Discovering Network Topology in the Presence of Byzantine Faults

Mikhail Nesterenko[1]* and Sébastien Tixeuil[2]**

[1] Computer Science Department Kent State University Kent, OH, 44242, USA,
mikhail@cs.kent.edu
[2] LRI-CNRS UMR 8623 & INRIA Grand Large
Université Paris Sud, France, tixeuil@lri.fr

May 9, 2005
Technical Report TR-KSU-CS-2005-1

Abstract. We present Byzantine-robust solutions to the topology discovery problem. Our programs allow each process to learn the complete topology of the network (up to the neighborhoods of the faulty nodes). The properties of our programs are as follows. They tolerate up to a fixed number of faults. They terminate. They handle arbitrary network topology. The individual processes do *not* know either the diameter or the size of the network. The execution model is asynchronous. The processes do not use cryptographic primitives such as digital signatures.

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.

Topology discovery is a fundamental problem in distributed computing (*e.g.* see [17]). It has direct applicability in practical systems. For example, link-state based routing protocols such as OSPF use topology discovery mechanisms to compute the routing tables. Recently, the problem came to the fore with the introduction of ad hoc wireless sensor networks, such as Berkeley motes [5], where topology discovery is essential for routing decisions.

As reliability demands on distributed systems increase, the interest in developing robust topology discovery programs grows. One of the strongest fault

* This author was supported in part by DARPA contract OSU-RF#F33615-01-C-1901 and by NSF CAREER Award 0347485.
** This author was supported in part by the FNS grants FRAGILE and SR2I from ACI "Sécurité et Informatique".

models is *Byzantine* [7]: the faulty node may behave arbitrarily. This model encompasses rich set of faulty behaviours. Moreover, Byzantine fault tolerance has security implications, as the behavior of an intruder can be modeled as Byzantine. One way to deal with Byzantine faults is by introducing security primitives such as digital signatures or certificates. However, this option may be unavailable. For example, wireless sensors do not have the capacity to manipulate digital signatures. A way to limit the power of a Byzantine process is to assume synchrony: each process 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.

Related work. Masuzawa [10] considers the problem of topology discovery and update. However, his fault model is not as general as Byzantine: he considers only transient and crash faults. Most Byzantine fault-tolerant programs described in the literature [1, 8, 9, 12] assume completely connected networks and cannot be easily extended to deal with arbitrary topology. Dolev [4] considers Byzantine agreement on arbitrary graphs. Dolev 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 [2, 6, 14]. However, in all cases the topology is assumed to be known. Bhandari and Vaidya [2] and Koo [6] assume two-dimentional grid. Pelc and Peleg [14] consider arbitrary topology but assume its a priori knowledge at each node.

A notable class of algorithms tolerates Byzantine faults locally [11, 13, 16]. Yet, the emphasis of these 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. Thus, local containment is not applicable to topology discovery.

Perrig *et al.* [15] survey robust routing methods in *ad hoc* sensor networks. The techniques covered assume that the nodes are capable of cryptographic operations.

Our contribution. In this paper we present two topology discovery programs for arbitrary networks. They can withstand up to a fixed k Byzantine faults. They perform correctly without synchrony assumptions on process execution speed. Our programs do not rely on cryptographic operations. Both programs are terminating and have polynomial message complexity. Moreover, our programs do not assume that the nodes know the parameters of the network such as the total number of nodes in the system or its diameter. To our knowledge, these are the first asynchronous Byzantine-robust solutions to topology discovery problem in general networks that do not use digital signatures.

Our programs allow the nodes to discover topology up to the neighborhood of a faulty node. Our first program — *Detector*, either discovers topology or detects the presence of a fault. In the latter case it invokes a more robust program — *Explorer*. *Detector*, however, has better message complexity than *Explorer*. *Detector* either determines topology or signals fault 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* assumes that the network is $(k+1)$-connected. *Explorer* requires $(2k+1)$-connectivity. We extend our programs to (a) discover a fixed number of routes instead of complete topology and (b) reliably propagate arbitrary information instead of topological data.

The rest of the paper is organized as follows. After stating our underlying model in Section 2, we first present *Detector* in Section 3. Then, in Section 4, we present *Explorer*. We discuss the composition of our programs and their extensions in Section 5 and conclude the paper in Section 6.

2 Notation, Definitions and Assumptions

Graphs. A distributed *system* (or *program*) consists of a set of processes and a binary reflexive symmetric relation between them. Each process has an identifier that is unique throughout the system. This relation is the system *topology*. The topology forms a graph G. Denote n and e to be the number of nodes[3] and edges in G respectively. Two processes are *neighbors* if there is an edge in G connecting them. A set P of neighbors of process p is *neighborhood* of p. In the sequel we use small letters to denote singleton variable 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 fact about graph connectivity that follows from Menger's theorem (see [18]): if a graph is k-connected 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 pair of neighbor processes share a pair of special variables called *channels*. We denote $Ch.b.c$ the channel form process b to process c. Process b is the *sender* and c is the *receiver*. A channel variable has a value chosen from the domain of (potentially infinite) sequences of messages.

[3] We use terms *process* and *node* interchangeably.

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 [3] of the form: $(\langle quantifier \rangle \langle bound\ variables \rangle : \langle range \rangle : \langle term \rangle)$, where *range* and *term* are boolean constructs.

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. Observe that this allows a faulty node to send arbitrary messages. We assume, however, that messages sent by such 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. We assume *oral record* [7] 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.

Semantics. An action of a process of the program is *enabled* in a certain state if its guard evaluates to true. An action of a faulty process is always enabled. 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 define 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.

Graph exploration. The processes discover the topology of 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 Detector

Outline. The objective of the *Detector* program is to make either every non-faulty process learn the topology of the system or one process to detect a Byzantine fault. Observe that in the presence of a faulty process, *Detector* does not have to ascertain correct topology information, just detect the inconsistency caused by the Byzantine process. *Detector* leverages the connectivity of the system 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. However, the graph connectivity assumption is used to detect 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 the number of faulty nodes. 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.

Detailed Description. We assume that the graph is $(k+1)$-connected. 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 keeps the upper bound k on the number of faulty processes. Process p maintains the following variables. Boolean variable *detect* indicates if p discovers a fault in the system. Boolean

```
process p
const
    P: set of neighbor identifiers of p
    k: 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) ∨
                   (path_number(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. Detector process

variable *start* guards the execution of the action that sends p's neighborhood information to its neighbors. Set TOP 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 **path_number** 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 the graph G' by adding an edge to every pair of unexplored processes present in TOP. We calculate the value of **path_number** 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 **path_number** returns ∞. Otherwise the function returns the minimum number of internally node disjoint paths between two explored nodes in G'. In the correctness proof for this program we show that unless there is a fake node, the **path_number** of G' is no smaller than the connectivity of G.

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 $(k+1)$-connected p indicates that it detected a fault by setting *detect* to **true**. Alternatively, if p did not previously have the information about r, p updates TOP and sends the received information to all its neighbors.

Correctness proof. 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.

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

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 $detect = \textbf{false}$, 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 $start.a = \textbf{false}$ 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* = **false**. 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 u is not in COR and it receives (a, A), then either u 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* $detect = \mathbf{true}$ *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.

According to Lemma 3, eventually, either $detect = \mathbf{true}$ 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 $detect = \mathbf{true}$. □

The below theorem follows from Lemmas 3 and 4.

Theorem 1 (Liveness). *Every computation of the* Detector *program contains a state where either a Byzantine process is detected or each non-faulty process contains correct neighborhood information about of every non-faulty process present in the system and no information about processes not present in the system.*

Lemma 5. *If the system does not have a faulty process, then in every computation, for each process, the path_number of the explored subgraph G' is greater than k.*

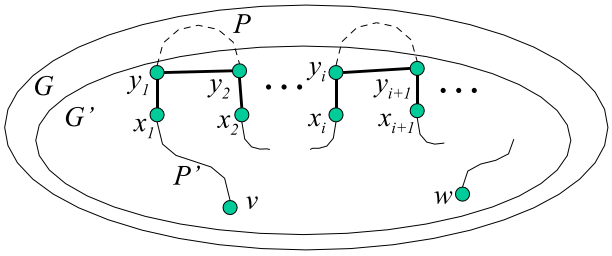

Fig. 2. Illustration for the proof of Lemma 5: construction of path $P' \subset G'$ on the basis of path $P \subset G$

Proof: Observe that if there are no faulty processes, only correct topology information is circulated in the system. Hence, for each process u, $TOP.u$ contains the subgraph of the system graph G. In this case, $G'.u$ is an arbitrary set of explored processes from G and the unexplored members of their neighborhoods. By the construction of $G'.u$, every pair of unexplored processes is connected by an edge.

Let v and w be an arbitrary pair of explored nodes in $G'.u$. And let P be a path connecting v and w in G. We claim that there exists a path P' in $G'.u$ connecting v and w that is also a node-subset of P. That is, every node that belongs to P' also belongs to P. See Figure 2 for the illustration. If P only contains the nodes explored in $G'.u$, our claim holds for $P' = P$. Let P contain unexplored nodes as well. In general, P contains alternating segments of explored and unexplored nodes. Let $\langle x_i, y_i, \cdots, y_{i+1}, x_{i+1} \rangle$ be any such unexplored segment, where x_i, x_{i+1} are explored and $y_i, \cdots, y_{i+1}$ are not. Observe that y_i and y_{i+1} have explored neighbors — x_i and x_{i+1} respectively. This means that both y_i and y_{i+1} belong to $G'.u$. Since y_i and y_{i+1} are unexplored, $G'.u$ contains an edge connecting them. We construct P' to contain every explored segment of P; we replace every unexplored segment by the edge that links unexplored nodes in $G'.u$. Observe that by construction, $P' \in G'.u$ and P' contains a subset of the nodes of P. Thus, our claim holds.

Let P_1 and P_2 be two internally node disjoint paths connecting v and w in G. According to the just proved claim, there exist P_1' and P_2' belonging $G'.u$ that connect v and w. Moreover, P_1' contains a subset of nodes of P_1 and P_2' contains a subset of nodes of P_2. Since P_1 and P_2 are internally node disjoint, so are P_1' and P_2'.

Recall that G is assumed to be $(k+1)$-connected. This means that for every two vertices v and w there exist $k+1$ internally node disjoint paths between v and w. Thus, the number of internally node disjoint paths for v and w in $G'.u$ is at least $k+1$. Hence, the path number of $G'.u$ is greater than k. □

Theorem 2 (Safety). *Any computation of a detector program contains a state where a Byzantine process is detected only if there indeed is a Byzantine process in the system.*

Proof: A non-faulty process sets *detect* to **true** if it encounters divergent information about some nodes' neighborhood or when it detects that the path number is less than $k+1$. 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 path number never falls below $k+1$. Hence, *detect* is set to **true** only if indeed the system contains a faulty process. □

Efficiency evaluation. Since we consider an asynchronous model, the number of messages a Byzantine process can send in a computation is infinite. To evaluate the efficiency of *Detector* we assume that each process is familiar with the upper bound on the number of processes in the system and this upper bound is in $O(n)$. A non-faulty process then detects a fault if the number of processes it explores exceeds this bound or if it receives more than one identical message from the

same neighbor. We assume that the process stops and does not send or receive any more messages if it detects a fault.

In this case we can estimate the number of messages that are received by non-faulty processes before one of them detects a fault or before the computation terminates. To make the estimation fair, the 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 a 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 send by *Detector* is $2en$. 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 $O(\delta n^3)$.

4 Explorer

Outline. The objective of the *Explorer* program is to make every non-faulty process learn the topology of the system. Unlike the *Detector* program, *Explorer* cannot quit after detection of a Byzantine process. *Explorer* has to establish correct topology despite the interference of a faulty process. 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. If 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 one Byzantine fault. We describe how this version can be extended to tolerate multiple faults in the end of the section.

Description. We assume that the graph is $(2k + 1)$-connected. The program is shown in Figure 3. 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 $uTOP$ contains the topology info that is not confirmed. On the other hand, $cTOP$ 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 neighbors 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 visited set.

Processes exchange messages where, along 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 form p's neighbor — q.

First, *accept* checks if the the information about r is already confirmed. If so, the only manipulation is to record the received information in $uTOP$. Actually, in this case, even this update of $uTOP$ is extraneous, but it makes the 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.

Correctness proof. Just like for the *Detector* program 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 axillary 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 broadcast 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 $SENT$ of the neighboring processes allows us to derive a formal proof for *Explorer*.

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

Lemma 6. *The following predicate is an invariant of the* Explorer *program.*

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

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

```
process p
const
    P, set of neighbor identifiers of p
parameter
    q : P
var
    start : boolean, initially true, controls sending of p's neighbor ids
    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) ⊂ {r}))
                then
                    cTOP := cTOP ∪ {(r, R)},
                    (∀j : j ∈ P : send (r, R, ∅) to j)
            uTOP := uTOP ∪ {(r, R, S ∪ {q})}
     ]
```

Fig. 3. Explorer process

to $cTOP.c$. A *tree carries* neighborhood information A' about process a. A tree contains two types of nodes: a root and non-root. If process c is non-root, then for some V, $(a, A', V) \in SEND.c$ 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 $(a, A', V) \in SEND.c$ 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 below lemma follows from Lemma 6.

Lemma 7. *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 8. *If a non-faulty node c confirms (a, A'), then $A' = A$ and a is real.*

Proof: Let us 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. 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. According to Lemma 7, V and V' contain the respective identifiers of the roots of b's and c's trees. Since the intersection of V and V' is a subset of $\{a\}$, if b and c belong to the same tree, this tree is rooted in a. 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, if b and c belong to different trees then $A' = A$.

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 9. *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 $(2k+1)$-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 $SEND.b$ 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 Lemma 6, $Ch.b.c$ 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 $SEND.c$ 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 lemma follows from Lemma 6.

Lemma 10. *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 11. *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 $(2k + 1)$-connected, there exists a simple cycle containing a and not containing the faulty process. According to Lemma 9, 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 Lemma 10, 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 9, each non-faulty process in the system joins at least one of these trees. Since the network is at least $(2k + 1)$-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 Lemma 10, c confirms (a, A).

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

The below theorem is a direct consequence of Lemmas 8 and 11.

Theorem 3 (Liveness). *Every computation of* Explorer *contains a state where each non-faulty process contains confirmed neighborhood information about every non-faulty process present in the system and no-information about processes not present in the system.*

The next theorem follows from Lemma 8.

Theorem 4 (Safety). *No computation of* Explorer *contains a state with a non-faulty process containing confirmed incorrect information about another process in the system or information about a process not present in the system altogether.*

Observe a non-faulty node may potentially confirm incorrect information about a Byzantine node. That is, an edge reported by the faulty process is either missing or fake. The other adjacent node may or may not be faulty. This incorrect information conflicts with the data confirmed by the adjacent non-faulty node. However, this information may not conflict if the adjacent node is also Byzantine. The following corollary clarifies the extent of this conflict. An edge in confirmed topology is *suspect* if the neighborhood information of one of the adjacent nodes does not contain it.

Corollary 1. *The confirmed topology in any non-faulty node contains a suspect edge only if one of the adjacent nodes is Byzantine. This topology contains incorrect edge information (the edge is either missing or a fake edge present) only if both nodes adjacent to the edge are Byzantine.*

Efficiency Evaluation. Unlike *Detector*, *Explorer* does not quit when a fault is discovered. Thus, the number of messages a faulty node may send 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 $O(n^4)$.

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 path. Thus, the program can handle a single Byzantine fault. The explorer can handle $k > 1$ faults, if it waits until it receives $2k + 1$ 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.

5 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.

Termination. A Byzantine process may send messages indefinitely. To capture this, we weaken the definition of termination. We consider a Byzantine-tolerant program *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 of 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 Theorem 3 states that each process eventually confirms neighborhoods of all correct 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 infor-

mation 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.

Other extentions. 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 node disjoint paths from the source or from other confirming nodes. 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 node. Out of these paths, at least $k + 1$ ultimately lead to the distinguished node.

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.

6 Conclusion

In conclusion, we would like to outline a couple of interesting avenues of further research.

The existence of Byzantine-robust topology discovery solutions opens 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 belive that

this can be accomplished using the technique used by Dolev [4]. 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 further its further messages.

References

1. Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics.* McGraw-Hill Publishing Company, New York, May 1998. 6.
2. Vartika Bhandari and Nitin 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.
3. Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, Berlin, 1990.
4. D. Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, 1982.
5. J.L. Hill and D.E. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, November/December 2002.
6. Chiu-Yuen 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.
7. Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.
8. 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.
9. Dahlia Malkhi, Yishay Mansour, and Michael K. Reiter. Diffusion without false rumors: on propagating updates in a Byzantine environment. *Theoretical Computer Science*, 299(1–3):289–306, April 2003.
10. 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.
11. Toshimitsu Masuzawa and Sébastien Tixeuil. A self-stabilizing link-coloring protocol resilient to unbounded byzantine faults in arbitrary networks. Technical Report 1396, Laboratoire de Recherche en Informatique, January 2005.
12. Yaron Minsky and Fred B. Schneider. Tolerating malicious gossip. *Distributed Computing*, 16(1):49–68, 2003.
13. Mikhail Nesterenko and Anish Arora. Tolerance to unbounded byzantine faults. In *Proceedings of 21st IEEE Symposium on Reliable Distributed Systems*, pages 22–29, 2002.
14. A. Pelc and D. Peleg. Broadcasting with locally bounded byzantine faults. *Information Processing Letters*, 93:109–115, 2005.

15. Adrian Perrig, John Stankovic, and David Wagner. Security in wireless sensor networks. *Communications of the ACM*, 47(6):53–57, June 2004.
16. Yusuke Sakurai, Fukuhito Ooshita, and Toshimitsu 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.
17. J. M. Spinelli and R. G. Gallager. Event-driven topology broadcast without sequence numbers. *IEEE trans. on commun.*, COM-37, 5:468–474, 1989.
18. Jay Yellen and Jonathan L. Gross. *Graph Theory & Its Applications*. CRC Press, 1998. ISBN: 0–849–33982–0.