Bounds on Topology Discovery 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

Abstract. This report is a companion to another technical report [3] to present the formal proofs that could not fit into the conference proceedings of this article [4] due to space limitations.

In this report we revisit the problem of Byzantine-robust topology discovery. We formally state the weak and strong versions of the problem. We focus on non-cryptographic solutions to these problems and 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.

April 13, 2006
Technical Report TR-KSU-CS-2006-1

1 Introduction

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.

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 a edge between

* An abbreviated version of these results was presented in [4].

** 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".

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.

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[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 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 fact about graph connectivity that follows from Menger's theorem (see [5]): 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 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.

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 [1] 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

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

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 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.

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. 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. 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* [2] 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.

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 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 computation 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.

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.

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 exists no node- and 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 that $B_1 \not\subseteq B_2$. 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 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 $q \in A_1$ behaves as follows. In the channels connecting q to the nodes of $C_1 \subset G_3$, each round q outputs the messages as in s_1. Similarly, in the channels connecting q to the nodes of $B_2 \subset G_3$, q 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, $B_1 \not\subseteq B_2$. Moreover, none of the nodes in B_2 are faulty. If

this is the case then either s_3 violates the safety property of the problem or the assumed solution is not adjacent-edge complete. The theorem follows. □

Observe that for $(k+1)$-connected graphs an adjacent-edge complete solution is also node complete.

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 exists no node- and 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 $2k$, $A_1 = A_2$, $B_1 \cap C_1 = \varnothing$, $B_2 \cap C_2 = \varnothing$, and $B_1 \not\subseteq B_2$. 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 that none of the nodes in B_2 are faulty, this implies that either s_3 violates the safety property of the problem or the assumed solution is not adjacent-edge complete. The theorem follows. □

References

1. Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, Berlin, 1990.
2. Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.
3. Mikhail Nesterenko and Sébastien Tixeuil. Discovering network topology in the presence of byzantine faults. Technical Report TR-KSU-CS-2005-01, Dept. of Computer Science, Kent State University, 2005. http://www.cs.kent.edu/techreps/TR-KSU-CS-2005-01.pdf.
4. Mikhail Nesterenko and Sébastien Tixeuil. Discovering network topology in the presence of byzantine faults. In *3th Colloquium on Structural Information and Communication Complexity (SIROCCO 2006), to appear*, July 2006.
5. Jay Yellen and Jonathan L. Gross. *Graph Theory & Its Applications*. CRC Press, 1998. ISBN: 0–849–33982–0.