Discovering the Topology in an Unknown Network

Silviu Mărîi
Supervisor(s): Jérémie Decouchant, Bart Cox

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
22-6-2022

I. Abstract

Discovering the topology in an unknown network is a fundamental problem for the distributed systems that faces several backlashes due to the proneness of such systems to Byzantine (i.e. arbitrary or malicious) failures. During the past decades, several protocols were developed to allow a network to be resilient up to a certain number of Byzantine processes and guarantee a reliable communication between the correct processes. However, they still suffer from high complexity or unrealistic assumptions which makes them impractical. In this paper modifications made to one of the current state-of-art protocols are presented, modifications that can potentially reduce the load of the network and improve the Reliable Communication layer. We employ empirical analysis in order to prove that reliable communication can still be achieved with these modifications in place and to measure the performance. The findings of this study show that our improved protocol has a reduction of 18.38% in the number of exchanged messages.

***Keywords* – Reliable Communication, Topology Discovery, Explorer2**

II. Introduction

In this paper, the topology reconstruction of an arbitrary network in the presence of Byzantine processes is investigated. Each process may not be aware of its neighbors, but it is guaranteed that every process has an arbitrary number of direct links to other processes with a lower bound of $2 * f + 1$ and it needs to be able to reconstruct the topology of the entire network. The variable f represents the maximum number of faulty processes the network can tolerate. A Byzantine process is by definition a process that behaves arbitrarily: it can malfunction or crash, send arbitrarily messages or it can be malicious and attempt to disrupt other connections.

The Byzantine Generals is one of the most famous computer science problems and provides a clear example of the challenges in a communication network [1]. As it is impossible to have in reality a direct connection between each process in an ever-growing network, a process may need to rely on other processes to forward a message to the desired target, but these processes might prove to be Byzantine and compromise the integrity of the communication. Dolev addressed this issue in 1981 and developed a broadcast protocol which solves the reliable communication problem in an unknown network with at most f faulty processes, although this protocol is not feasible in large systems as it floods the network with messages, generating a factorial complexity [2]. An alternative version of Dolev's protocol that can potentially reduce the message complexity from factorial to polynomial was proposed [3], but it faces several problems: the improvements made may become insignificant in an asynchronous network and the complexity of the worst-case scenario is still factorial. Other protocols that have an optimal complexity were designed, but these protocols assume that the network topology is known by every process and therefore, their practical applications are greatly reduced [4]. Thus, reconstructing the topology of an unknown network became essential in distributed computing. Nesterenko and Tixeuil divised a protocol called Explorer to solve this problem, but it was later shown that it fails to guarantee one of the properties of the reliable communication [5]. Recently, Farina proposed a new solution which employs several modifications of the Dolev protocol and has a polynomial complexity of reliable communication instances [6]. Yet, it is still unclear whether this complexity can be further reduced.

In this study we explore the fundamental relations of topology discovery. We select the weakest practical state models for a communication network and the current assumptions made by the aforementioned state-of-art protocols. The main contribution of this study is examining if it is possible to reconstruct the topology using Explorer2 when only a part of the processes broadcast initially (in contrast to the current protocol in which all processes broadcast initially) and how the processes can optimize Reliable Communication layer based on the found topology. To achieve this goal, the processes will be replicated separately in order to simulate the conditions of a real network and sample actual latency and network throughput data.

In section III insights about the related work are provided, in section IV we state the system model and the problem statement. In section V we detail our approach and in section VI we introduce the changes made to the Explorer2 protocol in order to reduce the complexity. In section VII we present our experimental work and in section VIII we discuss the ethical aspect of our research. In section IX we provide a discussion of the results and in section X we present the conclusion and the questions that remain unanswered.

III. Related Work

In this section we present in detail the necessary and sufficient conditions identified that are required to solve the reliable communication problem, the state-of-art protocols that approach the reliable communication or the topology reconstruction problems and the mathematical basis of some assumptions.

The reliable communication problem

Techniques for reconstructing the topology require any-to-any reliable communication, meaning that any correct process is able to communicate reliably with any other correct process present in the network. An essential condition formulated by Dolev is that the any-to-any RC can be achieved only when the process connectivity is strictly greater than $2 * f$, the maximum number of assumed faults, and when f is strictly smaller than a third of the total number of processes.

Dolev protocol

Dolev protocol assumes a partially connected network and uses a flooding algorithm for broadcasting a message. The message contains the id's of the sender and the target, the content and the path. Specifically, a process p_i sens a message m to a target process p_j by broadcasting m to all of its neighbors. Every correct process that receives m will append the id of the process that forwarded m to the path and relay it to any other neighbor that is not already in m's path. Eventually p_j will receive m through $f+1$ disjoint paths and deliver m. The complexity of this protocol is factorial in the size of the network.

Modified Dolev protocol

Modified Dolev protocol or BFT presents several improvements made that significantly reduce the number of messages exchanged between the processes, enabling reliable communication in larger systems [7].

Explorer

Explorer protocol supports a static asynchronous network with unicast links and a known neighborhood. In this protocol, each process p_i broadcasts a message that contains its neighborhood $\Gamma(i)$ and when a process p_j delivers this message via a reliable communication instance, it will save $\Gamma(i)$ in a dictionary data structure $cTop_j := \bigcup < i, \Gamma(i) >$. However, this protocol fails to guarantee the liveness property, which will be explained in the next section.

Explorer2

Explorer2 protocol represents an improved version that can also handle the weaker scenario in which the neighborhood is unknown. Furthermore, it introduces a failure detector that can correct the topology reconstructed under certain assumptions. It has a complexity of O(n^2) reliable communication instances [6]. The complete analysis of Explorer2 can be found in *Background*.

Graphs and disjoint paths

Definition. The local node connectivity $k_{i,j}$ between two nodes v_i, $v_j \in V$ is the minimum number of nodes that have to be removed from G to disconnect v_i from v_j [6].

Definition. A disjoint paths solution $\Pi_{i,j}$ between two nodes v_i, v_j is a collection of node disjoint paths between v_i and v_j [6].

Definition. The node connectivity of a graph is the minimum value k for the local node connectivity $k_{i,j}$ [6].

Definition. Many paths between two nodes are node disjoint (or simply disjoint) if they share no vertex except for their endpoints [6].

Theorem [Menger's theorem [8]]. The local node connectivity between two nodes is equal to the maximum number of node disjoint paths that exist between them.

IV. System Model and Problem Statement

This section explains the motivation of our research by first introducing the relevant background information and then the questions answered in this study. Furthermore, an overall description of the system model is provided.

A. Background

A solution to the Reliable Communication problem needs to meet the following properties, where p_i and p_j are both correct processes in an arbitrary network:

- "safety: if p_i delivers a message m received from p_j, then p_j sent m previously.
- liveness: if p_i sends a message m to p_j, eventually p_j delivers m" [6].

The safety property guarantees the authenticity of a message as no malicious process should be able to impersonate a correct process. In the state-of-art protocols that are mentioned in this study, it is accomplished in the following way: when a process receives a message m, it will append to m's path the id of the process that forwarded m and not its' own id. The liveness property refers to fact that each correct process must be able to send a message m to any other correct process present in the network. This means that there must be at least $2 * f + 1$ disjoint paths available between every two processes, as there can be at most f disjoint paths that contain one or more faulty processes and the target process needs to receive the message from at least $f + 1$ disjoint paths in order to deliver m. Following Menger's Theorem, it results that the local connectivity between two arbitrary nodes must be at least $2 * f + 1$, or in other words, a k-connected network, where $k > 2 * f$.

In Dolev's protocol for unknown networks, each process broadcasts a message of the format <sourceId,*, content,∅> to its neighbors $p_j \in \Gamma$(i). When receiving a message m from a neighbor p_a, a process relays
<sourceId,*, content, path∪p_a> to all $p_b \in \Gamma$(i) and $p_b \notin$ path and delivers a message m if it received m from $f+1$ disjoint paths.

Practical Reliable Broadcast Protocol (BFT) is a modified version of the Dolev protocol that may reduce significantly the message complexity whilst achieving reliable communication under certain system models [6]. It is important to note that in BFT, a process p_t is said to deliver a message m from a process p_s if there is a direct link between them or if p_t received m from at least $f + 1$ disjoint paths.

Explorer2 protocol is defined by employing a BFT primitive and a failure detector that allows the processes to eventually output the same topology. In this protocol, each process broadcasts its neighborhood Γ(i) and stores any Γ(j) delivered via a BFT instance in a dictionary data structure $cTop_i$. According to Farina [6], the topology $G_i(V_i,E_i)$ is reconstructed by each process as follows:

- if $< v, \Gamma(v) > \in cTop_i$, then v is inserted in V_i.
- if v is declared as neighbor by at least $f+1$ nodes and $u_0, u_1, ..., u_f \in cTop_i$, then v is inserted in V_i.
- if $v, u \in V_i$ and one declares the other as neighbor, then $(v, u) \in E_i$.

As a process can only know that its neighborhood is at least $f + 1$, each time the neighborhood grows, it will broadcast it again. Thus, processes need a failure detector to identify and discard incorrect messages: if a process p_i delivers $\Gamma(j)$ and $\Gamma(j)'$ from p_j such that $\Gamma(j)' \notin \Gamma(j)$ and $\Gamma(j) \in cTop_i$, then exclude process p_j from the reconstruction [6].

B. Problem Statement

The overall complexity is O(n^2) reliable communication instances for the weakest model [6]. We aim to reduce this complexity by reducing the number of processes that begin the broadcast procedure. We will assume that each process knows whether it was assigned as broadcaster or not. Furthermore, we introduce new modifications to the Explorer2 protocol in order to guarantee the properties of the reliable communication. Lastly, we show how to compute an optimal routing. Decreasing the complexity of Explorer2 and generating a routing table could prove to be extremely beneficial, because routing protocols that have a low complexity can be used afterwards to enable reliable communication in larger systems.

$\boldsymbol{Remark}$ Note that the worst-case complexity for a single reliable communication instance is factorial in the size of the network.

C. System Model

In this subsection details on the system model and all assumptions made are provided. Specifically, our system model consists of p_1, p_2, p_3 ... p_n independent processes or nodes that have physical connections between them. These connections can are unicast links, which are simple connections between two different processes, or broadcast links in which a process p_i is attached to several others p_a,p_b.... Note that in the case of unicast links the complexity is implicitly higher and therefore our study focuses only on this type of link.

The processes or nodes of a network G can either be correct or faulty, while the edges or links between two processes can be divided into: (i) correct, (ii) one-faulty, the edge interconnects a correct node and a faulty one, and (iii) two-faulty, the edge interconnects two faulty nodes [6].

We select a global failure model, where there are at most $f <$ [N / 3] faulty processes, N being the number of processes in the network. The network is k-connected where $k > 2 * f$, a requirement needed in order to achieve reliable communication from any process to any other process [9]. To outline everything, we consider a static asynchronous communication network with unicast links and a global failure model. In a static network no new processes appear over time, communication links are always available and bidirectional and by asynchronous we refer to the each process's capability of executing local computations and message exchanges in an unpredictable amount of time.

The knowledge of each process is the number f and whether it is a broadcaster. Cryptographic solutions or any other way of validating the authenticity of a message are not considered for this study.

V. Approach

This section provides technical details on the methodology used, data collected and the metrics used for analysing the data.

A. Methodology

The test cases, graph generation and the methods for data processing were implemented in Python, while in C++ we implemented the BFT reliable communication and Explorer2 protocols as this language offers a better performance compared to the former and we made use of Salticidae, a minimal asynchronous library, to connect the processes. In order to obtain relevant data regarding network throughput and latency, the processes were simulated separately using Docker containers. The Docker image is Ubuntu version 22.10. For the experiments, we used a Intel I7-9750H machine with 16 GB of RAM.

For each different run, the network has three editable parameters namely, the total number of nodes, the number of faulty nodes and the connectivity. The number of faulty nodes is restricted to be less than $[k/3]$, where k is the connectivity. According to these parameters, a random graph is generated which can either be general k-connected or multipartite wheel. This data was gathered and analysed so that we could find out whether the improved version of Explorer2 is still reliable.

B. Collecting data

The processes were designed to output in several files the topology discovered, the time for each delivery for computing the network latency, the number of BFT delivers for computing the network throughput. The topology discovered by each process was compared to the topology of the network, excluding the edges that contain one or two faulty processes.

C. Validation and metrics

Before measuring the performance of the modified Explorer2, it was necessary to check whether the functionality behaves as expected. Empirical analysis was applied for this topic by making each process output the topology reconstructed in an individual file. After every run, the results were compared with the actual topology of the network, with and without the faulty edges. Moreover, it was verified that each process has at least f + 1 connections in order to assert the properties of the reliable communication.

For measuring the possible improvements in the performance, we ran the same network topology and configuration in parallel on the unmodified implementation of Explorer2. The comparisons made will focus mainly on the message and delivery complexity and communication latency where relevant. Time complexity will not be considered since our system model is based on an asynchronous behaviour. We define message complexity as the total number of messages exchanged by the processes in a network and the delivery complexity as the local computational cost required by a single process to validate a content.

Network throughput is computed by averaging $countDelivers/(maxTime - minTime)$ values of each process, where $maxTime$ and $minTime$ represent the last timestamp a message was delivered and the first timestamp a message was delivered respectively.

Remark Communication latency plays a secondary role in our evaluation due to our experimental setup. The processes run in independent Docker containers, yet the channels used for communication are still on the same machine and thus, they have a low and similar latency if not the same and elements like jitter are hard to simulate.

VI. Improving Explorer2

In this section we present the problems encountered when trying to reduce the number of broadcasts, the observations made and lastly, how to compute the routing paths.

A. Explorer2

We implemented the Explorer2 protocol according to the pseudocode found in Algorithm 1 and the code is available in [10]. The topology reconstruction process is triggered every time a different message is dolev delivered (line 10). The neighborhood Γ received is added to $cTop$ if the size of the respective entry is 0 (line 20), or otherwise it is goes through the failure detector step (lines 30-38) and if necessary the topology is updated accordingly. The $counter$ array is incremented (lines 42-43) on the respective indices of the elements present in the received neighborhood. For each element p in the neighborhood, if it was already declared by at least $f + 1$ processes as neighbor then the edge $(p, broadcasterId)$ is added to $edges$ (lines 50-52), otherwise $broadcasterId$ is pushed in $edge\_valid[p]$ (lines 58-59). In the case in which p was just declared by $f + 1$ processes (lines 53-57), for each element b in $edge\_valid[m]$, the edge (b, p) is added to $edges$ (lines 50-52, 60-67).

This protocol enables any correct process to partially reconstruct the topology and eventually all correct processes will share the same topology. However, ”the sets of assumptions (i) $k \geq 2 * f$, (ii) unicast links, (iii) unknown neighborhood, (iv) no two-faulty edges in G or (a) $k \geq 2*f$, (b) known neighborhood are not sufficient to enable every correct process to to compute a disjoint path solution towards every other correct process” [6].

Explorer2 guarantees the properties of RC when:

- ”$k > 3 * f$
- $k > 2 * f + [f/2]$ with local broadcast links
- $k \geq 2 * f + [f/2]$ with known neighborhood” [6]

Algorithm 1 *Explorer2*

```
 1: Global Parameters:
 2:         $cTop$: matrix containing $\Gamma$ of broadcasters
 3:         $edges$: vector containing edges in final topology
 4:         $edge\_valid$: vector containing edges not validated yet
 5:         $counter$: counter for the number of times each process
    was declared as neighbor by some other process
 6:         $alreadyProcessed$: bool vector keeping track of ver-
    tices accepted.
 7:         $vertices$: vector containing nodes in final topology
 8: upon event $\langle Dolev, \texttt{Deliver} \rangle$ do
 9:         $saveNeighborhood(broadcasterId, msg.neighborhood)$
10: Function saveNeighborhood(bId: int, neighbors: vector)
11:     if not saveToStructure() then
12:         checkBuild()
13:     end if
14:
15: Function saveToStructure(bId: int, neighbors: vector)
16:     if cTop[bId].size() = 0 then
17:         cTop[bId] := neighbors
18:         return false
19:     else if cTop[bId].size() < neighbors.size() then
20:         detectChange(bId, neighbors, false)
21:         return true
22:     else
23:         detectChange(bId, neighbors, true)
24:         return true
25:     end if
26:
27: Function detectChange(bId: int, neighbors: vector,
    larger: bool)
28:     if larger and cTop[bId] not in neighbors then
29:         return
30:     else if not larger and cTop[bId] not in neighbors
    then
31:         return
32:     end if
33:     arr := neighbors not in cTop[bId]
34:     for it in arr do
35:         counter[it] := counter[it] + 1
36:     end for
37:     buildTopology(bId)
38:
39: Function checkBuild(bId: int)
40:     if bId not in vertices then
41:         vertices.add(bId)
42:     end if
43:     counter[bId] := counter[bId] + numFaulty + 1
44:     for it in cTop[bId] do
45:         counter[it] := counter[it] + 1
46:     end for
47:     buildTopology(bId)
48:
49: Function buildTopology(bId: int)
50:     for i in cTop[bId] do
51:         if counter[i] $\geq$ numFaulty + 1 then
52:             if alreadyProcessed[i] and (i, bId) not in
    edges then
53:                 edges.add((i, bId))
54:                 continue
55:             else
56:                 if i not in vertices then
57:                     vertices.add(bId)
58:                 end if
59:                 edge_valid[i].push_back(bId)
60:                 addEdges(i)
61:                 alreadyProcessed[i] := True
62:             end if
63:         else if counter[i] $\leq$ numFaulty then
64:             edge_valid[i].push_back(bId)
65:         end if
66:     end for
67:
68: Function addEdges(node: int)
69:     for e in edge_valid[node] do
70:         if e not in edges then
71:             edges.add((i,j))
72:             cnt := cnt + 1
73:         end if
74:     end for
75:
```

B. Reducing the number of broadcasters

We assume that the nodes that are declared as broadcasters are always correct and $k > 3 * f$. Let us denote from now on the set of broadcasters as β. The solution would be to use the unmodified version of Explorer2 with a reduced number of broadcasters, but we must be able to guarantee the liveness property. Take the network presented in Figure 1 for instance, where all the nodes that are not neighbors of 14 will not be able to reach this node using a reliable communication primitive. Since $k = 8$ it results that $f = 2$ and $\Pi_{p_i,14} = 4 < 2 * f + 1$, $\forall p_i \notin \Gamma(14)$. The disjoint path solution is four, as $|\beta \cup \Gamma(14)| = 4$ and the rest of the links node 14 has will remain undiscovered.

Taking this into account, our initial assumption that $2 * f + 1$ broadcasters would be enough is not realistic and an in-depth analysis is required in order to find the number of broadcasters needed to guarantee the liveness property. However, the range we need to look into is significantly reduced, as it is shown that at least 2/3 of the processes need to broadcast initially.

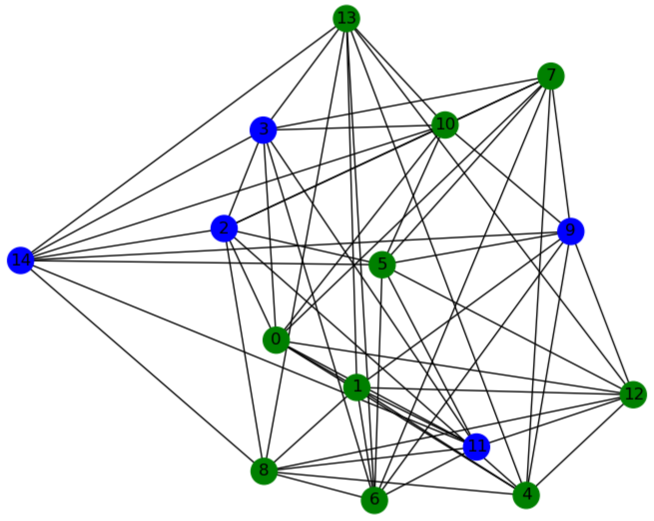

Fig. 1: A k-connected network of 15 nodes where k = 8. The green nodes represent the broadcasters.

C. Getting back the paths lost

The main problem is the low number of edges belonging to one or more processes that are discovered. As was described in the section above, when a process p_i is not selected as a broadcaster, other processes will be able to discover only a part of $\Gamma(i)$. Precisely, the other processes in the network will discover an edge $e(e_i, e_k) \in \Gamma(i)$ only if $p_k \in \beta$. This means that in order for a process p_k to compute $\Pi_{k,i}$, $p_k \in G$, either $p_i \in \beta$, $p_k \in \Gamma(i)$ or $|\Gamma(i) \cap \beta| \geq 2 * f + 1$. The latter condition can be used to employ a new modification with ease, as each process knows f and the number of neighbors that are broadcasters is equal to the number direct dolev delivers.

Using this observation, we could enable a smaller number of processes to broadcast initially and still guarantee the liveness property. A downside might be that more processes than the optimal number will end up broadcasting, although in an asynchronous communication it can prove to be highly unlikely as this translates to the case where at least two processes that are neighbors and are not in β will finish delivering all the messages broadcasted at exactly the same time.

D. Routing

After the broadcast rounds are over, each process needs to compute the disjoint paths solution Π to every other process present in the network G for routing future messages. We enabled the processes to generate the routing table using the Ford-Fulkerson algorithm.

A process p_i will assign its id as the source and the id of a process p_j as the sink and call the max-flow function to get $\Pi_{i,j}$. Each edge $e(e_1, e_2)$ reconstructed and its inverse $e'(e_2, e_1)$ are added with a capacity of 1 in the graph used for the max-flow algorithm. This procedure is done for every process $p_j \in G$ resulting in a total complexity of O(E*N), where E is the number of edges and N is the number of processes. Ford-Fulkerson was selected as an alternative for other max-flow algorithms in an effort to achieve optimal routing, as the disjoint paths are retrieved in ascending order (from the shortest to the longest) since the algorithm is based on a breadth-first approach and consequently, no additional sorting will be needed. This procedure can be found in Algorithm 2, while maximizeFlow function and the objects used can be found in Appendix A. Data structure $routingTable$ will contain the final Π.

Algorithm 2 Routing

```
1: Parameters:
2: $routingTable$ : map<int, vector<vector<int>>
3:
4: vector<Node> connections
5: for node in vertices do
6:     connections[node] := Node(node)
7: end for
8:
9: for it in edges do
10:     connections[it.first].addEdge(it.second)
11: end for
12:
13: for node in vertices do
14:     Graph g(connections, myid, node)
15:     routingTable.add(maximizeFlow(g, myid, node))
16: end for
```

VII. Evaluation

A. Experimental work

The focus of the experimental work was to find the lower limit of broadcasters that would guarantee the RC properties for every process in the network. Due to the heavy load of a network with a high connectivity and a significant number of nodes running locally, the Salticidae library is prone to connection errors at runtime. Having nodes crashing during execution is undesirable, as it could tamper the data and lead to erroneous results and for this reason, we decided to set the connectivity to 10 for all experiments and the total number of processes in the range of [15, 32]. For each test run, a new k-connected graph and β set are generated at random. Using scripts made in Python, the topologies outputted by each process were compared to the initial topology in order to identify the potential missing edges. The results can be seen in Figure 2. The highest percentage of nodes that broadcast that has an invalid outcome is 81.25%.

Another subject of interest regarding these results is the percentage of processes that are not able to generate the Π (referred to as invalid nodes) when the number of broadcasters is not big enough. The percentage is quite high, as it can be seen in Figure 3. The reason for this was explained in *Reducing the number of broadcasters*.

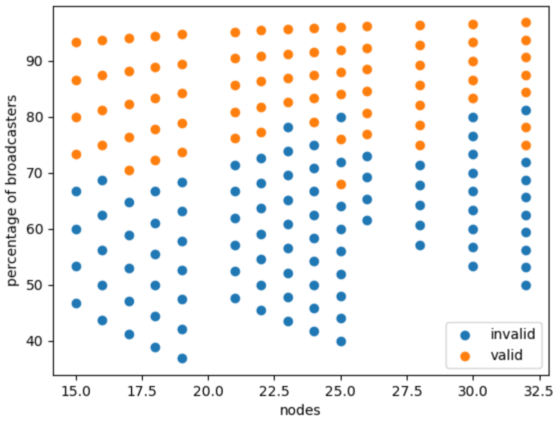

Fig. 2: Outcome of our main experiment. Valid refers to the case where all processes are able to generate the Π to every other process in the network.

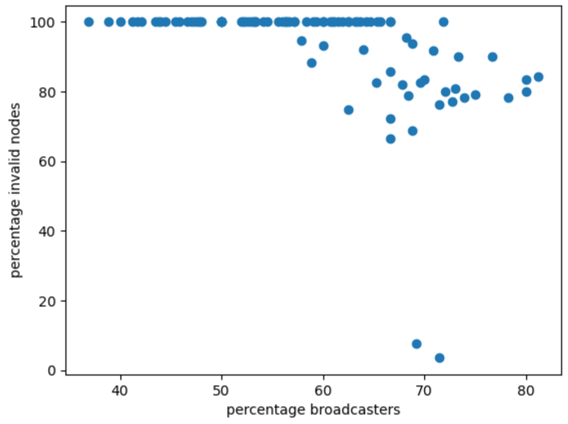

Fig. 3: Percentage of invalid nodes matched against the number of nodes that broadcasted.

A more interesting distribution of the data can be found in Figure 4, where the averaged number of undiscovered nodes for each node is compared to the percentage of nodes that broadcasted. A node p_j is undiscovered for a node p_i, if p_i is not able to compute a $|\Pi_{i,j}| > 2 * f$. This offers a much better perspective on how the protocol fares with different numbers of broadcasters. It is worth mentioning that usually the undiscovered nodes are the same for each node during an execution.

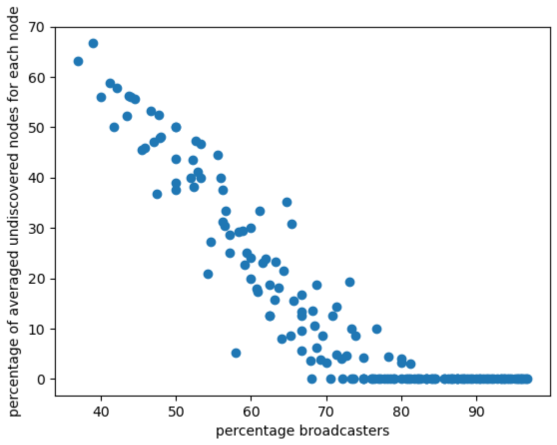

Fig. 4: Percentage of averaged undiscovered nodes for each node matched against percentage of nodes that broadcasted.

B. Impact of improvements

The results of the metrics for performance analysis can be found in Table 1. For the k-connected and multipartite wheel graphs we kept the configuration across the experiments while interchanging only the set β and connectivity was 10. On average, the reduction of 20% in the number of broadcasters seems to be reflected on the values of the metrics, as the difference in the number of messages exchanged is in the first case 21.71% and in the second 18.38%. The values of throughput in the second case should not be considered, as the data was clearly affected by the machine's computational power. The latency is also improved, both cases having a similar rate of improvement.

k-connected with 20 nodes			
Percentage of nodes	No. of messages	Throughput (delivers/s)	Latency (ms)
80%	5499	43	233.5
	5939	47	229.3
	6929	37	326.2
100%	8962	31	354.0
	8286	37	297.3
	6210	44	281.9
Improvement	21.71%	11.82%	15.44%

multipartite wheel graph with 30 nodes			
Percentage of nodes	No. of messages	Throughput (delivers/s)	Latency (ms)
80%	23188	3	9263
	24575	2	9945
	27338	3	10690
100%	31005	1	12250
	32448	2	12740
	28460	2	11667
Improvement	18.38%		18.44%

VIII. Responsible Research

Explorer2 was reimplemented to the best of our abilities and according to the indications provided in the literature and we provided the explicit pseudocode to ensure the reproducibility of our experiments. Furthermore, the results are expected to be similar, regardless of the choice of the topology configuration or the set β. Experiments in which one or more nodes crashed at runtime were discarded and with them any chances of using tampered data. We presented an explicit and complete pseudocode for the routing process and description of the methods and metrics used. All experiments are reproducible and we mentioned the values used for each parameter.

IX. Discussion

The averaged number of broadcasters required to enable any-to-any RC in a network is higher than expected, being around 80% out of the total number of processes. However, the modification detailed in *Getting back the paths lost* could reduce the number of broadcasters even more, but most importantly it guarantees the properties of RC. The metrics used indicate that the performance is better, especially when it comes to the number of messages the processes exchanged.

Regarding the results mapped in Figure 3, it is likely that the percentage of these invalid processes in the range of 60% to 80% broadcasters will drop if the connectivity increases. Moreover, we did not take into account the fact that if a process $p_j \in \Gamma(i)$ and $|\Pi_{i,j}| < 2 * f + 1$, p_i, then it is not necessary for p_i to know $2 * f + 1$ disjoint paths towards p_j in order to achieve reliable communication as they have a direct link. Thus, the percentage of invalid processes might be smaller.

X. Conclusions and Future Work

In this paper we described the pseudocode of the Explorer2 protocol, shown that it is possible to reduce the number of broadcasters required down to 81.25% of the total number of process and provided a new modification meant to ensure that the RC properties are preserved. The modification employed might decrease significantly the number of broadcasters needed, but more experiments with the aim of finding an optimal number of broadcasters are necessary since it is difficult to predict its behaviour in an asynchronous network. Moreover, we provided a pseudocode for generating the disjoint paths in an efficient way with a complexity of O($N * E$).

The outcome was not what we expected initially, but while the delivery complexity remains rather unchanged, decreasing the message complexity by 20% is still a significant improvement, given that the worst-case scenario is factorial. However, this work could be extended in the future by designing a new way to reconstruct the topology.

Appendix A

Objects pseudocode

```
class Graph
    Parameters:
    nodes : vector<Node>
    source : Node
    sink : Node

    Function Graph(nodes: vector<Node>, source: int, sink: int)
        this.nodes = nodes
        this.source = this.nodes[sourceId]
        this.sink = this.nodes[sinkId]

end class

class Node
    Parameters:
    id : int
    edges : vector<Node>

    Function Node(id: int)
        this.id := id

    Function addEdge(to: Node, lower: int, capacity: int)
        Edge e := Edge(lower, upper, this.id, to.id)
        this.edges.push_back(e)
        to.edges.push_back(e.getBackwards())

end class

class Edge
    Parameters:
    to : int
    from : int
    flow : int
    capacity : int
    lower : int
    backwards : Edge

    Function augmentFlow(add: int)
        flow := flow + add
        backwards.flow := getResidual()

    Function Edge(lower: int, capacity: int, from: int, to: int)
        this.lower := lower
        this.capacity := capacity
        this.from := from
        this.to := to
        this.flow := 0
        this.backwards := Edge(this)


    Function Edge(e: Edge)
        this.lower := 0
        this.flow := e.capacity
        this.capacity := e.capacity
        this.from := e.to
        this.to := e.from
        this.backwards := e

    Function getBackwards()
        return backwards

    Function getResidual()
        return capacity - flow

```

Function maximizeFlow

```
Function maximizeFlow(g: Graph, source: int, sink: int)
    vector<Edge> path
    vector<vector<Edge>> solution
    while not (path := findPath(g, source, sink).empty() do
        for e dodge in path
            r := min(r, e.capacity - e.flow)
        end for
        for edge in path do
            g.nodes[e.from].edges[e].augmentFlow(r)
        end for
        f := f + r
        solution.push_back(path)
    end while

Function findPath(g: Graph, start: Node, end: Node
    map<Node, Edge> mapPath
    queue<Node> sQueue
    Node currentNode := start
    sQueue.push(currentNode)
    while not sQueue.empty() and currentNode != end do
        currentNode = sQueue.pop()
        for edge in currentNode.edges do
            if edge.to != start.id and edge.to not in pathMap and edge.capacity > edge.flow then
                sQUeue.push(edge.to)
                mapPath.insert( (edge.to, edge) )
            end if
        end for
    end while
    vector<Edge> path
    if sQueue.empty() and currentNode = end then
        return path
    end if
    Node current = end
    while mapPath.find(current) do
        path.insert(mapPath[current])
        current = mapPath[current].from
    end while
    return path
=0
```

References

[1] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *Concurrency: the Works of Leslie Lamport*, pages 203–226. 2019.

[2] Dolev D. Unanimity in an unknown and unreliable environment. *IEEE Computer Society*, 22nd Annual Symposium on Foundations of Computer Science (sfcs 1981):159–168, October 1981.

[3] Silvia Bonomi, Giovanni Farina, and Sébastien Tixeuil. Multi-hop byzantine reliable broadcast with honest dealer made practical, Sep 2019.

[4] Danny Dolev, Joseph Y. Halpern, Barbara Simons, and H.Raymond Strong. A new look at fault-tolerant network routing, Nov 2004.

[5] Nesterenko M. and Tixeuil S. Discovering network topology in the presence of byzantine faults. In *Structural Information and Communication Complexity*, pages 212–226, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[6] Giovanni Farina. *Tractable Reliable Communication in Compromised Networks*. PhD thesis, Sorbonne Université; Sapienza Università di Roma (Italie), 2020.

[7] Silvia Bonomi, Giovanni Farina, and Sébastien Tixeuil. Multi-hop byzantine reliable broadcast with honest dealer made practical. *Journal of the Brazilian Computer Society*, 25(1):1–23, 2019.

[8] Karl Menger. Zur allgemeinen kurventheorie. *Fund. Math.*, 10:96–1159, 1927.

[9] Danny Dolev. The byzantine generals strike again. *Journal of algorithms*, 3(1):14–30, 1982.

[10] Explorer2 code implementation and python scripts for processing the metrics. https://gitlab.tudelft.nl/cse3000-2022-reliable-communications/silviu-discover-network-topology, June 2022.