Automated Stream-Based Analysis of Fault-Tolerance[⋆]

Scott D. Stoller[1] and Fred B. Schneider[2]

[1] Computer Science Dept., Indiana University, Bloomington, IN 47405, USA
`stoller@cs.indiana.edu`
[2] Dept. of Computer Science, Cornell University, Ithaca, NY 14853, USA
`fbs@cs.cornell.edu`

Abstract. A rigorous, automated approach to analyzing fault-tolerance of distributed systems is presented. The method is based on a stream model of computation that incorporates approximation mechanisms. One application is described: a protocol for fault-tolerant moving agents.

1 Introduction

As computers are integrated into systems having stringent fault-tolerance requirements, there is a growing need for practical techniques to establish that these requirements are satisfied. This paper describes such an analysis method. Automated analysis methods address an important need, because informal arguments do not supply the desired level of assurance for critical systems, and practitioners often lack the background needed to construct the formal proofs required by proof-based methods, such as those in [ORSvH95,CdR93,PJ94,JJ96,Sch96]. Automated verification techniques based on exhaustive exploration of finite state-spaces [CGL94,Hol91,Kur94,CS96] have made great progress in the last decade. But relatively little work has been done on automated verification of fault-tolerant software systems, partly because exhaustive search of the state-space of these systems is infeasible in many cases.

This paper discusses a specialized approach to analysis of fault-tolerance properties for distributed systems. It is a novel hybrid of ideas from stream-processing (or data-flow) models of networks of processes [Kah74,Bro87] and abstract interpretation of programs [AH87]. An important feature of our approach is its emphasis on communication (rather than state), consistent with the thesis that distributed systems have natural descriptions in terms of communication.

In stream-processing models, each component of a system is represented by an *input-output function* describing its input/output behavior. For simplicity, we assume processes communicate only by messages transmitted along unbounded FIFO channels. Behaviors of a system can be determined from input-output functions describing its components by doing a fixed-point calculation; this provides a clean algorithmic basis for our analysis.

The fixed-point calculation produces a graph, called a *message flow graph* (MFG), representing possible communication behaviors of the system. Each node of the graph corresponds to a component, and each edge is labeled with a description of the sequence of messages sent from the source node to the target

[⋆] This material is based on work supported in part by NSF/DARPA Grant No. CCR-9014363, NASA/DARPA grant NAG-2-893, and AFOSR grant F49620-94-1-0198. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these agencies.

node. An exact computation of all possible sequences of messages that might be sent is generally infeasible. To help make automated analysis feasible, our framework supports flexible and powerful approximations, or abstractions, as they are called in the literature on abstract interpretation [AH87]. Traditionally, stream-processing models have not incorporated approximations. The approximations in our framework enable compact representation of the highly non-deterministic behavior characteristic of severe failures and also support abstraction from irrelevant aspects of a system's failure-free behavior. The latter reflects a separation of concerns that is crucial for making the fault-tolerance analysis tractable.

A common approach to modeling failures is to treat them as events that occur non-deterministically during a computation (e.g., [CdR93,PJ94,LM94]), but this makes it difficult to separate the effects of failures from other aspects of the system's behavior and, consequently, to model the former more finely than the latter. In particular, one often wants to avoid case analysis corresponding to non-determinism in a system's failure-free behavior, while case analysis corresponding to different combinations of failures appears unavoidable in general in automated analysis of fault-tolerance. A *failure scenario* for a system is an assignment of component failures to a subset of the system's components. In our approach, each input-output function is parameterized by possible failures in the corresponding component; system behavior is analyzed separately for each failure scenario of interest.

In our framework, possible communications (in a given failure scenario) between two components are characterized by approximations of *values* (the data transmitted in messages), *multiplicities* (the number of times each value is sent), and *message orderings* (the order in which values are sent). Values and multiplicities are approximated using a form of abstract interpretation and a form of symbolic computation. Message orderings are approximated using partial (instead of total) orders.

Our analysis method is implemented in a prototype tool called CRAFT [Sto97]. We have used CRAFT to analyze a protocol for fault-tolerant moving agents and the Oral Messages algorithm for Byzantine Agreement [LSP82].

A formal presentation of our analysis method, including a semantics and a proof of soundness with respect to that semantics, appears in [Sto97]. A discussion of related and future work, including comparisons with the abstraction methods of [CGL94,Kur94], can also be found there.

2 Analysis Method

We start by describing how data is approximated in our framework and then how sets and sequences of messages are approximated. This leads directly to definitions of MFGs and input-output functions.

Values. As in abstract interpretation, we introduce a set *AVal* of *abstract values*. Each abstract value represents a set of concrete values. For example, we use abstract value $\mathbf{N}$ to represent the set of 64-bit numbers. In Section 3, we use abstract value $Arb(kcs, ms)$ to represent the concrete values that can be generated using encryption keys in the set kcs and ciphertexts in the set ms.

Abstract values alone capture too little information about relationships between concrete values. For example, consider a system containing a majority voter. The voter's outputs depend on equality relationships among its inputs. If two inputs both have abstract value $\mathbf{N}$, there is no way to tell from this whether they are equal. So, we introduce a set *SVal* of *symbolic values*, which are expressions composed of constants and variables. All occurrences of a symbol (i.e., a constant or variable) in a single MFG represent the same value. For example, if two inputs of a 3-way voter contain the same symbolic value, then that symbolic values represents the majority value and therefore represents the voter's output.

A *constant* represents the same value in every execution of a system; most constants are typeset in a sans-serif font. The meaning of a constant is specified by an *interpretation*, which maps constants to concrete values. A *variable* represents values that may be different in different executions of a system. Variables are useful for modeling outputs that are not completely determined by a component's inputs. Such outputs commonly arise with components that interact with an environment that is not modeled explicitly; they also arise when a component's behavior is approximated. Each variable is *local to* a single component, whose outputs in a given execution determine the value represented by that variable. Making each variable local to a single component enables independent verification that each input-output function faithfully represents the behavior of the corresponding process (as described in [Sto97]). We also include in *SVal* a special *wildcard* symbol "_", which is used when a value is not known to have any interesting relationships to other values. Different occurrences of the wildcard in a MFG do *not* necessarily represent the same concrete value.

A symbolic value and an abstract value together are often sufficient to characterize the possible data in a message. Analysis of a non-deterministic system might yield multiple such pairs, each representing some of the possibilities for the data in a message. So, we use a set of such pairs to represent values, and define $\mathit{Val} \triangleq \mathit{Set}(\mathit{SVal} \times \mathit{AVal}) \setminus \{\emptyset\}$, where $\mathit{Set}(S)$ is the powerset of a set S. Since abstract values are analogous to types, we usually write $\langle s, a\rangle \in \mathit{SVal} \times \mathit{AVal}$ as $s\!:\!a$. We usually omit braces around singleton sets; for example, $\{\langle X, \mathbf{N}\rangle\} \in \mathit{Val}$ may be written $X : \mathbf{N}$. Since a wildcard is similar in meaning to omission of a symbolic value, we usually elide the wildcard; for example, $\{\langle \_, \mathbf{N}\rangle\} \in \mathit{Val}$ may be written $\mathbf{N}$.

Multiplicities. Uncertainty in the number of messages sent during a computation may stem from various sources, including non-determinism of components (especially faulty components), non-determinism of message arrival order, and approximation of values. For example, a component subject to Byzantine failures[1] might emit outputs with an arbitrary multiplicity. To compactly represent these possibilities, multiplicity (i.e., the number of messages) also needs to be approximated. Thinking of multiplicities as natural numbers suggests representing them in the same way as data. Thus, we define $\mathit{Mul} \triangleq \mathit{Set}(\mathit{SVal} \times \mathit{AMul}) \setminus \{\emptyset\}$, where the set $\mathit{AMul} \subseteq \mathit{AVal}$ of *abstract multiplicities* contains abstract values whose meanings are subsets of the natural numbers, excluding $\emptyset$ and $\{0\}$.

The symbolic values in multiplicities are useful for efficient analysis of systems with crash failures [SS97]. Abstract multiplicities are analogous to superscripts in regular expressions. To promote the resemblance, we assume *AVal* contains the following: 1, denoting $\{1\}$; ?, denoting $\{0, 1\}$; +, denoting the set of positive natural numbers; and $*$, denoting the set of natural numbers. The notational conventions for *Val* also apply to *Mul*; for example, $\{\langle \_, *\rangle\} \in \mathit{Mul}$ may be written $*$.

[1] A *Byzantine failure* causes a component to exhibit arbitrary behavior.

Partially-ordered sets of messages. A set of messages is approximated in our framework by a *ms-atom* (mnemonic for "message-set atom"). Each ms-atom approximates a set of messages, using an element of Val to characterize the concrete values in the messages and an element of Mul to characterize the number of messages (i.e., the cardinality of the set). For example, a ms-atom with value $X : \mathbf{N}$ and multiplicity $*$ represents a set S of messages such that: (1) the concrete value in each message is an element of (the set represented by) $\mathbf{N}$ and is represented by variable X (hence all the messages in S contain the same concrete value), and (2) the number of messages in S is arbitrary (but finite). Similarly, a ms-atom with value $\_ : \mathbf{N}$ and multiplicity $*$ represents an arbitrary-sized set of messages, with each message containing a (possibly different) element of $\mathbf{N}$.

A sequence of messages is approximated in our framework by a partially-ordered set (abbreviated as "poset") of ms-atoms. A poset over a set A is a pair $\langle S, \prec \rangle$, where $S \subseteq A$ and $\prec$ is an irreflexive, transitive, and acyclic binary relation on S. For a poset $\langle S, \prec \rangle$ of ms-atoms, the meaning of the partial order is: if $x \prec y$, then during an execution of the system, the messages represented by x would be sent (and received, since channels are FIFO) before the messages represented by y. As a technicality, in order to allow multiple ms-atoms with the same value and multiplicity to appear in a poset, we include in each ms-atom a tag from the set Tag. In examples, we take Tag to be the natural numbers. Thus, the signature of ms-atoms is $MSA \triangleq Val \times Mul \times Tag$. To promote the resemblance to regular expressions, we usually write an ms-atom $\langle val, mul, 0 \rangle$ as val^{mul}; if the multiplicity mul is 1, we usually elide it.

Message Flow Graphs. A system comprises a set of named components, with names from the set $Name$. The signature $Hist$ of *histories* is $Hist \triangleq Name \rightarrow POSet(MSA)$, where $POSet(MSA)$ is the set of posets over MSA. When a history h is used to represent the inputs to a component x, $h(y)$ represents the sequence of messages sent by y to x; when a history h is used to represent as the outputs of a component x, $h(y)$ represents the sequence of messages sent by x to y. Possible behaviors of a system are represented by a MFG, which has signature $MFG \triangleq Name \rightarrow Hist$. A concrete MFG g is interpreted, by convention, as a labeled directed graph with nodes in $Name$ and with edge $\langle x, y \rangle$ labeled with $g(y)(x)$. Thus, by convention, $g(y)$ is the input history of component y in g.

Input-output Functions. Since a component's behavior depends on what failures it suffers, input-output functions are parameterized by the possible failures of the corresponding component. Let $Fail$ denote the set of all possible failures for the components of a system. For example, $Fail$ might contain an element Byz corresponding to Byzantine failures. By convention, $Fail$ contains an element OK corresponding to absence of failure. The behavior of a process is approximated by an input-output function with signature

$$IOF \triangleq \{f \in Fail \rightharpoonup (Hist \rightarrow Hist) \mid tagUniform(f)\}, \qquad (1)$$

where the one-hooked arrow indicates a partial function and $tagUniform(f)$ asserts that renaming of tags in the input ms-atoms causes no change in the output ms-atoms except possibly renaming of tags (this requirement is sensible because tags do not appear in actual messages). For $f \in IOF$, domain(f) is the set of failures that the component might suffer, and for each $fail \in$ domain(f), $f(fail)$ characterizes the component's behavior when failure $fail$ occurs. Specifically, $f(fail)$ maps a history h representing a component's inputs to a history

$f(fail)(h)$ representing that component's outputs on those inputs. A failure scenario is a function in $FS \triangleq Name \rightarrow Fail$ that maps each component to one of its possible failures.

A system is represented by a function $nf \in Name \rightarrow IOF$ ("nf" is mnemonic for "name to input-output function"). A MFG representing a system's behavior is computed using the function

$$step_{nf,fs}(g) \triangleq (\lambda y\!:\!Name.\ (\lambda x\!:\!Name.\ nf(x)(fs(x))(g(x))(y))). \tag{2}$$

Informally, $step_{nf,fs}(g)$ is the MFG representing the result of each component in system nf in failure scenario fs processing its inputs in the possibly-incomplete executions represented by MFG g and producing possibly-extended outputs. The behavior of a system nf in failure scenario fs is represented by the MFG $\mathrm{lfp}(step_{nf,fs})$, if it exists, where lfp indicates the least fixed-point, and the partial ordering on MFG, defined in [Sto97], corresponds informally to the prefix ordering on sequences. This fixed-point might not exist; one reason, roughly, is that MFGs do not have canonical forms [Sto97]. The tool searches for a fixed-point by starting with the "empty" MFG $(\lambda x\!:\!Name.\ \lambda y\!:\!Name.\ \langle\emptyset,\emptyset\rangle)$ and repeatedly applying $step_{nf,fs}$. If the fixed-point does not exist, this procedure diverges.[2]

Fault-Tolerance Requirements. A fault-tolerance requirement is expressed in our framework as a function b such that for each failure scenario fs, $b(fs)$ is a predicate on MFGs. A system satisfies fault-tolerance requirement b if, for each failure scenario, the MFG computed as a fixed-point of $step_{nf,fs}$ satisfies $b(fs)$.

3 Analysis of Fault-Tolerant Moving Agent Protocol

An interesting paradigm for programming distributed systems is *moving agents*. In this paradigm, an agent moves from site to site in a network. For example, an agent that starts at site S might move to site S_1 in order to access some service (e.g., a database) available there. The agent might then determine that it needs to access a service located at site S_2 and move there. If the agent has gathered all of the information it needs, it might finish by moving to a final site A to deliver the result of the computation. The sequence of sites visited by a moving agent is generally not known when the computation starts, since it may depend on information obtained as the computation proceeds.

Replicated Two-Stage Moving Agent. To illustrate the fault-tolerance problems that arise with moving agents, we consider a "two-stage" moving agent that visits two replicated services. The moving agent starts at a source S, accesses service F, which is replicated at sites F_1, F_2, F_3, and then accesses service G, which is replicated at sites G_1, G_2, G_3. Since G is the last service it needs, the agent moves to a consolidator B, which is responsible for delivering the result of the computation to the destination, or "actuator", A. The consolidator computes the majority of the values it receives and sends the result to the actuator; in addition, as discussed below, the consolidator tests validity of received values and excludes invalid values from the vote.

[2] The user can interrupt the calculation and, by inspection of a few MFGs in the sequence, try to determine the "cause" of the divergence.

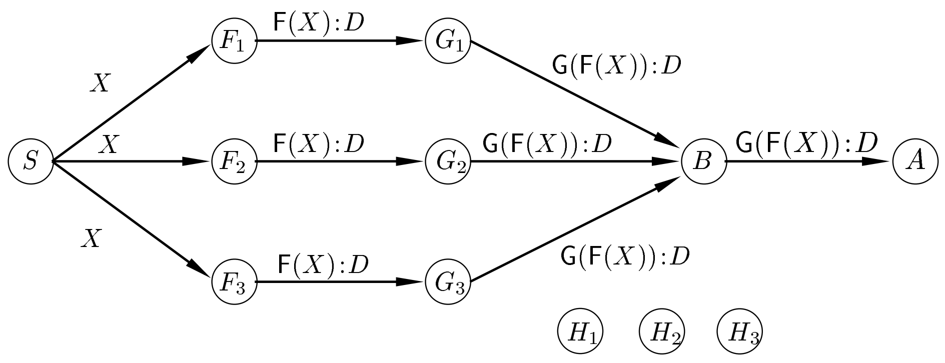

Fig. 1. MFG for replicated two-stage moving agent.

The failure-free behavior of this moving agent is represented by the MFG in Figure 1. Constants F and G represent the processing done by services F and G, respectively. A typical moving agent accesses only some of the available services. To reflect this, the system shown in Figure 1 includes a service H, replicated at sites H_1–H_3, which is not used by this particular agent. The fault-tolerance requirement is:

> **MA-FTR.** Inputs to the actuator should be unaffected by Byzantine failure of a minority of the replicas of each service used by the moving agent and by Byzantine failure of any number of replicas of each service not used by the moving agent.

Suppose faulty components can spoof (i.e., send messages that appear to be from other components) and eavesdrop (i.e., obtain copies of messages sent to other components). From the perspective of the recipient of a message, the possibility of spoofing causes uncertainty about the identity of the actual sender of the message. We model this uncertainty by using input-output functions that are independent of the names of the senders in the input history.

To eavesdrop on a component x, a faulty component (the "eavesdropper") sends a distinguished value *evsdrp* to x. The output history of a component that receives *evsdrp* must allow the possibility of sending copies of all subsequent outputs to the eavesdropper.[3] We assume that a faulty server is able to eavesdrop on all components except actuators.

Consider the consolidator B in Figure 1. How does it decide which inputs are valid? One might be tempted to say that B should treat messages from G_1–G_3 as valid and messages from other components as invalid. This criterion is inappropriate for moving agents, because it assumes B knows in advance that the last service visited by the moving agent will be service G; however, the sequence of services visited by a moving agent is generally not known in advance.

At the other extreme, suppose B considers all inputs valid: whenever B receives the same value from a majority of the replicas of some service, it sends that value to the actuator. (We assume every component knows which service is provided by each server.) It is easy to see that this scheme does not tolerate

[3] For this purpose, we allow an exception to the rule in the previous paragraph.

failure of a majority of the replicas of services (e.g., H_1–H_3) not used by the moving agent.

Informally, a message embodying a moving agent should be considered valid if it has visited the same sequence of services as the corresponding failure-free moving agent. We consider here a protocol in which digital signatures are used by the consolidator to determine validity. We assume digital signatures are implemented using public-key cryptography and that each component knows its own private key and the public key of every other component.

Each message sent by a source or server is signed and augmented with information about the sequence of services that should be visited. Each source or server includes in each outgoing message the name of the "destination", i.e., the next service or consolidator to be visited by the moving agent embodied in that message. A consolidator must verify the entire "history" of the moving agent (i.e., the entire sequence of visited services), so a server x also includes in the outgoing message the incoming message that embodied the arrival of that moving agent at x; by induction, that incoming message contains the history of the moving agent up to the arrival of the moving agent at x. The signatures on these recursively included messages provide a chain of evidence documenting the sequence of services actually visited by the moving agent.

A consolidator tests whether a message is valid by checking that it was originated by a (legitimate) source, that the consolidator itself is the declared destination of the message, and that the sequence of declared destinations (obtained from the included messages) are consistent with the chain of signatures on the included messages. Of course, the consolidator also verifies each of the signatures and considers the message invalid if any of those verifications fail. We say a set S of messages is valid if: (1) each message in S is valid; (2) all the messages in S contain the same sequence of declared destinations; (3) the final signatures on the messages in S are (collectively) from a majority of the replicas of some service. When the consolidator receives a valid set of messages, those messages should all contain the same data, which the consolidator forwards to the actuator.

To describe this protocol in our framework, we introduce some definitions. Let $D \in \mathit{AVal}$ be the "type" of data carried by moving agents. Let $\mathit{Svc} \subseteq \mathit{Con}$ denote the set of (names of) services that can be accessed by a moving agent. The processing done by a service $S \in \mathit{Svc}$ is represented by an operator $\mathsf{S} \in \mathit{Con}$, as in Figure 1. We assume component names can be used as constant symbols, i.e., that $\mathit{Name} \subseteq \mathit{Con}$. Let $\mathit{Src} \subseteq \mathit{Name}$ be the set of names of (legitimate) sources. For $x \in \mathit{Name}$, let $\mathsf{K}_x \in \mathit{KC}$ represents x's private key (used to sign messages); we assume each component's private key is unique. The set of constants representing private keys is $\mathit{KC} = \bigcup_{x \in \mathit{Name}} \mathsf{K}_x$.

To conveniently represent messages sent by sources, we introduce a constant $\mathsf{msg}_0 \in \mathit{Con}$ with the following interpretation: symbolic value $\mathsf{msg}_0(k, \mathit{data}, \mathit{dest})$ represents a message signed with key $k \in \mathit{KC}$, carrying data represented by symbolic value data, and with destination (either a service or the name of a consolidator) represented by symbolic value dest.

To conveniently represent messages sent by servers, we introduce a constant $\mathsf{msg} \in \mathit{Con}$ with the following interpretation: symbolic value $\mathsf{msg}(k, \mathit{data}, \mathit{dest}, \mathsf{msg})$ represents a message signed with key $k \in \mathit{KC}$, carrying data represented by symbolic value data, with destination (either a service or the name of a consolidator) represented by symbolic value dest, and with symbolic value msg representing a message that caused the server that received it to send this message.

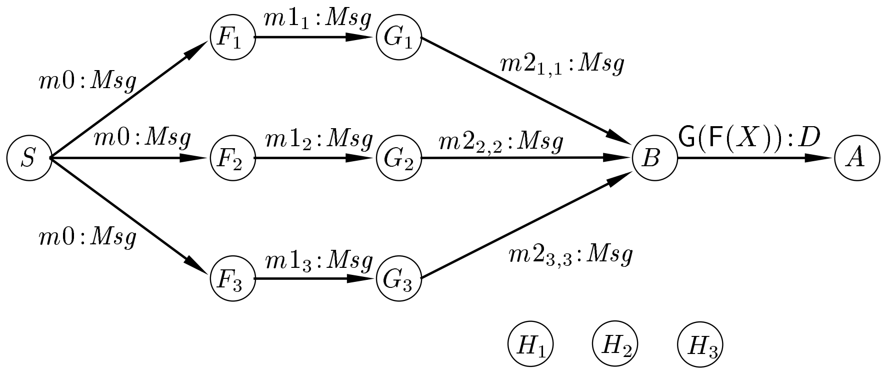

Fig. 2. Run of replicated two-stage moving agent, with authentication.

The MFG in Figure 2 shows the behavior of this protocol for the replicated two-stage moving agent described above, using the following abbreviations:

$$\begin{aligned} m0 &= \mathsf{msg_0}(\mathsf{K}_S, X, F) \\ m1_i &= \mathsf{msg}(\mathsf{K}_{F_i}, \mathsf{F}(X), G, m0) \\ m2_{i,j} &= \mathsf{msg}(\mathsf{K}_{G_j}, \mathsf{G}(\mathsf{F}(X)), B, m1_i). \end{aligned}$$

Tolerating Failure of Multiple Visited Services. The above protocol provides some fault-tolerance but does not satisfy MA-FTR. For example, the above protocol does not tolerate simultaneous failure of F_1 and G_2, because two of the consolidator's three inputs might be corrupted by these failures.

To make the moving agent more robust, each server sends its outgoing messages to *all* replicas of the next service, instead of just one, and validity tests and voting are incorporated into each stage of the computation after the first. The validity test and voting are as just described for consolidators.[4] Thus, a server sends messages only after receiving a valid set S of messages; to document the sequence of services visited by the moving agent, the server includes some message from S in the outgoing messages.[5] The behavior of the revised protocol is shown in Figure 3. Each server G_j might include any one of its three input messages in its output, so the value in its outputs is a set of three possibilities; specifically, the value is $(ms_j \times \{Msg\}) \in Val$, where

$$ms_j = \{m2_{1,j}, m2_{2,j}, m2_{3,j}\}.$$

Detailed input-output functions for this protocol appear in [Sto97].

3.1 Analysis Results

To determine whether the above protocol satisfies MA-FTR, an MFG representing the protocol's behavior is computed for each failure scenario in which a

[4] The only remaining differences between a server and a consolidator are: (1) a consolidator does not perform application-specific computation, i.e., does not apply an operator to the data carried by the moving agent; (2) a consolidator does not include authentication information in its outputs, because the channel between the consolidator and the actuator is assumed to be secure.

[5] The reader who wonders whether multiple messages from S should be included in the outgoing messages is referred to the comments in Section 3.1.

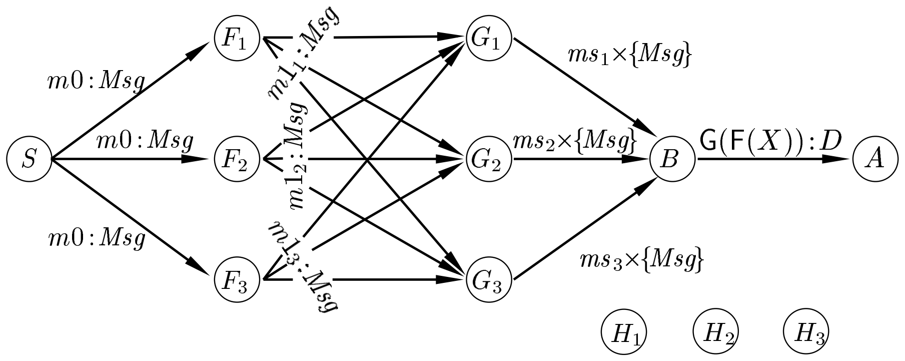

Fig. 3. Run of replicated two-stage moving agent, with authentication and with voting after each stage. Each skewed ms-atom labels each of the three edges it crosses.

minority of the replicas of each service used by the moving agent fail and any number of replicas of each service not used by the moving agent fail. MA-FTR is satisfied if, in each of these failure scenarios, the input to the actuator is represented by $\mathsf{G}(\mathsf{F}(X)):D$, as in the failure-free computation. We describe below the MFGs obtained for a few representative failure scenarios.

Failure of Visited Servers Only. Consider the failure scenario in which F_1 and G_2 fail. Let $N = \{F_1, F_2, F_3, G_1, G_2, G_3, H_1, H_2, H_3, B\}$. The fixed-point computed for this failure scenario is the same MFG as in Figure 3, except that the outputs of the faulty components are different, and other components send messages to the faulty components as a result of eavesdropping. Specifically, for $x \in \{F_1, G_2\}$, for $y \in N \setminus \{x\}$, edge $\langle x, y\rangle$ is labeled with the ms-atom

$$\{evsdrp, Arb(\{\mathsf{K}_{F_1}, \mathsf{K}_{G_2}\}, \{m0, m1_2, m1_3, m2_{2,1}, m2_{3,1}, m2_{2,3}, m2_{3,3}\})\}^*.$$

Arb was described in Section 2. Also, for $x \in \{F_1, G_2\}$ and $y \in N \setminus \{F_1, G_2\}$, edge $\langle y, x\rangle$ is labeled with all the output ms-atoms of component y in Figure 3, but with the multiplicities changed to ?.

Failure of Unvisited Servers Only. Consider the failure scenario in which H_1, H_2, and H_3 fail. The fixed-point computed for this failure scenario is the same MFG as in Figure 3, except that the outputs of the faulty components are different, and other components send messages to the faulty components as a result of eavesdropping. Specifically, for $x \in \{H_1, H_2, H_3\}$, for $y \in N \setminus \{x\}$, edge $\langle x, y\rangle$ is labeled with the ms-atom

$$\{evsdrp, Arb(\{\mathsf{K}_{H_1}, \mathsf{K}_{H_2}, \mathsf{K}_{H_3}\}, \bigcup_{i,j\in\{1,2,3\}} \{m0, m1_i, m2_{i,j}\})\}^*.$$

Also, for $x \in \{H_1, H_2, H_3\}$ and $y \in N \setminus \{H_1, H_2, H_3\}$, edge $\langle y, x\rangle$ is labeled with all the output ms-atoms of component y in Figure 3, but with the multiplicities changed to ?.

Failure of Visited and Unvisited Servers. Consider the failure scenario in which F_1, H_1, and H_2 fail. As the reader may have suspected, the protocol violates

MA-FTR in this failure scenario. Tracing the first three iterations of the fixed-point computation shows why. Due to space limitations, we omit those MFGs and describe the behavior informally. F_1 includes $m0$ in a signed message m' with declared destination H and carrying arbitrary data and sends m' to H_1 and H_2, who each include m' in a signed message with declared destination B and carrying the same (but otherwise arbitrary) data and send that message to B. These two messages cause the consolidatorto send arbitrary data to the actuator. One way to fix the protocol is to have servers include in each output message input messages from a majority of the replicas of some service. The analysis of the corrected protocol is similar to the analysis sketched here.

References

[AH87] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis-Horwood, 1987.

[Bro87] M. Broy. Semantics of finite and infinite networks of concurrent communicating agents. *Distributed Computing*, 2(1):13–31, 1987.

[CdR93] A. Cau and W.-P. de Roever. Using relative refinement for fault tolerance. In *Proc. 1st Intl. Symposium of Formal Methods Europe*, pages 19–41, 1993.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. on Prog. Lang. and Sys.*, 16(5):1512–1542, 1994.

[CS96] R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In *Proc. CAV '96*, volume 1102 of *LNCS*, pages 394–397. Springer-Verlag, 1996.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[JJ96] T. Janowski and M. Joseph. Dynamic scheduling in the presence of faults: Specification and verification. In *Proc. FTRTFT '96*, volume 1135 of *LNCS*, pages 279–297. Springer-Verlag, 1996.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing 74: Proc. IFIP Congress 74*, pages 471–475. North-Holland, 1974.

[Kur94] R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.

[LM94] L. Lamport and S. Merz. Specifying and verifying fault-tolerant systems. In *Proc. Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *LNCS*, pages 41–76. Springer-Verlag, 1994.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. on Prog. Languages and Systems*, 4(3):382–401, July 1982.

[MvRSS96] Y. Minsky, R. van Renesse, F. B. Schneider, and S. D. Stoller. Cryptographic support for fault-tolerant distributed computing. In *Proc. Seventh ACM SIGOPS European Workshop*, pages 109–114. ACM Press, 1996.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[PJ94] D. Peled and M. Joseph. A compositional framework for fault-tolerance by specification transformation. *Theoretical Computer Science*, 128(1-2):99–125, 1994.

[Sch96] H. Schepers. Real-time systems and fault-tolerance. In M. Joseph, editor, *Mathematics of Dependable Systems*, chapter 7. Prentice-Hall, 1996.

[SS97] S. D. Stoller and F. B. Schneider. Automated analysis of fault-tolerance in distributed systems. In *Proc. First ACM SIGPLAN Workshop on Automated Analysis of Software*, pages 33–44, 1997. Available via http://www.cs.indiana.edu/~stoller/ .

[Sto97] S. D. Stoller. *A Method and Tool for Analyzing Fault-Tolerance in Systems*. PhD thesis, Cornell University, May 1997. Available via http://www.cs.indiana.edu/~stoller/ .