Discovering Network Topology in the Presence of Byzantine Faults

Mikhail Nesterenko and Sébastien Tixeuil

Abstract—We pose and study the problem of Byzantine-robust topology discovery in an arbitrary asynchronous network. The problem is an abstraction of fault-tolerant routing. We formally state the weak and strong versions of the problem. The weak version requires that either each node discovers the topology of the network or at least one node detects the presence of a faulty node. The strong version requires that each node discovers the topology regardless of faults. We focus on non-cryptographic solutions to these problems. We explore their bounds. We prove that the weak topology discovery problem is solvable only if the connectivity of the network exceeds the number of faults in the system. Similarly, we show that the strong version of the problem is solvable only if the network connectivity is more than twice the number of faults. We present solutions to both versions of the problem. The presented algorithms match the established graph connectivity bounds. The algorithms do not require the individual nodes to know either the diameter or the size of the network. The message complexity of both programs is low polynomial with respect to the network size. We describe how our solutions can be extended to add the property of termination, handle topology changes and perform neighborhood discovery.

Index Terms—Fault tolerance, topology discovery, Byzantine faults.

1 Introduction

In this paper, we investigate the problem of Byzantine-tolerant distributed topology discovery in an arbitrary network. Each node is only aware of its neighboring peers and it needs to learn the topology of the entire network.

As reliability demands on distributed systems increase, the interest in developing robust distributed applications grows. One of the strongest fault models is *Byzantine* [12]: the faulty node behaves arbitrarily. This model encompasses a rich set of fault scenarios. Moreover, Byzantine fault tolerance has security implications, as the behavior of an intruder can be modeled as Byzantine.

However, in most studies to date, Byzantine faults are considered in completely connected networks. That is, the routing between the nodes in the system is assumed to be established in spite of the faults. In this paper we address the problem of Byzantine-robust routing. To allow us to abstract from the details of routing table establishment and maintenance, we state the problem of the topology discovery and study its properties and solutions to it. Besides being attractive in its own right, the solutions to the topology discovery problem easily translate into routing algorithms.

- *M. Nesterenko is with the Computer Science Department, Kent State University, Kent, OH, 44240, USA, he was supported in part by DARPA contract OSU-RF#F33615-01-C-1901 and by NSF CAREER Award 0347485. Part of this work was done while this author was visiting Paris Sud University.*

- *S. Tixeuil is with Université Pierre & Marie Curie - Paris 6, France. He was supported in part by the ANR grant SOGEA from "Sécurité, Systèmes Embarqués et Intelligence Ambiante" program, and ANR grant SHAMAN from VERSO program. Part of this work was done while the author was visiting Kent State University.*

- *Some of the results in this article were presented at the 13th Colloquium on Structural Information and Communication Complexity, Chester, UK in July 2006*

One approach to deal with Byzantine faults is by enabling the nodes to use cryptographic operations such as digital signatures or certificates. This limits the power of a Byzantine node as a non-faulty node can verify the validity of received topology information and authenticate the sender across multiple hops. However, this option may not be available. For example, the nodes may not have enough resources to manipulate digital signatures. Moreover, cryptographic operations implicitly assume the presence of trust infrastructure: secure channels to a key server or a public key infrastructure. Establishing and maintaining such infrastructure in the presence of Byzantine faults may be problematic.

Another way to limit the power of a Byzantine process is to assume synchrony: all processes proceed in lock-step. Indeed, if a process is required to send a message with each pulse, a Byzantine process cannot refuse to send a message without being detected. However, the synchrony assumption may be too restrictive for practical systems.

Our contribution. In this study we explore the fundamental properties of topology discovery. We select the weakest practical programming model, establish the limits on the solutions and present the programs matching those limits.

Specifically, we consider networks of arbitrary topology where up to fixed number of nodes k are faulty. The execution model is asynchronous. We are interested in solutions that do not use cryptographic primitives. The solutions should be terminating and the individual processes should not be aware of the network parameters such as network diameter or its total number of nodes.

We state two variants of the topology discovery problem: *weak* and *strong*. In the former — either each non-faulty node learns the topology of the network or one of them detects a fault; in the latter — each non-faulty node has to learn the topology of the network regardless of the presence of faults.

As negative results we show that any solution to the weak topology discovery problem can not ascertain the presence of an edge between two faulty nodes. Similarly, any solution to the strong variant can not determine the presence of an edge between a pair of nodes at least one of which is faulty. Moreover, the solution to the weak variant requires the network to be at least $(k+1)$-connected. In case of the strong variant the network must be at least $(2k+1)$-connected.

The main contribution of this study are the algorithms that solve the two problems: *Detector* and *Explorer*. The algorithms match the respective connectivity lower bounds. To the best of our knowledge, these are the first asynchronous Byzantine-robust solutions to the topology discovery problem that do not use cryptographic operations. In practice these bounds can be viewed as working tolerances of the algorithms. That is, the algorithms are specified to operate correctly until the number of faults exceeds the theoretical tolerance limit.

Explorer solves the stronger problem. However, *Detector* has better message complexity. In both algorithms, faulty nodes may generate arbitrary number of messages. However, in fault-free operation, *Detector* either determines in $O(\delta n^3)$ messages where δ and n are the maximum neighborhood size and the number of nodes in the system respectively. *Explorer* finishes in $O(n^4)$ messages.

Detector is not as robust as *Explorer*. *Detector* just alerts the system about the presence of the fault instead of masking it. The system has to have an external mechanism to cope with faults. For example, the system operators may have to manually take measures to neutralize the faulty node. However, *Detector* requires weaker connectivity and it is asymptotically more efficient. Thus, the system designers may consider this variant of the solution in case the faults are rare or the system cannot guarantee the necessary connectivity requirements for a more robust solution.

We extend our algorithms to *(a)* terminate *(b)* handle topology changes *(c)* discover neighbors if ports are known *(d)* discover a fixed number of routes instead of complete topology and *(e)* reliably propagate arbitrary information instead of topological data.

Related work. A number of researchers employ cryptographic operations to counter Byzantine faults. Avromopolus et al [2] consider the problem of secure routing. Therein see the references to other secure routing solutions that rely on cryptography. Perrig et al [21] survey robust routing methods in ad hoc sensor networks. The techniques covered there also assume that the processes are capable of cryptographic operations.

A naïve approach of solving the topology discovery problem without cryptography would be to use a Byzantine-resilient broadcast [3], [7], [10], [20]: each node advertises its neighborhood. However all existing solutions for arbitrary topology known to us require that the graph topology is *a priori* known to the nodes.

Let us survey the non-cryptography based approaches to Byzantine fault-tolerance. Most programs described in the literature [1], [13], [14], [18] assume completely connected networks and can not be easily extended to deal with arbitrary topology. Dolev [7] considers Byzantine agreement on arbitrary graphs. He states that for agreement in the presence of up to k Byzantine nodes, it is necessary and sufficient that the network is $(2k+1)$-connected and the number of nodes in the system is at least $3k+1$. However, his solution requires that the nodes are aware of the topology in advance. Also, this solution assumes the synchronous execution model. Recently, the problem of Byzantine-robust reliable broadcast has attracted attention [3], [10], [20]. However, in all cases the topology is assumed to be known. Bhandari and Vaidya [3] and Koo [10] assume two-dimensional grid. Pelc and Peleg [20] consider arbitrary topology but assume that each node knows the exact topology a priori. A notable class of algorithms tolerates Byzantine faults with either space [17], [19], [22] or time [16] locality. Yet, the emphasis of space local algorithms is on containing the fault as close to its source as possible. This is only applicable to the problems where the information from remote nodes is unimportant such as vertex coloring, link coloring or dining philosophers. Also, time local algorithms presented so far can hold at most one Byzantine node and are not able to mask the effect of Byzantine actions. Thus, local containment approach is not applicable to topology discovery.

Masuzawa [15] considers the problem of topology discovery and update. However, Masuzawa is interested in designing a self-stabilizing solution to the problem and thus his fault model is not as general as Byzantine: he considers only transient and crash faults.

Paper outline. The rest of the paper is organized as follows. After stating our programming model and notation in Section 2, we formulate the topology discovery problems, as well as state the impossibility results in Section 3. We present *Detector* and *Explorer* in Sections 4 and 5 respectively. We discuss the composition of our programs and their extensions in Section 6. In conclusion we state several problems that our research opens in Section 7.

2 Notation, Definitions and Assumptions

Graphs. A distributed *system* (or *program*) consists of a set of processes and a *neighbor* relation between them. This relation is the system *topology*. The topology forms a graph G. Denote n and e to be the number of nodes[1] and edges in G respectively. Two processes are *neighbors* if they share an edge in G. A set P of neighbors of process p is *neighborhood* of p. In the sequel we use small letters to denote singleton variables and capital letters to denote sets. In particular, we use a small letter for a process and a matching capital one for this process' neighborhood. Since the topology is symmetric, if $q \in P$ then $p \in Q$. Denote δ to be the maximum number of nodes in a neighborhood.

A *node-cut* of a graph is the set of nodes U such that $G \setminus U$ is disconnected or trivial. A *node-connectivity* (or just *connectivity*) of a graph is the minimum cardinality of a node-cut of this graph. In this paper we make use of the following

1. We use terms *process* and *node* interchangeably.

fact about graph connectivity that follows from Menger's theorem (see [24]): if a graph is k-connected (where k is some constant), then for every two vertices u and v there exists at least k internally node-disjoint paths connecting u and v in this graph.

Program model. A process contains a set of variables. When it is clear from the context, we refer to a variable var of process p as $var.p$. Every variable ranges over a fixed domain of values. For each variable, certain values are *initial*. Each process p, initially contains a set of neighbor identifiers P. Each pair of neighbor processes share a pair of special variables called *channels*. We denote $Ch.b.c$ the channel from process b to process c. Process b is the *sender* and c is the *receiver*. The value for a channel variable is chosen from the domain of (potentially infinite) sequences of messages. That is, each channel is a FIFO queue. Each process is capable of recognizing the sender of the message.

A *state* of the program is the assignment of a value to every variable of each process from its corresponding domain. A state is *initial* if every variable has initial value. Each process contains a set of actions. An action has the form $\langle name\rangle : \langle guard\rangle \longrightarrow \langle command\rangle$. A *guard* is a boolean predicate over the variables of the process. A *command* is sequence of assignment and branching statements. A guard may be a receive-statement that accesses the incoming channel. A command may contain a send-statement that modifies the outgoing channel. A parameter is used to define a set of actions as one parameterized action. For example, let j be a parameter ranging over values 2, 5 and 9; then a parameterized action $ac.j$ defines the set of actions

$$ac.(j = 2) \ [] \ ac.(j = 5) \ [] \ ac.(j = 9).$$

Either guard or command can contain quantified constructs [6] of the form: $(\langle quantifier\rangle\langle bound\ variables\rangle : \langle range\rangle : \langle term\rangle)$, where *range* and *term* are boolean constructs.

Semantics. An action of a process of the program is *enabled* in a certain state if its guard evaluates to **true**. An action containing receive-statement is enabled when appropriate message is at the head of the incoming channel. The execution of the command of an action updates variables of the process. The execution of an action containing receive-statement removes the received message from the head of the incoming channel and inserts the value the message contains into the specified variables. The execution of send-statement appends the specified message to the tail of the outgoing message.

A *computation* of the program is a maximal fair sequence of states of the program such that the first state s_0 is initial and for each state s_i the state s_{i+1} is obtained by executing the command of an action whose state is enabled in s_i. That is, we assume that the action execution is *atomic*. The maximality of a computation means that the computation is either infinite or it terminates in a state where none of the actions are enabled. The fairness means that if an action is enabled in all but finitely many states of an infinite computation then this action is executed infinitely often. That is, we assume *weak fairness* of action execution. Notice that we require the receive statement to appear as a standalone guard of an action. This means, that if a message of the appropriate type is at the head of the incoming channel, the receive action is enabled. Due to weak fairness assumption, this leads to *fair message receipt* assumption: each message in the channel is eventually received. Observe that our definition of a computation considers *asynchronous* computations.

To reason about program behavior we define boolean predicates on program states. A program *invariant* is a predicate that is **true** in every initial state of the program and if the predicate holds before the execution of the program action, it also holds afterwards. Notice that by this definition a program invariant holds in each state of every program computation.

Faults. Throughout a computation, a process may be either Byzantine (faulty) or non-faulty. A Byzantine process contains an action that assigns to each local variable an arbitrary value from its domain. This action is always enabled. Yet, the weak fairness assumption does not apply to this action. That is, we consider computations where a faulty process does not execute any actions. Observe that we allow a faulty node to send arbitrary messages. We assume, however, that messages sent by such a node conform to the format specified by the algorithm: each message carries the specified number of values, and the values are drawn from appropriate domains. This assumption is not difficult to implement as message syntax checking logic can be incorporated in receive-action of each process. We assume *oral record* [12] of message transmission: the receiver can always correctly identify the message sender. The channels are reliable: the messages are delivered in FIFO order and without loss or corruption. Throughout the paper we assume that the maximum number of faulty nodes in the system is bounded by some constant k.

Graph exploration. The processes discover the topology of the system by exchanging messages. Each message contains the identifier of the process and its neighborhood. Process p *explored* process q if p received a message with (q, Q). When it is clear from the context, we omit the mention of p. An *explored* subgraph of a graph contains only explored processes. A Byzantine process may potentially circulate information about the processes that do not exist in the system altogether. A process is *fake* if it does not exist in the system, a process is *real* otherwise.

3 The Topology Discovery Problem: Statement and Solution Bounds

Problem statement.

Definition 1 (Weak Topology Discovery Problem): A program is a solution to the weak topology discovery problem if each of the program's computation satisfies the following properties: termination *— either all non-faulty processes determine the system topology or at least one process detects a fault;* safety *— for each non-faulty process, the determined topology is a subset of the actual system topology;* validity *— the fault is detected only if there are faulty processes in the system.*

Definition 2 (Strong Topology Discovery Problem): A program is a solution to the strong topology discovery problem if each of the program's computations satisfies the following properties: termination — *all non-faulty processes determine the system topology;* safety — *the determined topology is a subset of the actual system topology.*

According to the safety property of both problem definitions each non-faulty process is only required to discover a subset of the actual system topology. However, the desired objective is for each node to discover as much of it as possible. The following definitions capture this idea. A solution to a topology discovery problem is *complete* if every non-faulty process always discovers the complete topology of the system. A solution to the problem is *node-complete* if every non-faulty process discovers all nodes of the system. A solution is *adjacent-edge complete* if every non-faulty node discovers each edge adjacent to at least one non-faulty node. A solution is *two-adjacent-edge complete* if every non-faulty node discovers each edge adjacent to two non-faulty nodes.

Observe that for $(k+1)$-connected graphs an adjacent-edge complete solution is also node complete and, therefore, simply complete. If $k = 1$, then an edge complete solution is also node complete and simply complete.

Solution bounds. To simplify the presentation of the negative results in this section we assume more restrictive execution semantics. Each channel contains at most one message. The computation is synchronous and proceeds in rounds. In a single round, each process consumes all messages in its incoming channels and outputs its own messages into the outgoing channels. Notice that the negative results established for this semantics apply for the more general semantics used in the rest of the paper.

Theorem 1: There does not exist a complete solution to the weak topology discovery problem if the number of faults k is greater than one.

Proof: Assume there exists a complete solution to the problem. Consider $k \geq 2$ and topology G_1 that is not completely connected. Let none of the nodes in G_1 be faulty. By the validity property, none of the nodes may detect a fault in such topology. Consider a computation s_1 of the solution program where each node discovers G_1. Let $p \in G_1$, $q \neq p$, and $r \neq p$ be three nodes in G_1, with q and r being non-neighbor nodes in G_1. Since G_1 is not completely connected we can always find two such nodes.

We form topology G_2 by connecting q and r in G_1. Let q and r be faulty in G_2. We construct a computation s_2 which is identical to s_1. That is, q and r, being faulty, in every round output the same messages as in s_1. Since s_2 is otherwise identical to s_1, process p determines that the topology of the system is $G_1 \neq G_2$. Thus, the assumed solution is not complete. □

Theorem 2: There does not exist a node- or adjacent-edge complete solution to the weak topology problem if the connectivity of the graph is lower or equal to the total number of faults k.

Proof: Assume the opposite. Let there be a node- and adjacent-edge complete program that solves the problem for graphs whose connectivity is k or less. Let G_1 and G_2 be two graphs of connectivity k.

This means that G_1 and G_2 contain the respective cut node sets A_1 and A_2 whose cardinality is k. Rename the processes in G_2 such that $A_1 = A_2$. By definition, A_1 separates G_1 into two disconnected sets B_1 and C_1. Similarly, A_2 separates G_2 into B_2 and C_2. Assume without loss of generality that $B_1 \neq B_2$. Specifically, there is a node q_1 such that $q_1 \in B_1$ but $q_1 \notin B_2$ and a pair of nodes q_2 and q_3 that share an edge in B_1 but not in B_2.

Since $A_1 = A_2$ we can form graph G_3 as $A_1 \cup B_2 \cup C_1$.

Let s_1 be any computation of the assumed program in the system of topology G_1 and no faulty nodes. Since the program solves the weak topology discovery problem, the computation has to comply with all the properties of the problem. By validity property, no fault is detected in s_1. By termination property, each node in G_1, including some node $p \in C_1$, eventually discovers the system topology.

By safety property the topology discovered by p is a subset of G_1. Since the solution is complete, the discovered topology is G_1 exactly. Let s_2 be any computation of the assumed program in the system of topology G_2 and no faulty nodes. Again, none of the nodes detects a fault and all of them discover the complete topology of G_2 in s_2.

We construct a new computation s_3 of the assumed program as follows. The system topology for s_3 is G_3 where all nodes in A_1 are faulty. Each faulty node $r \in A_1$ behaves as follows. In the channels connecting r to the nodes of $C_1 \subset G_3$, each round r outputs the messages as in s_1. Similarly, in the channels connecting r to the nodes of $B_2 \subset G_3$, r outputs the messages as in s_2. The non-faulty nodes of B_2 and C_1 behave as in s_1 and s_2 respectively.

Observe that for the nodes of B_2, the topology and communication is indistinguishable from that of s_2. Similarly, for the nodes of C_1 the topology and communication is indistinguishable from that of s_1. Notice that this means that none of the non-faulty nodes detect a fault in the system. Moreover, node $p \in C_1$ decides that the system topology is the subset of G_1. Yet, by construction, $G_1 \neq G_3$. Specifically, $B_1 \neq B_2$. None of the nodes in B_2 are faulty. If this is the case then, by the assumptions on B_1 and B_2, either s_3 violates the safety property of the problem or the assumed solution is neither adjacent-edge nor node-complete. The theorem follows. □

Theorem 3: There does not exist an adjacent-edge complete solution to the strong topology discovery problem.

Proof: Assume such a solution exists. Consider system graph G_1 that is not completely connected. Let $p \in G_1$ be an arbitrary node. Let $q \neq p$ and $r \neq p$ be two non-neighbor nodes of G_1. We form topology G_2 by connecting q and r in G_1.

We construct computations s_1 and s_2 as follows. Let s_1 and s_2 be executed on G_1 and G_2 respectively. And let q be faulty in s_1 and r be faulty in s_2. Set the output of q in each round to be identical in s_1 and s_2. Similarly, set the output of r to be identical in both computations as well. Since the output of q and r in both computations is identical, we construct the behavior of the rest of the nodes in s_1 and s_2 to be the same.

Due to termination property, p has to decide on the system topology in both computations. Due to the safety property, in s_1 process p has to determine that the topology of the graph is a subset of G_1. However, since the behavior of p in s_2 is identical to that in s_1, p decides that the topology of the system graph is G_1 in s_2 as well. This means p does not include the edge between q and r to the explored topology in s_2. Yet, one of the nodes adjacent to this edge, namely q, is not faulty. An adjacent-edge complete program should include such edges in the discovered topology. Therefore, the assumed program is not adjacent-edge complete. □

Theorem 4: There does not exist a node- or two-adjacent-edge complete solution to the strong topology problem if the connectivity of the graph is less than or equal to twice the total number of faults k.

Proof: Assume that there is a program that solves the problem for graphs whose connectivity is $2k$ or less. Let G_1 and G_2 be two different graphs whose connectivity is $2k$. Similar to the the proof of Theorem 2, we assume that $G_1 = A_1 \cup B_1 \cup C_1$ and $G_2 = A_2 \cup B_2 \cup C_2$ where the cardinality of A_1 and A_2 are $2k$, $A_1 = A_2$, $B_1 \cap C_1 = \varnothing$, $B_2 \cap C_2 = \varnothing$. Also $B_1 \neq B_2$. Specifically, there is a node q_1 such that $q_1 \in B_1$ but $q_1 \notin B_2$ and a pair of nodes q_2 and q_3 that share an edge in B_1 but not in B_2.

Form $G_3 = A_1 \cup B_2 \cup C_1$. Divide A_1 into two subsets A_1' and A_1'' of the same number of nodes. Construct a computation s_1 with system topology G_1 where all nodes in A_1' are faulty; and another computation s_3 with system topology G_3 where all nodes in A_1'' are faulty. The faulty nodes in s_1 in the channels connecting A_1' to C_1 communicate as the (non-faulty) nodes of A_1' in s_3. Similarly, the faulty nodes in s_3 in the channels connecting A_1'' to C_1 communicate as the nodes of A_1'' in s_1. Observe that s_1 and s_3 are indistinguishable to the nodes in C_1. Let the nodes in C_1, including $p \in C_1$ behave identically in both computations. According to the termination property of the strong topology discovery problem every node, including p, has to determine the system topology in both s_1 and s_3. Due to safety, the topology that p determines in s_1 is a subset of G_1. However, p behaves identically in s_3. This means that p decides that the system topology in s_3 is also a subset of G_1. Since $G_1 \neq G_3$ (specifically, $B_1 \not\subseteq B_2$) and none of the nodes in B_2 are faulty, then either s_3 violates the safety property of the problem or the assumed is neither two-adjacent-edge nor node complete. The theorem follows. □

4 DETECTOR

Outline. *Detector* solves the weak topology discovery problem for system graphs whose connectivity exceeds the number of faulty nodes k. The algorithm leverages the connectivity of the graph. For each pair of nodes, the graph guarantees the presence of at least one path that does not include a faulty node. The topology data travels along every path of the graph. Hence, the process that collects information about another process can find the potential inconsistency between the information that proceeds along the path containing faulty nodes and the path containing only non-faulty ones.

Care is taken to detect the fake nodes whose information is introduced by faulty processes. Since the processes do not know the size of the system, a faulty process may potentially introduce an infinite number of fake nodes. Graph connectivity is leveraged to detect such fake nodes. As faulty processes are the only source of information about fake nodes, all the paths from the real nodes to the fake ones have to contain a faulty node. Yet, the graph connectivity is assumed to be greater than k. If a fake node is ever introduced, one of the non-faulty processes eventually detects a graph with too few paths leading to the fake node.

To simplify the presentation, we describe *Detector* assuming that the system contains at most one faulty node ($k = 1$). We then explain how to generalize the algorithm to an arbitrary fixed k.

```
process p
const
    P: set of neighbor identifiers of p
    k = 1: integer, upper bound on the number of
        faulty processes
parameter
    q : P
var
    detect : boolean, initially false, signals fault
    start : boolean, initially true, controls sending of
            p's neighborhood info
    TOP : set of tuples, initially {(p, P)},
            (process ids, neighbor id set) received by p
    *[
init:   start ⟶
            start := false,
            (∀j : j ∈ P : send (p, P) to j)
     []
accept: receive (r, R) from q ⟶
            if (∃s, S : (s, S) ∈ TOP : s = r ∧ S ≠ R) ∨
                (connectivity(TOP ∪ {(r, R)}) < k + 1)
            then
                detect := true
            else
                if (∄s, S : (s, S) ∈ TOP : s = r) then
                    TOP := TOP ∪ {(r, R)},
                    (∀j : j ∈ P : send (r, R) to j)
    ]
```

Fig. 1. Process of *Detector*

Detailed description. The program is shown in Figure 1. Each process p stores the identifiers of its immediate neighbors. They are kept in set P. Each process is aware of the upper bound $k = 1$ on the number of faulty processes in the system. Process p maintains the following variables. Boolean variable $detect$ indicates if p discovers a fault in the system. Boolean variable *start* guards the execution of the action that sends p's neighborhood information to its neighbors. Set TOP (for topology) stores the subgraph explored by p; TOP contains tuples of the form: (*process identifier, its neighborhood*). In the initial state, TOP contains (p, P).

Function **connectivity** evaluates the topology of the subgraph stored in TOP. Recall that a node u is unexplored by p if for every tuple $(s, S) \in TOP$, s is not the same as u. That is u may appear in S only. We construct graph G' by adding an edge to every pair of unexplored processes present

in TOP. We calculate the value of **connectivity** as follows. If the information of TOP is inconsistent, that is:

$$(\exists u, v, U, V : ((u, U) \in TOP) \wedge ((v, V) \in TOP) : \\ (u \in V) \wedge (v \notin U))$$

then **connectivity** returns 0. Otherwise the function returns the connectivity of graph G'. In the proof of algorithm's correctness we demonstrate that the connectivity G' may fall below k only if there is a fault in the graph.

Processes exchange messages of the form (*process identifier, its neighborhood id set*). A process contains two actions: *init* and *accept*. Action *init* starts the propagation of p's neighborhood throughout the system. Action *accept* receives the neighborhood data of some process, records it, checks against other data already available for p and possibly further disseminates the data. If the data received from neighbor q about a process r contradicts what p already holds about r in TOP or if the newly arrived information implies that G is less than 2-connected p indicates that it detected a fault by setting $detect$ to **true**. That is, the connectivity of the resultant graph is less than $k+1$. Alternatively, if p did not previously have the information about r, p updates TOP and sends the received information to all its neighbors.

Observe that the propagation of information about the neighborhood of a certain process is independent of the information propagation of another process. Thus, we will focus on the propagation of the information about a particular non-faulty process a.

Let COR contain each process b such that b is not faulty and $TOP.b$ holds (a, A). Let a itself belong to COR if $start.a$ is **false**.

Lemma 1: The following predicate is an invariant of Detector.

$$(\forall \text{ non-faulty } b, c : b \in COR, c \in B : \\ (c \in COR) \vee \\ ((a, A) \in Ch.b.c)) \vee \\ (\exists \text{ non-faulty } j : j \in N : detect.j = \textbf{true}) \tag{1}$$

The predicate states that unless one of the non-faulty processes in the program detects a fault, if a process b belongs to COR then each neighbor c of b either belongs to COR as well or the channel from b to c contains (a, A).

Proof: To prove that Predicate 1 is an invariant of the program, we need to show that it holds in the initial state of any computation and it is closed under the execution of actions of Byzantine as well as non-faulty processes. The predicate holds initially as the first disjunct is vacuously true.

Note that no action of a Byzantine process immediately affects the validity of the predicate. Observe also that a non-faulty process can only set $detect$ to **true**. Thus, once this happens the predicate holds throughout the rest of the computation. Suppose $detect$ is **false** in all processes of the program. Then the predicate is violated only if there is a non-faulty pair of neighbors b and c such that b belongs to COR, c does not and there is no message (a, A) in the channel from b to c. Notice that a non-faulty process adds the first value (r, R) to TOP and never changes it afterwards. Thus, provided that $detect =$ **false**, to violate the predicate, a process has to join COR without sending (a, A) to its neighbors or consume a message with (a, A) without joining COR. Let us examine the actions of a non-faulty process and ensure that neither of this happens.

Observe that *init* is only of interest in a. This action sets $start.a =$ **false** which, by definition, adds a to COR. Also, *init* atomically sends (a, A) to all neighbors of a. Thus, the predicate is not violated by the execution of *init*.

Let us now consider *accept* in an arbitrary non-faulty process u. Let the message received by u carry (r, R). Observe that *accept* affects Predicate 1 only if $r = a$. *accept* may make u join COR or consume a message with (a, A). Notice, that if u is already in COR the receipt of a message with (a, A) does not violate the predicate. Also, u joins COR only if it receives (a, A). Hence, the only case we have to consider is when u does not belong to COR before the execution of *accept*, u receives (a, A) and joins COR.

The behavior of u in this case depends on whether it has an element (s, S) in $TOP.u$ such that $s = a$. Since $u \notin COR$, if $(a, S) \in TOP.u$, then S differs from A. In this case if u receives (a, A) then it sets $detect =$ **true**. This preserves the validity of the predicate. Alternatively, if such an entry in $TOP.u$ does not exist, then the receipt of (a, A) causes u to join COR and forward (a, A) to all its neighbors. This preserves the predicate as well.

Thus, Predicate 1 holds in the initial state of every computation of the program and is preserved by its every action. Which means that this predicate is an invariant of the program. □

Lemma 2: If a computation of Detector *contains a state where there is a process u that belongs to COR that has a non-faulty neighbor v that does not, then further in the computation, either some non-faulty process sets $detect =$* **true** *or v joins COR.*

Proof: According to Lemma 1, Predicate 1 is an invariant of the program. Hence, if u belongs to COR and its non-faulty neighbor v does not, then channel $Ch.u.v$ contains a message with (a, A). Due to fair message receipt assumption, (a, A) is received. Observe that if v is not in COR and it receives (a, A), then either v sets $detect =$ **true** or joins COR. □

Lemma 3: Every computation of Detector *contains a state where either $detect =$* **true** *in some non-faulty process or every non-faulty process belongs to COR.*

Proof: The proof is by induction on the number of non-faulty processes in the program. As a base case, we show that a itself eventually joins COR. Recall, that we assume that a itself is not faulty. Observe that the program starts in a state where $start.a$ is **true**. If this is so, *init* is enabled. Moreover, *init* is the only action that sets $start.a$ to **false**. Thus, *init* stays enabled until executed. By weak fairness assumption, *init* is eventually executed. When this happens, a joins COR.

Assume that COR contains i: $1 \leq i < n$ processes at some state of a computation and there is a non-faulty process that does not belong to COR. We assume that the connectivity of the graph exceeds the maximum number of faulty processes. Thus, there is a non-faulty process $u \in COR$ that has a non-faulty neighbor $v \notin COR$. According to Lemma 2, this

computation contains a state where COR contains v. Thus, every non-faulty process eventually joins COR. □

Lemma 4: If a computation of Detector *contains a state where non-faulty process u explores a fake process v, then this computation contains a state where $detect =$* **true** *in some non-faulty process.*

Proof: Observe that the only source of fake process information is a Byzantine process. Hence, if u explores a fake process v, then every path to v leads through a Byzantine process. Thus, in a graph with a fake node, the maximum number of node-disjoint paths between a real and a fake node is no more than $k = 1$.

According to Lemma 3, eventually, either $detect =$ **true** at a non-faulty process or u explores every non-faulty process in the system. In this case u detects that all paths to the fake node v lead through no more than k processes and sets $detect =$ **true**. □

Lemma 5: If the system does not contain a faulty process, then the connectivity of the explored graph G' is at least $k+1$.

Proof: We prove the lemma by inductively removing each node p of G that is not present in G' and replacing it with edges between p's neighbors. If a graph is $k+1$-connected then each node has at least $k+1$-neighbors. Note that at least one node is always explored in G'. Thus, the number of nodes in G' is no less than $k+1$. In our proof we make use of the Menger's theorem and show that such removal does not decrease the number of node-independent paths connecting two arbitrary nodes below $k+1$.

Let $G' = G$. If G is $k+1$-connected, so is G'. Let $G'.i$ be a graph with $i > (k+1)$ nodes of G remaining in G'. Assume that the connectivity of $G'.i$ is greater than $k+1$.

Consider how the removal of an arbitrary node $p \in G'.i$ affects node-independent paths connecting a pair of arbitrary nodes. Note that it is sufficient to only consider paths whose ends are the neighbors of p as other affected paths include the neighbors of p as segments.

Let us start with the case where p shares an edge with every node in $G'.i$. If p is removed, $G'.(i-1)$ becomes completely connected and its connectivity is $k+1 > k$. Now, let us consider the case where there is a node u that is not a neighbor of p in $G'.i$. Let v and w be two neighbors of p.

Consider the case v and w do not share an edge. There may be an independent path v, p, w that connects the two nodes in $G'.i$. After the removal of p, v and w are connected. This means that a path v, p, w is replaced by another path: v, w. That is, the independent path is preserved in $G'.(i-1)$.

Consider the case where v and w share an edge in $G'.i$. That is, an edge that connects them is not added in $G'.(i-1)$. In this case the two nodes may lose an internally node disjoint path v, p, w that connects them. Assume the number of nodes in the neighborhood of p is greater than $k+1$. Since the neighborhood is completely connected in $G'.(i-1)$, there are at least $k+1$ internally node disjoint paths between v and w in $G'.(i-1)$. Assume that the number of nodes in p's neighborhood is exactly $k+1$. In this case, there are only k internally node disjoint paths between v and w that only use the neighbors of p. We now show that there is another path.

According to Menger's theorem, three are at least $k+1$ internally node-disjoint paths that connect p and u in $G'.i$. Each path then has to go through a different neighbor of p. This includes v and w. That is there exist two paths $p, v, \cdots, u$ and $p, w, \cdots, u$ in $G'.i$ that do not share nodes except p and u. Neither do these paths contain any other neighbors of p. Consider path $v, \cdots, u, \cdots, w$ composed of the segments of these two paths. The new path does not contain neighbors of p. This new path exists in $G'.(i-1)$ and it is internally node disjoint with the other k paths that connect v and w. That is, the connectivity of $G.(i-1)$ remains at least $k+1$. The lemma follows by induction. □

Lemma 6: Any computation of a detector program contains a state where a Byzantine process is detected only if there is indeed a Byzantine process in the system.

Proof: A non-faulty process sets $detect$ to **true** if it encounters divergent information about some node's neighborhood or when it detects that **connectivity** is less than 2 (i.e. $k+1$). However, a non-faulty process never modifies the neighborhood information about other processes. Hence, if the program does not have a faulty process, all the information about a particular neighborhood that is circulated in the system is identical. Also, according to Lemma 5 if there are no faulty processes in the system, the **connectivity** never falls below 2 ($k+1$). Hence, $detect$ is set to **true** only if indeed the system contains a faulty process. □

Theorem 5: Detector *is an adjacent-edge complete solution to the weak topology discovery problem in case the connectivity of system topology graph exceeds the number of faults.*

Proof: To prove the theorem we show that every computation of *Detector* conforms to the properties of the problem. We then show that the discovered topology is adjacent-edge complete.

Termination property follows from Lemma 3, safety — from Lemma 4, while validity follows from Lemma 6. Notice that Lemma 3 states that unless a fault is detected, the neighborhood of every non-faulty process is added to COR. That is, edges adjacent to a non-faulty processes are detected by every non-faulty processes. Thus, *Detector* is adjacent-edge complete. Hence the theorem. □

Note that with $k = 1$ an adjacent-edge complete solution is also, node and simply-complete. Hence the following corollary.

Corollary 1: Detector provides a complete solution to the topology discovery problem in case the system is doubly connected and contains at most one fault.

Efficiency evaluation. Since we consider an asynchronous model, the number of messages a Byzantine process can send in a computation is infinite. We evaluate the efficiency of *Detector* in case there are no faults in the system. To make the estimation fair, we assume that the unit is $log(n)$ bits. Since it takes that many bits to assign unique process identifiers to n processes, we assume that one identifier is exactly one unit of information. A message in *Detector* carries up to $\delta + 1$ identifiers, where δ is the maximum number of nodes in the neighborhood of a process. Observe that a process can receive at most n messages from each incoming channel. Thus, the total number of messages that can be sent by *Detector* is $2en$,

where e is the number of edges in the graph. The message complexity of the program is in $O(2en\delta)$. If e is proportional to n^2, then the complexity of the program is in $O(\delta n^3)$.

Modification to handle $k > 1$ faults. Observe that the above discussion and correctness proofs are still valid if the possible number of faulty nodes is greater than one and the required network graph connectivity is $k + 1$.

5 EXPLORER

Outline. The main idea of *Explorer* is for each process to collect information about some node's neighborhood such that the information goes along more than twice as many paths as the maximum number of Byzantine nodes. While the paths are node-disjoint, the information is correct if it comes across the majority of the paths. In this case the recipient is in possession of confirmed information. It turns out that the topology information does not have to come directly from the source. Instead it can come from processes with confirmed information. The detailed description of *Explorer* follows.

To simplify the presentation, we describe and prove correct the version of *Explorer* that tolerates only one Byzantine fault. We describe how this version can be extended to tolerate multiple faults in the end of the section.

Description. Since we first describe the 1-fault tolerant version of *Explorer* we assume that the graph is 3-connected. The program is shown in Figure 2. Similar to *Detector*, each process p in *Explorer*, stores the ids of its immediate neighbors. Process p maintains the variable *start*, whose function is to guard the execution of the action that initiates the propagation of p's own neighborhood. Unlike *Detector*, however, p maintains two sets that store the topology information of the network: $uTOP$ and $cTOP$. Set $uTOP$ stores the topology data that is unconfirmed; $cTOP$ stores confirmed topology data. Set $uTOP$ contains the tuples of neighborhood information that p received from other nodes. Besides the process id and the set of its neighbor ids, each such tuple contains a set of process identifiers, that relayed the information. We call it *visited set*. The tuples in $cTOP$ do not require a visited set.

Processes exchange messages where, along with the neighbor identifiers for a certain process, a visited set is propagated. A process contains two actions: *init* and *accept*. The purpose of *init* is similar to that in the process of *Detector*. Action *accept* receives the neighborhood information of some process r, its neighborhood R which was relayed by nodes in set S. The information is received from p's neighbor — q.

First, *accept* checks if the information about r is already confirmed. If so, the only manipulation is to record the received information in $uTOP$. Actually, this update of $uTOP$ is not necessary for the correct operation of the program, but it makes the its proof of correctness easier to follow.

If the received information does not concern already confirmed process, *accept* checks if this information differs from what is already recorded in $uTOP$ either in r or in R. In either case the information is broadcast to all neighbors of p. Before broadcasting, p appends the sender — q to the visited set S.

If the information about r and R has already been received and recorded in $uTOP$, *accept* checks if the previously recorded information came along an internally node disjoint path. If so, the information about r is added to $cTOP$. In this case, this information is also broadcast to all p's neighbors. Note, however, that p is now sure of the information it received. Hence, the visited set of nodes in the broadcast message is empty.

```
process p
const
    P, set of neighbor identifiers of p
parameter
    q : P
var
    start : boolean, initially true, controls sending of p's neighbors
    cTOP : set of tuples, initially {(p, P)},
        (process id, neighbor id set) confirmed topology info
    uTOP : set of tuples, initially ∅,
        (process id, neighbor id set, visited id set)
        unconfirmed topology info
    *[
init:   start ⟶
            start := false,
            (∀j : j ∈ P : send (p, P, ∅) to j)
    []
accept: receive (r, R, S) from q ⟶
            if (∀t, T : (t, T) ∈ cTOP : t ≠ r) then
                if (∀t, T, U : (t, T, U) ∈ uTOP : t ≠ r ∨ T ≠ R)
                then
                    (∀j : j ∈ P : send (r, R, S ∪ {q}) to j)
                elsif (∃t, T, U : (t, T, U) ∈ uTOP :
                        t = r ∧ R = T ∧ ((U ∩ (S ∪ {q}))) ⊂ {r}))
                then
                    cTOP := cTOP ∪ {(r, R)},
                    (∀j : j ∈ P : send (r, R, ∅) to j)
            uTOP := uTOP ∪ {(r, R, S ∪ {q})}
    ]
```

Fig. 2. Process of *Explorer*

Correctness proof. Just like for the *Detector* algorithm, we are focusing on the propagation of the neighborhood information A of a singular non-faulty process a. Notice that we use A to denote the correct neighborhood info. We use A' for the neighborhood information of a that may not necessarily be correct.

To aid us in the argument, we introduce an auxiliary set *SENT* to be maintained by each process. Since this set does not restrict the behavior of processes, we assume that the Byzantine process maintains this set as well. *SENT* contains each message sent by the process throughout the computation. Notice that $uTOP$ records every message received by the process in the computation. Hence, the comparison of $uTOP$ and $SENT$ allows us to establish the channel contents.

Since, a message cannot be received without being sent and vice versa, the following proposition states the invariant of the predicate that affirms it.

Proposition 1: The following predicate is an invariant of

the Explorer *program.*

$$\begin{aligned}&(\forall b, \text{non-faulty } c, A', V : c \in B :\\ &(((a, A', V) \in Ch.b.c) \vee\\ &((a, A', V \cup \{b\}) \in uTOP.c)) \Leftrightarrow\\ &((a, A', V) \in SENT.b))\end{aligned} \tag{2}$$

The predicate states that for any process b and its non-faulty neighbor c the information about the neighborhood of a is recorded in $SENT.b$ if and only if this information is en route from b to c or is recorded in $uTOP.c$ with b appended to the sequence of visited nodes V.

Before we proceed with the correctness argument we have to introduce additional notation. We say that some process c *confirms* (a, A') if it adds this tuple to $cTOP.c$. We view the propagation of A' as construction of a *tree* of processes that relayed A'. This tree *carries* A'. A tree contains two types of nodes: a root and non-root. If process c is non-root, then for some V, $(a, A', V) \in SENT.c$ and $(a, A', V) \in uTOP.c$. That is, a non-root is a process that forwarded the information received from elsewhere without alteration. If c is a root, then $(a, A', V) \in SENT.c$ but $(a, A', V) \notin uTOP.c$. Node c's *ancestor* in a tree is the node that lies on a path from c to the root.

Observe that the root of a tree can only be the process a itself, the Byzantine node or a node that confirms (a, A'). Notice also that since each non-faulty process c sends a message about a's information at most twice, c can belong to at most two trees. Moreover, c has to be the root of one of those trees.

The proposition below follows from Proposition 1.

Proposition 2: If some process d is the ancestor of another process c in a tree carrying (a, A') and $(a, A', V) \in uTOP.c$, then $d \in V$.

Lemma 7: If a non-faulty node c confirms (a, A'), then $A' = A$ and a is real.

Proof: Let us first suppose that a is real. Further, suppose c is the first non-faulty process in the system, besides a, to confirm (a, A'). To add (a, A') to $cTOP.c$ any process $c \neq a$ has to contain $(a, A', V) \in uTOP.c$ and receive a message from one of its neighbors b carrying (a, A', V') such that $V \cap V' \subset \{a\}$. In our notation this means that c belongs to a tree that carries (a, A') and receives a message from b (possibly belonging to a different tree) that carries the same information: (a, A'). Let us consider if b and c belong to the same or different trees.

Suppose b and c belong to the same tree. If this is the case the messages that c receives have to share nodes in the visited sets V and V'. However, for c to confirm (a, A') the intersection of V and V' has to be a subset of $\{a\}$. That is, the only common node between the two sets is a. Observe that a does not forward the information about its own neighborhood if it receives it from elsewhere. Thus, if a belongs to a tree then a is its root. In this case $A' = A$.

Suppose b and c belong to different trees. Recall that for c to confirm (a, A'), both of these trees have to carry (a, A'). However, if $A' \neq A$ then the root of the tree is either the faulty node or another node that confirmed (a, A'). Yet, we assumed that c is the first node to do so. Thus, if c receives a message from b, the only tree that carries the information (a, A') such that $A' \neq A$ is rooted in the faulty node. Thus, even if b and c belong to different trees, $A' = A$.

Similarly, if a is fake, unless another node confirms (a, A') there is only one tree that carries (a, A') and it is rooted in the faulty node. In this case, no other node confirms (a, A'). □

Lemma 8: Every computation of Explorer *contains a state where each non-faulty process belongs to at least one tree carrying (a, A).*

Proof: We prove the lemma by induction on the number of nodes in the system. To prove the base case we observe that the *init* action is enabled in a in the beginning of every computation. This action stays enabled unless executed. Thus, due to weak-fairness of action execution assumption, *init* is eventually executed in a. When it is executed, a forms a tree carrying (a, A).

Let us assume that there are i: $1 \leq i < n$ non-faulty nodes that belong to trees carrying (a, A). Since the network is at least 3-connected, there exists a non-faulty process c that does not belong to such a tree but has a neighbor b that does.

If b belongs to a tree carrying (a, A) then $SENT.b$ contains an entry (a, A, V) for some set of visited nodes V. If c does not belong to such a tree then, by definition, $(a, A, V') \notin uTOP.c$. In this case, according to Proposition 1, $Ch.b.c$ contains (a, A, V). Similar argument applies to the other neighbors of c that belong to trees carrying (a, A). That is, c has incoming messages from every such neighbor.

According to the fair message receipt assumption, these messages are eventually received. We can assume, without loss of generality, that c receives a message from b first. Since c does not contain an entry (a, A, V') in $uTOP.c$, upon receipt of the message from b, c sends a message with $(a, A, V \cup \{b\})$, attaches this message to $SENT.c$ and includes it in $uTOP.c$. This means that c joins the tree carrying (a, A).

Thus, every non-faulty node eventually joins a tree carrying correct neighborhood information about a. □

A *branch* of a tree is either a subtree without the root or the root process alone. The following proposition follows from Proposition 1.

Proposition 3: If a computation of Explorer *contains a state where a non-faulty node c and its neighbor b either belong to two different trees carrying the same information (a, A) or to two different branches of the tree rooted in a, then this computation also contains a state where c confirms (a, A).*

Lemma 9: Every non-faulty process c eventually confirms (a, A).

Proof: The proof is by induction on the number of nodes in the system. The base case trivially holds as a itself confirms (a, A) in the beginning of every computation. Assume that i non-faulty processes have (a, A) in $cTOP$, where $1 \leq i < n$. We show that if there exists another non-faulty process c, it eventually confirms (a, A). Two cases have to be considered: there exists only one tree carrying (a, A), and there are multiple such trees.

Let us consider the first case. Notice, that in every computation there eventually appears a tree rooted in a. In this case, we may only consider a tree so rooted. Since the network is at least 3-connected, there exists a simple cycle containing a and not containing the faulty process. According to Lemma 8, every process in the cycle eventually joins this tree. Observe that, by our definition of a tree branch, there always is a pair of neighbor processes b and c that belong to different branches of a tree rooted in a and carrying (a, A). In this case, according to Proposition 3, one of the two nodes eventually confirms (a, A).

Let us now consider the case of multiple trees carrying (a, A). Again, according to Lemma 8, each non-faulty process in the system joins at least one of these trees. Since the network is at least 3-connected there exists a non-faulty process c belonging to one tree that has a neighbor b belonging to a different tree. In this case, according to Proposition 3, c confirms (a, A).

By induction, every non-faulty process in the system eventually confirms (a, A). □

Theorem 6: Explorer *is a two-adjacent-edge complete solution to the strong topology discovery problem in case of one fault and the system topology graph is at least* 3*-connected.*

Proof: Explorer conforms to the termination and safety properties of the problem as a consequence of Lemmas 9 and 7 respectively.

Observe that a non-faulty node may potentially confirm incorrect neighborhood information about a Byzantine node. That is, an edge reported by the faulty process is either missing or fake. However, due to the two above lemmas, if two nodes are non-faulty the information whether there is an adjacent edge between them is discovered by every non-faulty node. Hence *Explorer* is two-adjacent-edge complete. □

Modification to handle $k > 1$ faults. Observe that *Explorer* confirms the topology information about a node's neighborhood, when it receives two messages carrying it over internally node disjoint paths. Thus, the program can handle a single Byzantine fault. *Explorer* can handle $k > 1$ faults, if it waits until it receives $k + 1$ messages before it confirms the topology info. All the messages have to travel along internally node disjoint paths. For the correctness of the algorithm, the topology graph has to be $(2k + 1)$-connected.

Proposition 4: Explorer *is a two-adjacent-edge complete solution to the strong topology discovery problem in case of* k *faults and the system topology graph is at least* $(2k + 1)$*-connected.*

Efficiency evaluation. Similar to *Detector*, the number of messages a faulty node may send in *Explorer* is arbitrary large. However, we can estimate the message complexity of *Explorer* in the absence of faults. Each message carries a process identifier, a neighborhood of this process and a visited set. The number of the identifiers in a neighborhood is no more than δ, and the number of identifiers in the visited set can be as large as n. Hence the message size is bounded by $\delta + n + 1$ which is in $O(n)$.

Notice, that for the neighborhood A of each process a, every process broadcasts a message twice: when it first receives the information, and when it confirms it. Thus, the total number of sent messages is $4e \cdot n$ and the overall message complexity of *Explorer* if no faults are detected is in $O(n^4)$.

6 Composition and Extensions

Composing *Detector* and *Explorer*. Observe that *Detector* has better message complexity than *Explorer* if the neighborhood size is bounded. Hence, if the incidence of faults is low, it is advantageous to run *Detector* and invoke *Explorer* only if a fault is detected. We assume that the processes can distinguish between message types of *Explorer* and *Detector*. In the combined program, a process running *Detector* switches to *Explorer* if it discovers a fault. Other processes follow suit, when they receive their first *Explorer* messages. They ignore *Detector* messages henceforth. A Byzantine process may potentially send an *Explorer* message as well, which leads to the whole system switching to *Explorer*. Observe that if there are no faults, the system will not invoke *Explorer*. Thus, the complexity of the combined program in the absence of faults is the same as that of *Detector*. Notice that even though *Detector* alone only needs $(k + 1)$-connectivity of the system topology, the combined program requires $(2k + 1)$-connectivity.

Message Termination. We have shown that *Detector* and *Explorer* comply with the functional termination properties of the topology discovery problem. That is, all processes eventually discover topology. However, the performance aspect of termination, viz. message termination, is also of interest. Usually, an algorithm is said to be message terminating if all its computations contain a finite number of sent messages [5].

However, a Byzantine process may send messages indefinitely. To capture this, we weaken the definition of message termination. We consider a Byzantine-tolerant program *message terminating* if the system eventually arrives at a state where: (a) all channels are empty except for the outgoing channels of a faulty process; (b) all actions in non-faulty processes are disabled except for possibly the receive-actions of the incoming channels from Byzantine processes, these receive-actions do not update the variables of the process. That is, in a terminating program, each non-faulty process starts to eventually discard messages it receives from its Byzantine neighbors.

Making *Detector* terminating is fairly straightforward. As one process detects a fault, the process floods the announcement throughout the system. Since the topology graph for *Detector* is assumed $(k+1)$-connected, every process receives such announcement. As the process learns of the detection, it stops processing or forwarding the messages. Notice that the initiation of the flood by a Byzantine node itself only accelerates the termination of *Detector* as the other processes quickly learn of the faulty node's existence.

The addition of termination to *Explorer* is more involved. To ensure termination, restrictions have to be placed on message processing and forwarding. However, the restrictions should be delicate as they may compromise the liveness properties of the program. By the design of *Explorer*, each process may send at

most one message about its own neighborhood to its neighbors. Hence, the subsequent messages can be ignored. However, a faulty process may send messages about neighborhoods of other processes. These processes may be real or fake. We discuss these cases separately.

Note that each process in *Explorer* can eventually obtain an estimate of the identities of the processes in the system and disregard fake process information. Indeed, a path to a fake node can only lead through faulty processes. Thus, if a process discovers that there may be at most k internally node disjoint paths between itself and a certain node, this node is fake. Therefore, the process may cease to process messages about the fake node's neighborhood. Notice, that since the system is $(2k+1)$-connected, messages about real nodes will always be processed. Therefore, the liveness properties of *Explorer* are not affected.

As to the real processes, they can be either Byzantine or non-faulty. Recall that each non-faulty process of *Explorer* eventually confirms neighborhoods of all other non-faulty processes. After the neighborhood of a process is confirmed, further messages about it are ignored.

The last case is a Byzantine process u sending a message to its correct neighbor v about the neighborhood of another Byzantine process w. By the design of *Explorer*, v relays the message about w provided that the neighborhood information about w differs from what previously received about w. As we discussed above, eventually v estimates the identities of all real processes in the system. Therefore, there is a finite number of possible different neighborhoods of w that u can create. Hence, eventually they will be exhausted, and v starts ignoring further messages form u about w.

Thus, *Explorer* can be made terminating as well.

Handling topology updates. In the topology discovery problem statement, it is assumed that the system topology does not change. However, *Detector* and *Explorer* can be adapted to manage topology changes as well. Observe, that apart from detecting fake nodes in *Explorer*, both algorithms propagate the information of one process neighborhood independently of the others. We first describe how this propagation can be done in case the topology changes and then address the fake node detection. Each time the neighborhood of a process p changes, p starts a new *version* of the topology discovery algorithm for its neighborhood. Observe that a faulty process may also start a new version for p.

The versions are distinguished by version numbers. Each process maintains the version numbers of p. Each related message carries the version number. Each process outputs the discovered neighborhood of p with the highest received version number. Observe that in the case of *Explorer* the processes only output confirmed information. Notice that if a faulty process sends incorrect information about p's neighborhood with a certain version number, this incorrect info will be handled by the basic *Detector* or *Explorer* within that version. For example, the faulty messages of version i about p's neighborhood will be countered by the correct messages of the same version. Notice that a faulty process in *Explorer* may start a version j for p's neighborhood such that it is higher than the highest version i that p itself started. However, according to the basic *Explorer*, the incorrect information in version j will not be confirmed.

There are two specific modifications to the basic *Detector*. If the faulty process sends a message concerning p with the version number higher than that of p, p itself detects the fault. To detect fake nodes generated by a faulty process, each node has to compile the topology TOP graph of the highest version number for each node in the system and ensure that its connectivity does not fall below $k+1$. Observe that *Detector* is unable to differentiate between temporary lack of connectivity from malicious behavior of the faulty nodes. Therefore, the connectivity of the discovered network at each node should never fall below $k+1$. For that, we assume that throughout a computation the intersection of all system topologies is $k+1$-connected. Similarly, the topology has to always be $2k+1$-connected for *Explorer* to operate correctly.

The topology update mechanism can be optimized in obvious ways. For *Detector*, each process has to keep the information for p with only the highest version number. Obsolete information can be safely discarded. For *Detector*, the process may keep the latest version of *confirmed* neighborhood information. Observe that this extension of the topology discovery algorithms assumes infinite-size counters. Care must be taken when implementing these counters in the actual hardware, as the faulty processes may try to compromise topology discovery if the counter values are reused. Hence, such an implementation would require a Byzantine-robust counter synchronization algorithm. Lamport and Melliar-Smith [11] proposed such algorithm for completely connected systems. Extending it to arbitrary topology systems is an attractive avenue of further research.

Discovering neighbors. As described, in the initial state of *Detector* and *Explorer*, each process has access to correct information about its immediate neighborhood. Note that, in general, obtaining this information in the presence of Byzantine processes may be difficult as they can mount a Sybil attack [8]. In such an attack, a faulty process is able to use an arbitrary identifier while a correct process cannot determine whether two messages were sent by the same or by different process. A Sybil attack is difficult to handle [4], [23]. However, *Detector* and *Explorer* can be modified to handle neighborhood discovery with *known ports*. That is, each process does not know the identities of its neighbors but can determine if a message is coming from the same process. Also note that we assume that there are no actual topology changes during the computation as it interferes with the neighborhood discovery. Observe that with known ports a faulty process may not be able to use more than one identifier per correct neighbor without being detected.

Note that there are certain limits of what can be discovered under these assumptions. First, no algorithm can determine the identifier of a faulty process as it may assume an arbitrary one. Moreover, with more than one faulty process, the system is subject to "black hole" attack: a pair of colluding faulty nodes may deceive their non-faulty neighbors into believing that they share an edge. When communicating to a non-faulty

node a, its faulty neighbor b assumes the identity of another non-faulty node d. Similarly, a faulty neighbor c of d assumes the identity of a. This way, non-faulty nodes a and d are led to believe they share an edge.

The modified algorithms contain two phases: neighborhood discovery phase and topology discovery proper phase. In the first phase, each process broadcasts its identifier to its neighbors. Observe that faulty processes may not send these initial messages at all. Thus, the process should not wait for a message from every possible neighbor. Instead, as soon as each process p gets a message from a new port with q in its identifier, p may start the second phase with $\{q\}$ as its neighborhood. Every time p gets a new distinct identity from a fresh port, p treats it as topology update, increments its counter and re-initiates the topology discovery. Note that a faulty process may select an identifier of another neighbor r. Due to known ports, the recipient will be able to distinguish between the two processes. In case of *Detector*, the recipient can immediately detect a fault. In case of *Explorer* the proposed procedure is as follows. The recipient process identifies its neighbors with the same identities r and r' and propagates the information to the other processes in this form. The other processes treat these identities as distinct.

This identity discovery procedure can be further streamlined. Recall that for *Detector* and *Explorer* the topology graph has to be respectively $k+1$ and $2k+1$-connected. Thus, depending on the algorithm, each process is guaranteed to have at most 1 or $k+1$ non-faulty neighbors. Therefore, each process may delay initiating topology discovery until it gets this minimum number of distinct identities. However, though the updates, each process learns the identities of all correct processes in the system.

Other extensions. Observe that *Explorer* is designed to disseminate the information about the complete topology to all processes in the system. However, it may be desirable to just establish the routes from all processes in the system to one or a fixed number of distinguished ones. To accomplish this *Explorer* needs to be modified as follows. No neighborhood information is propagated. Instead of the visited set, each message carries the propagation path of the message. That is, the order of the relays is significant.

Only the distinguished processes initiate the message propagation. The other processes only relay the messages. Just as in the original *Explorer*, a process confirms a path to another process only if it receives $2k+1$ internally process disjoint paths from the source or from other confirming processes. Again, like in *Explorer*, such process rebroadcasts the message, but empties the propagation path. In the outcome of this program, for every distinguished process, each non-faulty process will contain paths to at least $2k+1$ processes that lead to this distinguished process. Out of these paths, at least $k+1$ ultimately lead to the distinguished process.

In *Explorer*, for each process the propagation of its neighborhood information is independent of the other neighborhoods. Thus, instead of topology, *Explorer* can be used for efficient fault-tolerant propagation of arbitrary information from the processes to the rest of the network.

7 Conclusion and Open Problems

In this article we explored an unorthodox approach to Byzantine-robust routing. It relies on topology itself rather than on cryptographic primitives. It is best suited where the more conventional cryptography-based solution is infeasible due to lack of computing, storage and transmission resources such as on motes [9]. However, our approach may be an attractive alternative even if the platform technically allows cryptographic operation. In topology-based Byzantine tolerance, the fault-recovery is based on route-based redundancy of information propagation. If the system allows such redundancy of routes, then our approach promises significant computation savings since no cryptographic operations are required.

Note that our approach requires that the network is designed to be dense enough to tolerate the acceptable number of faults. Alternatively, the connectivity or density of the network dictates the maximum number of faults it is able to tolerate.

In conclusion, we would like to outline a several interesting research directions that this paper opens. The existence of Byzantine-robust topology discovery solutions poses the question of theoretical limits of efficiency of such programs. The obvious lower bound on message complexity can be derived as follows. Every process must transmit its neighborhood to the rest of the nodes in the system. Transmitting information to every node requires at least n messages, so the overall message complexity is at least δn^2. If k processes are Byzantine, they may not relay the messages of other nodes. Thus, to ensure that other nodes learn about its neighborhood, each process has to send at least $k+1$ messages. Thus, the complexity of any Byzantine-robust solution to the topology discovery problem is at least in $\Omega(\delta n^2 k)$.

Observe that *Explorer* and *Detector* may not explicitly identify faulty nodes or the inconsistent view of the their immediate neighborhoods. We believe that this identification can be accomplished using the technique used by Dolev [7]. In case there are $3k+1$ non-faulty processes, they may exchange the topologies they collected to discover the inconsistencies. This approach, may potentially expedite termination of *Explorer* at the expense of greater message complexity: if a certain Byzantine node is discovered, the other processes may ignore its further messages.

In Section 6 we briefly considered the problem of transformation of known ports to oral record [12] model in the context of topology discovery. However, the problem can be posed in a more general setting. That is, what are the necessary and sufficient conditions for enable such transformation in an arbitrary system?

Note that our solution assumes that the failures may affect arbitrary nodes. As the system scale increases, this may lead the designers to build the system with a lot of redundant links. It would be interesting to study a problem with less potent faults; for example where the placement of faulty nodes is randomized.

References

[1] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. McGraw-Hill Publishing Company, New York, May 1998. 6.

[2] I.C. Avramopoulos, H. Kobayashi, R. Wang, and A. Krishnamurthy. Highly secure and efficient routing. In *Proceedings of INFOCOM*, Hong Kong, March 2004.

[3] V. Bhandari and N.H. Vaidya. On reliable broadcast in a radio network. In *Proceedings of the Twenty-Fourth Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2005)*, page to appear, Las Vegas, Nevada, July 2005.

[4] Sylvie Delaët, Partha Sarathi Mandal, Mariusz Rokicki, and Sébastien Tixeuil. Deterministic secure positioning in wireless sensor networks. In *Proceedings of the ACM/IEEE International Conference on Distributed Computing in Sensor Networks (DCOSS 2008)*, Lecture Notes in Computer Science, Santorini, Greece, June 2008. Springer-Verlag.

[5] E.W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[6] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, Berlin, 1990.

[7] D. Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, 1982.

[8] J.R. Douceur. The Sybil attack. In *First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, MA, USA, March 2002.

[9] J.L. Hill and D.E. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, November/December 2002.

[10] C.-Y. Koo. Broadcast in radio networks tolerating byzantine adversarial behavior. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 275–282, New York, NY, USA, 2004. ACM Press.

[11] L. Lamport and P.M. Melliar-Smith. Byzantine clock synchronization. *Operating Systems Review*, 20(3):10–16, 1986.

[12] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[13] D. Malkhi, Y. Mansour, and M.K. Reiter. Diffusion without false rumors: on propagating updates in a Byzantine environment. *Theoretical Computer Science*, 299(1–3):289–306, April 2003.

[14] D. Malkhi, M. Reiter, O. Rodeh, and Y. Sella. Efficient update diffusion in byzantine environments. In *The 20th IEEE Symposium on Reliable Distributed Systems (SRDS '01)*, pages 90–98, Washington - Brussels - Tokyo, October 2001. IEEE.

[15] T. Masuzawa. A fault-tolerant and self-stabilizing protocol for the topology problem. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 1.1–1.15, 1995.

[16] Toshimitsu Masuzawa and Sébastien Tixeuil. Bounding the impact of unbounded attacks in stabilization. In Ajoy Kumar Datta and Maria Gradinariu, editors, *SSS*, volume 4280 of *Lecture Notes in Computer Science*, pages 440–453. Springer, 2006.

[17] Toshimitsu Masuzawa and Sébastien Tixeuil. Stabilizing link-coloration of arbitrary networks with unbounded byzantine faults. *International Journal of Principles and Applications of Information Science and Technology (PAIST)*, 1(1):1–13, December 2007.

[18] Y. Minsky and F.B. Schneider. Tolerating malicious gossip. *Distributed Computing*, 16(1):49–68, 2003.

[19] M. Nesterenko and A. Arora. Tolerance to unbounded byzantine faults. In *Proceedings of 21st IEEE Symposium on Reliable Distributed Systems*, pages 22–29, 2002.

[20] A. Pelc and D. Peleg. Broadcasting with locally bounded byzantine faults. *Information Processing Letters*, 93:109–115, 2005.

[21] A. Perrig, J. Stankovic, and D. Wagner. Security in wireless sensor networks. *Communications of the ACM*, 47(6):53–57, June 2004.

[22] Y. Sakurai, F. Ooshita, and T. Masuzawa. A self-stabilizing link-coloring protocol resilient to byzantine faults in tree networks. In *Proceedings of the 2004 International Conference on Principles of Distributed Systems (OPODIS'2004)*, Lecture Notes in Computer Science. Springer-Verlag, December 2004.

[23] Adnam Vora, Mikhail Nesterenko, Sébastien Tixeuil, and Sylvie Delaët. Universe detectors for sybil defense in ad hoc wireles networks. In *International Conference on Stabilization, Safety, and Security (SSS 2008)*, Lecture Notes in Computer Science. Springer-Verlag, November 2008.

[24] J. Yellen and J.L. Gross. *Graph Theory & Its Applications*. CRC Press, 1998. ISBN: 0–849–33982–0.

PLACE PHOTO HERE

Mikhail Nesterenko got his PhD in 1998 from Kansas State University. Presently he is an associate professor at Kent State University. He is interested in distributed algorithms.

PLACE PHOTO HERE

Sébastien Tixeuil got his PhD in 2000 from Université Paris Sud-XI. He is now a professor at Université Pierre & Marie Curie - Paris 6. His research interests include fault and attack tolerance in networks and distributed systems.