 0. It is defined as $F_1 = 2 * ((P * R)/(P + R))$ . In the previous example the F1-measure would be $2 * ((1 * 0.01)/(1 + 0.01)) =$ 2%, thus telling us that the algorithm had very poor performances.

We will now present a collection of simulations. Each simulation runs the topology inference algorithm on a test network that models a common LAN scenario. The performance of the algorithm is discussed for each simulation.

**Simulation one.**

In the first simulation we setup a test network with a single router and two subnets. Two sensors are used to run the network topology inference algorithm. Each sensor is placed in a different subnet. We actually run three simulations. In the first one, each subnet includes 3 hosts. In the second one, each subnet includes 10 hosts. In the third one, each subnet includes 50 hosts.

Figure 16 shows the network topology when 50 hosts per subnet are used. Figure 17 shows one of the reconstructed topologies as exported from a blockchain node. Blue nodes represent responding nodes, grey nodes represents anonymous nodes.

From now on we will not show samples of the reconstructed topologies for space reasons.

Figure 18 shows a graphical representation of the evaluated metrics. We see that for this simple topology both precision and recall are good, indeed they are larger than 90%.

Anyway, the number of false positives grows linearly with the number of hosts in each subnetwork. We recognized that this problem is due to the network emulator. Indeed in certain deterministic cases some nodes do not reply to the traceroute probe, thus creating a hole in the traces that the algorithm (correctly) translates into an anonymous router.

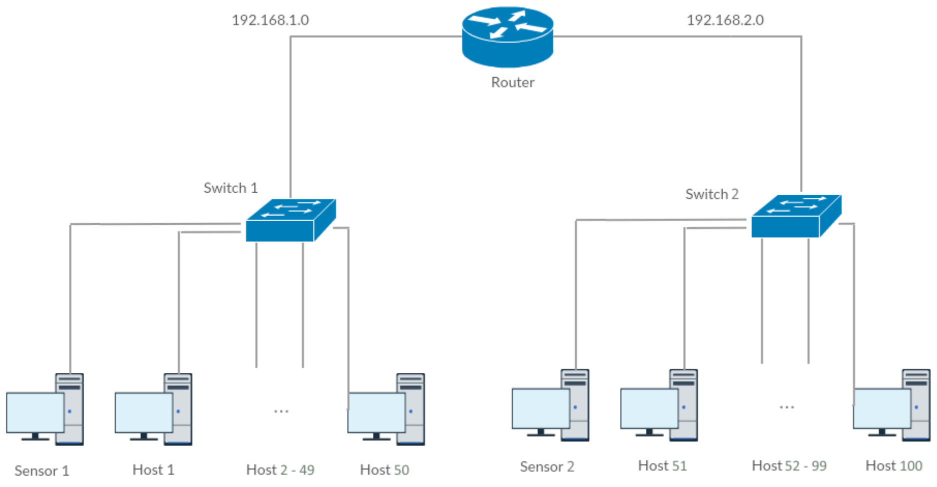


**Figure 16: Network topology for simulation one with 50 hosts per subnet.**

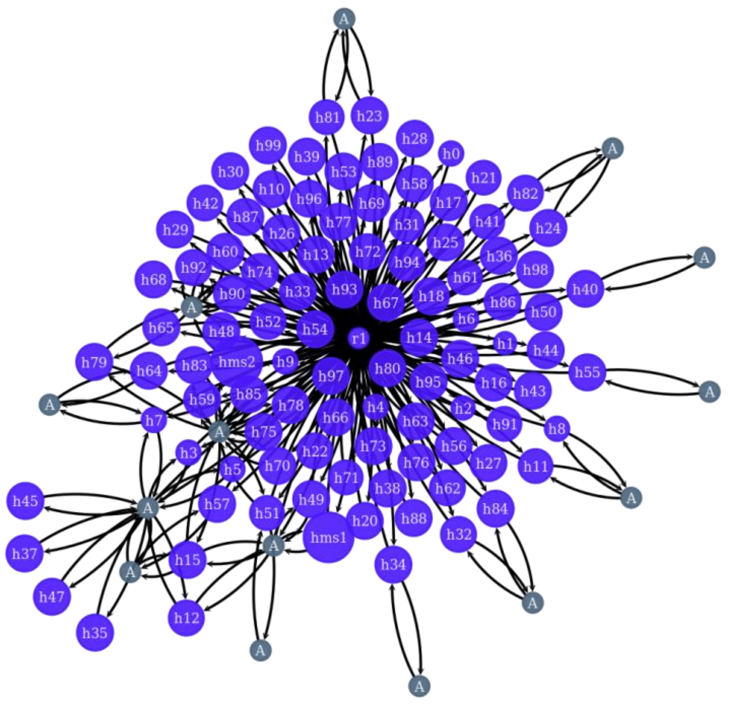


**Figure 17: Topology inferred by the algorithm when 50 nodes per subnet are used.**

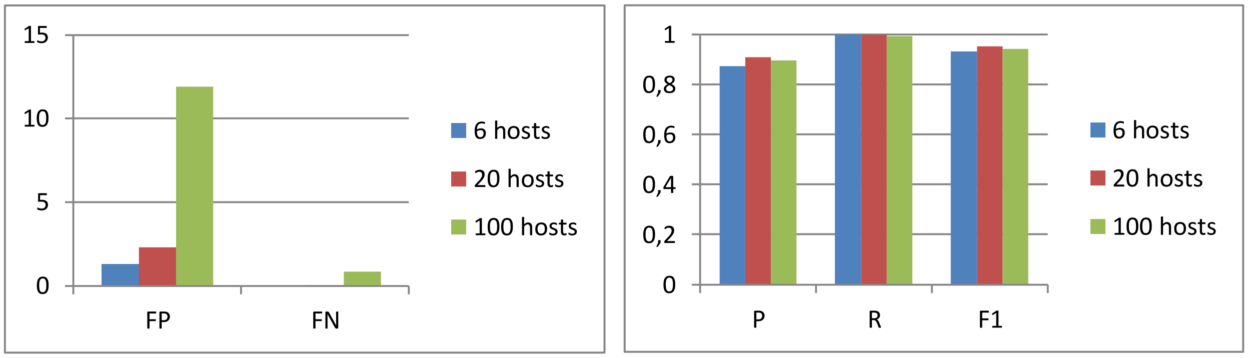


**Figure 18: Metrics of simulation one.**

## Simulation two

In the second simulation we setup a network with one router and three subnets. These three subnets represent the Internet, the demilitarized zone and a LAN. A sensor is placed in each subnet between the subnet hosts and the router, in a position that allows it to easily capture traffic from or to other subnets. Communication is possible among all the subnets.

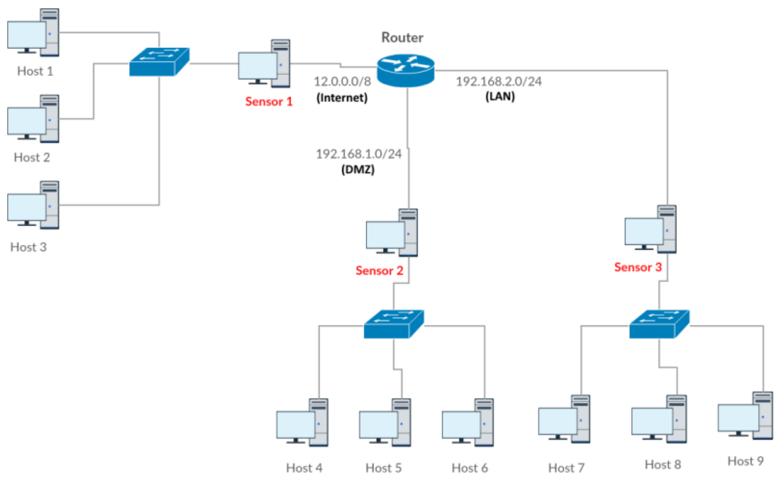
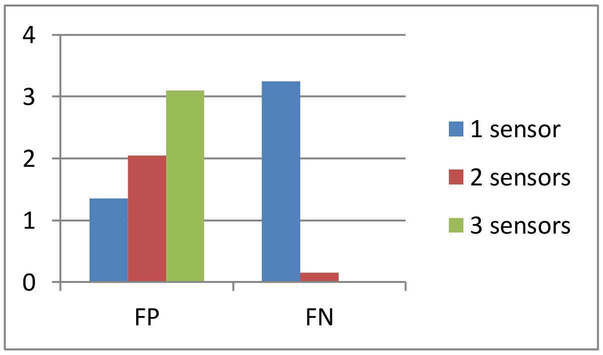


**Figure 19: Network topology for simulation two.**

Figure 20 shows the evaluated metrics for simulation two. Experiments have been carried out by using one, two and three random sensors. The recall is maximum and equal to one when three sensors are used. The precision slightly decreases with three sensors because of the introduction of false positives due to the network emulator. The F1-measure is about 0.9 using at least two sensors and indeed the topology reconstructed by the algorithm is good despite the presence of some false positive.



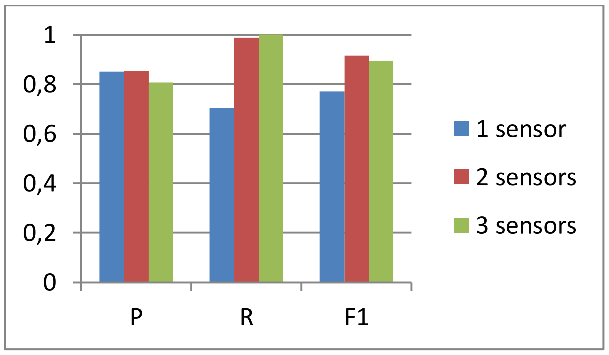


**Figure 20: Metrics of simulation two.**

### Simulation three

In the third simulation we setup a network that is analogous to the network in simulation 2, with the only exception that we insert a firewall instead of the router to allow the communications only between the demilitarized zone and the Internet. The traffic inside the LAN is completely segregated from the other networks.

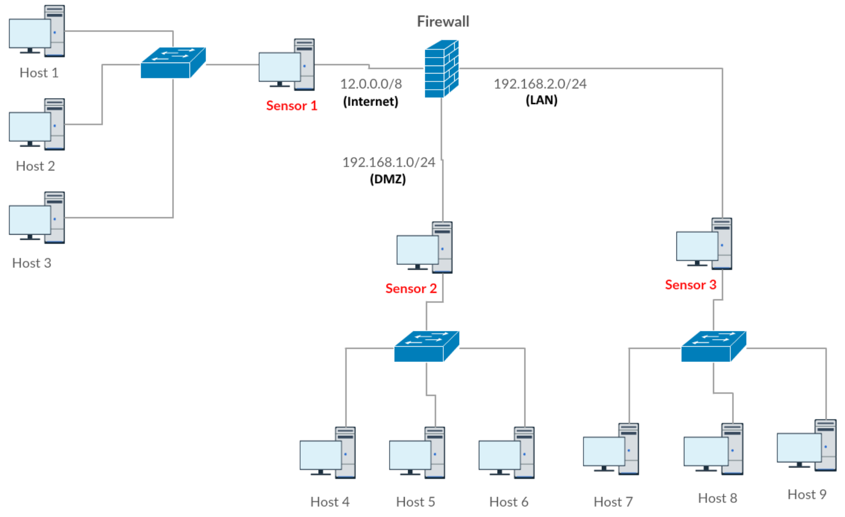


**Figure 21: Network topology for simulation three.**

Figure 22 shows the evaluated metrics for simulation three. Experiments have been carried out by using one, two and three random sensors. The precision is between 0.85 and 0.90 for all the choices of sensors. Indeed, using a larger number of sensors decreases the number of false negatives but, at the same time, it increases the number of false positives for the aforementioned reason, thus keeping the precision more or less constant. The recall is instead maximum and equal to 1 when three sensors are used. The high number of false negatives when only one random sensor is used is due to the fact that the networks are, at least partially, segregated. Hence, the sensor in the LAN subnet has no chance to capture traffic from the demilitarized zone or the Internet. So in this case the best choice is using all the three sensors.

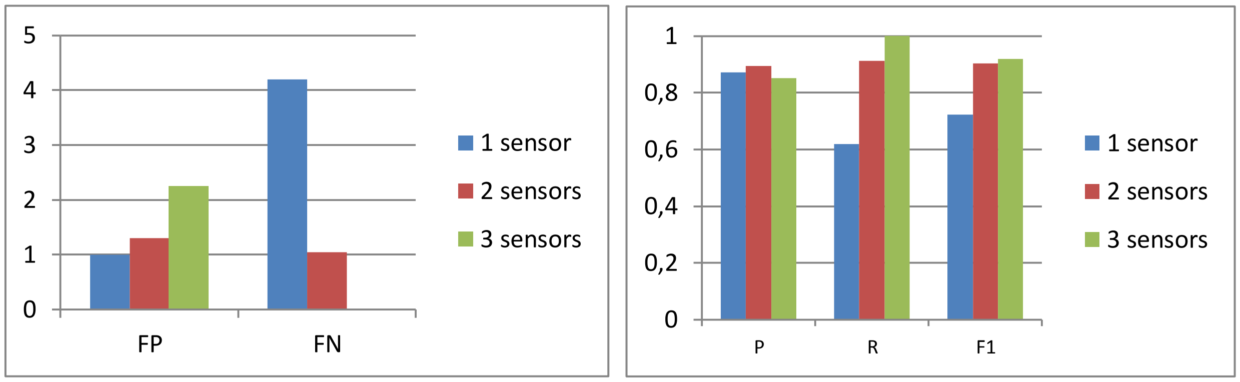


**Figure 22: Metrics of simulation three.**

## Simulation four

In the fourth simulation we setup a network with three routers and four subnets. . The subnets represent the Internet, the demilitarized zone and two LANs. A sensor is placed in each subnet between the hosts and the router. Communication is possible among all the subnets.

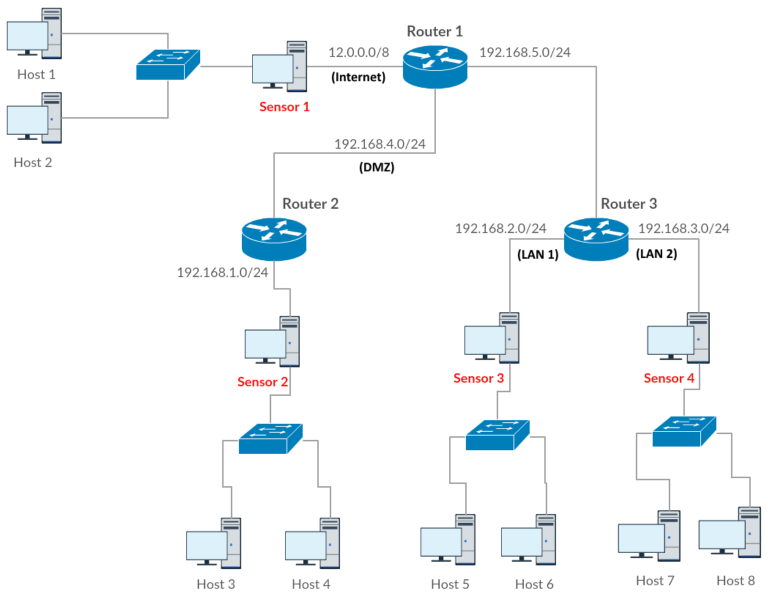


**Figure 23: Network topology for simulation four.**

Figure 24 shows the evaluated metrics for simulation four. Experiments have been carried out by using one, two, three and four random sensors. The precision slightly decreases with 3 and 4 sensors, due to the increasing number of false positives. The recall is maximum and equal to 1 when using at least two sensors. False negatives are zero if at least two sensors are employed. The F1-measure is best with two sensors. Since networks are not segregated, the best solution uses two sensors: all the nodes in the original topology are present in the inferred topology and the number of false positives is smaller than when using 3 or 4 sensors.

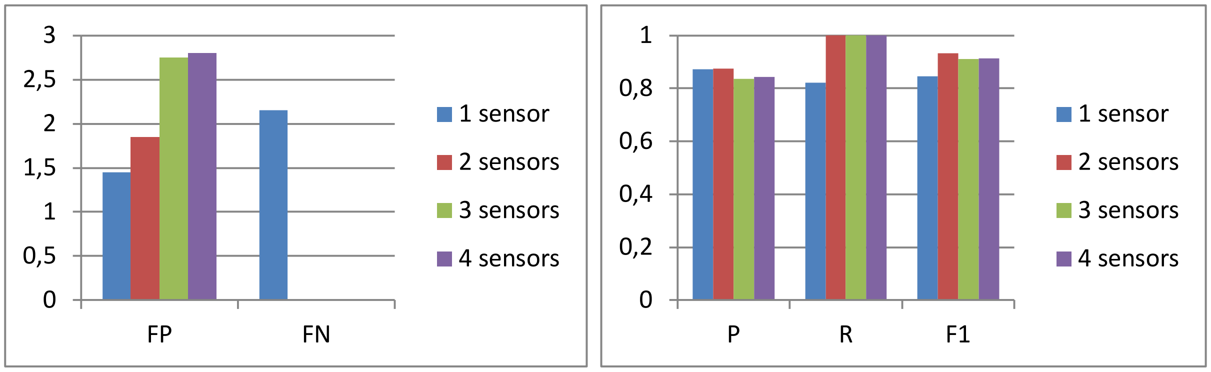


Figure 24: Metrics of simulation four.

## Simulation five

In the fifth simulation we setup a network with one router, two firewalls and four subnets. The subnets represent the Internet, the demilitarized zone and two LANs. A sensor is placed in each subnet between the hosts and the router or firewall. Communication is only possible between the Internet and the demilitarized zone and between the two LANs.

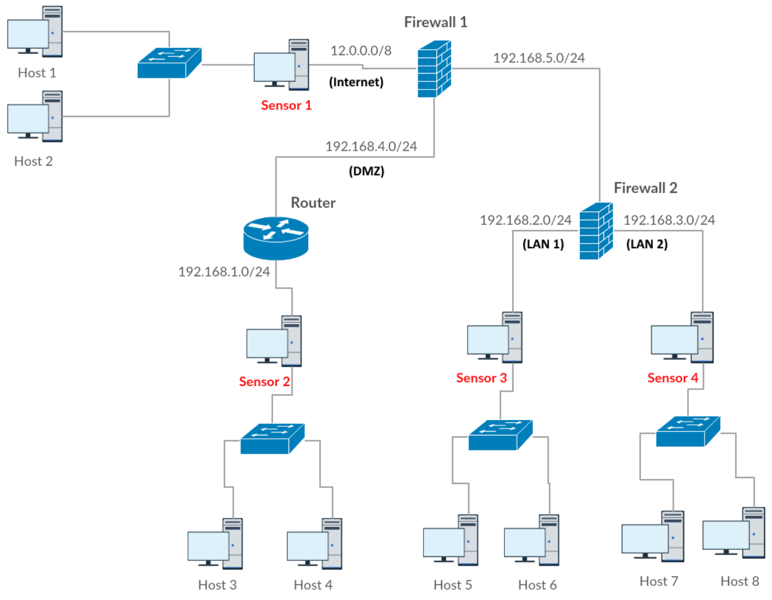


Figure 25: Network topology for simulation five.

Figure 26 shows the evaluated metrics for simulation five. Experiments have been carried out by using one, two, three and four random sensors. The effects of traffic segregation are clear: all the metrics, with the usual exception of false positives, are enhanced by using more sensors. Here the main focus is on the number of false negatives, that sharply decreases by increasing the number of sensors. Thus in this simulation the best solution uses all the 4 sensors available, that is one sensor per subnet.

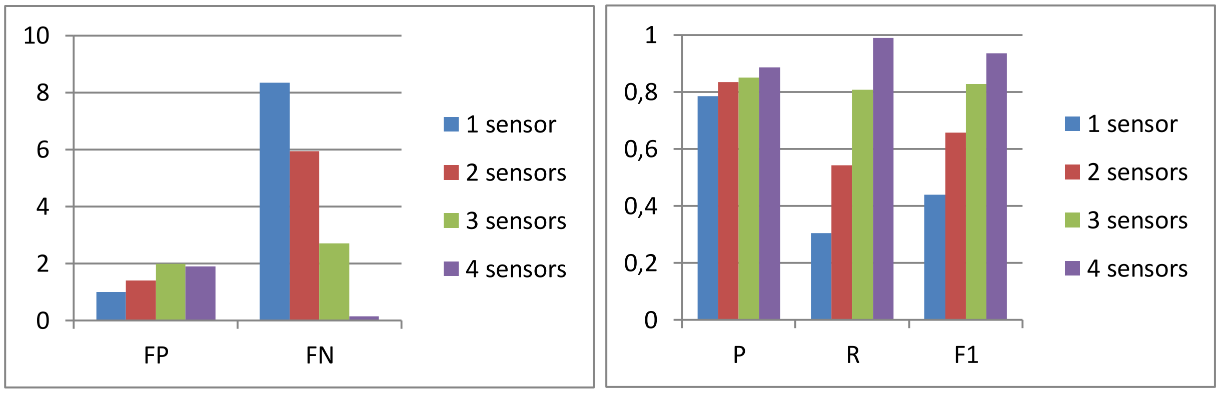


**Figure 26: Metrics of simulation five.**

## Simulation six

In the sixth simulation we setup a network with three routers and four subnets, where the root router is anonymous. The subnets represent the Internet, the demilitarized zone and two LANs. A sensor is placed in each subnet between the hosts and the router. Communication is possible among all the subnets.

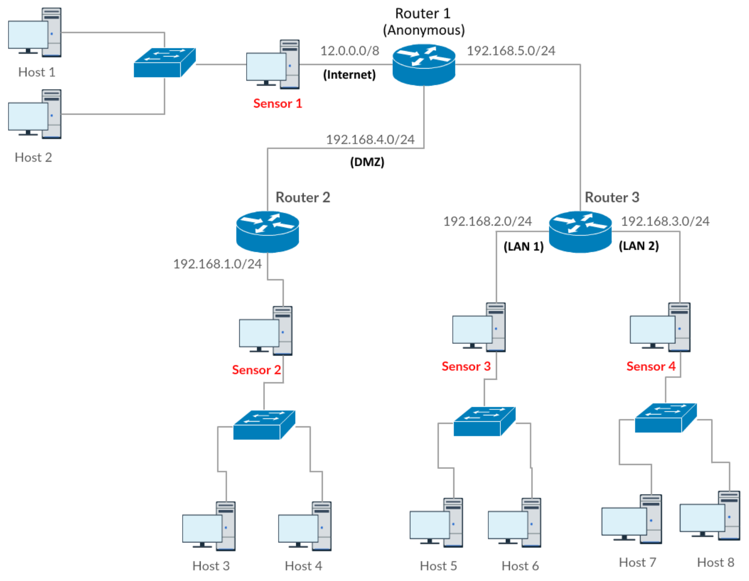


**Figure 27: Network topology for simulation six.**

Figure 28 shows the evaluated metrics for simulation six. Experiments have been carried out by using one, two, three and four random sensors. The precision decreases as the number of sensors increases because more sensors also means more false positives. As already said, this is due to the misbehavior of the emulator. The recall is maximum and equal to 1 for any choice of sensors. Anyway when using one sensor the reconstructed topology includes two connected components, thus not representing faithfully the original topology. So here we should use at least two sensors. The best choice, based on the F1-score, is using three sensors.

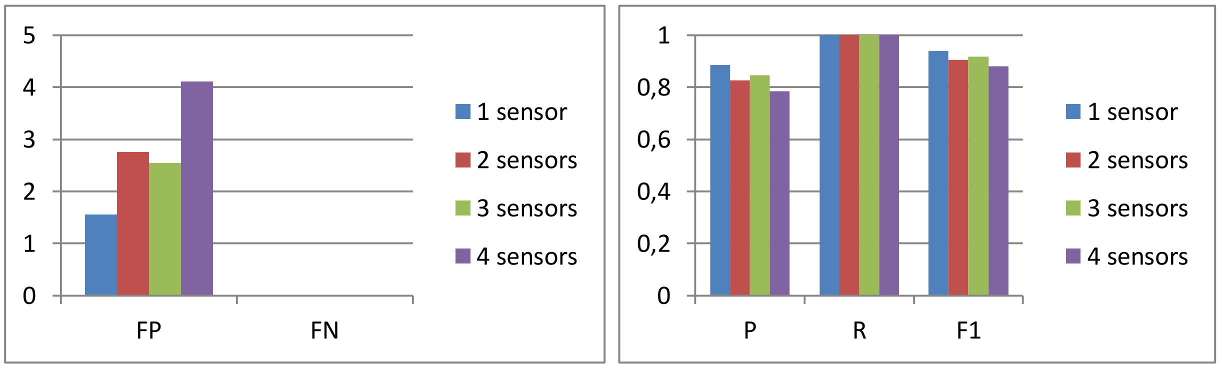


**Figure 28: Metrics of simulation six.**

### Simulation seven

In the seventh simulation we setup a network with three routers and four subnets, where the root router is blocking. The subnets represent the Internet, the demilitarized zone and two LANs. A sensor is placed in each subnet between the hosts and the router. The blocking router drops all the incoming packets and this prevents the communication between hosts in distinct subnetworks. Only LAN1 and LAN2 can communicate with each other, because the corresponding packets do not cross the blocking router.

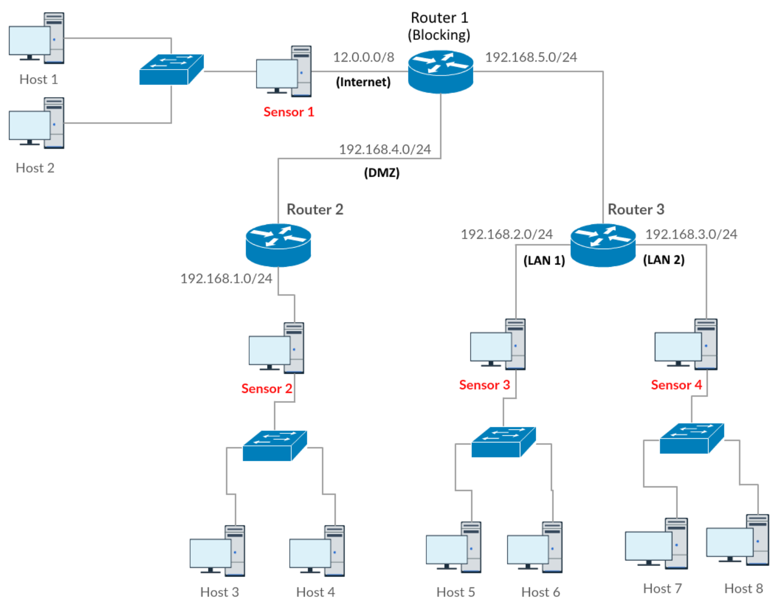


**Figure 29: Network topology for simulation seven.**

Here the reconstructed topology is poor because the blocking router prevents the sensors from having a global view of the topology. Each sensor reconstructs the portion of network that it can observe and sends the inferred topology to the blockchain nodes. These nodes put all the received topologies together, but the final topology consists of three disconnected components (the Internet, the DMZ and the LANs) because the blocking router never answers to a traceroute probe. This creates a hole in the traces that prevents the connection of the three disconnected components. A centralized NOC that collects the traces from all the sensors and then runs the network topology inference algorithm could reconstruct a single connected component, but this approach would nullify the benefits that we gain from decentralization. Instead, we have devised a heuristic that allows the reconstruction of a single connected component when a blocking router is present by exploiting some knowledge of the topology.

## Heuristic

The idea is that, since we are trying to infer the network topology of a LAN, we may know some information about it. For example, we could know that the routers and the hosts of the topology are arranged according to a given **pattern**. What we don't know is, for instance, which are the exact nodes in the network, and that's the reason why we are using a network topology inference algorithm.

If we know that the nodes are arranged to form a tree, and we know the identity of the blocking root node, we can reconstruct the correct topology that consists of a single connected component. We have tested this idea by implementing a tree pattern. Each sensor is told which is the root node and which is the node, belonging to the inspected branch of the tree, that is connected to the root. This way each sensor sends extra transactions to the blockchain nodes that allow them to join the disconnected components to form a single tree.

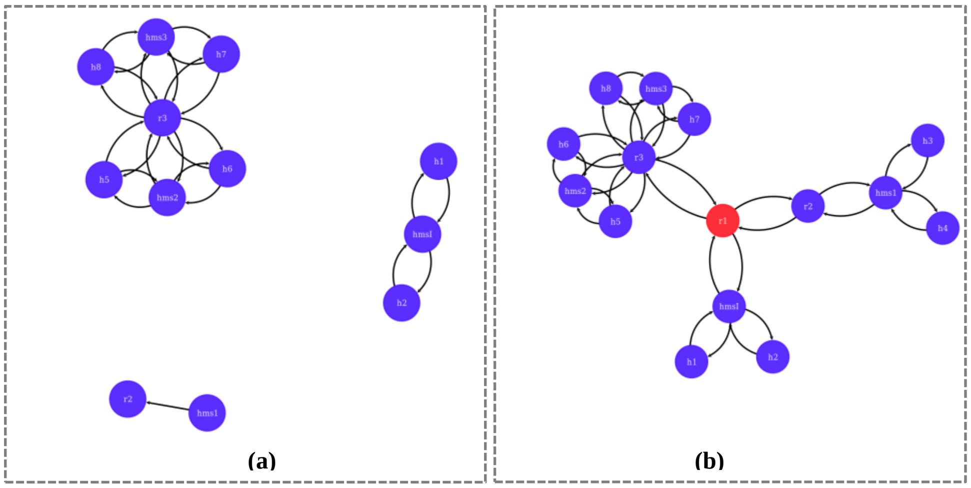


**Figure 30: Reconstructed topology for simulation seven without (a) and with (b) the heuristic. In Figure b the special "pattern" node is depicted in red.**

### Simulation eight

In the eighth simulation we analyze a test network structured as a tree of subnets. The test network is made of four firewalls and five subnets that represent three LANs, a demilitarized zone and the Internet. With the only exception of the subnet modelling the Internet, for which we have deployed just one host, each subnet includes seven hosts and one sensor. The firewalls are configured to allow the following communications:

- Firewall 1: allows the communication between h11 (LAN1) and h21 (LAN2).
- Firewall 2: allows the communication between h31 (LAN3) and h21 (LAN2).
- Firewall 3: allows the communication between h41 (LAN1) and h21 (LAN2).
- Firewall 4: allows the communication between any host in the Internet and any host in the DMZ.

Figure 32 shows the evaluated metrics for simulation eight. Experiments have been carried out by using one, two, three and four random sensors. The best results are produced when employing 4 sensors. In this case, false negatives are zero. The network emulator did not introduce any false positive in these experiments, so when using four sensors a F1-score equal to 1 is achieved.

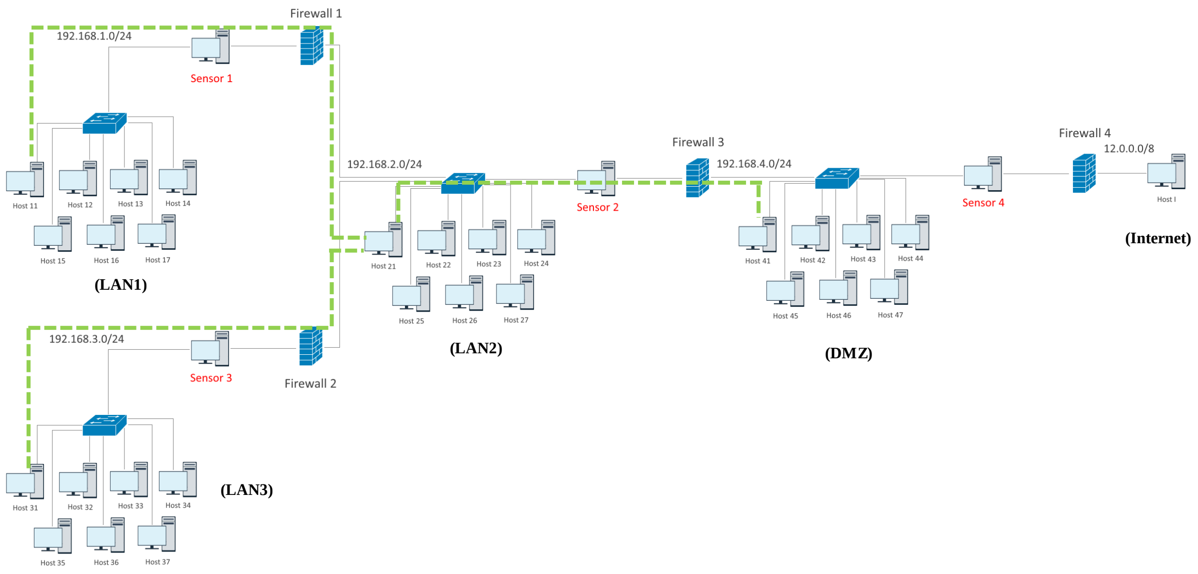


**Figure 31: Network topology for simulation eight. The green dotted lines represent the communications allowed by firewalls 1, 2 and 3. Firewall 4 allows the communications between any host in the DMZ and any host in the Internet.**

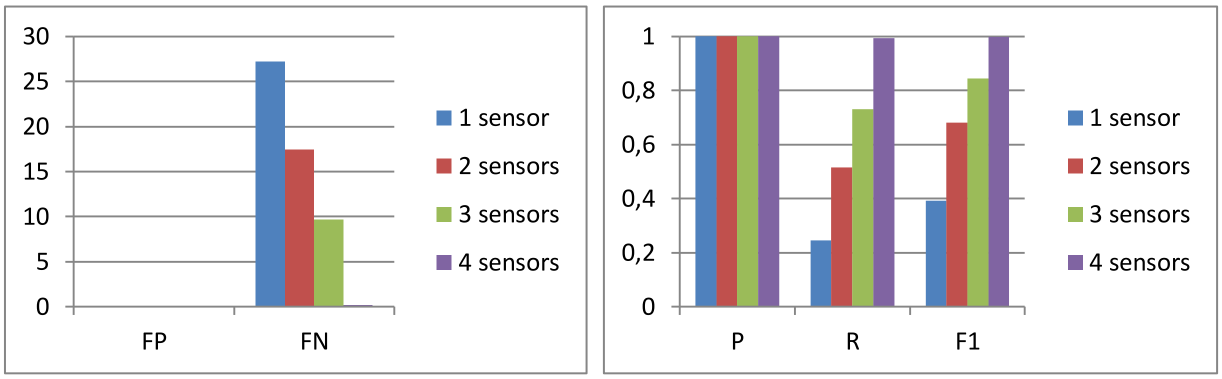


**Figure 32: Metrics of simulation eight.**

## 5.2 Blockchain

We measured the time that the blockchain nodes take to reach consensus by varying the quorum values and the number of malicious nodes in the blockchain.

For each simulation, we ran 20 experiments with the same parameters and then we computed the average of the experiment execution times.

The architecture is shown in figure 33. There is a client sending (honest) transactions to the six server nodes storing the blockchain. Each server has a UNL made of the other 5 blockchain nodes. Each server node runs on a virtual machine. Each virtual machine is run on a separate physical machine. The nodes communicate via a 802.11ac wireless router reserved for the experiments. All the blockchain nodes are honest with the exception of nodes 2 and 3 that, depending on the experiments, may be malicious and may try to insert also fraudulent transactions into the ledger.

The plotted execution times are always referred to node 1, which is a honest node but whose time to reach consensus varies based on the number of transactions that it has to process and on the presence of malicious nodes. Consensus is reached in all the experiments.

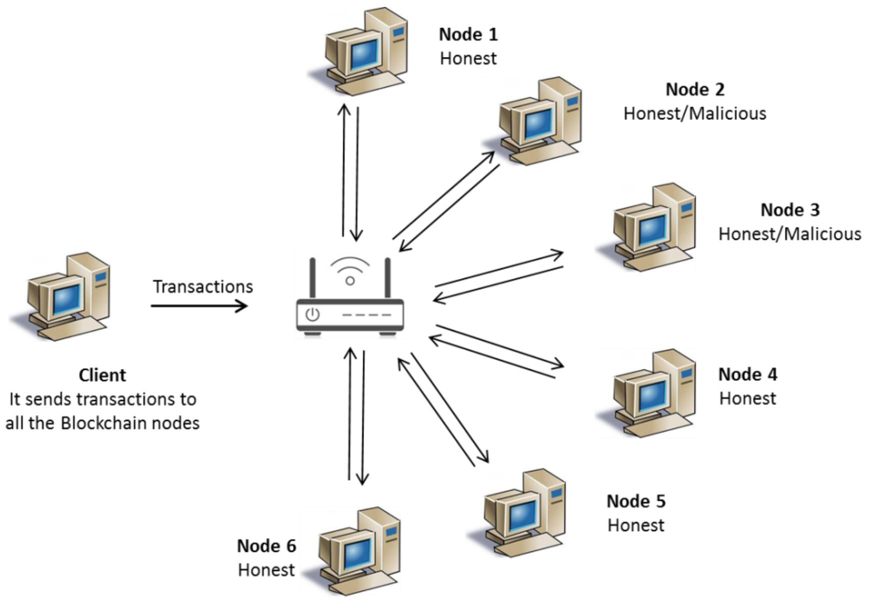


**Figure 33: Architecture for the blockchain simulations.**

Finally, we should note that the time to reach consensus also depends on the constants defined in the configuration file of each node. These constants affect the minimum and maximum amount of time that is spent in each consensus phase and were presented in section 2.2.1. All the experiments were conducted with the following constants:

- `LEDGER_MIN_CLOSE = 40s`
- `LEDGER_MAX_CLOSE = 60s`
- `LEDGER_MIN_CONSENSUS = 20s`
- `LEDGER_MAX_CONSENSUS = 60s`

**Simulation one**

The client sends 500 transactions to the blockchain nodes. We assume a quorum equal to 60% of the nodes in the UNL. Since each UNL includes 5 nodes, a node declares the consensus reached if at least 3 nodes in its UNL have validated the same ledger. Figure 34 compares the execution time of one consensus round of node 1 when:

- All nodes are honest.
- Node 2 is malicious.
- Node 2 and node 3 are malicious.

The execution times are plotted as a function of the number of fraudulent transactions inserted by the malicious nodes. The malicious nodes send the fraudulent transactions in addition to the honest transactions.

The execution times are highest when there are two malicious nodes inserting 500 fraudulent transactions. This delay is intuitive: although the fraudulent transactions are not inserted in the ledger, they have to be processed and discarded by the nodes.

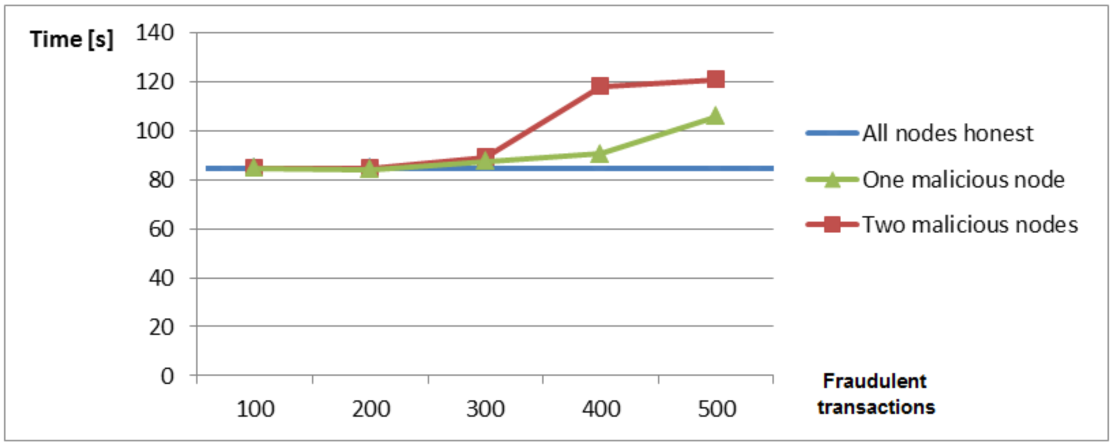


**Figure 34: Execution times for one consensus round.**

**Simulation two**

Simulation two compares the execution times of the three phases (open, establish and accept) of the consensus round when malicious nodes are present.

We actually carried out three simulations. For each simulation we have plotted a bar chart to give a better visualization of how the execution times of the various phases change depending on the additional fraudulent transactions inserted by the malicious nodes. In all the cases the client sends 500 transactions to the blockchain nodes while the malicious nodes try to validate a varying number of fraudulent transactions too.

Simulation **2a** compares the execution times when the quorum is equal to 60% and one malicious node is present. The execution times increase when the malicious node inserts more transactions.

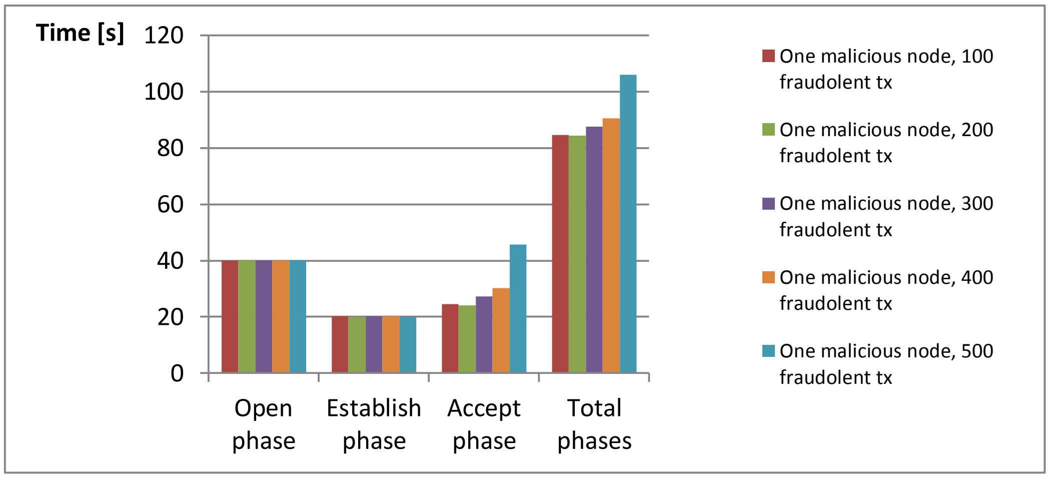


**Figure 35: Simulation 2a. Execution times of the various phases with one malicious node and quorum = 60%.**

Simulation **2b** compares the execution times when the quorum is equal to 60% and two malicious nodes are present inside the blockchain. The execution times are largest when the malicious nodes insert more transactions. The worst case execution time is larger than the worst case execution time of simulation 2a. So two malicious nodes slow down, but not prevent, the consensus process more than a single malicious node.

Simulation **2c** compares the execution times when the quorum is equal to 80% and one malicious node is present. The execution times grow with the number of fraudulent transactions inserted by malicious nodes and are comparable with the ones of simulation

**2a**. This suggests that the different quorum values do not affect the time to reach consensus. Instead, the presence of malicious nodes inserting additional fraudulent transactions slows down the consensus process.

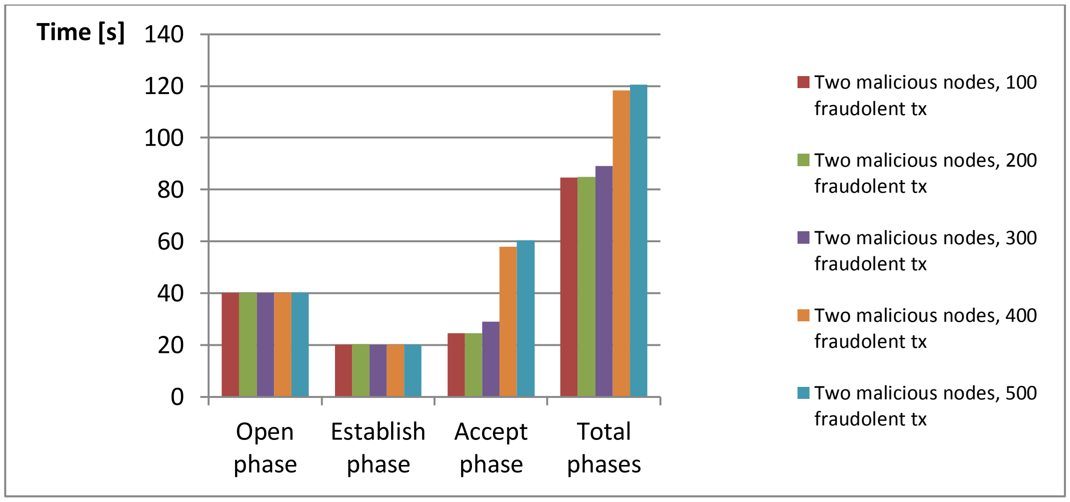


**Figure 36: Simulation 2b. Execution times of the various phases with two malicious nodes and quorum = 60%.**

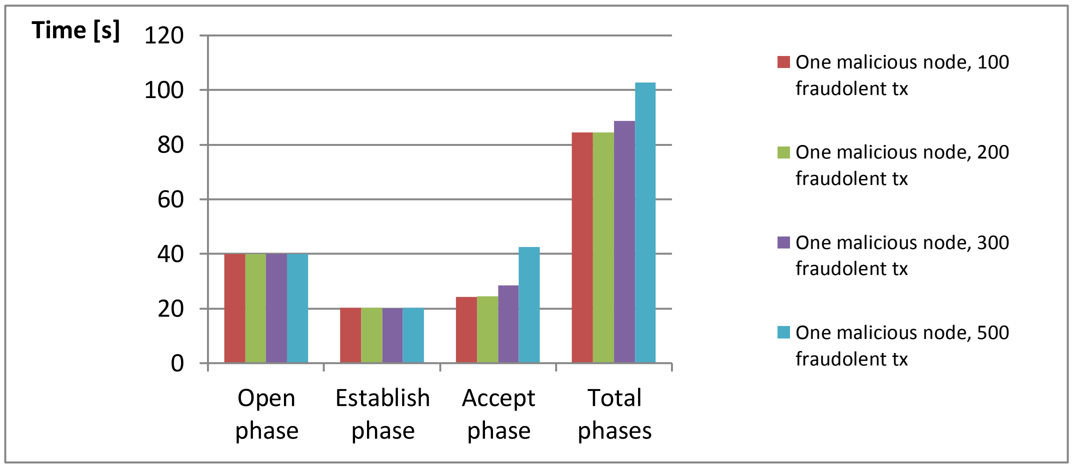


**Figure 37: Simulation 2c. Execution times of the various phases with one malicious node and quorum = 80%.**

## Simulation three

The third simulation assesses the impact of distinct quorum values on the time to reach consensus. The client sends a variable number of honest transactions (100, 200, 300, 400 or 500) to the blockchain nodes. All the blockchain nodes are honest. We consider three different values for the quorum: 60%, 80% and 100%.

In Figure 38 we plot the average execution times of one consensus round as a function of the number of transactions sent by the client. In all the three cases, the execution time grows linearly with the number of inserted transactions. Instead, the different quorum values do not affect the execution times. This is a further evidence that the time to reach consensus depends on the number of transactions to be validated and on the number of malicious nodes but not on the quorum values.

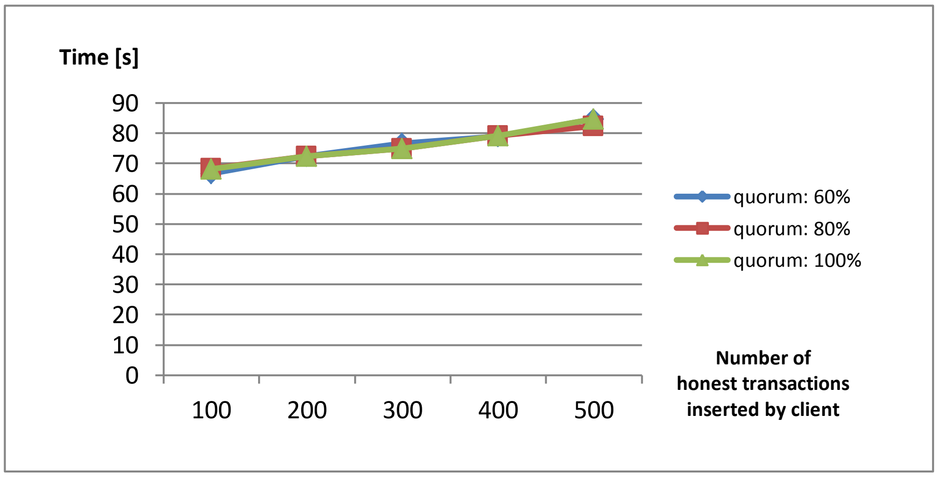


**Figure 38: Execution times for one consensus round as a function of the number of inserted transactions.**

## Simulation four

In the fourth simulation we assume a quorum equal to 80% and we simulate a malicious node that does not take part actively to the consensus process. Figure 39 compares the execution times of a consensus round when all the nodes are honest and when such a malicious node is present. The times differ in the accept phase. Indeed each honest node waits up to `LEDGER_MAX_CONSENSUS` time to receive the ledger from all its peers. Since the malicious node is not going to send any ledger, the honest nodes wait for the maximum amount of time. So such a malicious node does not prevent the honest nodes from reaching the consensus. Its only slows down the consensus process.

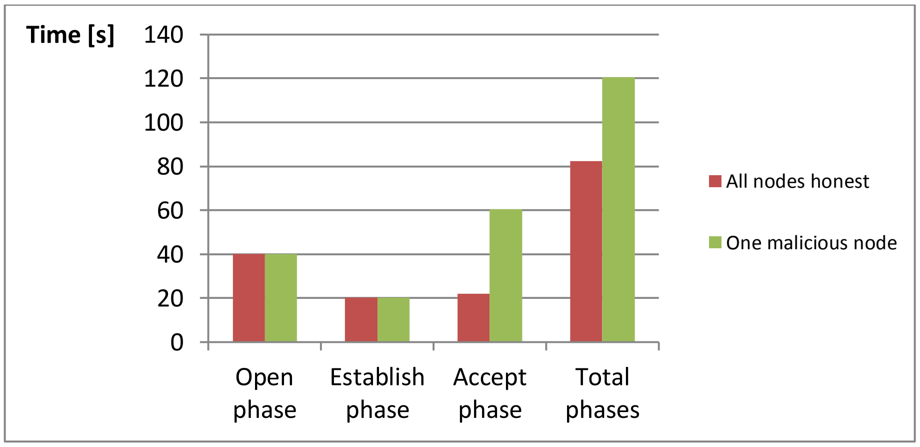


**Figure 39: Execution times for one consensus round when all nodes are honest (red bars) and when one node does not take part to the consensus process (green bars).**

# 6. Conclusions

We implemented and evaluated a tool that automatically discovers the topology of a network and securely stores it using a blockchain.
A map of the network simplifies network management tasks such as network diagnosis and resource management and may also allow the detection of unauthorized nodes inside the network. The inferred topology is stored into a blockchain in order to make it resistant to the tampering attempts by malicious nodes.

In order to evaluate the network topology inference algorithm, we set up a collection of tests that cover the most common topologies in a LAN scenario and we ran the algorithm on these networks. The algorithm is run by sensors, network nodes that capture and analyze traffic to discover new nodes inside the network and periodically check the reachability of the known network nodes. This way, they are able to track the evolution of the network topology. Every time they discover a change in the network topology, they run the network topology inference algorithm and store the new inferred topology into the blockchain.
Despite the introduction of false positive nodes in the inferred topology due to the network emulator, we have discovered that in general the deployment of one sensor per subnet results in the deduction of a topology which is equal or very close to the real one even in presence of firewalls or anonymous routers. When blocking routers are present, each sensor can only reconstruct the subnetwork before the block. This results in a wrong reconstructed topology that consists of several disconnected components. In order to reconstruct a more faithful topology we devised a heuristic that exploits prior knowledge about the network.

In order to evaluate the blockchain, we set up a network of six nodes. Each node ran on a different physical machine and executed the distributed consensus protocol with the other nodes. Several simulations were ran by varying the quorum value, the number of malicious nodes and the type of attack. We always chose values for the quorum and for the number of malicious nodes that allowed to reach the consensus on a ledger made of honest transactions. We observed that distinct quorum values do not affect the time to reach consensus. Instead, the transmission of fraudulent transactions to the blockchain nodes in addition to honest transactions slows down the consensus process because fraudulent transactions have to be processed before they can be discarded. This suggests that malicious nodes may attack the blockchain by carrying out denial-of-service (DOS) attacks. In the last simulation we assessed the limit case in which a fraudulent node carries out an attack which consists in never sending any proposal or ledger to the other nodes. This models a faulty node which cannot send transactions to the other blockchain nodes. Anyway, consensus is still reached in finite time if a number of nodes larger than the quorum reach the consensus on some ledger, because the maximum time to wait for the missing ledger is bounded.

Future developments include the integration of alias resolution techniques into the network topology inference algorithm. Furthermore, the discovered network topology could be fed to a vulnerability scanner to assess the vulnerabilities of the nodes in the network. The network topology and the list of vulnerabilities affecting network nodes could be used by tools that assess and manage the cyber risk in the network. An example is the Haruspex suite, a collection of tools that starting from the system topology and the list of vulnerabilities affecting the network nodes is able to quantitatively assess the risk of an ICT system by implementing a Monte Carlo method that simulates the agent plans [4,5,6,7].

# Acknowledgments

I sincerely thank Professor Baiardi for all the help he gave me during the writing of this thesis. His advices have been precious to me and I mainly due my interest in cyber security topics to its lectures.

I really thank Jessica for supporting me in the hard days divided between work and study. As long as we'll stay together, I know that I'll be able to climb even the steepest walls.

I thank my family for always giving me what I needed.

I thank my friends for always being there for me.

# References

[1] D. Achlioptas, A. Clauset, D. Kempe and C. Moore, “On the bias of traceroute sampling: or, power-law degree distributions in regular graphs”, *Journal of the ACM*, Vol. 56, no. 4, Article 21, 2005.

[2] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien and R. Teixeira, “Avoiding traceroute anomalies with Paris traceroute”, *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement* , New York, NY, USA, 2006, pp.153-158.

[3] B. Augustin, T. Friedman and R. Teixeira, “Multipath tracing with Paris traceroute”, *E2EMON 2007 - 5th IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services*, Munich, Germany, May 2007, pp.1-8.

[4] F. Baiardi, F. Corò, F. Tonelli, A. Bertolini, R. Bertolotti and D. Pestonesi, "Assessing and Managing ICT Risk with Partial Information", *CSS 2014*, Paris, France, 20-22 August 2014.

[5] F. Baiardi, F. Corò, F. Tonelli, L. Guidi, and D. Sgandurra, “Simulating Attack Plans Against ICT Infrastructures”, *ICVRAM 2014,* Liverpool, United Kingdom, 13-16 July 2014.

[6] F. Baiardi, F. Corò, F. Tonelli, and D. Sgandurra, “Automating the Assessment of ICT Risk”, *Journal of Information Security and Applications,* vol. 19, no.3, 2014, pp.182-193.

[7] F. Baiardi, F. Corò, F. Tonelli, and D. Sgandurra, “A Scenario Method to Automatically Assess ICT Risk”, *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing,* Torino, 2014, pp. 544-551.

[8] R. Castro, M. Coates, G. Liang, R. Nowak, R., and B. Yu, “Network tomography: Recent developments”, *Statistical Science*, vol.*19*, no. 3, 2004, pp.499-51.

[9] B. Chase and E. MacBrough, “Analysis of the XRP Ledger Consensus Protocol”, 2018.

[10] J. Chen and S. Micali, “ALGORAND”, 2017.

[11] J. Chen, S. Gorbunov, S. Micali, and G. Vlachos, “ALGORAND AGREEMENT: Super Fast and Partition Resilient Byzantine Agreement”, *IACR Cryptology ePrint Archive*,2018.

[12] *Cisco Visual Networking Index: Forecast and Trends, 2017–2022*, <https://www.cisco.com/c/en/us/solutions/collateral/service-provider/ visual-networking- index-vni/white-paper-c11-741490.html>, 09 February 2019.

[13] *Consensus and Validation*, 2018, < https://github.com/ripple/ rippled/blob/develop/docs/consensus.md>, 09 February 2019.

[14] Coates, A. O. Hero III, R. Nowak and Bin Yu, "Internet tomography," *IEEE Signal Processing Magazine*, vol. 19, no. 3, May 2002, pp. 47-65.

[15] *Discovering and Securely Storing a Network Topology*, <https://github.com/Balzu/network-inference-with-Blockchain>, 10 February 2019.

[16] B. Donnet and T. Friedman, "Internet topology discovery: a survey", *IEEE Communications Surveys & Tutorials*, vol. 9, no. 4, Fourth Quarter 2007, pp. 56-69.

[17] N. G. Duffield, J. Horowitz, F. Lo Presti and D. Towsley, "Multicast topology Inference from measured end-to-end loss," *IEEE Transactions on Information Theory*, vol. 48, no. 1, Jan. 2002, pp. 26-45.

[18] N. G. Duffield and F. Lo Presti, "Network tomography from measured end-to-end delay covariance", *IEEE/ACM Transactions on Networking*, vol. 12, no. 6, Dec. 2004, pp. 978-992.

[19] El-Shekeil, A. Pal and K. Kant, "CloudMiner: A Systematic Failure Diagnosis Framework in Enterprise Cloud Environments," *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Nicosia, 2018, pp. 73-80.

[20] *European countries join Blockchain Partnership*, 2018, <https://ec.europa.eu/ digital-single-market/en/news/european-countries-join-blockchain-partnership>, 09 February 2019.

[21] M. J. Fischer, N. A. Lynch and M.S. Paterson, “Impossibility of Distributed Consensus with One Faulty Process”, *J. ACM*, vol. 32, 1985, pp. 374-382.

[22] Y. Gilad, R. Hemo, S. Micali, G. Vlachos and N. Zeldovich, “Algorand: Scaling Byzantine Agreements for Cryptocurrencies”, *IACR Cryptology ePrint Archive*, 2017.

[23] *Graph-Tool*, <https://graph-tool.skewed.de>, 09 February 2019.

[24] Govindan and H. Tangmunarunkit, "Heuristics for Internet map discovery", *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, Tel Aviv, Israel, 2000, pp. 1371-1380.

[25] M. H. Gunes and K. Sarac, "Analytical IP Alias Resolution," *2006 IEEE International Conference on Communications*, Istanbul, 2006, pp. 459-464.

[26] H. Haddadi, M. Rio, G. Iannaccone, A. Moore and R. Mortier, "Network topologies: inference, modeling, and generation," *IEEE Communications Surveys & Tutorials*, vol. 10, no. 2, Second Quarter 2008, pp. 48-69.

[27] B. Holbert, S. Tati, S. Silvestri, T. F. La Porta and A. Swami, "Network Topology Inference With Partial Information," *IEEE Transactions on Network and Service Management*, vol. 12, no. 3, Sept. 2015, pp. 406-419.

[28] B. Huffaker, D. Plummer, D. Moore and K. Claffy, "Topology discovery by active probing", *Proceedings 2002 Symposium on Applications and the Internet (SAINT) Workshops*, Nara, Japan, 2002, pp. 90-96.

[29] X. Jin, W.-P. Yiu, S.-H. Chan, and Y. Wang, “Network topology inference based on end-to-end measurements*”,IEEE Journal on Selected Areas in Communications*, vol. 24, no. 12, 2006, pp. 2182–2195.

[30] A. Lakhina, J. W. Byers, M. Crovella and P. Xie, "Sampling biases in IP topology measurements", *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies*, vol.1, San Francisco, CA, 2003, pp. 332-34.

[31] B. Lantz, B. Heller and N. McKeown, “A Network in a Laptop: Rapid Prototyping for Software-Defined Networks”, *9th ACM Workshop on Hot Topics in Networks*, Monterey, CA, October 2010.

[32] S.D. Leopold, N. Englesson, “How Eco friendly is our money and is there an alternative?”, 2017.

[33] M. Luckie, “Scamper: a Scalable and Extensible Packet Prober for Active Measurement of the Internet”, *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement (IMC '10)*, New York, NY, USA, 2010, pp. 239-245.

[34] P. Marchetta, A. Montieri, V. Persico, A. Pescapé, Í. Cunha and E. Katz-Bassett, "How and how much traceroute confuses our understanding of network paths", *2016 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, Rome, 2016, pp. 1-7.

[35] S. Nakamoto, “Bitcoin: a Peer-to-Peer Electronic Cash System”, 2008.

[36] *Pure Python RSA implementation*, <https://github.com/sybrenstuvel/python-rsa>, 09 February 2019.

[37] D. Schwartz, N. Youngs and A. Britto, “The Ripple Protocol Consensus Algorithm”, 2014.

[38] *Siemens invests in LO3 Energy and strengthens existing partnership*, 2017, <https://www.siemens.com/press/en/pressrelease/?press=/en/pressrelease/2017/energymanagement/pr2017120121emen.htm>, 09 February 2019.

[39] *Transaction*, <https://en.bitcoin.it/wiki/Transaction>, 09 February 2019.

[40] B. Yao, R. Viswanathan, F. Chang, and D. Waddington, “Topology inference in the presence of anonymous routers”, *IEEE INFOCOM*, 2003.

[41] B. Zhang, R. Liu, D. Massey, and L. Zhang, “Collecting the Internet AS-level Topology”, *ACM SIGCOMM CCR*, vol. 35, no. 1, 2005, pp. 53–61.