Automated Monitor Based Diagnosis in Distributed Systems

Abstract

In today's world where distributed systems form many of our critical infrastructures, dependability outages are becoming increasingly common. In many situations, it is necessary to not just detect a failure, but also to diagnose the failure, i.e., to identify the source of the failure. Diagnosis is challenging since high throughput applications with frequent interactions between the different components allow fast error propagation. It is desirable to consider applications as black-boxes for the diagnosis process. In this paper, we propose a Monitor architecture for diagnosing failures in large-scale network protocols. The Monitor only observes the message exchanges between the protocol entities (PEs) remotely and does not access internal protocol state. At runtime, it builds a causal graph between the PEs based on their communication and uses this together with a rule base of allowed state transition paths to diagnose the failure. The hierarchical Monitor framework allows distributed diagnosis tolerating Byzantine failures at individual Monitors. The framework is implemented and applied to a reliable multicast protocol executing on our campus-wide network. Fault injection experiments are carried out to evaluate the accuracy and latency of the diagnosis.

Keywords: Distributed system diagnosis, runtime monitoring, hierarchical Monitor system, fault injection based evaluation.

1 Introduction

The wide deployment of high-speed computer networks has made distributed systems ubiquitous in today's connected world. These form the backbone of much of the information technology infrastructure of the world today. The infrastructure, however, is increasingly facing the challenge of dependability outages resulting from both accidental failures and malicious failures, like security attacks, collectively referred to as *failures* in this paper. The potential causes of accidental failures are hardware failures, software defects, and operator failures, including mis-configurations, while the malicious attacks may be launched by external or internal users. The consequences of downtime of distributed systems are catastrophic. An extent of the financial losses can be gauged from a survey by Meta Group Inc. of 21 industrial sectors in 2000 [1], which found the mean loss of revenue due to an hour of computer system downtime to be $1.01M. Compare this to the average cost of $205 per hour of employee downtime! Also, compare the cost today to the average of $82,500 in 1993 [2] and the trend becomes clear.

In order to build robust infrastructures capable of tolerating the two classes of failures, it is required to provide detection and diagnosis primitives as part of a fault tolerance infrastructure. Following the definitions in [27], a fault is an invalid state or bug underlying in the system, which when triggered becomes an error. An external manifestation of this error leads to a failure. Error propagation might take place in a distributed system and a resulting failure can be flagged by the detection system. The role of the diagnosis system is to identify the entity that originated the failure. We take the approach of structuring the combined system into two clearly segmented parts with well-defined mutual interactions – an observer or monitor system, which provides detection and diagnosis, and an observed or payload system, which comprises the protocol entities (PEs). This paper builds diagnosis functionality on an existing detection framework presented in [4]. The diagnosis problem is significant in distributed applications that have closely interacting PEs, which allows for error propagation. This may lead to the failure being detected at an entity distant from the entity that originated the cascaded chain of errors.

There are certain design motivations for the monitor system both for detection and diagnosis. First, it is desirable that the monitor system operate asynchronously to the payload system so that its throughput does not suffer due to the checking overhead. This requirement imposes the requirement of fast detection and diagnosis, so that substantial damage due to cascaded failures is avoided. Third, it is desirable that the monitor system not be intrusive to the payload system. This rules out the possibility making changes to the PEs or creating special tests that they respond to, and argues in favor of having the payload system be viewed as a black-box by the monitor system. While it is possible to build very optimized and specialized mechanisms for specific applications (example of the approach applied to the reliable multicast protocol used here called TRAM [3]), such solutions typically do not generalize very well across applications. Thus, it is important to design the monitor system to have an application neutral architecture. Related to this, is the requirement that the monitor system be easily deployable to a new application.

The above requirements are challenging to meet in many of today's distributed applications. The machines on which the applications are hosted are heterogeneous in nature, the applications often run legacy code without the availability of their source code, the systems are of very large scales (e.g., of the order of tens of thousands of protocol participants for a system with DNS clients and servers), and the systems often have soft real-time guarantees. In this paper, we propose a generic Monitor architecture to provide diagnosis primitives to distributed applications, meeting all the design requirements mentioned above.

We use a hierarchical Monitor architecture to perform diagnosis of failures in the underlying protocol. The Monitor snoops on the communication between the PEs and performs diagnosis of the faulty PE once a failure is detected. We use the terminology “the Monitor *verifies* a PE” to represent the detection and the diagnosis process. The Monitor can only use externally observable message exchanges as input and *not* the internal state of any of the PEs. The relevant state is deduced and matched against rulebase for diagnosis. The Monitor architecture is generic and applicable to a large class of message passing based distributed applications, and it is the specification of the rule base that makes the Monitor specialized for an application. Once a detection alarm is raised by a Monitor, the diagnosis protocol starts executing. For the diagnosis, the Monitors treat the PEs as black-box and only the causal relation amongst the messages deduced from the send-receive ordering along with the formal protocol specification are used to perform the diagnosis. For the diagnosis, the PEs are not exercised with additional tests since that would make the Monitor system more invasive to the application protocol. Instead state that has already been collected by the Monitors during normal operation is used for the diagnosis process. The Monitors coordinate to perform distributed diagnosis if PEs lie under different Monitors’ verification domains. No assumptions are made on the communication channel between the protocol participants or on the clocks at the different Monitors. We assume Byzantine failures may occur in the Monitor system as well and use replication to mask them. The replicated Monitors use atomic multicast to achieve ordering on the alarm messages. The traditional replication degree bound of 3f+1 for atomic multicast is reduced by enforcing a hybrid failure model by the use an existing distributed security kernel called the Trusted Timely Computing Base (TTCB) [15], which embeds a small fail-silent component into each node hosting a redundant Monitor.

There are other research efforts that have provided the separation into an observer and an observed system and performed online monitoring ([5][6][7]). However, their goal has been restricted to the detection of the failure. The anomaly detection systems that provide a mechanism for local response, such as terminating the connection from a suspect IP address, make the implicit assumption that the root cause of the failure was the entity at which the detection was achieved [8], thus ruling out error propagation. The more traditional approach to diagnosis in distributed systems does not have the separation into the observer and the observed ([10][11] for review). The diagnosis uses tests which are executed by each PE on other entities and appears to require intrusive changes to the protocol and devising special (and oftentimes perfect) test sequences.

The Monitor system is implemented and deployed on our university’s campus-wide network and used to provide detection and diagnosis functionality to a streaming video application running over a reliable multicast application called TRAM [12]. Latency and accuracy of diagnosis are measured, through performing fault injection experiments. The Monitor accuracy is found to decrease with increasing data rate using a pessimistic version of the matching algorithm. However, switching to an optimistic version gives improved diagnosis accuracy of 85% at 175 KB/s compared to 63% in the pessimistic case. However, this comes at the cost of higher latency of diagnosis.

The paper makes the following contributions:

1. It provides a distributed protocol for accurate diagnosis of arbitrary failures. The diagnosis protocol is asymptotically optimal with increasing number of PEs.
2. It maintains a useful abstraction of the observer and the observed systems with non-intrusive interactions between them.
3. In the Monitor system, diagnosis can be achieved in the face of Byzantine failures in the fault tolerance framework itself and error propagation across the entire payload system.
4. The system is demonstrated on a real-world third-party application and on a real-world production network. The evaluation covers both the performance and the fault tolerance aspects of the diagnosis framework.

The rest of the paper is organized as follows. Section 2 presents the diagnosis protocol for the PEs assuming failure free Monitor hierarchy. Section 3 deals with how Monitor failures are handled. Section 4 presents the analysis of diagnosis accuracy. Section 5 presents the implementation, the experiments, and the results. Section 6 reviews related work and Section 7 concludes the paper.

2 Diagnosing Failures

This section details the diagnosis protocol that is executed in the system to determine the cause of the failure. We assume in this section that diagnosis is performed by a failure free Monitor hierarchy verifying the PE but explain in Section 3 how diagnosis is handled in case of failures in Monitors.

2.1 System Model

The Monitor preserves state across the messages and employs a stateful model for rule matching to perform detection and diagnosis. The Monitor contains rule-base which consists of combinatorial (valid for all points in time in the lifetime of the application) or temporal (valid for limited time periods) rules. Monitor observes only the external messages of the PEs and can be placed anywhere in the network but would not be co-hosted with the PEs because it can add performance constraints on PEs. Smaller latency of detection and diagnosis drives placement of the Monitor in the vicinity of the PEs. The Monitor architecture consists of Data Capturer, Rule Matching Engine, State Maintainer, and Decision Maker. The Data Capturer snoops over the communication medium to obtain messages. It can be implemented using *active forwarding* by the PEs to the Monitor or by a *passive snooping* mechanism. In passive snooping the Monitor captures the communication over the channel without any cooperation from the PE, e.g., through the promiscuous mode in a LAN or using router support. The message exchanges correspond to events in the rule base of the Monitor. The Rule Matching engine is used to match the incoming events with rules for those events in the rule base. The State Maintainer maintains the state transition diagram (STD) and the current state of each verified PE. Finally, the Decision Maker is responsible for making decisions based on the outcome from the Rule Matching Engine. This architecture is derived from the Monitor architecture in [4].

The system comprises multiple Monitors logically organized into Local, Intermediate, and Global Monitors. The *Local Monitors* (LMs) directly observe and verify the PEs. An *Intermediate Monitor* (IM) collects information from several Local Monitors. An LM performs filtering of packets and sends only the processed information, such as aggregated count of messages of a type, to the IM. There may be multiple levels of IMs depending on the number of PEs, their geographical spread, and the capacity of the host on which an IM is executing. There is only a single Global Monitor (GM), which has a global view of the protocol and only verifies the overall properties of the network. An example of the hierarchical setup with a single level of IM used in our experiments is shown in Figure 3.

The Monitor's functionality of verification and diagnosis is completely asynchronous to the protocol. Each Monitor maintains a local logical clock (*LC*) for each PE it is verifying, which it updates at each observable event (send or receive) for that PE. The logical clock used is similar to Lamport's clock with the events of importance being message send or receive. We assume that the jitter on the PE→Monitor link for any particular PE verified by a Monitor is bounded by Δt in logical clock time. This implies if two consecutive events happen at a PE, they cannot be observed at the Monitor more than Δt apart in logical clock time. This bound is referred to as *phase* in the rest of the paper. This assumption is much weaker than synchrony. We do not assume any bound on the clock drift.

We assume that PEs can fail arbitrarily exhibiting a Byzantine failure. Errors can propagate from one PE to another through the messages which are exchanged between them. Failures in the PEs are detected by the Monitor infrastructure by comparing the observed message exchanges against the *normal* rule base as opposed to the *strict rule base* used during diagnosis (Section 2.2.4). An anomaly in the behavior of the PEs detected by flagging of a rule triggers the diagnosis procedure.

2.2 Diagnosis Protocol

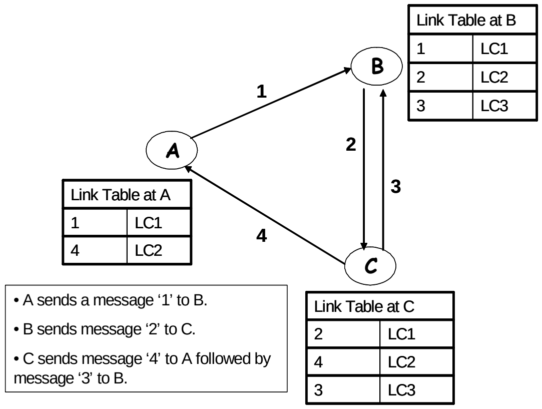
Diagnosis in a distributed manner based on observing only external message exchanges poses significant challenges. It is essential while formulating a diagnosis protocol to consider the phenomenon of propagated errors to avoid penalizing a correct node in which the failure first manifested as a deviation from the normal protocol behavior. As the Monitor has access only to external message exchanges and not to internal state, any diagnosis has to be done based only on the messages captured by the Data Capturer. PEs exchanging messages may lie within the domains of different LMs. Hence the diagnosis itself has to be a distributed effort spanning multiple Monitors over different levels. In order to identify the faulty node from among a set of suspect nodes, each node is subjected to a *test* procedure. However, it is desirable that the Monitor testing be non-intrusive and the Monitor treat each PE as a black–box. Hence the Monitor is not aware of the valid request-response for the protocol and cannot send any explicit *test* message to the PEs. Moreover, the PE may not currently be in the same state as the one in which the fault was triggered. A simple request-response test will therefore not work since it will not reconstruct the old PE state. These constraints drive the design of the diagnosis protocol for our hierarchical Monitor infrastructure. A failure manifested at the PE could be because of a fault which originated at this PE or because of error propagation through a message which the PE received. If the error is propagated through a message then it must causally precede the message which resulted in failure detection. As explained in Section 2.1, the Monitor maintains a logical clock for each PE which provides causal order to the messages for each PE as seen by the Monitor. A causal graph is used to maintain this order.

2.2.1 Causal Graph

The causal graph is updated during the normal operation of the protocol. A causal graph at a Monitor *m* is denoted by CG_m and is a graph (V, E) where

(i) V contains all the PEs verified by *m*.
(ii) An edge *e* contained in E, between vertices *v1* and *v2* indicates interaction between *v1* and *v2* and contains state about all observed message exchanges between the corresponding PEs. The edges are directed, and are stored separately as incoming and outgoing, with respect to a given node. The edges shall be referred to as *links* from now on.

A logical clock *LC* is maintained independently for every PE at the Monitor and is incremented for every observable event at that PE, i.e., a packet send or a packet reception. The value of *LC* at the instant of sending or receiving a packet is stored in the link in the causal graph. The other information maintained for each edge is the event and state in which that event took place. The links are also time-stamped with the local (physical) time at the Monitor, at which the link is created.



Link Table at B	
1	LC1
2	LC2
3	LC3

Link Table at A	
1	LC1
4	LC2

Link Table at C	
2	LC1
4	LC2
3	LC3

- A sends a message '1' to B.
- B sends message '2' to C.
- C sends message '4' to A followed by message '3' to B.

Figure 1: A sample causal graph

An example of a causal graph is given in Figure 1 for the sequence of events described on the lower left corner. For example in the Link Table for node C, message '4' is assigned a logical clock time 2. Message "3" is causally preceded by message "2" which is causally preceded by message "1". If message "3" causes a failure and hence an alarm to be flagged, the error could have been in entity *C* or it may have propagated through the causally preceding message "2", sent by entity *B*. It is important to note that since the underlying system is not synchronous, messages observed by the Monitor might not be in the same order as that generated by the PEs.

2.2.2 Cycle and Phase

In modern distributed protocols, there could be thousands of communicating protocol entities. If the diagnosis algorithm tries to test using every causally preceding message for the entire system, it would not be scalable or provide real time results. We define a time window over which the diagnosis protocol tests nodes. The start point of the time window as the *Cycle*, namely, how far should a diagnosis algorithm go in history to detect faulty nodes. Cycle boundaries can be decided either by using the STD of the application or error latency of the application in actual physical time or logical time. First, we present the definition using the STD.

In the Monitor design, a transition from one state to the next state depends solely on the current state and the event that occurs in the current state. This in fact limits the types of applications that can be handled by the Monitor system, namely, those that can be represented as a Discrete Time Markov Chain (DTMC). Let there be *n* PEs verified by the Monitor infrastructure. A reduced STD is maintained at the LM for every verified PE_k, denoted as STD_k. Owing to the reduced and finite nature of the STD, it can be assumed that there are repetitions in the set of states traversed by a PE over a long enough time interval.

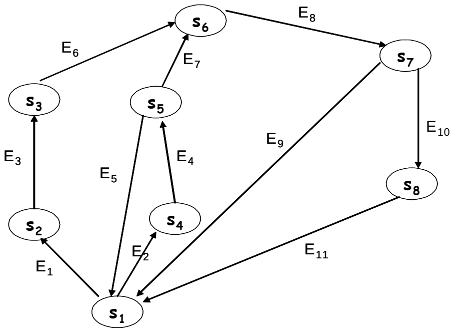


Figure 2: Sample STD for a PE, illustration of *Cycle*

Let the STD for PE P_1 be represented by Figure 2 and let S_1 be the starting state. P_1 initially starts out at state S_1, and returns to S_1 after a finite number of transitions. The collection of transitions owing to the ordered sequence of events is referred to as a *run*. In Figure 2, events $\{E_2, E_4, E_5\}$ and $\{E_1, E_3, E_6, E_8, E_9\}$ constitute two of several possible runs for P_1. The durations of runs for different PEs as well as for the same PE are different. Let S_{1k} denote the starting state of the PE *k* being verified. At an arbitrary starting time t_0, the states of the *n* PEs would be $\{S_{11}, S_{12}, S_{13}\ldots S_{1n}\}$. Let time t_1 be the minimum time at which all the PEs have completed one run. Without loss of generality, assume that P_1 completes its run at t_1. The future transitions of P_1 depend entirely on the current state and events that occur in the current state. Hence there is no way the system can differentiate between the state of P_1 at t_0 and that at t_1. A possible scenario is that each PE has the property that a fault is going to be manifested within a run, if at all. Thus, when a cycle has elapsed, the history of the previous cycle need not be maintained, and the corresponding entries in the causal graph can be flushed.

One can also use the error detection latency in the system (e.g., [25]) to come up with the cycle boundary. If we can provide a bound that any error in the application will manifest in δ seconds, we can limit the messages which need to be checked for errors. If the application does not offer any such property then the point for flushing the state has to be determined using heuristics based on the amount of state storage available at the Monitor and the tolerable diagnosis latency. This may cause a loss of diagnosis accuracy. The cycle also serves as a garbage collection point for state in the Monitor.

Let us consider two links in the causal graph *L* that have been time-stamped with logical times t_{L1} *and* t_{L2} by the Monitor. Given $t_{L2} > t_{L1}$ we cannot conclude anything about the actual order of these events. This is because the causal graph is constructed based on events seen by the Monitor, and these events may not be seen in the order in which they actually occurred. Since there is no synchrony assumption on the underlying system, hence a PE *v* sending two messages to PE *w* can result in the messages being received out of order at *w,* or being received in order at *w,* but out of order at the Monitor. We have a more relaxed assumption about the communication channel. Consider that a Monitor *M* is verifying two PEs – sender *S* and receiver *R*. The assumption required by the diagnosis protocol is that the variation in the latency on the *S*-*M* channel as well as the variation in the sum of the latency in the *S-R* and *R-M* channels is going to be less than a constant Δt, called the *phase*, which is known *a priori*. Thus, if two messages M_1 and M_2, corresponding to two send events at *S*, are received at Monitor M_1 at (logical) times t_1 and t_2, by the assumed property of the *S-M* channel, it is guaranteed that send event M_1 happened before M_2 if $t_{L2} \geq t_{L1}+\Delta t$. Phase also determines the logical time window over which a diagnosis rule can operate.

2.2.3 Suspicion Set

Flagging of a rule corresponding to a PE represented by node *N* in the causal graph indicates a failure *F* in the protocol and starts the diagnosis procedure. Henceforth, we will use the expression "failure at node *N*" for a failure manifested at the PE corresponding to the causal graph node *N*. Diagnosis is always started with the node where the rule is initially flagged, proceeding to other nodes *suspected* in connection with the failure at node *N*. All such nodes along with the link information (i.e. state and event type) form a *Suspicion Set* for failure *F* at node *N* denoted as SS_{FN}.

The suspicion set of a node N consists of all the nodes which have sent it messages in the past denoted by SS_N. If a failure is detected at node N then initially $SS_{FN}=\{SS_N\}$. Assume SS_N consists of nodes $\{n_1, n_2\ldots..n_k\}$. Each of the nodes in SS_{FN} is tested using a test procedure which is discussed in Section 2.2.4. If none of the nodes is found to be faulty then the suspicion set for the failure F is expanded to include the suspicion set of all the nodes contained in SS_N. Thus, in the next iteration $SS_{FN} = \{SS_{n_1}, SS_{n_2}\ldots, SS_{n_k}\}$. Arriving at the set of nodes that have sent messages to *N* in this time window can be done easily from the causal graph. Consider that the packet that triggered diagnosis is sent by *N* at time τ_S. Then, all the senders of all incoming links into node *N* with time-stamp *t* satisfying $C \leq t \leq \tau_S+\Delta t$ are added to the suspicion list, where Δt is the phase parameter and *C* is the cycle boundary. The procedure of contracting and expanding the Suspicion Set repeats recursively until the faulty node is identified or the cycle boundary is reached thereby terminating the diagnosis.

2.2.4 Test Procedure

The diagnosis protocol uses the already formed causal graph to arrive at decisions and adheres to the initial assumption of the PEs being unaware of the Monitors and the Monitors being unaware of the internal operations of PEs. To achieve this, we define the test procedure for a PE to be a set of rules that are to be matched based on the state of the PE as maintained by the Monitor in the causal graph. This set of rules is called the *strict rule base (SRB)* and like the *normal rule base* consists of temporal and combinatorial rules. The SRB is a detailed set of rules highlighting basic protocol behavior in terms of cause and effect relations between message exchanges in the PEs. The SRB is based on the intuition that a violation does not deterministically lead to a violation of the protocol correctness, and in many cases gets masked. However, in the case of a fault being manifested through the violation of a rule in the normal rule base as a failure, a violation of a rule in the SRB is regarded as a contributory factor. The strict rules are of form – <*Type*> <*State*$_1$> <*Event*$_1$> <*Count*$_1$> <*State*$_2$> <*Event*$_2$> <*Count*$_2$>where, *(State*$_1$*, Event*$_1$*, Count*$_1$*)* forms the precondition to be matched, while *(State*$_2$*, Event*$_2$*, Count*$_2$*)* forms the post-condition that should be satisfied for the node to be deemed *not* faulty. A condition consisting of <state *S*, event *E*, count *C*> refers to the fact that the event *E* should have been detected in the state *S* at least count *C* number of times. Note that a node may appear multiple times in the Suspicion Set, e.g., in different states, and may be checked multiple times during the diagnosis procedure. Also, the tests are run on state maintained at the Monitor without involving the PE, thus satisfying the design goal mentioned earlier.

When an SRB rule is used to test a given link in the causal graph, it operates over a logical window of Δt, the phase, on either side of the link's logical time. This is because the faulty packet may be displaced from the link under investigation by as much as Δt due to the assumption about the jitter in the underlying system. The rules in SRB help in diagnosis but are not considered to be perfect since a given rule only checks for a specific kind of event around the link under investigation. Therefore, a message sent by an entity in the Suspicion Set is tested by running multiple rules from the SRB on it. Consider the simple case that flagging by any of the rules results in a diagnosis. Then the probability of diagnosis becomes $1\text{-}(1\text{-}c)^T$, where *T* is the total number of tests and *c* is the coverage of each test. This quantity monotonically increases with *T*. The complete analysis is shown in Section 4. Like the normal rule base, the rules in the SRB are dependent on the state and the event of the link. The number of rules in the SRB is typically much larger than that in the normal rule base. Hence, it is conceivable that the system administrator would not tolerate the overhead of checking against the SRB during normal protocol operation .

A new diagnosis procedure is started for every rule that is flagged at the Monitor. Multiple faulty nodes exhibiting uncorrelated faulty behavior nearly concurrently would result in multiple rules being flagged, leading to separate, and independent diagnosis procedures for each of them. Conversely, a single fault may lead to multiple failures and hence multiple diagnoses, each detecting a single node as being faulty, would execute. If there are multiple faults which lead to a single failure then the diagnosis protocol stops after detecting a single faulty PE.

2.2.5 Diagnosis Protocol: Flow

This section illustrates the flow of control of the diagnosis protocol and the interactions in the Monitor infrastructure to arrive at a correct diagnosis. We illustrate the set of steps for a failure at a single PE. The protocol for distributed diagnosis amongst the Monitors comes into play when a suspect node identified by an LM lies outside its domain, i.e. the PE required to be tested is not verified by *this* LM. The LM does not contain the causal graph information for the suspect node, and hence requests the corresponding LM verifying the suspect node to carry out the test (step 4).

1. A failure *F* at PE *N* is detected by the local Monitor LM_1 verifying it.
2. LM_1 constructs the suspicion set SS_N for the failure and adds it to SS_{FN}.
3. For every $N' \in SS_N$ that belongs to the domain of LM_1 , LM_1 *tests* N' for correctness for the suspect link L′ using rules from the SRB for that particular event and state. If N' is not faulty, then it is removed from SS_N and SS_N' is added to the SS_{FN} queue.
4. For every N'' belonging to SS_N that is not under the domain of *M* but under the domain of another Monitor *M'*, *M* sends a *test request* for N'' and faulty link L″ recursively to higher level Monitors till a common parent for *M* and *M'* is found, which routes it to *M'*. *M'* tests N'' and sends the result of the test back to *M* through the same route. If N'' is not faulty, then *M'* also sends the suspicion set corresponding to link L″ for N''.
5. The diagnosis procedure repeats recursively till a node is diagnosed as faulty, or till the cycle boundary is reached.

3 Diagnosis in the Presence of Faulty Monitors

An external fault-free "oracle" performing detection and diagnosis although desirable, is not realistic. In our framework, the Monitors are also considered susceptible to faults. The goal of this section is to show how the diagnosis of faulty PEs can be carried out in face of arbitrary failures of the Monitors. We assume Monitors are susceptible to runtime Byzantine failures due to environmental errors like misconfigurations, transient errors and runtime synchronization errors to name a few. The set of failures that can be tolerated is identical to the set that can be tolerated using traditional replication, *without* the assumption of design diversity in the replicas.

3.1 Faults at Local Monitors

If LM is faulty then it may exhibit arbitrary behavior by sending false alarms to higher level Monitors or may drop a valid alarm. In such scenarios an LM cannot be allowed to perform the diagnosis procedure. We use replication to mask failures at the LMs, by allowing multiple LMs to verify a PE.

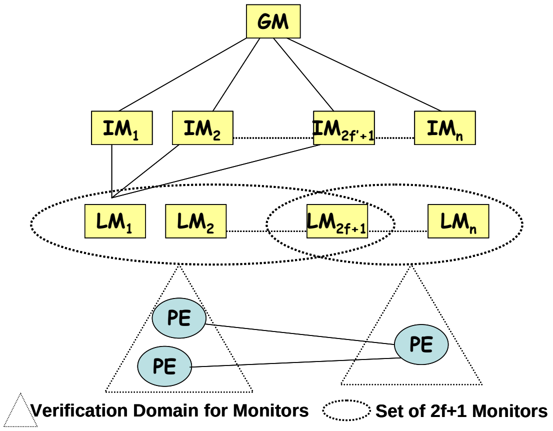


Figure 3: Redundancy in the Monitor hierarchy

Assuming there can be up to *f* LM failures, each PE is verified by *2f+1* LMs, called the Collaborative LM Set (denoted CS_{LM}). An IM can accept that there is an error in the PE being monitored, if it receives *f+1* identical alarms from the different LMs verifying the same PE. Note that if there is a set of entities (the LMs) whose responses are "voted on" by a fault-free "oracle" (the IM), then only *2f+1* entities are required under the Byzantine failure model. The communication between LMs and IMs is authenticated, to avoid multiple alarms being sent by the same LM. Although all the LMs in the CS_{LM} verify the same PE, they are spatially disjoint leading to possibly different views of the state of the PE. However, for our system, we need that all correct LMs in a CS_{LM} agree on the failure alarms they send to the IM. Another requirement is defining an order among the alarms sent out by the LMs in a CS_{LM}.

The solution to both issues is based on an *atomic or total order multicast protocol* (see definition in [13]) This problem is known to be equivalent to consensus [18], which requires a minimum of *3f+1* process replicas to be solvable in asynchronous systems with Byzantine faults [14]. We reduce this number of LM replicas to *2f+1* using an existing approach called the architectural hybrid model [17]. The algorithm is the following. When the Monitor is initialized, each LM starts a counter with 0. When a rule in an LM raises an alarm, it atomically multicasts that alarm to all LMs in CS_{LM} (including itself). When the atomic multicast delivers an LM $(f+1)^{th}$ copy of the same alarm sent by different LMs in CS_{LM}, it gives that alarm the number indicated by the counter, increases the counter, and sends the message to the IM. This algorithm guarantees that all correct LMs agree on the same alarms, and give the same order number to the same alarms, conforming to the properties of atomic multicast. Therefore, the algorithm guarantees that an IM receives identical alarms from all correct LMs verifying a PE.

3.1.1 TTCB and architectural-hybrid fault model

In this paper, we use the *architectural-hybrid fault model* provided by a distributed security kernel called the Trusted Timely Computing Base (TTCB). The notion of architectural-hybrid fault model is simple: we assume different fault models for different parts of the system. Specifically, we assume that most of the system can fail arbitrarily, or in a Byzantine manner, but also that there is a *distributed security kernel* in the system that can only fail by crashing the TTCB [15]. The TTCB can be considered a “hard-core” component that provides a small set of secure services, such as Byzantine resilient consensus, to a collection of external components, like the LMs. These components communicate in a world full of threats, some of them may even be malicious and try to cheat, but there is an “oracle” that correct components can trust and use for the efficient execution of their protocol.

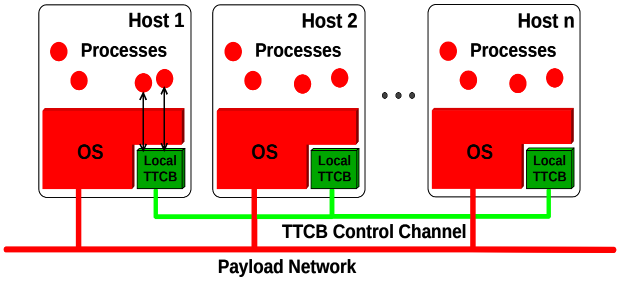


Figure 4: Architecture of *n* hosts with a TTCB

The design and implementation of the TTCB was discussed at length in [16] and here we give a brief overview relevant to its application in the Monitor system. This kernel is a component with local parts in hosts (local TTCBs), which are interconnected by a dedicated control channel (Figure 4). The local TTCBs can be protected by being inside some kind of software secure compartment or hardware appliance, like a security coprocessor. The security of the control channel can be guaranteed using a private LAN.

An important design principle of the TTCB is *verifiability*: its implementation must be formally verified using some formal verification techniques, which are applicable only to small sized software components. This principle implies that the TTCB cannot be used to protect complex components, like LMs or IMs, but only to provide very simple services with a well defined, and therefore possible to secure, interface. This is the case of the service we use to implement the TTCB-based atomic multicast protocol.

3.1.2 Atomic multicast protocol

The atomic multicast primitive provides the following properties: (1) All correct recipients deliver the same messages; (2) If the sender is correct, then the correct recipients deliver the sender’s message; (3) All messages are delivered in the same order by all correct recipients. The Byzantine-resilient atomic multicast tolerant to *f* out of *2f+1* faulty replicas is presented in detail in [17]. Here we describe briefly how it is applied to the Monitor system. Notice that only the nodes with LMs need to have a local TTCB, not the nodes with IMs or the GM. The reason is that the local TTCBs at the different entities need to be connected through a dedicated control channel. While it may be feasible to connect the LMs monitoring a specific PE cluster, which are likely to be geographically closely placed, through such a control channel, it is unwieldy for IMs that are unlikely to have geographical proximity.

The core of the solution we use is one of the simple services provided by the TTCB, the Trusted Multicast Ordering (TMO) [15]. Being a TTCB service, its code lies inside the local TTCBs and its communication goes the TTCB control channel. When an LM wants to atomically multicast, it gives the TMO a *hash* of M obtained using a *cryptographic hash function*, e.g., SHA-1. A cryptographic hash function can be used as a unique identifier for a message since it has essentially two properties: (1) its output has constant length (160 bits for SHA-1); (2) it is computationally infeasible to find two different inputs that hash to the same output. When a recipient receives a message M it also gives the TMO a hash of the message. Notice that the messages are sent through the normal payload network, i.e., outside the TTCB. However, they are sent using channels that guarantee the authenticity and integrity of the messages. These channels could be implemented using SSL or TLS.

3.2 Faults at the Intermediate Monitors

Next we augment the model to allow IM failures by having a redundant number of IMs. Therefore all LMs in a Collaborative LM Set CS_{LM} send alarms to all IMs in a Collaborative IM Set, denoted CS_{IM}. Faults at the IM level can be masked by considering at least *2f’+1* IM replicas, where *f’* is the maximum number of IM replicas which can fail. The output of these replicas is voted on by a simple voter (GM in our case). The model now has the GM acting as a simple voter and a data collector. The simplicity of the GM and the fact that it is not distributed makes it reasonable to assume that efforts can reasonably be made to make it fault free. The common approach to achieve this is through formal verification.

The possibility of faults in the Monitors leads to the design requirement that a diagnosis procedure not be started by a faulty Monitor, owing to the relatively higher overhead involved with diagnosis compared to detection. For this purpose, the initiation of diagnosis must be done at a higher level Monitor, which is considered sacrosanct. It is the responsibility of the higher level Monitor to send test requests to the lower level Monitor’s replicas and collect replies from the appropriate number of the replicas. It should be noted that faults in the Monitors are not diagnosed, but simply masked. An alternative design choice is to control the entire diagnosis protocol from the lower level (failure prone) Monitors through the use of consensus. This was considered to have unacceptable overhead in number of messages and rounds for consensus, which would be required for every member of the suspicion set. Also, if the suspicion set spans boundaries of the LM, higher level Monitors would anyway be needed for distributed diagnosis.

3.3 Flow of Control of Diagnosis with Failing Monitors

Assume that an IM initiates the diagnosis.

1. A failure F at PE *N* is detected by the CS_{LM} verifying it.
2. LM_1 constructs the suspicion set SS_N for the failure and adds it to SS_{FN}.
3. The LMs assign an order to the alarm using atomic broadcast protocol and send alarm along with SS_{FN} up to the IM.
4. The IM waits for *f+1* identical alarms and then starts the diagnosis procedure.
5. For every $N' \in SS_{FN}$ IM sends a test request to the CS_{LM} verifying *N'*.
6. Each $LM \in CS_{LM}$ *tests N'* for correctness of the suspect link L′ using multiple rules from the SRB for the particular event and state of the link.
7. The test results are sent above to the IM who votes on the *f+1* identical responses to decide if *N'* is faulty. If *N′* is not faulty, then it is removed from SS_N and $SS_N{}'$ is added to the SS_{FN}.
8. If a PE *N″* lies outside the verification domain of the IM then a *test request* for *N″* and faulty link L″ is sent recursively to higher level Monitors, which send the request down the tree to the relevant set of Monitors ($CS_{LM'}$) monitoring *N″*, The result of the test is sent back to the IM through the same route. If *N″* is not faulty, then corresponding suspicion set is also sent along.
9. The diagnosis procedure repeats recursively till a node is diagnosed as faulty, or till the cycle boundary is reached.

4 Analysis of Diagnosis Accuracy

For easier understanding and comparison, we follow similar notation to that in [26]. Consider a graphical model where the PE nodes are the vertices and the edges are the messages which have been exchanged between the PEs. Assume that the graph is regular with respect to the incoming links with degree *k*, i.e. each node has *k* incoming links. It is important to note that we are only considering the incoming links to a particular node. A node is faulty with probability λ. An error can propagate through a message sent by the node with probability ρ, given that the node is faulty. Then the probability of error propagation through the message is ρλ. An error in the node could be because of the fault which is present in the node or due to an error propagation through one of the incoming links. A test executed by the Monitor on the node has a coverage c_i if the node n_i is faulty and a coverage d_i if the node has an error which has propagated from some incoming links. For an ideal test, c_i=1 and d_i=1. For simplicity, we do not distinguish between the propagation of single or multiple errors. Let *c* and *d* be the average values for the detection coverage for fault and propagated error over all nodes. Let the number of tests from SRB performed on the node be *T* and the total number of nodes be *N*. Each test yields an output O ϵ {0, 1}, where an output 0 means the node passes the test and 1 that it fails the test. Assume that a node is determined to be faulty if there are *z* or more ones in the total number of tests, z ϵ (0,*T*). Let π be the event that a node is faulty and π′ be the complement event. Based on the model:

A = Prob(test=1|π) = c ; ***[1(a)]***

B = Prob(test=1| π′) = $d(1-(1-\rho\lambda)^k)$; ***[1(b)]***

Prob(*z-ones*| π) = *C*(*T,z*) $A^z(1-A)^{T-z}$ (where *C* is the binomial coefficient) ; ***[1(c)]***

Prob(*z-ones*| π′) = C(*T,z*) $B^z(1-B)^{T-z}$; ***[1(d)]***

One figure of merit for the diagnosis process is the probability of detecting the original faulty node that caused the chain of errors to propagate finally leading to the failure. Now, the *posterior* probability is given by:

Prob(π| *z-ones*) = Prob(*z-ones*| π).Prob(π) / Prob(*z-ones*) ; where Prob(z-ones) is given by 1(c) λ + 1(d)(1-λ) using the total probability formula.

$$= \frac{\lambda C(T,z)A^z(1-A)^{T-z}}{\lambda C(T,z)A^z(1-A)^{T-z} + (1-\lambda)C(T,z)B^z(1-B)^{T-z}} = \frac{1}{1+\frac{(1-\lambda)}{\lambda}\left(\frac{B}{A}\right)^z\left(\frac{1-B}{1-A}\right)^{T-z}} \text{ ; } \boldsymbol{[1(e)]}$$

This equation matches with the one derived by Fussel and Rangarajan (FR) [22] with the following mapping: *R* (number of rounds) there maps to *T* here, since in each round of the FR algorithm, the same test is performed.

Now consider B from equation 1(b)

B = $d(1-(1-\rho\lambda)^k)$, taking the number of messages to be very large we can assume that k→∞ reduces to $d(1-e^{k\rho\lambda})$ because ρλ →0 .

We can rewrite the equation 1(e) as: Prob(π| *z-ones*) = 1 / 1 + F(z) ; where F(z) = **(**(1-λ)/ λ).(B/A)^z .((1-B)/(1-A))$^{T-z}$

We claim that 1(e) is a monotonically increasing function of z. Note that A and B ε (0,1). Also, for realistic situations, the probability of a node being faulty is much greater than the probability of a propagated error affecting a node (this is a common assumption in the fault tolerance literature ([9][28]). Any reasonable diagnosis test should be able to distinguish between a node being the originator of a fault (high probability of π=1) and one which is the victim of a propagated error (low probability of π=1). Therefore, A>B. Let us represent F(z) as $k.\beta^z\mu^{T-z}$. For A>B, β<1 and μ>1 and therefore Prob(π| *z-ones*) increases with *z*. This can also be proved through showing *d(Prob(*π*| z-ones))/dz* > 0 . This implies that the higher the value of *z* for a fixed *T* the greater is the confidence in the diagnosis process. In other words, the diagnosis process is well behaved [26].

Theorem*:* The diagnosis algorithm provides asymptotically correct diagnosis for $N\to\infty$ for $k\geq 2$ and $T\geq \alpha(N)log(N)$, where $\alpha(N)\to\infty$ arbitrarily slowly as $N\to\infty$. It is also optimal in diagnosis accuracy among diagnosis algorithms in its class.

Proof: For this, we use the result proved in [26] and simply map our algorithm's testing behavior to theirs.

Asymptotic implies $N\to\infty$. In [26], the number of tests grows with N as $\alpha(N)log(N)$ and thus asymptotically also tends to ∞, though the growth is not as fast as *N*. Our algorithm falls in the 3AM (*m*-threshold local diagnosis) category as defined in [26] since (i) all testing is done with local knowledge, and (ii) a threshold number of tests needs to fail for an entity to be diagnosed as faulty. The posterior probability given by equation 1(e) matches the posterior probability of the FR algorithm [22]. Hence the algorithm tends to perfect behavior in the asymptotic case of *N* when $k\geq 2$ and *T* grows as $\alpha(N)log(N)$.

Note that our diagnosis algorithm is also asymptotically correct for asymptotic behavior of *T*, independent of *N*. Considering the posterior probability equation 1(e), $\lim_{T\to\infty}\lim_{z\to T} Prob(\pi \mid z-ones)$ approaches 1.

Optimality follows by appropriate choice of *z*. Knowing the coverages *c* and *d*, the optimal choice of *z* is given by the optimality proof in theorems 2 and 3 in [26].

5 Implementation, Experiments, and Results

The diagnosis protocol is implemented in Java as a module adding to the detection module. The implementation in Java helps in portability to multiple platforms. The diagnosis protocol is demonstrated by running a streaming video application on top of TRAM. TRAM is a tree based reliable multicast protocol consisting of a single sender, multiple repair heads (RH), and receivers [12]. Data is multicast by the sender to the receivers with an RH being responsible for local repairs of lost packets. An ack message is sent by a receiver after every *ack window* worth of packets has been received, or an *ack interval* timer goes off. The RHs aggregate acks from all its members and send an aggregate ack up to the higher level to avoid the problem of ack implosion. During the start of the session, *beacon* packets are sent by the sender to advertise the session and to invite receivers. Receivers join using *head bind* (HB) messages and are accepted using *head acknowledge* (HA) messages from the sender or an RH. TRAM entities periodically exchange *hello* messages to detect failures.

In our experiments, the Monitor is fed the strict rule base (SRB) along with the STD and the normal rule base. An example of a temporal rule in the normal rule base is that the number of data packets observed during a time period of 5000 ms should be between 30 and 500. The thresholds are calculated using the maximum and minimum data rates required of TRAM as specified by the user. Another example is that there should not be two *head bind* messages sent by a receiver within 500ms during the data receiving state as the receiver could be malicious and be frequently switching RHs. An example of a strict rule used in our experiments for the sender is *SR1* : *HI S2 E11 1 S2 E9 1* : If in state S2, the receiver has received a data packet (E11) say with linkID as *d* then there must be an *ack* packet within the phase interval around *d*. This rule ensures the receiver sends an *ack* packet on receiving data packet(s). Another SRB rule bounds the hello to be only sent when an entity is in the data tranmission-reception state and if it has not received a hello during the *HelloInterval*. This prevents a malicious receiver from trying to flood the neighbors with hello messages and throttling the bandwidth. In our experiments we use 4-8 SRB rules to test a link depending upon the state of the link.

Lazy link building

During the fault-free operation of the protocol, the Monitor adopts a *lazy approach* (euphemistically, *optimistic approach*) to build the causal graph. During the normal operation, each incoming (outgoing) message to (from) a node is stored in a vector of incoming (outgoing) links for that node. A linkID which is the logical time stamp is assigned to the link along with the physical time, state, and event type. Link contains two IDs, one for the node which sent it and another for the receiving node. Since the Monitor maintains a logical clock for each node, therefore for this link to be formed, a matching is required between the sending and the receiving entities' messages. The link A-B will be matched once the messages sent by A and received by B are also seen by the Monitor. For high message throughput applications, including TRAM, if the Monitor tries to match all the incoming packets during runtime then it will entail an enormous overhead. This scheme is referred to as the *pessmistic approach*. This approach causes faster diagnosis runs but results in some links not being matched at runtime thereby causing a drop in the accuracy of the diagnosis protocol. Note that the matched links are not used if a failure is not detected in the same cycle and thus the computational pressure put on the Monitor host in the pessimistic case turns out to be of no use. Hence, in the optimistic approach, at runtime the Monitor simply stores the link in the causal graph and marks it as being unmatched. The link matching is carried out when diagnosis is triggered on failure. Our experiments give a comparative evaluation of the optimistic and the pessimistic approaches.

In order to reduce the link matching overhead during diagnosis, we use multi-level hash tables. The unmatched links are kept in a hash table (*temporaryLinks*) which have the node ID for the node which sent the corresponding message. The first level in the hash structure is a hash table that is indexed by the nodeID of the node being investigated (*nodeID*). The next level of the hash structure is a hash table (*otherNodeID*) keyed with the id of all the nodes which have sent messages to or received messages from the node in *nodeID*. Further, messages from one node in *nodeID* to a node in *otherNodeID* are arranged in another hash table indexed with the event type (*eventID*). During the matching process, each unmatched link is looked up in the *temporaryLinks* table. The node associated with the link is then looked up in the *nodeID* table and the other end of the link is found by following the pointer to the *otherNodeID* table and then to the *eventID* table. The matching is done by using the sequence number of each event. Thus the linkID of the i^{th} message sent by node n_1 to n_2 is matched with the i^{th} message received by n_2 from n_1. Once matched, the *temporaryLinks* table entry is filled up, which is used for creating the causal graph. Once the causal graph is updated, it is used to form the suspicion set as described in Section 2.2.3. The HashCode method in Java's Object class can be overridden to reduce collisions, thus guaranteeing close to constant order lookups in the multi-level hash structure.

5.1 Fault Injection

For exercising the diagnosis protocol, we perform *fault injection* in the header of the TRAM packets transmitted by the sender. We use the following kind of injections for a *burst length* period of time:

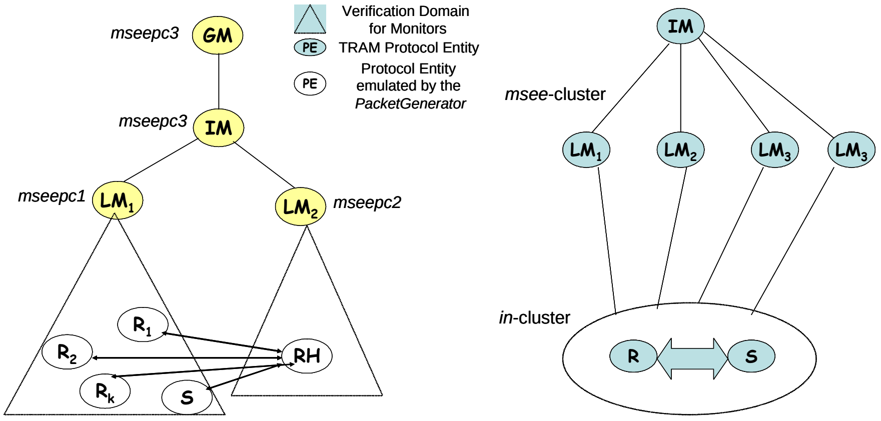
a. *Stuck-At injection* : A randomly selected header field value is changed to a random but valid value. The same value is injected in each packet for the entire burst length.

b. *Directed Injection* : A specific header field is chosen for one experiment and changed to a random but valid value. Each packet is injected with a different value.

c. *Specific Injection* : Specific kinds of injections are carried out, such as, slow data rate, dropping acks, and hello message flooding, to bring out specific characteristics of the diagnosis protocol.

5.2 Test Set Up and Topology

Figure 5(b) illustrates the topology used for the accuracy and latency experiments on TRAM with components distributed over the campus network (henceforth called TRAM-D), while Figure 5(a) shows the topology for the local deployment of TRAM (TRAM-L). TRAM-D is important since a real deployment will likely have receivers distant from the sender. TRAM-L lets us control the environment and therefore run a more extensive set of tests (e.g., with a large range of data rates). The PEs and the LMs are capable of failing, while we assume for these experiments that the IMs and the GM are fault free. The sender, the receivers, and the RHs do active forwarding of the packet to the respective LMs. When a failure is detected, the IMs coordinate the flow of the diagnosis protocol. The minimum data rate in TRAM needed to support the adequate quality of the video application is set at 25 Kbps. The Monitors are on the same LAN which is different from the LAN on which the PEs are located. The routers are interconnected through 1Gbps links and each cluster machine is connected to a router through a 100Mbps link.



(a) (b)

Figure 5: Topology used for accuracy and latency experiments in (a) TRAM-L (b) TRAM-D

5.3 Accuracy and Latency Results for TRAM-L

We measure the accuracy and latency for the diagnosis algorithm on the TRAM protocol through fault injection in the header of sender packets. We consider a single receiver receiving packets from an RH which is connected to the sender. Accuracy is defined as the ratio of the number of correct diagnosis to the total number of diagnosis protocols that were triggered. This definition eliminates any detection inaccuracy from the diagnosis performance. Diagnosis accuracy decreases if the algorithm terminates without diagnosing any node as faulty (*incomplete*) or if it flags a correct node to be faulty (*incorrect*). Latency is defined as the time elapsed between the initiation of diagnosis and diagnosing a node as being faulty, either correctly or incorrectly, or incomplete termination of the algorithm. We perform this set of experiment with two versions of diagnosis algorithm the optimistic approach and the pessimistic approach. There are thus two dimensions to the experiments – the approach (abbreviated as *Opt* and *Pes*) and the fault injection strategy (abbreviated as, *SA* for Stuck-at, *Dir* for Directed, and *Spec* for Specific). In the interest of space a representative sample of results is shown. The results are plotted for *Opt-SA*, *Opt-Dir*, and *Pes-Dir* with a fixed burst length of 300ms for each injected fault. Inter packet delay is varied to achieve the desired increase in the data rate. Delay of *d* is inserted using Gaussian random variable with mean *d* and standard deviation *0.01d*. Each point is averaged over 4 injections and between 20 and 58 diagnosis instances, depending on the number of detections, which in turn depends on the rate of incoming faulty packets.

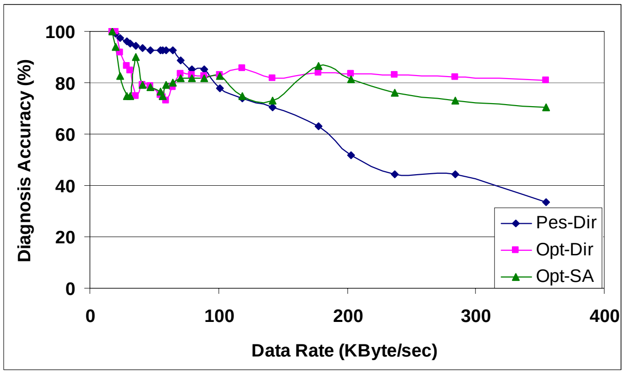


Figure 6: Variation of Accuracy with Data Rate

Figure 6 shows that for *Pes-Dir* accuracy is a monotonically decreasing function with data rate. Diagnosis accuracy drops to a low of 33% for data rate at 355 KByte/sec. This is due to the fact that there is a rate mismatch between the matching of links for causal graph creation (slower process) and the arrival of packets at high data rates (faster process). In the absence of adequate amount of buffering, some packets are dropped and never reflected in the causal graph leading to a drop in accuracy. Secondly, there is no synchronization between the causal graph formation process and the suspicion set creation and testing process. Thus, the latter may be triggered before the former completes leading to inaccuracies.

For *Opt-Dir*, the accuracy is high for small data rate but decreases with the increase in data rate. Unlike the *Pes-Dir* scenario, here the accuracy does not drop below 80%. The link matching and the causal graph completion take place only at the time the diagnosis starts, and the diagnosis algorithm tests the links only after the causal graph is complete resulting in higher accuracy compared to *Pes-Dir*. This advantage becomes significant at high data rates. Also, beyond a threshold, further increasing the data rate does not affect the latency because the number of incorrect packets increases, which helps diagnosis because the current algorithm stops as soon as a single faulty link is identified. The accuracy of *Opt-SA* is slightly lower than that of *Opt-Dir* since in the former, the same message type is injected for the entire burst. If a rule for the message type does not exist in the SRB, the diagnosis is incomplete.

Figure 7 graphs the latency of diagnosis with increasing data rate. Notice the significantly higher latency for the optimistic case compared to the pessimistic one (two separate y-axes are used). We can see that for the *Pes-Dir* case, the latency increases with data rate which is expected because there are more packets to be tested by each rule in the SRB. Latency tends to saturate at high data rates because the causal graph is not fully formed leading to early termination and hence incomplete diagnosis. On the other hand in the *Opt-Dir* scenario, the latency keeps increasing with data rate. Since in the *Opt* case the link matching happens during diagnosis, increasing the data rate causes more packets to be matched leading to high latency.

Effect of burst length: We study the impact of burst length on diagnosis accuracy for the pessimistic and the optimistic case. We keep the data rate low at 15 KByte/sec to isolate the effects due to high data rate. Diagnosis as shown in Figure 8 is accurate for low and high values of burst length. For small burst length values, low numbers of incorrect packets get injected leading to a low entropy in the payload system which is easy to detect. As the burst length increases, more incorrect packets are received by the Monitor which increases the entropy and hence decreases the accuracy. Beyond a certain burst length, the increased number of incorrect packets that come in help in the diagnosis process. For a given coverage for each diagnosis test, the greater is the number of incorrect packets, the greater is the chance of being flagged by the test. A more "systems level" explanation for the increasing part of the curve on the right side is that as the burst length increases, the proportion of SRB rules that match across the boundary of the burst length decreases. These are the SRB rules that are likely to lead to incorrect diagnosis since they are dealing with a mix of correct and incorrect packets.

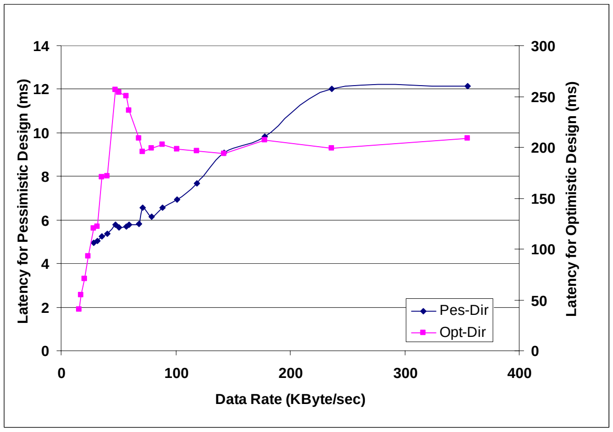


Figure 7: Variation of Latency with Data Rate for Optimistic and Pessimistic Approaches

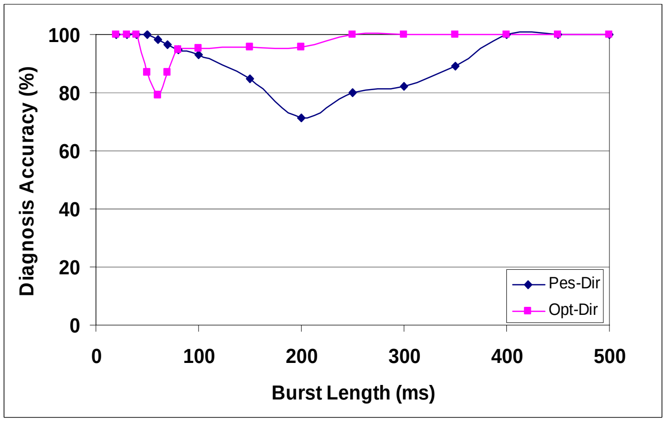
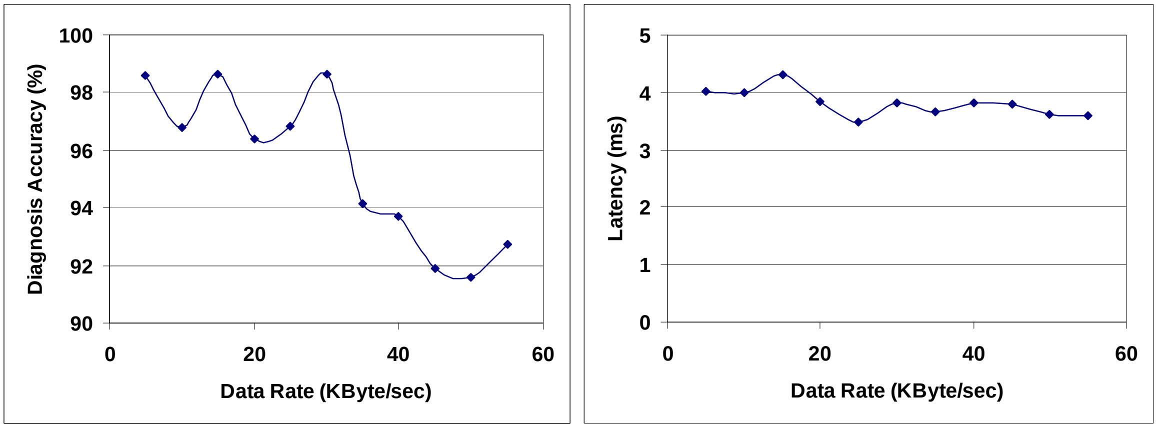


Figure 8: Diagnosis Accuracy with Burst Length for Optimistic and Pessimistic Approaches

5.4 Accuracy and Latency Results for TRAM-D

In this set of experiments we measure the accuracy and latency of the pessimistic approach of the diagnosis protocol on TRAM, while performing specific fault injection, namely, reducing the data rate from the sender (Figure 5(b)). Since there are only two PEs, each of which is sending packets to the other (sender sends *data* and receiver sends *ack*), the suspicion set SS_s and SS_r both contain two nodes <sender, receiver>. The latency and accuracy values are averaged over 200 diagnosis instances for each data rate.



(a) (b)

Figure 9: (a) Diagnosis Accuracy and (b) Latency variation with increasing data rate in TRAM

Figure 9(a) shows that the accuracy of diagnosis drops from a high of 98% at 15 KB/s to 91% for 50 KB/s. Let us say that the data rate is *d* and ack rate is *a*. The incoming data rate at each LM is 2(*d*+*a)*. As the data rate increases, the creation of links in the causal graph gets delayed as incoming packets are pushed off to a buffer for subsequent matching. If a diagnosis is triggered which needs to follow one of the missing links, it results in an incomplete diagnosis, leading to a drop in accuracy. This problem could be partially mitigated by having multiple threads do the causal graph formation, but the limited degree of parallelism and the single processor nature of the host machine lead us to believe that the gains will not be substantial. Figure 9(b) shows the latency of diagnosis with increasing data rate. Intuitively when the data rate increases, increasing load on the Monitor should cause the latency to increase. However, the data rate used is low enough that it has no significant effect. This is also borne out by the experiments with TRAM-L (Figure 7).

6 Related Work

Different problem. A preliminary step for diagnosis is detection of failures, whether accidental or malicious. There is a plethora of work on failure detection using heart beats, watchdogs, and Intrusion Detection Systems (IDS). They differ in the level of intrusiveness with respect to the application entities. Interestingly, the automated response mechanisms associated with many detectors take local responses assuming the detection site is the origin of the fault and that no error propagation has happened. This is clearly a leap of faith as has been shown repeatedly ([28][29]).

Different approaches to same problem. Diagnosis in distributed systems has been an important problem area and was first addressed in a seminal paper by Preparata *et al.* [20] known as the PMC method. The PMC approach, as several other deterministic models [10], assume that each test is perfect and mandates that each entity be tested a fixed number of times. The fault model assumed is often restrictive, such as permanent failures [24]. Probabilistic diagnosis was first introduced in [21]. Probabilistic diagnosis can only diagnose faulty nodes with a high probability but can relax assumptions about the nature of the fault (intermittent faulty nodes can be diagnosed) and the structure of the testing graph. Followup work focused on multiple syndrome testing [22][26] where multiple syndromes were generated for the same node proceeding in multiple lock steps. Both use the comparison based testing approach whereby a test workload is executed by multiple nodes and a difference indicates suspicion of failure. The probabilistic diagnosis algorithms have been categorized in terms of the level of knowledge needed by each node, the least knowledge being m-threshold local diagnosis (3AM). Our proposed Monitor framework, while being fundamentally different in approach, falls in this category with respect to required knowledge. We have shown that it obeys the property of most probable diagnosis as discussed in [26], while being non intrusive and providing practical tests. The goal of the work in [19] is to localize faults in communicating network objects, given alerts that are generated by the objects themselves and that may be inaccurate. The approach is centralized and assumes dependencies between the objects are known *a priori*. The membership and system diagnosis problems are viewed in a unified framework in [11]. More recently the authors in [33] propose a fully distributed algorithm that allows every fault-free node to achieve diagnosis in, at most, $(\log N)^2$ testing rounds and the authors apply it for network management. All of these approaches are fundamentally different from ours since the tested and the testing systems are the same and the explicit tests for diagnosis make the process intrusive to the tested entity.

Similar approach to different problem. There has been considerable work on diagnosing performance problems in distributed systems. They can be classified into active probing or perturbation and passive monitoring approaches. In the first class, in [30][31], the authors use respectively fault injection and forcible locks on shared objects to determine the location of performance bottlenecks. The second approach uses execution traces for black-box applications and has similarities to the Monitor approach ([32][34][35]). For example, in [32], the debugging system performs analysis of message traces to determine the causes of long latencies. However, in all of this work, the goal is not diagnosis of faults, but deduction of dependencies in distributed systems which may enable humans to debug performance problems. These may be regarded as point solutions in the broader class of diagnosis problems.

TTCB. There is abundance of work on consensus. Consensus has been applied to various kinds of environments, with different timing assumptions and types of failures, ranging from crash to arbitrary (see [23] for a survey of early work). Our approach of using TTCBs on the Monitor replicas for atomic multicast is derived from the work in [16][17], which showed how consensus can be achieved in a hybrid failure and communication model system.

7 Conclusions

We have presented a Monitor system for distributed diagnosis of failures in the protocol entities in a distributed application. The overall system is structured as a payload system and a Monitor system each of which may fail in arbitrary ways. The demonstration is given for a streaming video application running on top of a reliable multicast protocol called TRAM. The hierarchical Monitor system is shown to be able to perform diagnosis in the presence of error propagation and using cooperation between the individual Monitor elements. The diagnosis accuracy is higher than 90% for the streaming video application under a large range of scenarios. Next, we plan to explore the cooperative testing by multiple Monitors, testing in the face of uncertain information, and effect of placement of the Monitors.

References

[1] META Group, Inc.,"Quantifying Performance Loss: IT Performance Engineering and Measurement Strategies," 2000.

[2] FIND/SVP, "Costs of Computer Downtime to American Businesses," 1993.

[3] G. Khanna, J. Rogers, and S. Bagchi, "Failure Handling in a Reliable Multicast Protocol for Improving Buffer Utilization and Accommodating Heterogeneous Receivers," In the 10th IEEE Pacific Rim Dependable Computing Conference (PRDC), pp. 15-24, 2004.

[4] G. Khanna, P. Varadharajan, and S. Bagchi, "Self Checking Network Protocols: A Monitor Based Approach," In the 23rd IEEE Symp. on Reliable Distributed Systems (SRDS), pp.18-30, 2004.

[5] M. Diaz, G. Juanole, and J.-P. Courtiat, "Observer-A Concept for Formal On-Line Validation of Distributed Systems," In IEEE Trans. on Software Engineering, 20(12), pp. 900-913, 1994.

[6] M. Zulkernine and R. E. Seviora, "A Compositional Approach to Monitoring Distributed Systems," In IEEE Intl. Conference on Dependable Systems and Networks (DSN), pp. 763-772, 2002.

[7] K. Bhargavan, S. Chandra, P. J. McCann, and C. A. Gunter, "What Packets May Come: Automata for Network Monitoring," In ACM SIGPLAN Notices, 36(3), pp. 206-219, 2001.

[8] C. A. Carver, J. M.D. Hill, and U. W. Pooch, "Limiting Uncertainty in Intrusion Response," In IEEE Workshop on Information Assurance and Security United States Military Academy, 2001.

[9] S. Bagchi, Y. Liu, Z. Kalbarczyk, R. K. Iyer, Y. Levendel, and L. Votta, "A Framework for Database Audit and Control Flow Checking for a Wireless Telephone Network Controller," In the Intl. Conf. on Dependable Systems and Networks (DSN), pp. 225-234, 2001.

[10] R. Buskens and R. Bianchini Jr., "Distributed On-line Diagnosis in the Presence of Arbitrary Faults," In 23rd Intl. Symp. on Fault-Tolerant Computing (FTCS), 1993.

[11] M. A. Hiltunen, "Membership and System Diagnosis," In 14th IEEE Symp. on Reliable Distributed Systems (SRDS), 1995.

[12] D. M. Chiu, M. Kadansky, J. Provino, J. Wesley, H. Bischof, and H. Zhu, "A Congestion Control Algorithm for Tree-based Reliable Multicast Protocols," In INFOCOM '02, pp.1209-1217, 2002.

[13] T. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," In Journal of the ACM, 43(2), pp. 225–267, 1996.

[14] G. Bracha and S. Toueg , "Asynchronous Consensus and Broadcast Protocols," In J. of the ACM, 32(4), pp. 824–840, 1985.

[15] M. Correia, N. F. Neves, and P. Veríssimo, "How to Tolerate Half Less One Byzantine Nodes in Practical Distributed Systems," In the 23rd Intl. Symp. of Reliable and Distributed Systems (SRDS), pp. 174–183, 2004.

[16] M. Correia N. F. Neves, and P. Veríssimo, "The Design of a COTS Real-Time Distributed Security Kernel," In the 4th European Dependable Computing Conference (EDCC), pp. 234–252, 2002.

[17] Miguel Correia and Nuno Ferreira Neves and L. C. Lung and Paulo Veríssimo, "Low Complexity Byzantine-Resilient Consensus," In Distributed Computing, Springer-Verlag Heidelberg, 2004.

[18] A. Mostefaoui, M. Raynal, and C. Travers, "Crash-Resilient Time-Free Eventual Leadership," In the 23rd IEEE Intl. Symp. on Reliable Distributed Systems (SRDS), pp. 208-217, 2004.

[19] I. Katzela and M. Schwartz, "Schemes for Fault Identification in Communication Networks," In IEEE/ACM Trans. On Networking, 3(6), pp. 753-764, 1995.

[20] F.P. Preparata, G. Metze and R.T. Chien. "On the Connection Assignment Problem of Diagnosable Systems," In IEEE Trans. on Electronic Computers, Vol. EC-16, No. 6, pp. 848-854, 1967.

[21] S. Maheshwari and S. Hakimi, "On Models for Diagnosable Systems and Probabilistic Fault Diagnosis," In IEEE Trans. on Computers, C-25, pp. 228–236, 1976.

[22] D. Fussel and S. Rangarajan, "Probabilistic Diagnosis of Multiprocessor Systems with Arbitrary Connectivity," In the 19th Intl. IEEE Symp. on Fault-Tolerant Computing (FTCS), pp. 560-565, 1989.

[23] M. J. Fischer, "The Consensus Problem in Unreliable Distributed Systems (A Brief Survey)," In M. Karpinsky, editor, Foundations of Computing Theory, vol-158 of Lecture Notes in CS, pp. 127–140, Springer-Verlag, 1983.

[24] A. Bagchi and S. Hakimi, "An Optimal Algorithm for Distributed System Level Diagnosis," In the 21st Intl. Symp. on Fault Tolerant Computing (FTCS), pp. 214-221, 1991.

[25] R. Chillarege and R. K. Iyer, "Measurement-Based Analysis of Error Latency," In IEEE Trans. on Computers, 36(5), 1987.

[26] S. Lee and K.G. Shin, "On Probabilistic Diagnosis of Multiprocessor Systems Using Multiple Syndromes," In IEEE Trans. Parallel and Distributed Systems, 5(6), pp. 630 – 638, 1994.

[27] A. Avizienis and J.-C. Laprie, "Dependable Computing: From Concepts to Design Diversity" In Proc. of the IEEE, 74(5), pp. 629–638, 1986.

[28] S. Chandra and P. M. Chen, "How Fail-Stop are Faulty Programs?," In the 28th Annual Intl. Symp. on Fault-Tolerant Computing (FTCS), pp. 240-249, 1998.

[29] H. Madeira and J. G. Silva, "Experimental Evaluation of the Fail-Silent Behavior in Computers Without Error Masking," In the 24th Intl. Symp. on Fault-Tolerant Computing (FTCS), pp. 350-359, 1994.

[30] A. Brown, G. Kar, and A. Keller, "An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment," In the 7th Intl. Symp. on Integrated Network Management (IM), 2001.

[31] S. Bagchi, G. Kar, and J. L. Hellerstein, "Dependency Analysis in Distributed Systems Using Fault Injection: Application to Problem Determination in an E-Commerce Environment," In the 12th Intl. Workshop on Distributed Systems: Operations & Management (DSOM), 2001.

[32] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance Debugging for Distributed Systems of Black Boxes," In the ACM Symp. on Operating Systems Principles (SOSP), 2003.

[33] E. P. Duarte, T. Nanya, "A Hierarchical Adaptive Distributed System-Level Diagnosis Algorithm," IEEE Trans. on Computers, 47(1), pp. 34-45, 1998.

[34] J. L. Hellerstein, "A General-Purpose Algorithm for Quantitative Diagnosis of Performance Problems," Journal of Network and Systems Management, 2003.

[35] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, "Magpie: On-Line Modelling and Performance Aware Systems," ACM-HotOS-IX, pp. 85-90, 2003.