NON INTRUSIVE DETECTION AND DIAGNOSIS OF FAILURES IN HIGH THROUGHPUT DISTRIBUTED SYSTEMS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Gunjan Khanna

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2007

Purdue University

West Lafayette, Indiana

ACKNOWLEDGMENTS

Special thanks to Prof. Saurabh Bagchi without whose efforts and input this Thesis would not have made its way to the Committee Members. He gave me the inspiration to strive, think and achieve what stands in the form of this research work. I would like to thank my committee members Prof. Ness Shroff, Prof. Cristina Nita Rotaru, and Prof. Rudolf Eigenmann for their support and helpful criticisms. My manager at IBM Dr. Gautam Kar and my group members have provided a useful industry perspective to my research work and I am grateful for their support. I would also like to thank Dale Talcott and Bill Whitson from the ITAP department in Purdue whose help in installation and running of TRAM was priceless.

TABLE OF CONTENTS

Page

Page

Page

LIST OF FIGURES

ABSTRACT

Khanna, Gunjan PhD., Purdue University August 2007. Non Intrusive Detection and Diagnosis in High Throughput Distributed Systems. Major Professor: Saurabh Bagchi.

Distributed systems form an integral part of human life—from ATMs to the Domain Name Service. Typical distributed systems consist of distributed services interacting through messages. Failures in these systems are often the causes of huge financial loss or human catastrophes. Efficient fault detection and diagnosis of cascaded non fail-silent failures is extremely challenging because of legacy code, black-box nature of application entities, scalability and state space explosion. Current error detection and diagnosis protocols suffer from one or more of the following problems—very specific to one application, require intrusive changes to the application, lack of scalability, impose additional load on the application, are offline and cannot detect (or diagnose) the failures at runtime.

In this thesis, we propose Monitor, a scalable, autonomous, fault detection and diagnosis framework. The Monitor only observes the external messages between the components of the application and is unaware of any internal transition of the application entities. The Monitor uses a rule base of allowable behavior and does fast matching of incoming messages. The Monitor deduces actual causal dependencies from the protocol behavior to perform diagnosis. We propose state reduction mechanisms which reduce the number of states to be verified by the Monitor without affecting the accuracy of detection or diagnosis. We propose a sampling approach which adjusts a sampling rate in accordance with the incoming rate of packets such that the breakdown in the Monitor capacity is avoided.

We use a distributed deployment of Monitors across the Purdue WAN to demonstrate its effectiveness. We compare the performance of the Monitor in diagnosing faults in e-commerce applications with Pinpoint, the state-of-the-art diagnosis approach. Monitor outperforms Pinpoint in both accuracy and precision of diagnosis.

1. INTRODUCTION

1.1. Motivation

The wide deployment of high-speed computer networks has made distributed systems ubiquitous in today's connected world providing the backbone for the information infrastructure. The infrastructure, however, is increasingly facing the challenge of dependability outages resulting from both accidental & malicious failures, collectively referred to as *failures* in this thesis. 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 financial consequences can be gauged from a survey by Meta Group Inc. of 21 industrial sectors in 2000 [41], 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 computer system downtime cost today to the average of $82,500 in 1993 [33] and the trend becomes clear. Little wonder that distributed systems are called upon to provide always-available and trustworthy services.

We increasingly face the challenge of failures due to natural errors and malicious security attacks affecting these systems. Downtime of a system providing critical services in power systems, flight control, banking, air traffic control, and railways signaling could be catastrophic. For ease of exposition consider two distinct systems: the application system or distributed protocol which needs to be verified and the fault tolerance system which provides detection and diagnosis primitives. The application system is comprised of multiple services communicating through standard protocols. The communication is through externally observable messages. Example of such application services are web service and authentication service. The fault tolerance system sits in the network vicinity and provides detection and diagnosis.

In order to build robust infrastructures capable of tolerating the two classes of failures, it is foremost to provide *detection* of the failures. The role of the detection system is to raise an alarm when it detects that the application system is not operating according to the specified behavior. The behavior of the application system can be defined through *misuse* or *anomaly* semantics. In the misuse based detection systems, the incorrect behavior is specified while in anomaly based systems the correct behavior of the system is specified.

The next important component of the fault tolerance system consists is the diagnosis component which gets triggered once a detection of failure takes place. Following the definitions in [75], a fault is an invalid state or bug underlying in the system, which when triggered becomes an error. A failure is an external manifestation of an error at the systems' boundary. A failure in a distributed system may be caused by error propagation between processes, and detected by the detection system. The role of the diagnosis system is to identify the entity that originated the failure. The diagnosis problem is significant in distributed applications that have many closely interacting PEs, since this facilitates error propagation. Failure detection at some entity A could be because of a local fault at A or error propagation from some node B through message interactions. Cascaded failures are multiple entities failing because of the failure of a single entity. For example consider routing protocols in the internet (like BGP). If there is a faulty route entry in one of the routers, that can cause multiple faulty route entries in other routers because of route advertisement messages. This can cause the requests to fail for not only a small subnet linked to the initial faulty router, but also requests in the larger wide area network. Cascaded failures is exactly what happened on 10 August 1996 when a 1300-mw electrical line in southern Oregon sagged in the summer heat, initiating a chain reaction that cut power to more than 4 million people in 11 Western States [113]. This is an example where early detection and diagnosis of an initial failure could have prevented the catastrophic outcome.

There are several challenges to the problem of designing a detection system which can handle failures in the distributed systems of today. First, many existing systems run legacy code, the protocols have hundreds of participants, and systems often have soft real-time requirements. Legacy code is typically an old piece of software written by developers who are no longer around to support the code. In essence, understanding and making changes to legacy codes to add detection and diagnosis functionality is extremely difficult A common requirement is for the detection system to be non-intrusive to the distributed system being verified implying that significant changes to the application or to the environment in which they execute are undesirable. This requirement rules out executing heavyweight detectors in the same process space or even in the same host as the application entities. While it may be possible to devise very optimized solutions for individual distributed applications, such approaches are not very interesting from a research standpoint because of limited applicability. Trying to make changes to a particular protocol also requires in depth understanding of the code which is either unavailable or too complex to allow for the understanding under time constraints.

The other primitive of the fault tolerance system *i.e.*, diagnosis, comes with its own set of challenges. First, similar to detection, diagnosis also faces the challenge of legacy code, real time guarantees and large number of protocol entities. Second, there is imperfect observability of the components of the application due to their relative network placements and losses in the environment. This is particularly likely as the application is distributed with components spread out among possibly distant hosts. Third, a diagnosing entity might have limited resources and may drop some message interactions due to exhaustion of its resources (*e.g.*, buffer size) during periods of peak load. Fourth, any diagnostic test used by the diagnosis framework cannot be assumed to be perfect. Finally, several parameters of the environment are not known deterministically and have to be estimated at runtime. These include the ability of a component to stop the cascade of error propagation and the lossiness of the links between the application components as well as the application and the diagnosing entity.

Detection is called *stateful* if the detection approach uses the knowledge of the current and few previous states of the entity in providing detection. Providing stateful detection in such a setting is imperative because of the criticality of applications but it comes with more challenges. The rules are then based on the state, thus on aggregated information rather than instantaneous information. Stateful detection is looked upon as a powerful mechanism for building dependable distributed systems [114][115]. The stateful detection models can be specified using various formalisms, such as, State Transition Diagrams, PetriNets or UML. Stateful detection requires the fault tolerance infrastructure to keep track of the state of the protocol thus increasing the storage and the computation. Increased computation leads to high latency of detection and diagnosis impacting the real-time nature of these operations. Today's distributed systems operate at a very high data rate; any delay in detecting failure can cause catastrophic effects [113]. The distributed systems have a large number of participating entities leading to large states which increase exponentially with increasing number of entities. For a stateful system, tracking a protocol entity's state can be difficult in such circumstances because of increased computation causing the detection (or diagnosis) system to break. This is commonly known as *State space explosion*. Because of state space explosion, the detection system fails to scale with increasing number of entities.

In general, the fault tolerance system does not have perfect observability of the application system. For one, the internal state is not observable and in the best case, only the part of the state that is deducible from the external messages is available to the fault tolerance system. Additionally, the fault tolerance system is typically placed in the network vicinity of the application entities rather than co-hosting the two due to the performance impact on the application. This further hampers observability due to the nature of the communication medium (loss, delay, and jitter) and the distributed nature of the two systems (such as, non synchronized clocks). Resource constraints can further limit the observability of a detection and diagnosis framework. Thus, detection and diagnosis have to be provided to today's high throughput distributed systems under imperfect observability.

1.2. Design Goals and State of the Art

The solution for developing a detection system should foremost address the challenges mentioned previously. Because it is near impossible to obtain access to the internals of the system, the detection system should be oblivious to the protocol internals. The protocol participants are treated as black boxes and their internal state transitions should be invisible to the detection system. This requirement can also be restated as being non-intrusive to the underlying protocol. A non-intrusive methodology does not subject the verified protocol to additional tests for the purpose of providing detection or diagnosis. The fault tolerance system should be generic and applicable to a large class of distributed protocols. Any new proposed framework should address *scalability*. It should be capable of scaling to a large number of protocol entities. Further it should not be a performance bottleneck to the underlying protocol. The two metrics of primary interest for a detection system are accuracy (and the related metric of precision) and latency. Accuracy of detection is given by *1- missed alarms*. The latency of a detection system is the overall processing time taken to verify the correctness of the application protocol.

In a fault tolerant system, detection is logically followed by diagnosis. Because we would like to build a common framework for detection and diagnosis, several design goals of detection are also applicable to diagnosis. For example: black-box and non-intrusive diagnosis. The diagnosis approach should consider a realistic model with imperfect observability and imperfect diagnostic tests. The diagnosis should not rely on any *a priori* dependency model as input obtained through instrumentation of application or expert knowledge. The diagnosis protocol should address error propagation between the components of the application, with no artificial assumptions on the length of the error propagation chain.

Previous approaches of detection in distributed systems have varied from heartbeats, to watchdog [120]-[122]. There is previous work [123][124] that has approached the problem of detection and diagnosis in distributed applications modeled as communicating finite state machines. The designs have looked at a restricted set of errors (such as, livelocks) or depended on alerts from the PEs themselves. A detection approach using event graphs is proposed in [125], where the only property being verified is whether the number of usages of a resource, executions of a critical section, or some other event globally lies within an acceptable range. These approaches have focused on accuracy of fault detection and not scalability.

There has been considerable work in the area of diagnosis of embedded systems, particularly in automotive electronic systems. In [127], the authors target the detection and shut down of faulty actuators in embedded distributed systems employed in automotives. This class of work ([116][127]) focuses on making best use of constrained resources, such as processing power and communication bandwidth, while achieving real time functionality. There are several other offline tools that aid diagnosis, such as tools for data slicing [117], backtracking[118], and deterministic replay [119], but they all require manual effort in actually diagnosing the faulty components. Several existing diagnosis approaches are based on some knowledge about the interaction between the application entities like Dependency Graphs [111]. However accuracy of such interaction models is questionable. Diagnostic tests employed by several approaches (such as [98][111]) are not perfect and therefore cannot deterministically indict a component. Finally, several parameters of the environment are not known deterministically and have to be estimated at runtime. These include the ability of a component to stop the cascade of error propagation (error masking ability) and the unreliable links within the application components as well as the links between the application and the diagnosis module.

1.3. Solution Approach: Monitor System

In this thesis, we present Monitor architecture for detection and diagnosis of failures in distributed systems. The proposed design segments the overall system into an observer or a *Monitor system* and an observed or a *payload system*. The Monitor system comprises multiple Monitors and the payload system comprises potentially a large number of protocol entities (PEs). The Monitors are said to *verify* the PEs. The Monitors are designed to observe the external messages that are exchanged between the PEs, but none of the internal state transitions of the PEs. The Monitors use the observed messages to deduce a runtime state transition diagram (STD) executed by the PEs. Next, pre-specified rules in a rulebase are used to verify correctness of the behavior based on the reduced

STD. The rules can be either derived from the protocol specification or specified by the system administrator as QoS requirements. The system provides a rich syntax for the rule specifications, categorized into combinatorial (valid for the entire length of system operation except for transients) and temporal rules (valid only for specific times in the protocol operation) and optimized matching algorithms for each class of rules. The large part of the Monitor is generic, i.e., not hard wired to a specific application, and it is the specification of the rules in the rule base that makes a Monitor deployment application specific.

Monitor framework is divided into a hierarchy of Local, Intermediate and Global Monitors to achieve scalability. Local Monitors filter the local interactions and pass this filtered information to the Intermediate Monitors. In well-designed distributed protocols the bulk of interactions are local thus providing significant local filtering of messages. Intermediate Monitors verify the interactions between the local domains and the Global Monitor verifies the overall global correctness of the protocol. This aids in the scalability of the Monitor framework.

The Monitor is designed to meet a set of design requirements. First, the Monitor should not become a performance bottleneck for the payload system. This is achieved by making the Monitor's operation asynchronous to the payload system's operation and removing any requirement for co-locating the Monitors with the PEs. The protocol entities are not exercised with additional tests for either detection or diagnosis since that would make the Monitor system more invasive to the application protocol. Second, the Monitor should be scalable to a system with thousands of verifiable PEs. The design of the Monitor components, the hierarchical Monitor structure, and the fast rule matching algorithms help achieving the scalability goal. A sampling algorithm which samples incoming messages based on the rate of packets increases the Monitor's scalability to a large number of protocol entities or alternately, a higher message rate from the protocol entities. Third, the Monitor should be applicable to a large class of applications with minimal effort in moving from one application to another. To achieve this goal, the Monitor architecture is kept application neutral and the rulebase, specifiable in an intuitive formalism, is used for detection and diagnosis of failures in different applications. Finally, the Monitor should have a low latency of detection and diagnosis. This is critical since the Monitor functions asynchronously to the application and a high latency will make any subsequent containment or recovery action complicated. This is achieved by the same design features that support scalability.

The Monitor employs a stateful model for rule matching, *i.e.*, it preserves state across messages. The Monitor also maintains a reduced state transition diagram (STD) for every observed PE comprising only the externally visible state transitions. Reduced state transition diagrams can be obtained through two methods of reduction: Reducing invisible states and 2) Reducing ruleless states. Invisible states consist of the states whose incoming transitions are not visible to the Monitor. This could be because of firewall or relative placement of the Monitor. Ruleless states compose of states which do not have any rules to match in the rulebase. Existing methods of state reduction such as partial order reduction [92] or symbolic state space reduction [93][94] only provide a reduction of the *non-reachable* states. Partial order methods exploit interleaving of concurrent events while state space exploration techniques calculate the possible reachable states. Authors in [95] provide a hybrid approach using the underlying benefits of both approaches. In contrast, we provide the system administrator the flexibility to choose the states which he would like to verify by providing rules only for the states needing verification. The ruleless states will be automatically removed by the state reduction process. The reduction mechanisms are proved to have no effect on the detection or diagnosing capability of the Monitor.

In developing the diagnosis capabilities of the Monitor framework, we maintain the non-intrusive semantics. As previously mentioned in the design goals, the PEs are not exercised with any additional tests. Instead state that has already been deduced by the Monitors during normal operation through the observed external messages is used for the diagnostic process. Using already existing state also prevents any performance bottleneck on the underlying protocol possible because of additional tests. The Monitor performing diagnosis maintains a causal graph which maintains the causal relationship between the application components interaction. An edge in the causal graph is created when there is a message exchange between two protocol entities which indicates a possible path for error propagation.

An initial step toward diagnosis is made via the deterministic diagnosis approach. Here we assume that the Monitor has perfect observability and the diagnostic tests used by the Monitor are accurate. In reality however, messages may be dropped and diagnostic tests may be inaccurate. Therefore, we extend it to come up with a probabilistic approach in which the Monitor makes a probabilistic determination of the entities that may have initiated the propagation of errors. Our probabilistic solution rests on three basic techniques. First, the observed interaction between the components is used to build a causal dependency structure between the components. Second, when a failure is detected, the causal structure is traversed (till a well-defined bound) and each component is tested using diagnostic tests. Third, runtime observations are used to estimate and continually refine the estimates of parameters that bear on the possibility of error propagation, such as lossiness of links and error masking capabilities. We also show that the diagnosis protocol is optimal amongst its class of diagnosis algorithms. The classes are defined on the basis of the amount of information an algorithm needs to perform diagnosis.

We extend the Monitor framework to incorporate a sampling approach which adjusts the rate of messages to be verified by sampling the incoming stream of messages from the application entities. The adjustment is such that the breakdown in the Monitor capacity is avoided. The cost of processing each message increases and the accuracy decreases because the application state is no longer accurately known at the Monitor and instead rules pertaining to all of the possible states have to be matched. However, the overall detection cost is reduced due to the lower rate of messages processed. We show that even with sampling, the Monitor is able to provide stateful detection without significant degradation in accuracy.

The Monitor system is implemented and deployed across the Purdue wide area network. Test bed experiments using reliable multicast protocol (TRAM [71]) are performed to demonstrate the detection and diagnosis of the Monitor. TRAM is a tree based reliable multicast protocol. It has a single sender which multicasts the data to the receivers and the protocol guarantees all receivers receive an uninterrupted stream. Some receivers volunteer for local repair of lost packets and are called Repair heads. The performance of detection and diagnosis are individually evaluated by measuring the *accuracy* and the *latency*. Faults are injected for measuring the detection and diagnosis capabilities of the Monitor. The performance is evaluated under varying data rate, number of receivers, size of rulebase, and thread pool size.

Diagnosis experiments are also performed on an *e*-commerce system. The *e*-commerce system consists of a JBoss application server hosting a PetStore J2EE application. Monitor is also compared with Pinpoint, an existing state of the art approach for performing diagnosis in distributed systems. Pinpoint uses a dependency matrix of user transactions on the system components. It uses clustering to aggregate the components on which failed requests depend on with the underlying model that components which contribute to failures are likely to cluster with the failures. Monitor outperforms Pinpoint by achieving higher accuracy for the same precision values.

Further, to illustrate the efficacy of the sampling approach real test bed experiments are performed across Purdue's Network. The scalability of the Monitor framework is tested in a high throughput environment. We perform controlled experiments via emulating the TRAM protocol locally in a single cluster. The performance of the Monitor's sampling approach is tested against high rate stream of TRAM traffic. The sampling approach helps in achieving higher accuracy of detection at a higher incoming packet rate than possible with the baseline Monitor.

1.4. Summary of Contributions

Summarizing, the primary contributions of this thesis are.

- We proposed a generic hierarchical framework the Monitor, to provide non-intrusive detection and diagnosis in distributed systems. The Monitor system assumes no access to the internals of the distributed protocol.
- We developed a stateful detection mechanism that can scale to a high data rate of the application protocol.
- We develop a black-box diagnosis protocol to perform diagnosis of faults in distributed systems. The approach can account for uncertainties of the deployment environment as well as imperfect knowledge of the characteristic of the protocol

entities.

- We provide proof of the efficacy and correction of two state reduction methods proposed to address the problem of *state space explosion.*
- The scalability is achieved in the Monitor framework by incorporating a sampling approach which reduced the overall workload at the Monitor for a given message rate of the application.

1.5. Thesis Outline

The thesis is organized as follows. Chapter 2 describes the Monitor Detection approach. Chapter 3 and Chapter 4 describe the deterministic and probabilistic diagnosis respectively. In Chapter 5 we describe state reduction mechanisms to provide flexibility of verifying only chosen states. We compare the Monitor's diagnosis approach to Pinpoint in Chapter 6. We improve the scalability of the Monitor framework through the sampling approach in Chapter 7. Related research and conclusions are presented in Chapter 8 and Chapter 9 respectively.

2. MONITOR DETECTION

A first step in providing autonomic management is to provide detection of problems. The Monitor provides detection of problems in the underlying protocol through verification of messages, which are then diagnosed.

2.1. Monitor Architecture

The Monitor architecture consists of several modules classified according to their functional roles. These modules include, in order of their invocation, the Data Capturer, the State Maintainer, the Rule Matching Engine, the Decision Maker, the Interaction Component, and the Rule Classifier. Figure 2.1 gives a pictorial representation of the Monitor components.

The details of the structural and functional roles of the components have been described in [40] and only a basic description is given here. The Data Capturer is responsible for ‘capturing’ the messages exchanged between the protocol participants over the network, and passing it on for further processing by the Monitor. Message capturing can be through *passive* monitoring of traffic or using *active forwarding* support from the protocol entities. Monitor may be placed in the same domain (for e.g. LAN) as the protocol entities or in a completely different domain, with PEs providing active forwarding of messages. Port mirroring on switches and routers can also achieve forwarding of messages to the Monitor without PE cooperation.

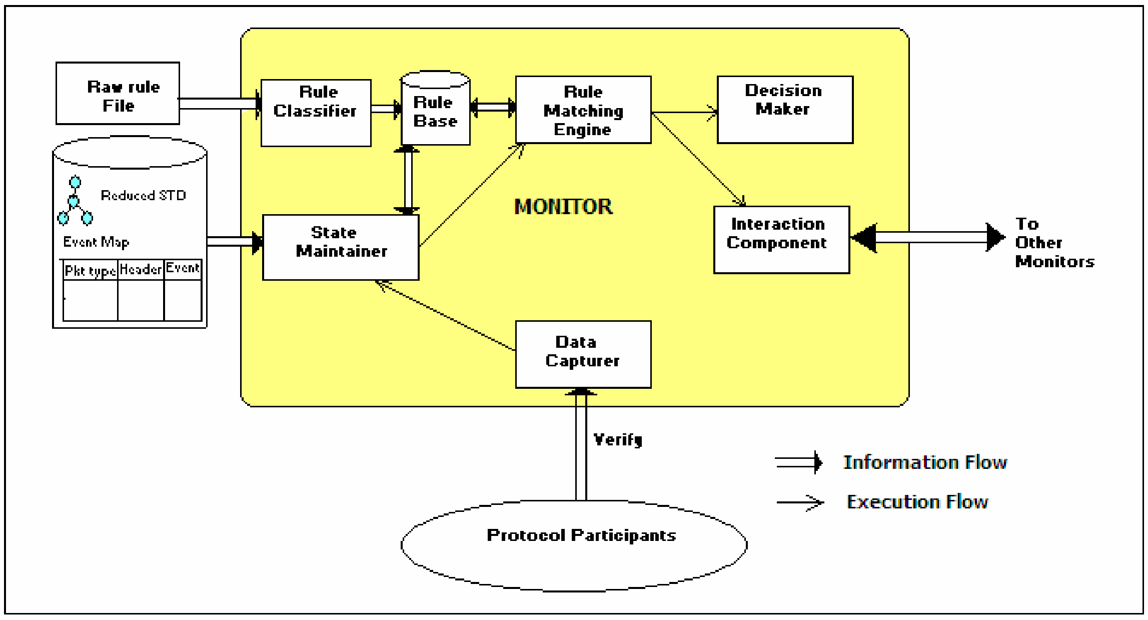


Figure 2.1: Monitor architecture with process flow and information flow among multiple components

The State Maintainer contains static information of the reduced state transition diagrams for each observed entity and dynamic information of the current state of each. The combination of current state and incoming event determines the set of rules to be matched. The Matching Engine is invoked by the state maintainer when an incoming packet triggers a rule that has to be matched. This component is highly optimized for speed to reduce the detection latency. It uses separate matching algorithms for temporal and combinatorial rules. Once the matching engine finishes its rule matching, the Decision Maker combines the results of rule matching for the different rules in the rulebase and raises an appropriate flag in case of error. The Interaction Component deals with communication between monitors at different levels in the hierarchical approach.

2.2. Fault Model

We follow the classical definition of faults being underlying defects that are triggered to become errors and some errors causing end-user visible failures. Errors can propagate from one PE to another through message exchanges and finally manifest as a detected failure. We assume that protocol entities (PEs) can fail arbitrarily, exhibiting Byzantine failures that are observable in external messages. These failures could be caused due to various reasons, e.g., incorrect deployment, software bug, security vulnerability, or performance problems.

In an abstract sense, the Monitor is capable of detecting any fault in a PE that manifests itself as a deviation from expected message exchange with other PEs in a distributed application. As introduced above, the Monitor uses a rulebase modeling expected behavior from each PE. The rulebase may include correctness properties – intrinsic to the PE itself (such as, a 404 error code should be returned by a web server when a non-existent page is requested), or to the specific deployment of the PE (such as, no "post" operations are allowed on the web server); and QoS properties (such as, a minimum and a maximum transaction rate of 20 kbps and 40 kbps respectively are expected from the multicast system). At a more concrete level, the error has to pertain to an error in a state in the reduced state transition diagram deduced by the Monitor and has to violate a rule in the rule base. The Monitor cannot observe any internal state transitions within a PE and it cannot exercise a PE with additional tests for detection. Therefore, the only faults that can be detected are those that are manifested at the external interface of the PE and in the state deduced by the Monitor. It may appear that all the checking performed at the Monitors can be moved into the PEs as application specific checks. However, the specification as well as the runtime checking have to be done in an application intrusive, ad hoc manner and would not be possible for black-box applications.

Monitor in its current form, does not handle failures due to collusion amongst PEs. This can be best explained using the following example. Consider entities A-B, representing 2 PEs of some distributed protocol. Assume a Monitor M is verifying the operation of the distributed protocol and also verifying the communication between A-B. If A and B collude and form an outside channel to communicate, this makes the communication un-observable by M. Because of the in-ability to see the communication, M will not be able to perform any verification and thus would not be accurate in this kind of a failure. Therefore, it would be inaccurate to consider the Monitor suitable for detecting security attacks in general because removing observability of the attack steps is often an intrinsic part of security attacks, such as in covert timing channels [128][129]. In essence the Monitor would be able to verify any externally observable behavior as long as it is captured in the rulebase.

2.3. Structure of the Rule Base

The rules can be obtained from two sources – formal protocol specification and deployment QoS conditions required by the administrator. The first class of rules in our case are derived from a complete state transition diagram (STD) specification of the protocol while the second class is specified by us based on the application requirements.

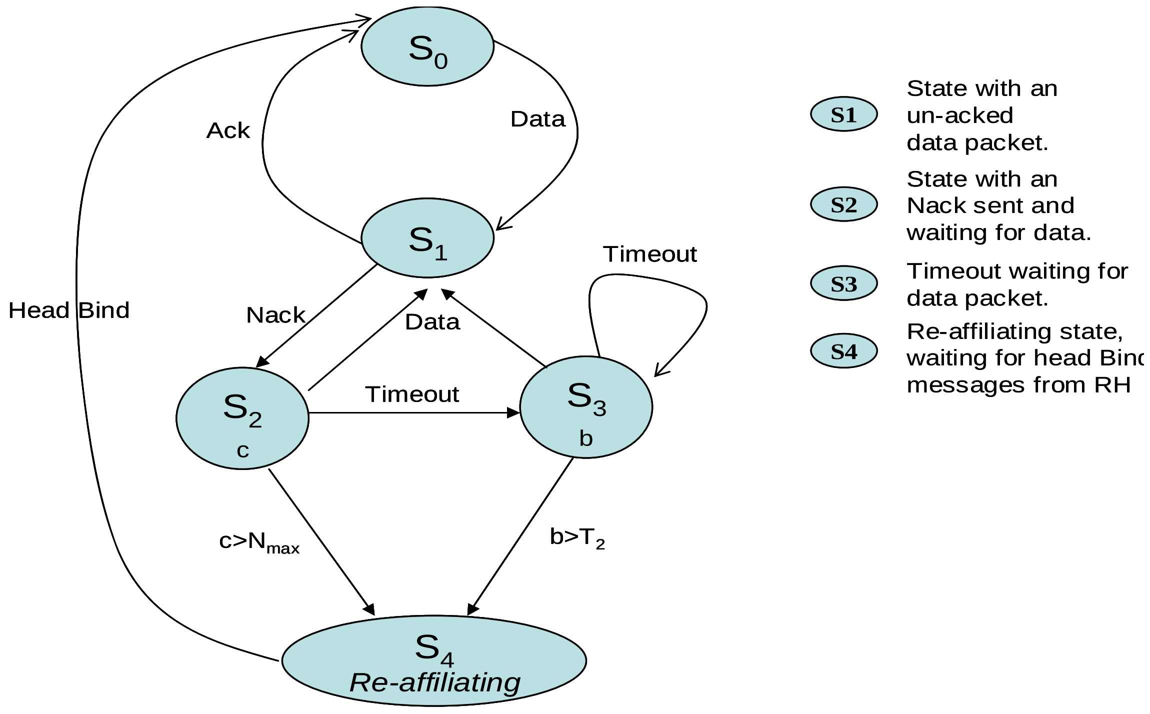

S0
Ack
Data
S1
Timeout
Head Bind
Nack
Data
S2
c
Timeout
S3
b
c>N_max
b>T_2
S4
Re-affiliating
S1 State with an un-acked data packet.
S2 State with an Nack sent and waiting for data.
S3 Timeout waiting for data packet.
S4 Re-affiliating state, waiting for head Binc messages from RH

Figure 2.2: State Transition Diagram for a receiver in TRAM protocol.

The running protocol we use as example is the TRAM protocol ([4][5]) for reliable multicast of data from a single sender to multiple receivers through intermediate routing nodes called repair head (RH). In TRAM, the receiver acks correct data packets and sends Nacks for missing data packets to the RH above. The receiver maintains a counter for the number of Nacks sent and if it crosses a threshold, receiver begins to re-join a different RH assuming the old RH is not functioning any more. The STD in Figure 2.2 shows an example STD for a receiver receiving data from the sender/RH. Rules can be derived from the STD using the *states, events, state variables* and *time of transitions*. Each state has a set of state variables. Events may cause transitions between states. In our context, events are message sends and receives. It is however important to note that not all events cause state transitions, e.g., in a simple sender receiver protocol where a timeout occurs state changes from S_2 to S_3 but subsequent timeouts only increment a state variable in the state S_3. In Figure 2.2, the receiver moves from state S_2 to state S_3 if there is a timeout and no packet is received. Hence a rule can be derived if for all $t \in (t_i, t_i+a)$, $S_2 \Lambda \neg P \Rightarrow \neg S_2$. Here predicate P implies packet receive. Also if S_4 is true then S_0 will be true at some time interval Δ_2 in future. Similarly if the number of Nacks is greater than N_{max}, then we must see a head bind message: $N_{max} \leq N_{ack} \Rightarrow H_{Bind}$. Hence rules can be derived from the STD specifications. The system administrator may add rules specifying QoS conditions that the application should meet, e.g., a minimum data rate that must be received at each receiver. In addition, the system administrator may augment the rule base with additional rules apart from the derived rules to catch manifestations of any protocol vulnerability.

We have a formally defined syntax for rules in the system. The syntax represents the expressibility of the system and by extension, its ability to detect different classes of failures. The syntax also determines the speed with which rule matching can be performed. The rules defined in the system could be derived from the specifications of the protocol or from the QoS requirements on the application. Further, the rules defined are anomaly based (i.e., specify acceptable state transitions), and not misuse based. A primary reason for the choice is that the space of misuse based rules could be very large. Combinatorial rules are expected to be valid for the entire period of execution of the system, except for transient periods of protocol instability. Details on combinatorial rules and its matching process can be found in [40].

2.3.1. Temporal Rules

We studied the properties of the rules for two applications – TRAM and SIP (Session Initiation Protocol), a signaling protocol used for exchanging control messages used to manage interactive multimedia sessions. After the study, we came up with following classification for the temporal rules.

– Type I: $(S_T=S_p)$ = true for $T\in(t_N, t_N+k) \Rightarrow (S_T=S_q)$ = true for $T\in(t_I, t_I+b)$, where $t_I > t_N$, and $k, b \geq 0$.

The above rule represents the fact that if for some time interval k starting at t_N, a node is in state S_p i.e., the state predicate $S_T=S_p$ is true, then it will cause the system to be in another state S_q for some time b starting from time t_I. The time t_N is when state changes to S_p, irrespective of which event causes the transition. This rule is defined completely in terms of states of the entity and no events or state variable.

– Type II: S_t is the state predicate of an object at time t : $S_t \neq S_{t+\Delta}$, if event E_i takes place at t,

the state S_t will not remain constant for Δ time units from t.

– Type III: $L \leq |V(t)| \leq U$, $t\in(t_i,t_i+k)$, t_i is the time of event E_i, where E_i occurs in state S_i

The state variable V in a particular state S_i will have its count bounded by L and U over a time window of k starting at time t_i when event E_i occurs.

– Type IV: $L \leq |V(t)| \leq U$, $t\in(t_i,t_i +k)$, t_i is the time of event $E_i \Rightarrow L' \leq |B(q)| \leq U'$, $q\in(t_n,t_n+b)$, $t_n > t_i$.

If a state variable V in a particular state S_i has a bounded count from above and below over a time window k, it will cause another state variable B_q to be bounded for a time window b starting from t_n. This rule is in fact the master rule and the three previous rule types are special cases of it. But we still need the first three rule types because matching this class of rule entails matching more variables, which increases the latency of detection.

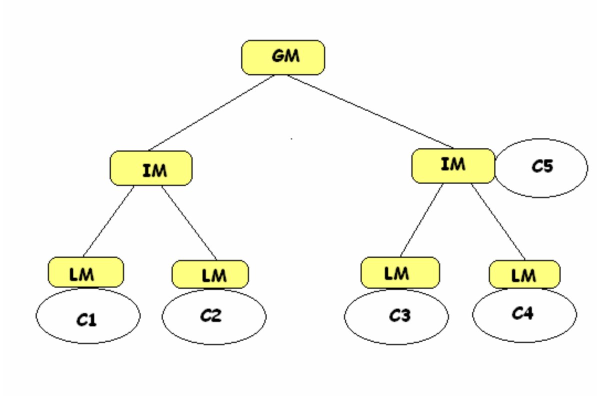
For Temporal rule matching we use two threads namely Variable Copier and Rule Matching. The Variable Copier thread copies the values of the relevant state variables into the Rule Object that is to be examined while the Rule Matcher thread performs the actual matching. Since the time between the capture of the state variable and the time when the rule is matched can be arbitrarily far apart in time, dependent on the specification in the rule, this mechanism ensures that a thread is not blocked for this arbitrary time.

2.4. Multi-level Monitor Architecture

A single Monitor approach has several drawbacks. It constitutes a single point of failure, a large number of protocol participants might overwhelm the Monitor increasing the latency of detection, it is not scalable, and an effort to make it scalable by observing partial views of the system by monitoring only select nodes may lead to reduced coverage. Therefore, in our system, we incorporate the idea of using a hierarchy of Monitors working in conjunction at multiple levels to detect failures. The entire structure is divided into Local, Intermediate, and Global Monitors. The Intermediate Monitor gathers information from several local Monitors, each verifying a set of PEs. In addition, the Intermediate Monitor may also be monitoring PEs directly. The Global Monitor has a global view of the protocol. Its functionality does not involve matching many rules as filtering is done at the local and the intermediate levels. In well designed protocols, most interaction among protocol participants is local, thus most messages are seen only at the Local Monitor. The intuition is that the large fraction of behavior in application elements can be verified locally, while the interactions that span multiple clusters have to be verified using multiple Local Monitors and Intermediate Monitor(s). Due to its simplicity, it can be reasonably assumed that the GM's failure mode is restricted to crash failures. Thus, using a suitable degree of replication of the GM, the GM cluster can be looked upon as fault free, e.g., a standby sparing approach of the GM replicas.

Each Monitor has the same architecture as described in Section 2.1 with the same rulebase. The rulebase is automatically partitioned into three classes – one which is not relevant to the Monitor, the second which is relevant to the Monitor alone and does not need to be observed by any other Monitor, and the third which is relevant to the Monitor plus some higher level Monitors.

The event corresponding to the second class of rule does not need to be forwarded up the Monitor chain and constitutes filtering by a Monitor. For an event corresponding to the third class of rules, the Monitor may optionally perform some computation on the event before forwarding it to a higher level Monitor, such as aggregation using counting.



C: Clusters; LM: Local Monitor; IM: Intermediate Monitor; GM: Global Monitor
Figure 2.3: Example topology of local, intermediate, and Global Monitor.

2.5. Rule Classification

The rule classification algorithm makes administering a hierarchical Monitor system relatively simple by allowing the system administrator to specify a single identical rulebase to all the Monitors. Knowing the set of PEs to verify and the compressed state transition diagram for each, the Monitor reasons about which rules are relevant to it. The rules are specified in terms of events and states. The state name space is unique to each entity being monitored. A message send or receive event is called an Elementary Event. An event may also be generated by processing one or a set of elementary events, and forwarded as a new message for rule matching at a higher level Monitor. Such an event is called a Derived Event. We will refer to a state as local to a Monitor if it is the state of an entity it verifies and an event or variable as local to a Monitor if it is for a local state.

2.6. Rule Categories

Given the rules in the entire rulebase, they are classified into the following categories *with respect to a given Monitor* – Local, Global, and Bypass.

Each Monitor has a domain of PEs to verify and all messages exchanged within the domain are visible to the Monitor. Inter domain message exchanges cannot be verified locally by the local Monitor but require support from other Monitors. A rule not being matched is called *flagging* of a rule.

1 *Local Rule*. A local rule is matched at the current Monitor only as it consists of states/events/state varables which pertain to entities solely in the local monitoring domain and does not require matching at any other Monitor.

2 *Global Rule*. A global rule generates event(s) to be forwarded for subsequent matching at other Monitors. This kind of rule is further categorized into the following sub-classes.

2.1 Forward only (FO). The current(local) Monitor simply forwards the event corresponding to this rule without doing any processing or checking. For example, consider a rule "$E11 \wedge E21$" where E11 corresponds to an event which happens at a PE within the monitoring domain and E21 is an event outside the domain of the current monitor. The rule indicates that both events E11 and E21 must have occurred for correct behavior.

2.2 *Process, don't flag, and forward (PNF)*. The current Monitor processes the rule which corresponds to states/events/state variable local to the Monitor. It generates a new event based on the processing and forwards the new event to the Monitor higher up in the hierarchy to perform the matching. This is done because the rule consists of both local and non local states/events/state variables and a local Monitor can only process the ones which are local. For example, rule R4 states: $0 < E11 < 10$ for 5000 ms in state S1 $\Rightarrow$ $10 < E21 < 20$ for a time window of 1000 ms. This rule states that if the event count for event E11 is between 0-10 in a state S1, then the event count of event E21 will be between 10-20. Here E11 and S1 are local and can be processed locally but E21 is non-local and cannot be processed. So processed information is generated that the precondition is true and sent to a higher level Monitor which can perform matching.

2.3 *Process, flag, and forward (PFF)*. This category is identical to category (ii) above, except that in case if there is a mismatch with the rule, the Monitor flags an alarm. For example, a conjunctive rule R3 with $L1 < |E_1| < U1$, $L2 < |E_2| < U2$, …, $LG < |E_G| < UG$, where E1 is a local event for which it generates the derived event E_G. All the other events are non-local.

3 *Bypass.* These rules have states and events that are all non-local to the Monitor. Such a rule is not relevant for the Monitor and hence is removed from the rulebase for the current Monitor.

2.7. Monitor Interactions

The distributed protocols deployed on a large scale (such as, over the internet) often go through periodic changes due to several reasons, such as version change, or participant changes, such as joins, leaves, or changes of properties of participants. Deploying a static Monitor in hierarchical fashion would be inefficient or inaccurate as the protocols runs for longer period of time. For example, a component may be verified by a Monitor that is placed far apart (source of inefficiency) or an entity may be unchecked due to failures of some Monitors (source of missed alarms, a form of inaccuracy). The Monitor interactions aim to handle these dynamic behaviors in the system. Each of the supported Monitor interactions is described below. We have already discussed one interaction in the discussion of rule classification (Section 0), namely, message processing and filtering by a child Monitor before forwarding to the parent Monitor. The automatic rule classification algorithm complements the Monitor interactions and needs to be executed after any change of the assignment of PEs to a Monitor.

1. **Heartbeat:** Each parent Monitor sends a periodic heartbeat to each of its child Monitors. We employ a smart heartbeat where the liveness of the Monitor is verified by receipt of a packet from it. A parent Monitor sends a heartbeat to the child Monitor if it has not received a packet for the last θ seconds. The parent Monitor must receive the *HeartBeatReply* within Φ seconds of sending the request, where Φ is a constant integral factor times the RTT. The time θ is determined by the rule specification and the RTT. The rate of packets sent by the child Monitors to the parent Monitor according to the

global rules depends on the wait time specified in these rules (D) and the rate of instantiation of these rules (R). For a rule *i*, the minimum time taken for sending a packet from the child to the parent Ti = (1/Ri + Di + c), where c is a constant incorporating rule matching latency and network delays. The median Ti over all the rules at the child Monitor is denoted by Tmed. Consider θ' as max{ Φ, Tmed }. This can lead to high detection latencies if Tmed is not bounded. So, we consider a time bound θub which is the maximum value θ can take. Hence, θ is min{θ', θub}. In case no reply is sent by the child Monitor within Φ seconds, then after θ seconds the parent Monitor sends another *HeartBeat* message. If the parent Monitor fails to receive the *HeartBeatReply* repeatedly (three times in our implementation) then that child Monitor is declared as *failed*.

2. **Load Balancing :** Section 2.12 shows that as the rate of packets input to a Monitor goes above a certain rate, say η, then the rule matching latency rises exponentially. We call this point where the Monitor gets over-loaded the *neck*. We define a Monitor to be over-loaded when it is operating in a region beyond η and to be under-loaded when it is operating below η/2. So the desirable range of operation, called the *Target Load Range* (TLR), is between η/2 and η. Since in a dynamic scenario, the data rate of the PEs may change, it causes varying load at the Monitor. Hence load balancing is required to ensure that each Monitor operates in the TLR. Due to the hierarchical Monitor structure, the parent Monitor is best placed to make decisions about load balancing amongst its child Monitors. In order to enable load balancing, each Monitor maintains a sliding window based measure of the rule matching rate. In case it exceeds η or falls below η/2 the Monitor sends an *Overload* or an *Underload* packet to its parent Monitor. If a Monitor is operating at a load λ and λ < η/2, then it is an under-loaded Monitor which can increase its load by f.(η/2-λ), which is called the *residual capacity*, where 1 ≤ f ≤ (η-λ)/(η/2-λ). The upper bound represents the maximum load before the latency of matching exceeds the neck. Similarly if λ> η, the Monitor needs to reduce its capacity by f*(λ - η), called the *overload capacity*, where 1 ≤ f ≤ (λ-η/2)/(λ-η). The number of additional PEs which a Monitor can accommodate or which needs to be removed is determined by the residual capacity or overload capacity, respectively. Each *Overload()* packet contains the list of nodes which the Monitor feels should be removed from its domain. Each *Underload()*

packet contains the additional number of rule matchings which a Monitor can accommodate. Each Monitor records the rate of rules and the corresponding latency in the *Rule Latency Table (RLT)*, which is maintained as a sliding window in time. A parent Monitor on receiving an Underload or an Overload packet, initiates redistribution of load till the Monitors reach the TLR. The overload in a child Monitor is shared by other child Monitors with residual capacities in the ratio of the residual capacity for a one-to-many mapping between overloaded Monitors to under-loaded Monitors. A similar proportionate division scheme is followed for the many-to-one mapping from the overloaded to the under-loaded Monitors. If there is no such Monitor with residual capacity, the parent Monitor expresses need through the *Need* packet and attempts to add a new Monitor through the process outlined in interaction 4 described below. The removal of nodes from the child Monitor is based on a heuristic which removes the PEs in the decreasing order of load each is generating, ensuring a that only a minimum number of entities are chosen. The heuristic can also consider geographical proximity to generate a geographically optimal configuration over time. To avoid the expensive load rebalancing due to transients, the parent Monitor initiates it only after aggregation of Underload or Overload packets. We describe the procedure for shuffling nodes later in interaction 5.

3. **Addition of New PEs:** If a new PE to be monitored comes up, the system administrator is responsible for selecting the appropriate Monitor and entering the information — the rulebase and the STD — into the Monitor. This is thus a manual bootstrap process. However, if an incorrect or inefficient allocation is done, the load balancing mechanisms can correct the situation.

4. **Addition of a New Monitor:** The architecture allows for addition of a Monitor in two ways, namely, Controlled and Self-Evolving. In the Controlled case, the administrator specifies the parent Monitor for this new Monitor. The new Monitor (NM) sends a join message to the parent Monitor (PM) to which the PM replies with an accept message if it wishes to accept the NM and a reject message otherwise. Condition for acceptance is if one or more of the current child Monitors is overloaded. Condition for rejection is if the PM has difficulty in accessing the subnet where the NM is placed,

possibly due to large network distance. The administrator has to specify a new PM if such a situation arises. In the Self-Evolving case, NM broadcasts its call for a parent Monitor through a beacon packet with increasing values of TTL. When any Monitor, say SM (for starting Monitor), sees the beacon message, it sends a confirm message to NM to prevent flooding of the beacon message. The NM then suspends its search, and initiates an advertisement window length of wait for it to be joined to the Monitor hierarchy. The SM sends an advertisement message to its parent and child Monitors, thereby indicating there is a new Monitor which wishes to join the Monitor hierarchy. This allows other Monitors a fair chance to adopt NM. Any intermediate Monitor sends the advertisement on all the links except the incoming link. Thus, if it received the packet from its parent Monitor, it sends it down to the child Monitor and vice-versa. All the IMs that have the need for a new Monitor, due to one or more overloaded child Monitors, send a need packet to NM indicating the number of PEs to be transferred. NM collects the need packets over the advertisement window and then sends a request to the Monitor that it wants to join, say JM (for Joining Monitor). The decision to join a particular Monitor is based on a weighted mean of factors which include the proximity, how overloaded the requesting Monitor is, and how much load NM can handle. On receiving the request packet, IM sends an accept or a reject to the new Monitor.

5. **Transfer of PEs:** This procedure is required to transfer PEs from one Monitor to a different sibling Monitor. Say α is the parent Monitor which is carrying out the transfer of nodes from Monitor A to Monitor B. IM α will send a *transferRequest* packet with the IP address of the PEs to be transferred to Monitor B. The Monitor B snoops over the LAN to ensure that it can actually receive the packets for the PEs which are being transferred. Monitor B sends an *acceptTransfer* packet to the IM α if it accepts the assigned additional load. IM α then transfers the STD for the nodes to Monitor B. Monitor B sends a *rejectTransfer* message if it feels the load transferred is excessive or it is unable to see messages from the transferred PEs. IM α then chooses another Monitor to transfer the PEs. Note that the global rulebase is stored in all the Monitors and B only needs to run the *rule classification* algorithm after the transfer.

2.8. Formal Property

Theorem: *At all points during the operation of the protocol except for a time duration ranging from 6* Φ to $3(T_{max} + \Phi)$, *every PE is monitored by at least one Monitor.*

Proof:

1. *Fault Free Scenario :* Each PE *p* is a leaf node of a heterogeneous tree, in which all leaf nodes are PEs and non-leaf nodes are Monitors. Each leaf node in a tree has a non-empty set of ancestors. A protocol participant is monitored by all the ancestor nodes and therefore the property holds.

2. *Load Balancing:* Let transfer of node *p* happen from Monitor A to Monitor B. Let P_{AB} be the parent of both these monitors. When the transfer is complete from A to B, the leaf node *p* obtains a second parent B for a transient period of time till P_{AB} sends a *endMonitoring* message to Monitor A. Hence *p* has at least one parent throughout.

3. *Monitor Crash:* Monitor crash is detected by three successive failures to receive *HeartBeatReply* packets. Let *p* be a PE verified by a local monitor M that crashes. Let T be the time for detection of monitor crash. Recollect that a parent Monitor sends a heartbeat to the child Monitor if it has not received a packet for the last θ seconds and Φ is the time from sending the request that the parent Monitor should get a reply. $\theta \leq \theta_{ub}$. Therefore, $T \leq 3(\theta_{ub} + \Phi)$. The minimum value that θ can take is Φ, as θ' is $\max\{\Phi, T_{med}\}$. Under this case, the wait time for each round is $\Phi+\theta = 2\Phi$ and since failure is declared after three timeouts, the total delay is 6Φ. Hence $T \geq 6\Phi$. During T, the path from the PE (leaf in the tree) to the GM (root of the tree) is broken, and *p* remains unmonitored. As soon as the crash of M is detected by its parent Monitor, it starts verifying the local rules as well. Assignment of another Monitor to verify *p* can be considered as a load balancing task and the property holds during the transitional period of load balancing as proved in case 1. Hence the property holds during Monitor crash.

2.9. TRAM

The detailed description of TRAM can be found in [4]. We provide an overview here and present details of the features relevant to the study. TRAM is distributed as a part of the Java Reliable Multicast Service (JRMS) by Sun Microsystems [71]. JRMS is a set of libraries and services for building multicast-aware applications. TRAM is designed for high scalability targeted towards multicasting streaming data from a single sender to a large number of receivers. TRAM ensures reliability by using a selective acknowledgement mechanism. An ack is sent in the form of an offset and a bit vector once every ack window (32 packets). It provides scalability by adopting a hierarchical tree-based repair mechanism. The receivers and the data source of a multicast session in TRAM interact with each other to dynamically form repair groups. These repair groups are linked together in a hierarchical manner to form a tree with the sender at the root of the tree. Figure 2.4 shows a typical TRAM repair tree. The nodes participating in TRAM play three roles, some nodes playing multiple roles – sender, receiver and repair head (RH). Every repair group has a receiver that functions as a group head; the rest function as group members which are said to be affiliated with their head. All members receive data multicast by the sender. The group members report lost and successfully received messages to the group head using a selective acknowledgement mechanism. Every ack message contains a start message number indicating the first missing message, and a bit vector, with a 1 denoting a missing packet and a 0 denoting a received packet. If no packets are missing, the message number indicates all messages prior to and including this one has been received and the bit vector is of zero length. An ack message is sent after every *ack window* worth of packets has been received, or an *ack interval* timer goes off. The RHs cache every message sent by the sender and provide repair service for messages that are reported as lost by the members. The RH's maintain a high and low water mark for monitoring cache occupancy. If the amount of buffer occupied by the packets goes beyond the high water mark, an attempt is made to purge the cache. Failure to do so is taken as an indication of congestion in the network. The RHs aggregate acks from all its members and send an aggregate ack up to the sender to avoid the problem of ack implosion, i.e., the situation in which the sender is overloaded with acks from every single receiver (and this can be a large number under the target systems for TRAM). The data rate sent out by the sender is bounded by maximum and minimum rates configured at the sender. Receivers that cannot keep up with the minimum data rate can be pruned from the repair tree. The TRAM protocol is also more challenging because of both unicast and multicast messages being exchanged amongst the entities. This makes the detection process of the Monitor to be competent enough to encompass both kinds of messages.

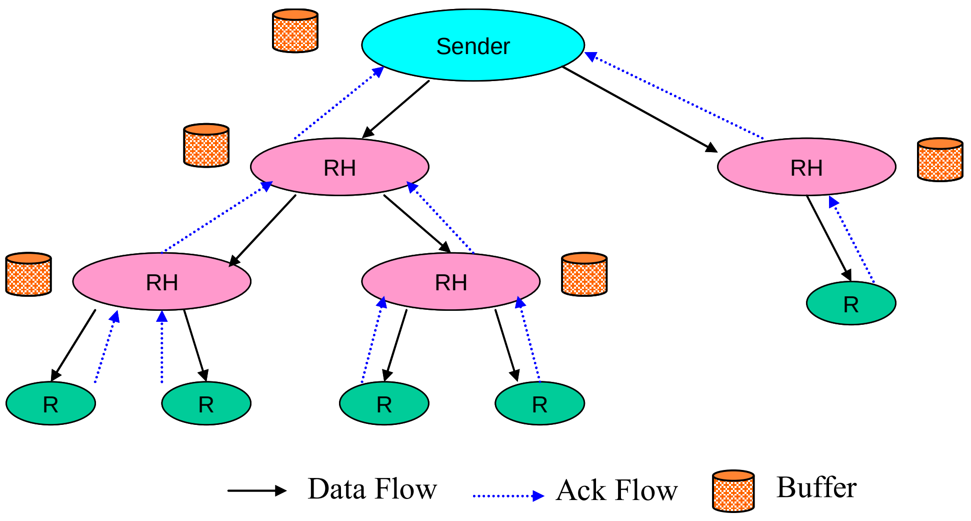


Figure 2.4: An example of TRAM deployment.

Figure 2.4(a) shows a TRAM deployment with a sender, two levels of RHs and multiple receivers connected through links over which bi-directional data and ack messages flow. Two examples of repair groups are shown, one involving the sender and the three RHs at the first level, and the second showing a RH and its receivers.

2.9.1. TRAM Implementation

The TRAM code is multi-threaded. These threads are responsible for carrying out the group management functions in addition to basic sending and receiving of data packets. *GroupMgmtThread* is the main thread which is responsible for starting up TRAM, initiation of the *beacon messages* by the sender and affiliation of the receivers to the senders or repair heads. The beacon messages are used to advertise the session and invite nodes to join the multicast session. This thread performs the task of sending periodic hello messages among the receivers and its head. Each receiver also maintains a backup list of heads which it can switch to if the current head resigns or fails. Once the data transmission phase starts, *InputDispThread* and *OutputDispThread* come into picture. *OutputDispThread* transmits the packets. *InputdispThread* gives the packet to all the listeners and hence, each entity calls its received packet method to get the desired packet. The sender and the repair head's sending functionality use *HeadAck* class to receive ack packets. The receivers use *MemberAck* class to receive data packets and to send acks. Repair head uses *MemberAck* class, as it is a receiver for the sender above, to send cumulative acks. **Figure 2.5** shows pictorially the threads or methods which are used for upstream and downstream communication in TRAM.

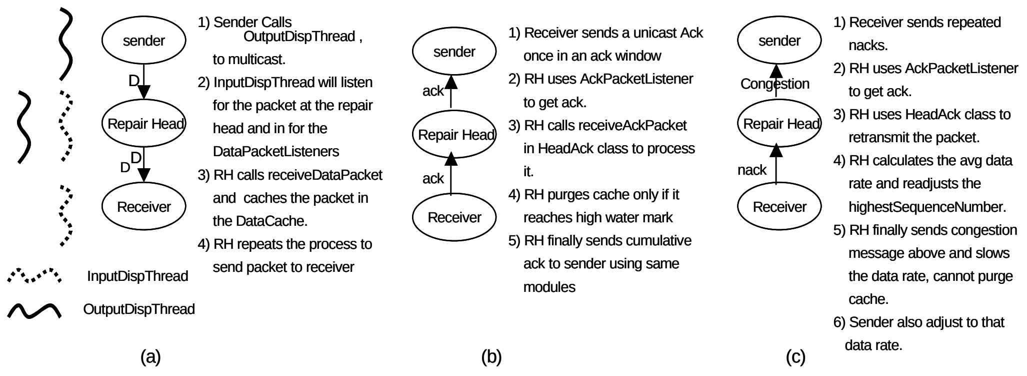

sender
Repair Head
Receiver
D
D
D
1) Sender Calls OutputDispThread , to multicast.
2) InputDispThread will listen for the packet at the repair head and in for the DataPacketListeners
3) RH calls receiveDataPacket and caches the packet in the DataCache.
4) RH repeats the process to send packet to receiver
InputDispThread
OutputDispThread
(a)
sender
Repair Head
Receiver
ack
ack
1) Receiver sends a unicast Ack once in an ack window
2) RH uses AckPacketListener to get ack.
3) RH calls receiveAckPacket in HeadAck class to process it.
4) RH purges cache only if it reaches high water mark
5) RH finally sends cumulative ack to sender using same modules
(b)
sender
Repair Head
Receiver
Congestion
nack
1) Receiver sends repeated nacks.
2) RH uses AckPacketListener to get ack.
3) RH uses HeadAck class to retransmit the packet.
4) RH calculates the avg data rate and readjusts the highestSequenceNumber.
5) RH finally sends congestion message above and slows the data rate, cannot purge cache.
6) Sender also adjust to that data rate.
(c)

Figure 2.5: TRAM message processing (a) downstream with no errors, (b) upstream with no errors, (c) upstream with message, node or link errors

In the case of errors, the downstream path is identical to the error free case. In the upstream path, the receiver sends nacks to the RH which transmits the requested packets; the RH adjusts the data rate and sends to the sender which finally adjusts the data rate.

2.9.2. TRAM LAN Optimization

TRAM performs various Tree based optimizations to improve the scalability of its multicast system. The characteristic of an optimal tree varies depending upon how the receivers are organized and their relative placement. Tree optimizations are necessary to have local repair regions and avoid incurring high latency while recovering the lost packet from a higher level Repair Head. TRAM tries to form a repair head in every LAN so that local repairs are very fast. At the beginning all the receivers in a particular LAN are directly connected to the sender. If every receiver has RH flag enabled then each receiver is likely to become a repair head, which is not desirable. One of the receivers (randomly chosen) is picked as the repair head to serve that LAN for repair of lost packets.

2.9.3. TRAM Flow Control

TRAM incorporates a receiver based flow control mechanism. Receiver advertises a window of packet which determines the amount of packets it can accept. This receiver window is sent by each receiver to its parent repair head and so on up to the sender. The sender chooses the minimum window to determine the number of packets it needs to send next. This mechanism is different from sender based flow control mechanisms. This receiver based flow control allows the receivers to control the amount of new data they can accept.

This however also can be a source of problem. A bug in one of the receivers can cause it to advertise a small window for packets. Also if a receiver is running on a machine with small buffer then this will cause it to advertise a small window. Since the sender chooses the minimum of all advertised windows from the receivers, it causes the whole TRAM tree to run at a slower rate because of a small window advertised by one receiver.

2.10. Monitor Rulebase for TRAM

In our implementation, the recipient of a packet does active forwarding of the packet to the Local Monitor. The packet consists of 1316 bytes of payload, 14 bytes of TRAM header, and 10-18 bytes of variable header depending upon the packet type. The Monitor follows a reduced STD for each verified PE, which is manually input. The reduced STD should cover all the states and events in the rule base and can only depend on externally visible message exchanges. The rule base consists of anomaly-based rules governing the execution of the protocol at the TRAM receiver. In a rule specification, the first letter (T/C) specifies whether the rule is temporal or combinatorial in nature, while (R1/R2/R3/R4) indicates the sub-type of the rule in the temporal category as defined in Section 2.3. An example of Rule 3 is given by *T R3 S2 E11 30 500 5000***.** The number of data packets observed during a time period of 5000 ms can be any number between 30 and 500. Similarly a Rule 4 can be constructed as T *R4 S2 E11 30 500 5000 S2 E9 1 8 500 7000 i.e.*, if there are between 30 and 500 data packets in 5000 ms in given state, then the number of ACK packets should be between 1 and 8 from 500ms to 7000ms in the same state. The numbers are characteristic of the protocol and QoS as specified by the system owner. An extensive list of rules (with explanation) used in our experiments is provided in Appendix B.

2.11. Experiments & Results

The Monitor is implemented in Java to allow portability across heterogeneous machines. The Monitor is deployed across the Purdue campus-wide WAN. We demonstrate the working of the Monitor architecture on TRAM and divide the experiments to quantify the performance and the coverage respectively.

2.12. Performance Results

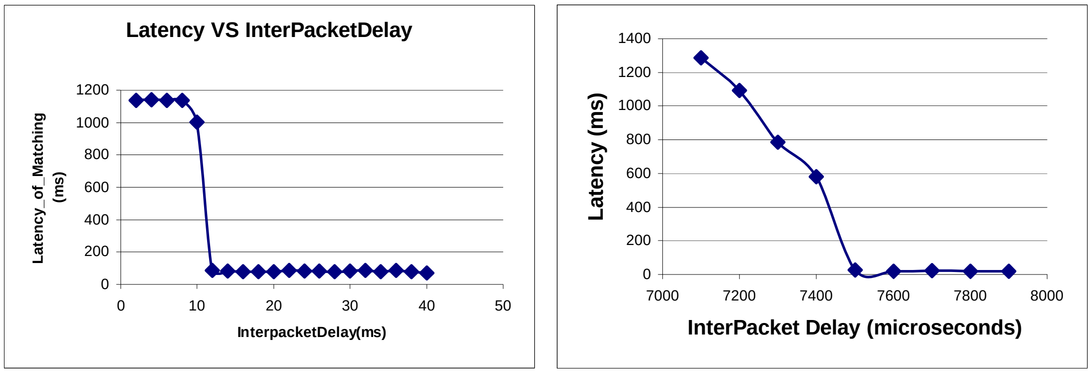
2.12.1. Experimental Setup

The goal of these experiments is to quantify the performance of a single level Monitor. This can lead to insights about the desired load level for the Monitor. For these experiments, the performance of the Monitor is defined by its latency of rule matching. The performance is evaluated under varying data rate, number of receivers, size of rulebase, and thread pool size. Let us denote by δ the difference in time between when the packet comes into the Monitor and when the matching of the corresponding event against a rule in the rulebase completes, either signaling an alarm (in case of mismatch) or not. Let us denote by ξ the waiting time specified in the rule, i.e., the time between the capture of the state variables and the rule matching. The latency of matching the rule is defined as δ-ξ. Note that ξ is subtracted from the total time since this is a function of the rule, which is determined by the characteristic of the payload system being verified and is not a reflection on the performance of the Monitor.

The scalability experiments would ideally be conducted in the TRAM configuration on the campus-wide network. However, this cannot be carried out in a controlled setting with exclusive access to the machines. Also, the data rate at the higher end of the range we are interested in stressing the Monitor with (~1.5 MBps) is much higher than the rate that can be stably sustained on the campus WAN (~40 kbps) as shown in our previous work [40]. Hence, a simple *Packet Generator* module is used, which emulates the receiver in TRAM. It sends 32 *datapackets* followed by an *ack.* The standard TRAM packet size used is 1400 Bytes. The Packet Generator can be used to emulate multiple receivers being verified by the Monitor. The Monitor is deployed on an unloaded machine in the same network as the Packet Generator. The Packet Generator actively forwards packets to the Monitor for matching, which uses the input rulebase for the TRAM receiver.

2.12.2. Scalability with Data Rate

We evaluate the performance of the Monitor under increasing data rate for a single receiver. Varying data rate is achieved by adjusting the interpacket delay deterministically in multiples of millisecond.



(a) (b)

Figure 2.6: (a)Variation of rule-matching latency with inter-packet delay at a receiver. (b) Microscopic view of the region of inter-packet delay between 7ms-8ms

.

We can see from Figure 2.6 that when the interpacket delay is small the latency of matching is high because the packets are coming in at higher data rate. As the interpacket delay increases beyond 7 ms, we see a sharp decrease in the latency because the degree of concurrency becomes lower than the parallelism available at the Monitor through its thread pool. The latency does not decrease to zero even as the interpacket delay decreases further because each rule matching has to go through a set of steps, such as mapping the packet to an event, searching rulebase for rules for an event, which takes a finite non-zero amount of time. For interpacket delay greater than 7 ms we see a flat curve leading to the conclusion that this is a desirable range of operation for the Monitor. We term the point of sharp increase as the *neck* and the region to the right of that point as the Target Load Range (TLR) (defined earlier in Section 2.7). To better characterize the rapid change between 7 ms and 8 ms, we perform another experiment by varying the data rate in smaller steps in this range. The result with increments of multiples of μs is shown in Figure 2.6 (b). This indicates the maximum data rate that can be handled by the single Monitor is 1.49 Mbps, corresponding to the inter-packet delay of 7.5 ms.

2.12.3. Latency with varying Rule Base

We measure the effect of size of the rulebase on the latency of matching. We vary the rulebase size by replicating the identical rulebase, once, 5 times, and 10 times for the experimental results shown here.

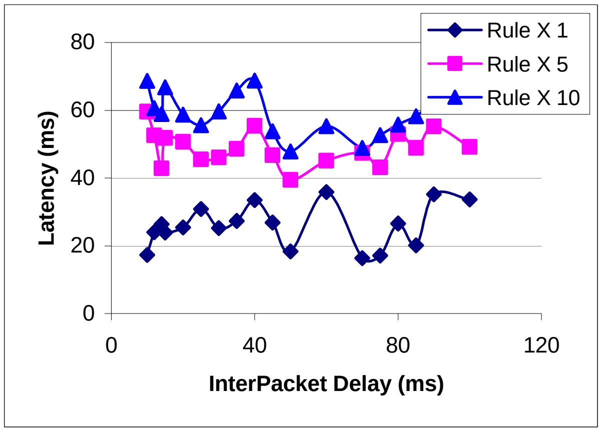
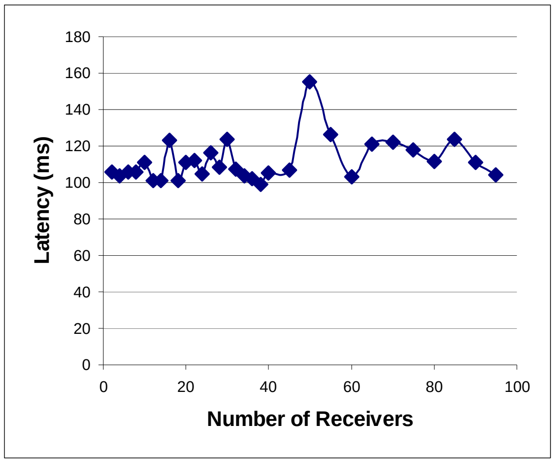


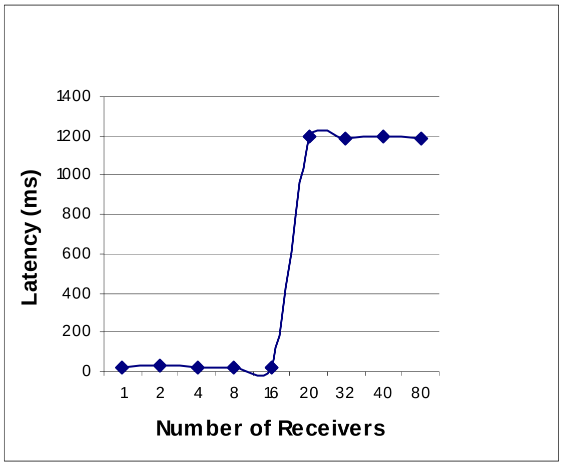
Figure 2.7: Variation of Latency with Rulebase Size

We see that latency curves for each rulebase size follow a similar horizontal trend with increasing inter-packet delay while the mean latency is higher for a larger rulebase size. The latency curve for a particular rulebase size is horizontal since we are operating in a region lower than the cutoff point for the particular thread pool size. The larger the rulebase size, the larger is the number of rules being matched and hence the higher latency. All the latency curves exhibit some non monotonicity and have peaks at nearly identical values of interpacket delay. This can be explained by the fact that for a certain value of the interpacket delay, say *D*, multiple rule matchings are scheduled concurrently leading to a higher latency for each individual matching. The increase in latency should be observed for the same delay D for larger sized rulebases as well since they are obtained by replication of the smaller sized rulebases. This may be characterized as harmonics the system is observing for the experimental conditions. If the experiment were to consider a larger set of distinct rules, such synchronized peaks will likely disappear.

2.12.4. Scalability with varying number of receivers

We vary the number of receivers verified by the Monitor and evaluate the latency of the rule matching. In this experiment we perform two tests. In the first test, we keep the aggregate data rate into the Monitor constant as we increase the number of receivers. Thus, if the number of receivers is doubled, the sender halves its data rate leading to each receiver's data rate being halved. We keep the aggregate data rate fixed at 280 kbps. In the second test, we keep the sender's data rate constant as we add more number of receivers for the Monitor to verify. Thus the aggregate data rate going into the Monitor increases linearly with the number of receivers. The number of threads is kept at 10 and a single instance of the rulebase is used.





(a) (b)

Figure 2.8: (a) Latency with number of receivers (Fixed Aggregate Data Rate = 280kbps) (b)Latency with number of receivers (Fixed sender data rate = 70kbps)

In the first test (Figure 2.8(a)), the latency of matching does not get affected by increasing the number of receivers. This is because the Monitor's thread pool is able to handle the overall data rate of these receivers. It might seem counter intuitive to see the latency not varying with the number of receivers. This can be explained as follows. All the temporal rules except those in category R2 are such that they allow only a single instance of the rule at any time for a single receiver. Let us call a rule which falls in this class, an "A-Rule". However the category R2 temporal rules allow multiple instances for a single receiver to be active concurrently (call this B-Rule). Higher the incoming event rate, higher will be the number of R2 rules that are active concurrently. In case of a few receivers with higher data rate for each (left hand side of the X-axis), the number of R2 rules will dominate over the number of A-Rules. The reverse is true for a large number of receivers with a lower data rate for each. The latency of matching an A-Rule is not significantly different from the latency of matching a B-Rule. Since the sum of the concurrently active rules of A and B type is approximately constant, we see a flat latency curve.

In the second test (Figure 2.8(b)), as the number of receivers increase beyond 16, the latency curve hits the neck and rises sharply till it reaches saturation for any further increases. This is because the thread parallelism fails to keep up beyond this point and the latency of matching rises to a very high value (about 25 times the value for smaller number of receivers). This value of 16 receivers can be taken as the cutoff point for the given data rate and the Monitor load should always be kept below the point. A load rebalancing interaction as described in Section 2.7 can be invoked if the load on a single Monitor increases beyond the cutoff point. We perform tests on sensitivity of Monitor with the ThreadPool size (plots omitted for space reasons). We observe that the sharp increase is still seen, though it occurs later because of parallel rule matching with increased thread pool size.

2.13. Coverage Results

2.13.1. Experimental Setup

The coverage experiments are performed to evaluate the detection accuracy of the Monitor system. A streaming video application running over TRAM is used as a workload with single sender and multiple receivers. A client can flag an error if it views degradation in its video quality because of slow data rate, which is represented by a threshold. The minimum and the maximum date rate specified by the client to TRAM are 20 KBytes/sec and 40 KBytes/sec. The TRAM sender provides a best effort service on the basis of these configuration parameters. We perform single level experiments where a single Monitor verifies the PEs followed by hierarchical experiments where Local and Global Monitors are deployed. Due to paucity of space, we only present the hierarchical results and show comparison with single level experiments. Details about the single level experiments can be found in [40]. The configuration used for our experiments is depicted in Figure 2.9.

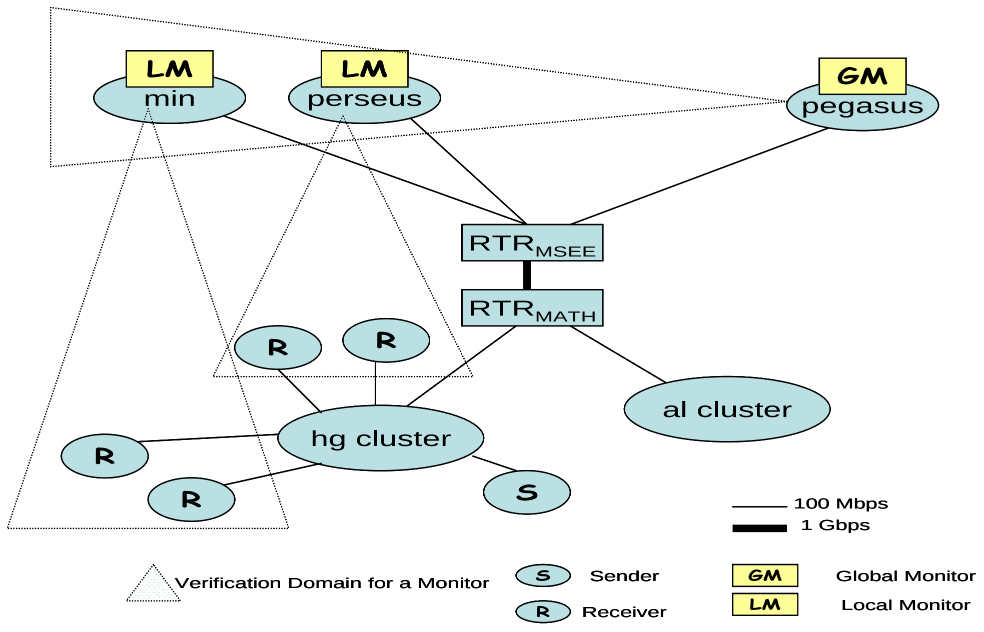


Figure 2.9: Physical Configuration of the Test bed used for Coverage Experiments

2.13.2. Fault Injection

Faults are injected into the protocol to cause invalid state transitions which should be detected by the Monitor. The faults are injected into the header of the TRAM packets before dispatching the packet to the receiver, which actively forwards it to the Monitor. This emulates the condition that the faulty packet is seen by the TRAM entities as well as the Monitor. The errors are injected continuously for a particular duration, denoted the *Burst Length*. This mode of error injection helps in emulating a real communication link where errors occur in bursts. The default burst length for the Monitor coverage measurements is 15 ms.

Three error models are used for the injections.

2. *Stuck at Fault*: In this error scenario, we simulate a stuck-at fault by changing a randomly selected header field into a different, valid, but incorrect value. The header field is always converted to the same value for all the packets in the entire burst length period.
3. *Directed*: The error injection is carried out into a randomly selected header field and its value changed to incorrect but valid values. Every packet is injected differently, unlike in the stuck at fault model.
4. *Random*: In this case, we choose a random header field and inject a random value into it. The injected value may not be valid with respect to the protocol.

We carry out two sets of run for each type of error injection, one with a *loose client* and another with a *tight client*. A *loose client* checks the data rate after every 4 Ack windows (approximately every 4.3 seconds) while a *tight client* checks the data rate after every Ack window. In practical terms, a *tight client* emulates a client less tolerant of transient slow downs in its received data rate.

There are four possible consequences of errors injected into the packets – exception is raised by the protocol (E), the client crashes (C), the client flags a low data rate error (DE), or no failure occurs (NF). It is possible for one, two, or all three of exception, crash and client data rate error to occur. The consequence of an error injection is represented as a tuple of up to three elements with the prefix "N" before a consequence denoting that the consequence did not occur. Thus (NE; NC; DE) denotes no exception, no crash, but client flagged a data rate error. When only a single consequence occurs, the notation can be abbreviated, as (DE) for the above case. Also, whenever an error is manifested in the protocol, the data rate ultimately drops leading to the data rate error (DE). If data rate error is *not* the only consequence, DE is dropped from the notation. The experimental runs, where the Monitor detects the failure before any of the protocol manifestations, are classified as Monitor detection. If the Monitor flags an alarm after an error has been manifested in the client (any of E, C, or DE), this is a case of error propagation and is classified as coverage miss. An error which does not lead to a failure but is flagged by the Monitor is categorized as a false alarm.

2.13.3. Hierarchical Monitor Results

The set up for hierarchical Monitor is shown in Figure 2.9. It is a two level hierarchy with each Local Monitor overseeing two receivers and a Global Monitor overseeing the two Local Monitors. Each kind of injection with each client (loose and tight) is carried out for 100 runs. A run is defined as an execution of the application with error injection where either the Monitor flags an error or the application has a failure or both. The first four columns are the different consequences of the error injection and are listed as: (Number of cases detected by the Monitor)/(Total number of such cases) (% Coverage of the Monitor). The definitions of the coverage misses have to be carefully considered in the hierarchical Monitor case. Consider a chain of overseeing Monitors for each receiver. A receiver is either verified by LM_1 (Local Monitor 1) and GM (Global Monitor), or LM_2 (Local Monitor 2) and GM. If either of the Monitors verifying an entity reports the error before the error manifests in the protocol, then the error is considered covered. The way the manifestation of the error in the protocol is defined differs for the Global and the Local Monitor. If the Global Monitor detects the error *after* the client reports the data error, it is still considered to be covered, while detection after an exception or crash is expectedly a miss. This relaxed definition accounts for the structure of the global rules, which imposes aggregation at the Local Monitor level and therefore, increases the delay between the erroneous packet being generated and rule matching at the Global Monitor. Also, detection by the Global Monitor can potentially convey more information about the error (such as, rate of spread) and a client data rate error is considered to be one which can be tolerated in the environment for transient periods while crashes or exceptions cannot.

The results from the injection are shown in Table 1. The results show the coverage miss by the Local Monitors and the entire Monitor system separately to bring out the advantages of deploying the two-level Monitor system. For the hierarchical Monitor system, the false alarm rate remains the same as for the single level case since all the false alarms come from the Local Monitors which remain identical in the two cases. The hierarchical Monitor system shows a high overall accuracy of 90.97%, an improvement of about 7% over the single level Monitor. The results from the injection are shown in Table 1. The results show the coverage miss by the Local Monitors and the entire Monitor system separately to bring out the advantages of deploying the two-level Monitor system. For the hierarchical Monitor system, the false alarm rate remains the same as for the single level case since all the false alarms come from the Local Monitors which remain identical in the two cases. The hierarchical Monitor system shows a high overall accuracy of 90.97%, an improvement of about 7% over the single level Monitor.

Table 1**:** Results of error injection with hierarchical Monitor

	No Exception No Crash Slow data rate (De)	**Exception No Crash (E;Nc;De)**	**Exceptio n Crash Slow data rate (E;C;De)**	**Missed Alarms by Hierarchica l Monitor System**	**False Alarm**	**Coverage By Hierarchical Monitor System**	**Coverage by Single Level Monitor**	**Improvemen t over Single Level**
Loose Random (LR)	29/29 (100%)	13/15 (87%)	2/2 (100%)	2/46 (4.34%)	8%	44/46 (95.66%)	42/46 (91.30%)	4.36%
Tight Random (TR)	28/29 (96.5%)	9/13 (69.2%)	3/3 (100%)	5/45 (11.1%)	10%	40/45 (88.88%)	37/45 (82.22%)	6.60%
Loose Directed (LD)	8/9 (89%)	30/32 (94%)	9/9 (100%)	3/50 (6.00%)	0%	47/50 (94.00%)	41/50 (82.00%)	12.00%
Tight Directed (TD)	12/14 (86%)	26/31 (83.8%)	5/5 (100%)	7/50 (14.0%)	0%	43/50 (86.00%)	41/50 (82.00%)	4.00%
Loose stuck at (LS)	22/22 (100%)	23/25 (92%)	1/1 (100%)	2/48 (4.17%)	4%	46/48 (95.83%)	42/48 (87.50%)	9.37%
Tight stuck at (TS)	24/26 (92%)	14/16 (88%)	4/7 (57%)	7/49 (14.2%)	2%	42/49 (85.80%)	39/49 (79.59%)	6.20%
				26/288 (9.02%)	12/300 (4%)	262/288 (90.97%)	242/288 (84.03%)	6.94%

This improvement is achieved by adding just two rules at the Global Monitor. The two rules correspond to aggregate data rate and nack rate. The results corroborate the need for a hierarchical setup of Monitors. The increase in coverage is most significant for the loose directed case (12%). On further investigation, it is found that the rule at the Global Monitor that checks the aggregate data rate is successful in pre-emptively detecting some cases which cause exceptions and crashes and therefore improves the coverage. As in the single level case, the system performs worse when the protocol's manifestation of error is exception, since it flags the error often after the exception has been raised. The Monitor system's performance in the directed and stuck-at injections with loose client is worse than for random injections. This can be attributed to the fact that in random injection, packets are injected with message type and sub-message type lying outside the defined set of protocol packet types. In such cases the packets are mostly discarded by the protocol. Thus the receiver does not see any data packet leading to it flagging the low data rate error. But in directed injection, different valid but incorrect types of packets are generated in every injection. This causes several invalid transitions in the protocol leading to an increase in the number of exceptions and crashes. However, the difference in performance is not sharp indicating that the global rules help to pre-emptively catch some of the failure cases. For the tight client in directed and stuck-at, the global rules do not make as much of a difference since the receiver data rate error detection dominates and often occurs before the global rules can flag the error.

2.14. Discussion

In this chapter we presented the basic structure of the Monitor framework. We laid the algorithmic framework for performing detection in distributed systems. Monitor performs detection while satisfying the design goals outlined in chapter 1. Fault detection is an important first step for developing a reliable system. The next step in making a distributed framework is providing accurate diagnosis of failures. Diagnosis helps in realizing the shortcoming in the designs and provides information about the environment in which the particular distributed system works successfully. This forms the next chapter of this thesis.

3. DETERMINISTIC DIAGNOSIS

We use the 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 mean the Monitor provides the *detection* and the *diagnosis* functionalities to the PE. 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 a rule base containing correctness and QoS rules 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 deduced by the Monitors during normal operation through the observed external messages is used for the diagnosis process. Loose assumption about the jitter on the communication channels, rather than the synchronous assumption, is made, while no assumption is made on the clocks at the different PEs or Monitors. A lower level Monitor, called the Local Monitor (LM), directly verifies a PE, while a higher level Monitor will match rules that span multiple LMs. 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.

The Monitors coordinate to perform distributed diagnosis if the verified PEs lie under different Monitors' verification domains. We assume Byzantine failures may occur in the Monitor system as well and we use replication to mask them. We enforce a hybrid failure model on the Monitors by the use an existing distributed security kernel called Trusted

Timely Computing Base (TTCB) [43].

3.1. System Model

The Monitor employs a stateful model for rule matching to perform detection and diagnosis, implying it maintains state that persists across messages. It contains a *rule base* consisting of *combinatorial* rules (valid for all points in time in the lifetime of the application) and/or *temporal* rules (valid for limited time periods). The Monitor observes only the external messages of the PEs. It can be placed anywhere in the infrastructure but typically not co-hosted with the PEs to avoid performance impact to the payload system. The desire to have low latency of detection and diagnosis suggests the placement of the Monitor in the vicinity of the PEs. The *Diagnosis Engine* is triggered when a failure is detected and it uses state information from the State Maintainer to make diagnosis decisions. The previous Monitor architecture in chapter 2 ([40])has been extended to add the diagnosis functionality.

The system comprises of multiple Monitors logically organized into Local, Intermediate, and Global Monitors. The *Local Monitors* (LMs) directly verify the PEs. An *Intermediate Monitor* (IM) collects information from several Local Monitors. An LM filters and sends only aggregate information to the IM. There may be multiple levels of IMs depending on the number of PEs, their geographical dispersion, and the capacity of the host on which an IM is executing. There is only a single Global Monitor (GM), which only verifies the overall properties of the network. The Monitor's functionality of detection 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 (similar to Lamport's clock [26][76]).

We assume that PEs can fail arbitrarily exhibiting Byzantine failures. 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 3.2.4). An anomaly in the behavior of the PEs detected by flagging of a rule triggers the diagnosis procedure. We assume that jitter on

PE $\rightarrow$Monitor link is bounded by *phase*(Δt). We further explain in Section 3.2.2 the need for such an assumption. It is important to note that this assumption is weaker than complete synchrony.

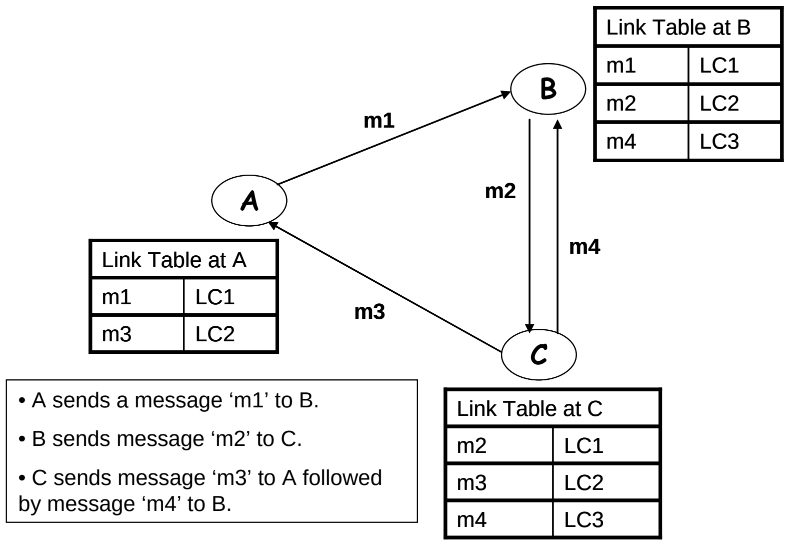
3.2. Deterministic Approach

Diagnosis in a distributed manner based on observing only external message exchanges poses significant challenges. It is essential 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, diagnosis must be based on these messages alone. In other words, the Monitor does not have perfect observability of the payload system's state. The PEs may lie within the domains of different LMs. In such cases, the diagnosis is a distributed effort spanning multiple Monitors at different levels (Local, Intermediate, and Global). In order to identify the faulty PE from among a set of suspect PEs, each PE is subjected to a *test* procedure. Since the Monitor treats PEs as black-boxes it is thus unaware 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 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. Causal order is obtained using the logical clock maintained by the Monitor for each verified PE, which is used to construct the *causal graph*.

3.2.1. 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[1] & (ii) An edge *e* contained in E, between vertices *v1* and *v2* (which represent PEs) indicates interaction between *v1* and *v2* and contains state about all observed message exchanges between the corresponding PEs including the logical clock (LC) at each end. 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. The links are also time-stamped with the local (physical) time at the Monitor, at which the link is created. An example of a causal graph is given in Figure 3.1 for the sequence of events described on the lower left corner.



Link Table at B	
m1	LC1
m2	LC2
m4	LC3

Link Table at A	
m1	LC1
m3	LC2

Link Table at C	
m2	LC1
m3	LC2
m4	LC3

- A sends a message 'm1' to B.
- B sends message 'm2' to C.
- C sends message 'm3' to A followed by message 'm4' to B.

Figure 3.1: A sample causal graph

For example in the Link Table for node C, message '4' is assigned a logical clock time 3. Message *m3* is causally preceded by message *m2* which is causally preceded by message *m1*. The messages may be received in different order at the Monitor because of the asynchronous nature of links.

3.2.2. Cycle and Phase

In modern distributed protocols, with thousands of communicating protocol entities, testing all the causally preceding messages is not feasible. We define a time window over

[1] We thus establish a correspondence between a PE in the payload system and a node in the causal graph. .

which the diagnosis protocol tests nodes. This time window is called a *Protocol Cycle* to differentiate it from a graph theoretic cycle in the causal graph. The start point of the *Protocol Cycle* denotes how far the diagnosis algorithm should go in history to detect faulty nodes[2]. 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. Let there be n PEs verified by the Monitor infrastructure. A reduced STD is maintained at the LM for every verified PE_k, denoted 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.

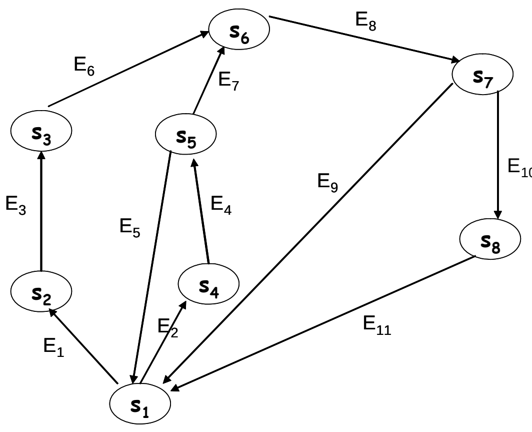


Figure 3.2: Sample STD for a PE P_1, illustration of Protocol *Cycle*

There could be several possible runs of different durations for a given PE each corresponding to a complete task (transaction) as defined in the protocol, e.g., a complete round of data and ack exchange. 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 $\vec{S}_{init} = \{S_{11}, S_{21}, S_{31}, \ldots, S_{n1}\}$. We define the *protocol cycle* as the completion of all the possible runs starting from $\vec{S}_{init}$. Each *protocol cycle* will encapsulate several graph cycles each of

[2] Henceforth, if there is no scope for confusion, we use the term cycle as shorthand for protocol cycle.

which includes the start state of the particular PE. Finding a protocol cycle is NP-complete since the known NP-complete problem of finding the Hamiltonian cycle can be reduced to it in polynomial time.

When a failure is detected in protocol cycle C_i, the checking has to be done till the beginning of C_{i-1} for a deterministic bug. The model for the deterministic bug is that if it manifests itself in state S_{ij} on receipt of event E_k for PE_i, then it *must* manifest itself every time PE_i goes through the same state and event. For a non-deterministic Heisenbug, the determination may have to go back to further cycle boundaries since by definition, a non-deterministic may not manifest itself repeatedly under the same conditions (same state and event). Alternate strategies may be needed if the number of states to be examined becomes too large through this approach. Then we can use the upper bound on the error detection latency in the system (e.g., as given through analysis in [22]) to come up with the cycle boundary. If we can provide a bound that any error in the application will manifest in time δ, we can limit the messages which need to be checked for errors as being no farther back in (physical) time than δ. If proactive recovery measures, such as periodic rebooting[77]**,** are used, then the time points at which the proactive recovery is performed can be taken as cycle boundaries. This is motivated by the claim that latent errors are eliminated at the proactive recovery points.

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. As the system is asynchronous and not FIFO, 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. Instead of the synchrony assumption, consider the following more relaxed assumption. 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*. If 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, it is guaranteed that send event M_1 happened before M_2 if $t_{L2} \geq t_{L1}+\Delta t$.

3.2.3. Suspicion Set

Flagging of a rule corresponding to a PE represented by node *N* in the causal graph indicates a failure *F* and starts the diagnosis procedure. Henceforth, we will use the expression "failure at node *N*" for a failure detected at the PE corresponding to the causal graph node *N*. Diagnosis starts at the node where the rule is initially flagged, proceeding to other nodes *suspected* for 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 $SS_{FN}=\{SS_N\}$. Let SS_N consist of nodes $\{n_1, n_2..., n_k\}$. Each of the nodes in SS_{FN} is tested using a test procedure which is discussed in Section 3.2.4. If a node $n_i \in SS_{FN}$ is found to be fault-free then it is removed from the suspicion set resulting in contraction of suspicion set. If none of the nodes is found to be faulty then in the next iteration suspicion set for the failure F is expanded to include the suspicion set of all the nodes which existed in SS_N in the previous iteration. Thus, in the next iteration $SS_{FN} = \{SS_{n_1}, SS_{n_2}..., SS_{n_k}\}$. Arriving at the set of nodes that have sent messages to *N* in this time window is done 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 $C \le t \le \tau_S + \Delta t$ 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.

3.2.4. Test Procedure

We define the test procedure for a PE to be a set of rules to be matched based on the state of the PE as maintained in the causal graph. This set of rules constitutes the *strict* *rule base (SRB)* and like the *normal rule base*, used for error detection, consists of temporal and combinatorial rules for expected patterns of message exchanges. 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 the 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. SRB of form <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 PE 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 of non-intrusiveness.

When an SRB rule is used to test a given link l_i in the causal graph, it uses as pre- and post-conditions in the rule events over a logical window of $\pm\Delta t$, the phase, measured from the logical time of l_i. This is attributed to the assumption of jitter bound on the communication link, namely, that a message at the Monitor cannot arrive out of order with respect to another message more than Δt away, originated at the same PE. Each rule in SRB has some *coverage* to verify a particular PE because it only tests a specific state and event. Therefore, a message sent by an entity in the Suspicion Set must be tested by running multiple rules from the SRB on it. We develop an analytical model on these assumptions in Section 3.4.2.

Like the normal rule base, the rules in the SRB are dependent on the state and the event of the link but the number of rules 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 faults manifesting nearly concurrently would result in multiple rules being flagged, leading to separate and independent diagnosis procedures for each of them.

3.2.5. 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 0).

(1) A failure F at PE N is detected by the local Monitor LM_i verifying it.

(2) LMi constructs the suspicion set SSN for the failure and adds it to SSFN.

(3) For every N'∈ SSN that belongs to the domain of LM1 , LM1 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 SSN and SSN' is added to the SSFN queue.

(4) For every N" belonging to SSN that is not under the domain of LM_i but under the domain of another Monitor LMj, LM_i sends a test request for N′′ and faulty link L" recursively to higher level Monitors till a common parent for LM_i and LM_j is found, which routes it to LM_j. LM_j tests N" and sends the result of the test back to LM_i through the same route. If N" is not faulty, then LM_j 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. In the first case, the node corresponding to which the link is diagnosed as faulty due to violation of rules in the SRB is considered as faulty. In the latter case, the diagnosis procedure terminates unsuccessfully.

3.3. Faults at Local Monitors

If an 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.

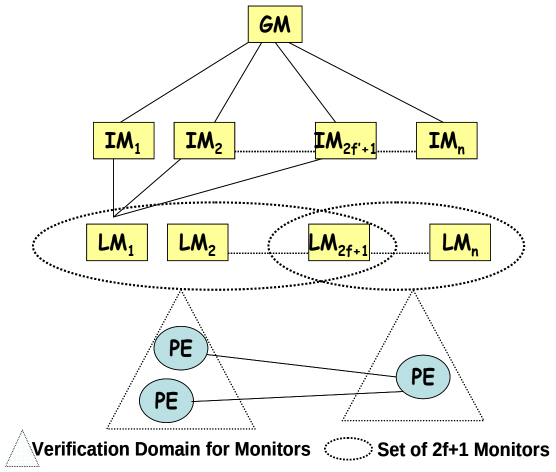


Figure 3.3: Redundancy in the Monitor hierarchy

Assuming there can be failures on up to *f* LMs, 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 alarm 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 fault 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 [28]). This problem is known to be equivalent to consensus [29], which requires a minimum of *3f+1* process replicas to be solvable in asynchronous systems with

Byzantine faults [43]. We reduce this number of LM replicas to *2f+1* using an existing method called the architectural-hybrid fault model [44] (Section 3.4.1.1).

The algorithm used by the LMs in a CS_{LM} to agree in an alarm 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 the $(f+1)^{th}$ 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. It guarantees that all correct LMs agree on the same alarms with a unique order number, ensuring an atomic order. Therefore, the algorithm guarantees that an IM receives identical alarms from all correct LMs verifying a PE.

3.3.1. TTCB and architectural-hybrid fault model

In this thesis, 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 (the TTCB) that can only fail by crashing [43]. 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 entities, like the LMs. These entities communicate in a world full of threats, some of them may even be malicious and try to cheat, but the TTCB is an "oracle" that correct entities can trust and use for the efficient execution of their protocol.

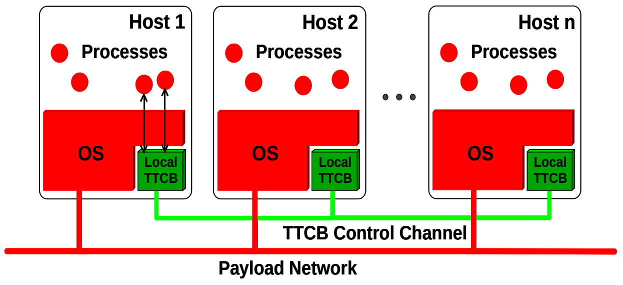


Figure 3.4: Architecture of *n* hosts with a TTCB

The design and implementation of the TTCB was discussed at length in [45] and here we give a brief overview relevant to its application in the Monitor system.
The local TTCB components are connected using a dedicated channel (Figure 3.4). 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.

3.3.2. 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 [44]. 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) [44]. Being a TTCB service, its code lies inside the local TTCBs and its communication goes in the TTCB control channel. When an LM wants to atomically multicast a message M, 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 an LM 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, these channels guarantee the authenticity and integrity of the messages. These channels could be implemented using SSL or TLS. Finally, when the TTCB has information that *f* LMs received M, it gives M & all LMs in CS_{LM} the next order number.

3.4. Faults at the Intermediate Monitors

Next we augment the model to allow IM failures by having a redundant number of IMs. To tolerate *f'* faults at the IM level at least *2f'+1* IM replicas must be used. Therefore all LMs in a Collaborative LM Set (CS_{LM}) send alarms to all IMs in a Collaborative IM Set, denoted by CS_{IM}. Output of replicas is voted on by a simple voter (GM in our case). 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. Secure coding methodologies, based on formal verification and static code analysis, can be used to build a fault-free GM. Possibility of faults in Monitors, forces an LM in CS_{LM} to accept a test request only if it receives *f+1* identical test requests from Monitors in CS_{IM}. An alternative design choice would be 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.

3.4.1. Flow of Control of Diagnosis with Failing Monitors

Assume that CS_{IM} initiates the diagnosis.

- Failure F at PE N is detected by the CS_{LM} verifying it, which constructs the suspicion set SS_N and adds it to SS_{FN}.
- The LMs assign an order to the alarm using the atomic broadcast protocol and send an alarm along with SS_{FN} up to all the IMs in CS_{IM}.
- The IMs wait for $f+1$ identical alarms and then start the diagnosis procedure.
- For every $N' \in SS_{FN}$ the (correct) IMs in a CS_{IM} send a test request to the CS_{LM} for verifying *N'*.
- Each $LM \in CS_{LM}$ that receives $f+1$ identical test requests from different IMs in CS_{IM} *tests* N' for correctness of the suspect link L' using multiple rules from the SRB for the particular event and state of the link.
- The test results are sent above to the IMs in CS_{IM} who vote 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}.
- If a PE N'' lies outside the verification domain of the IMs in CS_{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 Local Monitors verifying N''. The result of the test is sent back to the IMs through the same route. If N'' is not faulty, then the corresponding suspicion set is also sent along.
- The diagnosis procedure repeats recursively until a node is diagnosed as faulty, or until the cycle boundary is reached.

3.4.2. Analysis of Diagnosis Accuracy

For easier understanding and comparison, we follow a similar notation to that in [48]. Consider a *k*-regular directed graph with a node representing a PE and an edge representing message exchange between the PEs. 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. The probability of error propagation through the message is $\rho\lambda$. An error in the node can be caused by a fault in the node or due to an error propagated through one of the incoming links. A test executed on the node has a fault detection coverage c_i if the node n_i is faulty (i.e., probability of detecting a faulty node is c_i) 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. 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 $O \in \{0, 1\}$, 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, $z \in (0,T)$. 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 causing the failure. 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.

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

This equation matches with the one derived by Fussel and Rangarajan (FR) [48] 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 as $k\rightarrow\infty$ reduces to $d(1-e^{k\rho\lambda})$ because $\rho\lambda \rightarrow 0$. We can rewrite the equation 1(e) as:

Prob(π| *z-ones*) = 1 / 1 + F(z) ; where F(z) = $((1-\lambda)/\lambda).(B/A)^z.((1-B)/(1-A))^{T-z}$

We claim that 1(e) is a monotonically increasing function of *z*. Note that A and B $\in$ (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 [15][78]). 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 $A>B$, $\beta<1$ and $\mu>1$ and therefore Prob(π| *z-ones*) increases with *z*. This can also be proved through showing $d(Prob(\pi|\ z\text{-}ones))/dz > 0$. 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 as per the definition in [48].

***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 [48] and simply map our algorithm's testing behavior to theirs.

In [48], the number of tests grows with N as $\alpha(N)log(N)$ and thus asymptotically (w.r.t. *N*) 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 [48] 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 [48]. Hence the algorithm tends to perfect behavior asymptotically 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* since equation 1(e), $\lim_{T\to\infty}\lim_{z\to T} Prob(\pi\,|\,z-ones)$ approaches 1. Eqn. [1(e)] is an increasing function of *z*. Hence, we find the value $z = z_{th}$ which provides Prob(π| *z-ones*) = 0.5 and set the algorithm to conclude the node is faulty if $z > z_{th}$ and non-faulty otherwise. Equating eqn. 1(e) to 0.5 and simplifying we get:

$$z_{th} = \frac{\log(\frac{1-\lambda}{\lambda})}{\log(\frac{A(1-B)}{(1-A)B})} + T\frac{\log(\frac{1-B}{1-A})}{\log(\frac{A(1-B)}{(1-A)B})}$$

Therefore, using the property of Prob(π| *z-ones*) being an increasing function of *z* and Theorem 1 in [48], we conclude that our diagnosis algorithm is optimal in its class 3AM.

3.5. Experiments and Results

The diagnosis protocol implementation is demonstrated by running a streaming video application on top of TRAM. The Monitor is given the SRB along with the STD and the normal rule base as input. 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 by 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 transmission-reception state to prevent a malicious receiver from *hello* flooding. In our experiments the number of SRB rules to test a link varied from 4 to 8 depending on the state of the link. An extensive list of NRB and SRB used in our experiments are given in Appendix B.

3.5.1. Optimistic and Pessimistic Link Building

During the normal operation of the protocol, the Monitor adopts a *lazy approach* (euphemistically, *optimistic approach*) to build the causal graph. Each incoming (outgoing) message to (from) a node is stored in a vector of incoming (outgoing) links for that node. A linkID (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. For this link to be completed in the causal graph, a matching is required between the sending and the receiving PEs' messages. The link A→B will be matched once the message sent by A and the corresponding one received by B are seen at the Monitor. Matching all the incoming packets during runtime, referred to as the *pessimistic approach,* entails an enormous overhead. This approach results in low diagnosis latency but also results in some links not being matched at runtime due to overload 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. Hence, in the optimistic approach, at runtime, the Monitor simply stores the link in the causal graph and marks it as being unmatched. Link matching is performed when diagnosis is triggered on failure. We perform experiments give a comparative evaluation of the optimistic and the pessimistic approaches.

3.5.2. Fault Model & Fault Injection

For exercising the diagnosis protocol, we perform *fault injection* in the header of the TRAM packets transmitted by the sender. It must be noted that the faults are considered to be accidental faults, which may be of arbitrary nature. Malicious nodes launching deliberate attacks on the system are beyond the scope of this thesis. The Monitor inspects only the header and is not aware of the payload. Hence the faults are only injected into the packet header. The fault is *injected* by changing bits in the header after the PE has sent the message. Note that the emulated faults are not simply message errors, but may be symptomatic of faults in the protocol itself. For example, a faulty receiver may send a Nack instead of an Ack on successfully receiving a data packet. Errors in message transmission can indeed be detected by checksum computed on the header. However, the Monitor is responsible for detecting & diagnosing errors in the protocol itself, which are clearly outside the purview of checksum. As explained previously, the faults at the

Monitor level are masked through replication. The strict rules are used to diagnose the faults with each rule having some coverage. We use similar fault injection as for detection experiments. :*(a)Stuck-At injection*: For all packets in the burst length a randomly selected header field value is changed to a random but valid value. *(b) Directed Injection*: For each packet a specific header field is chosen for one experiment and changed to a random but valid value, with different values in different runs. *(c) Specific Injection*: Specific injections consist of slow data rate, dropping acks, and hello message flooding. Burst error is chosen as the fault model over single error since the protocol is robust enough that single errors are almost always tolerated by inbuilt mechanisms in the protocol.

3.5.3. Test Set Up and Topology

Figure 3.5(b) illustrates the topology used for the accuracy and the latency experiments on TRAM with components distributed over the campus network (henceforth called TRAM-D), while Figure 3.5(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. The min. data rate in TRAM needed to support the 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.

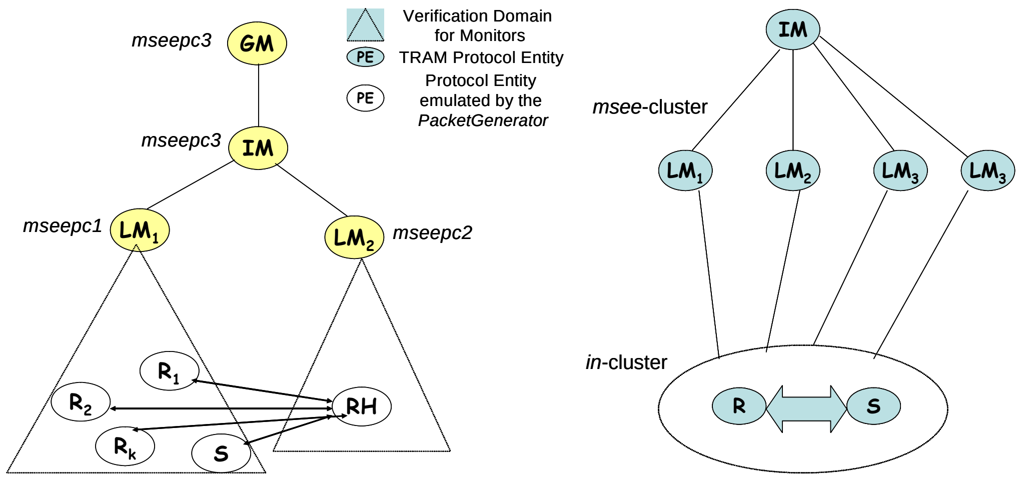


Figure 3.5: Topology used for accuracy and latency experiments in (a) TRAM-L (b) TRAM-D

3.5.4. 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 experiments with both the optimistic and the pessimistic approach of link building. There are thus two dimensions to the experiments – the link building 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.

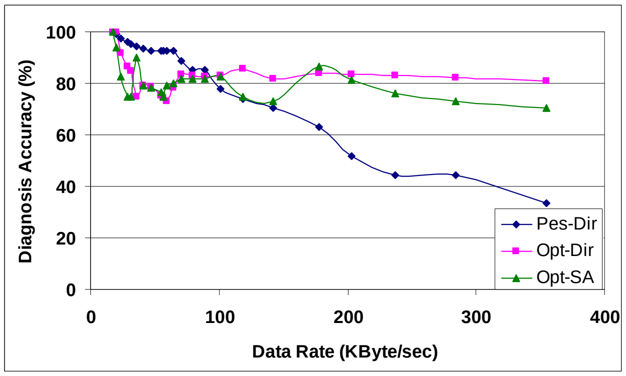


Figure 3.6: Variation of Accuracy with Data Rate

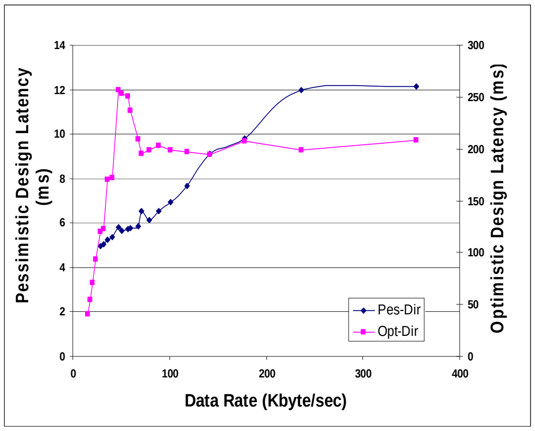
Figure 3.6 shows that for *Pes-Dir* accuracy is a monotonically decreasing function with data rate. Diagnosis accuracy drops to a low of 33% for data rate at 355 KByte/sec.

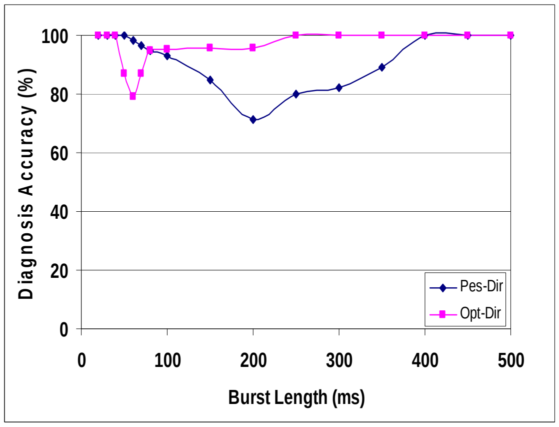
Rate mismatch between the matching of links for causal graph creation (slower process) and the arrival of packets at high data rates (faster process) causes this decrease. Lack of adequate buffer causes packet drops leading to missing links in the causal graph leading to a drop in accuracy. Another factor is lack of 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 *Pes-Dir*, here the accuracy does not drop below 80%. The link matching and the causal graph completion are triggered when 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 3.7 (a) graphs the latency of diagnosis with increasing data rate. Notice the significantly higher latency for the optimistic case compared to the pessimistic one. 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 of incomplete causal graph leading to an inaccurate early termination. On the other hand in the *Opt-Dir* scenario, the latency keeps increasing with data rate. This is attributed to the lazy link matching which happens during diagnosis, high 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 KBytes/sec to isolate the effects due to high data rate. Diagnosis as shown in Figure 3.7(b) is accurate for low and high values of burst length. For small burst length, a small number of incorrect packets gets 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, more incorrect packets come in, helping in diagnosis. 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.



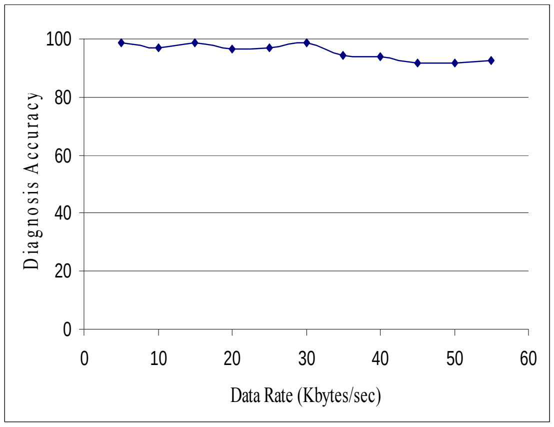


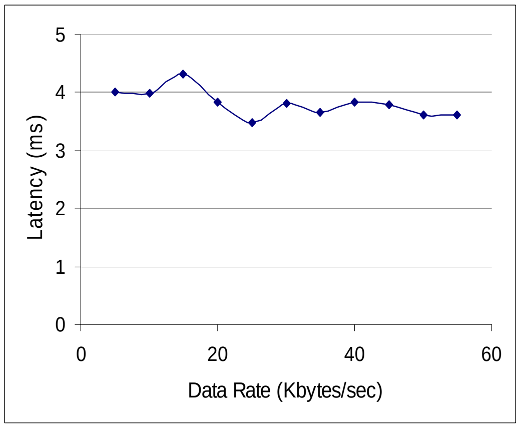
(a) (b)

Figure 3.7:(a) Variation of Latency with Data Rate and (b) Diagnosis Accuracy with Burst Length for Optimistic and Pessimistic Approaches.

3.5.5. 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. The latency and accuracy values are averaged over 200 diagnosis instances for each data rate. Figure 3.8 (a) shows that the accuracy of diagnosis drops from a high of 98% at 15 KB/s to 91% for 50 KB/s. 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.





(a) (b)

Figure 3.8:(a) Diagnosis Accuracy and (b) Latency variation with increasing data rate in TRAM

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. Figure 3.8 (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.

4. PROBABILISTIC DIAGNOSIS

The existing view of Monitor diagnosis is a deterministic process whereby the PE responsible for initiating the chain of errors can be deterministically identified. This is however, an over-simplification of reality. In practical deployments, it is often the case that the Monitor does not have perfect observability of the PE because the network between the Monitor and the PE is congested or intermittently connected. This is particularly feasible because the application is distributed with components spread out among possibly distant hosts, and the Monitor and the payload systems may be owned by different providers and run on different networks. Next, the Monitor itself has finite resources and may drop some message interactions from consideration due to exhaustion of its resources (e.g., buffers) during periods of peak load. It is desirable that the Monitor be non-intrusive to the payload system and therefore the testing process comprises testing invariants on properties of the payload system behavior deduced through the observation process. Thus, no additional test request is generated for the PE. However these tests are not perfect and may generate both missed and false alarms. Hence, a probabilistic model is needed to assess the reliabilities of the PEs. Finally, the nodes have different error masking abilities and thus different abilities to stop the cascade of error propagation. This masking ability is not known deterministically. All these factors necessitate the design of a probabilistic diagnosis protocol.

The goal of the probabilistic diagnosis process is to produce a vector of values called *Path Probability of Error Propagation* (**PPEP**). For the diagnosis executed due to a failure at node *n*, PPEP of a node *i* is the conditional probability that node *i* is the faulty PE that originated the cascaded chain of errors given the failure at node *n*. The PEs with the *k*-highest PPEP values or with PPEP values above a threshold may be chosen as the faulty entities.

Our approach to probabilistic diagnosis builds on the structures of Causal Graph and Suspicion Tree from the deterministic diagnosis protocol. A probabilistic model is now built for each of node reliability, error masking ability, link reliability, and Monitor overload. The probability values for some of the components are partially derived from history maintained at the Monitor in a structure called the *Aggregate Graph* (**AG**). A consequence of moving the fine-grained information from the CG to the summarized AG is that it reduces the amount of state to be maintained at the Monitor. The probability values from each component are combined for the nodes and the links in the path from a node *i* to the root of the Suspicion Tree to come up with *PPEP(i)*. The combination has to be done with care since the probabilities are not all independent. For example, overload condition at the Monitor is likely to persist across two consecutive messages in the payload system.

4.1. Probabilistic Diagnosis Model

The model of the payload system assumed in the deterministic diagnosis process is overly simplistic in several deployments. The relative placement of the Monitor and the verified PEs may cause imperfect observability of the external messages. The Monitor may be resource constrained and may not be able to accommodate periodic surges in the rate of exchanged messages among the PEs. The tests used to diagnose the PEs may be imperfect and the inherent characteristics of the PEs, e.g., their error masking capabilities, may not be accurately known to the Monitor system. The probabilistic diagnosis protocol handles these limitations, which were assumed away in the deterministic protocol. For a given failure the goal of the probabilistic diagnosis process is to assign a probability measure for every node to be the cause of that failure.

It may be infeasible storage wise to keep the entire state of all PE interactions till the cycle boundary. Perhaps more importantly, performing diagnosis on all the nodes till the cycle boundary will make the latency of the diagnosis process unacceptably long. However, it is not desirable to completely flush the old state in the CG as the prior information could be utilized to provide historical information about a PE’s behavior which may aid in the diagnostic process. Our solution is to aggregate the state information in the CG at specified time points and storing it in an *Aggregate Graph* (AG).

As in the deterministic diagnostic approach, incoming information in the Monitor is initially stored in a Temporary Links (TL) table where it is organized based on the source node, destination node and event type as the primary keys. For this link to be completed in the CG, a matching is required between the sending and the receiving PEs' messages. The link A→B will be matched once the message sent by A and the corresponding one received by B is seen at the Monitor. This information has to be matched and organized into the CG for diagnostic purposes. When the high water mark for the TL (HW_{TL}) is reached, then as many links as can be matched are transferred to the CG while those that cannot be matched but are within the phase from the latest message are kept in the TL. Remaining links in the TL are moved to the CG as unmatched links.

4.1.1. Aggregate Graph

The Aggregate Graph contains aggregate information about the protocol behavior averaged over the past. The AG is similar to CG in the structure i.e. a node represents a PE and a link represents a communication channel. The link is formed only if at least one message in the past has been exchanged between the entities. The links are directed and unlike the CG there is a single directed link between A and B for all the messages which are sent from A to B. Each node and link has some aggregated information stored and continuously updated which aids in the final diagnosis. The AG contains some node level information (such as, the node reliability) and some link level information (such as, the reliability of the link in the payload system). These information fields are formally defined in Section 4.1.2.

The time duration between consecutive CG to AG conversions is referred to as a *round*. State information is transferred from the CG to the AG if the high water mark (HW_{CG}) is reached after a TL to CG conversion. The amount of information kept in the CG is equivalent to that for a phase around the latest event. It is important for diagnosis accuracy that information stays for some time in the CG and is not immediately transferred to the AG. Therefore the size of the CG should be significantly higher than that of the TL.

4.1.2. Probabilistic Diagnosis Algorithm

The operation of the diagnosis protocol has two logical phases: (1) The actual diagnostic process that results in a set of nodes being diagnosed as cause of failure; (2) Information from the diagnostic process being used to update the information present in the AG. Let us first look at the diagnostic process.

4.1.3. Diagnosis Tree

As in the deterministic diagnosis case, the CG is used to calculate the set of suspicion nodes, tracing back from the node where the failure was detected. A *Diagnosis Tree (DT)* is formed for failure *F* at node *D*, denoted as $DT_{FD.}$ The tree is rooted at node D and the nodes which have directly sent messages to node D denoted by SS_{D1} are at depth 1 and so on. Since the CG is finite size, the tree is terminated when no causally preceding message is available in the CG after some depth *k*.

The sample DT created from the sample CG in Figure 3.1 is shown in Figure 4.1. The numbers at the links correspond to the link IDs. The path *P* from any node *N* to the root *D* constitutes a possible path for error propagation and the probability of path *P* being the chain of error propagation is given by the *Path Probability of Error Propagation* (PPEP).

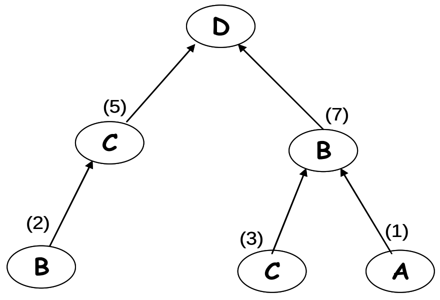


Figure 4.1: Sample DT for the CG in Figure 3.1.

Definition: ***PPEP(N, D)*** is defined as the probability of node *N* being faulty and causing this error to propagate on the path from *N* to *D*, leading to a failure at *D*. This metric depends on the following parameters:

(1) **Node reliability** – The node reliability is a quantitative measure of the PE corresponding to the node being faulty. The PPEP for a given node is proportional to its node reliability. The node reliability is obtained by running the tests from the SRB relevant to the current state and the event at the node in the CG. The result from the CG is aggregated with the previously computed node reliability (n_r) present in the AG. Let c be the combined coverage of the tests in the SRB. Then node reliability is updated as $n_r = (1 - \rho)c + \rho\, n_r$, where ρ is the weight used for current coverage. The node reliability is maintained for each node in the AG.

(2) **Link reliability** – The link reliability quantifies the Monitor's estimate of the reliability of a link in the payload system. Since the Monitor does not have a separate probe for the quality of the link, it estimates link reliability (l_r) by the fraction of matches of a message reported from the head of the edge (sender) with that reported from the tail of the edge (receiver). The PPEP for a given node is proportional to the link reliability, because high link reliability increases the probability of the path being used for propagating the error. The link reliability is maintained for each edge and each event in the AG.

(3) **Error Masking Capability (EMC)** – The error masking capability (e_m) quantifies the ability of a node to mask an error and not propagate it through the subsequent links on the DT towards the root node D. The PPEP for a given node is inversely proportional to the EMC values of nodes in the path since the intermediate nodes are less likely to have propagated the error to *D.* With the DT in Figure 4.1,

$PPEP(C, D) = n_r(\mathrm{C}) \cdot l_{r(C,D)}$; $PPEP(B, D) = n_r(\mathrm{B}) \cdot l_{r(B,C)} \cdot (1 - e_m(C)) \cdot l_{r(C,D)}$

Note that collusion among PEs reduces the coverage of the diagnosis when active forwarding is used. Thus, a sequence of PEs in a chain may omit to forward the messages to the Monitor though they are sent in the payload system. This will cause the Monitor to reduce the PPEP of the path through the colluding PEs. For an autonomous system, the parameters used in the diagnosis process should be automatically updated during the lifetime of the system as more failures and message interactions are observed and this forms the topic of our discussion next.

4.1.4. Calculating Node reliability

The objective is to assign node reliabilities to nodes corresponding to PEs, based on the results of the rules from the SRB. Let the set of tests that can be applied to the node *i* based on the event and the state be T_i. This set is partitioned into two sets A and A', depending respectively on if the test returned a value of 1 or 0. The weight of test $T_{i,j}$ is $w_{i,j}$. Then, the reliability of node *i* is given by $n(i) = \sum_{T_{i,j} \in A'} w_{i,j} / \sum_{T_{i,j}, \forall j} w_{i,j}$.

The weight of a test is proportional to the following factors: the frequency of invocations ($w^{(f)}$) where the test gave the correct result, i.e., agreed with the ultimate diagnosis by the Monitor; and whether the test examines state for a period of time greater than the transients in the system ($w^{(r)}$). The overall weight is calculated as $w_{i,j} = w_{i,j}^{(f)}.w_{i,j}^{(r)}$, where the two terms correspond to the two factors.

4.1.5. Calculating Link Reliability

At the time of formation of the AG , link reliability of the edge from A to B is calculated as follows:

$l_{r(A,B)} = n_m / n_t + n_m$ where n_m = Number of matched edges for A to B communication and n_t = Number of unmatched edges from A to B.

Subsequently, when CG to AG conversion takes place, link reliability in AG ($l_{r(A,B)}$) is updated with the link reliability for the current round ($l_{r\ c}$) as $l_{r(A,B)} = (1- \rho)l_{r\ c} + \rho l_{r(A,B)}$. Note that there may be multiple links between A and B for different states in which they have communicated. This design is influenced by the intuition that faults are state dependent. Notice that in the PPEP calculation, the edge reliabilities of adjoining edges are multiplied though the events are actually not independent. The explanation is given considering a linear chain of communication from C to B and B to A. The probability of a successful communication from C to A is P(C$\rightarrow$B is successful)$\cdot$P(B$\rightarrow$A is successful|C$\rightarrow$B is successful). In the link reliability formulation, the dependence is implicitly taken into account since the matched and unmatched link count on the B$\rightarrow$A link is affected by the events on the C$\rightarrow$B link.

4.1.6. Calculating Error Masking Capability

Assume that in the DT of Figure 4.1, nodes C and A at the same depth 2 are both faulty but PPEP(C) is the highest and PPEP(A) is low whereby node C is diagnosed as faulty and node A is not. In the Monitor system, node B is taken to have masked the error and not propagated it if the following three conditions are satisfied: (i) Running SRB rules on B yields a low value (*c(B)*); (ii) Running SRB rules on A yields a high value (*c(A)*); and (iii) Link reliabilities $l_{r(A,B)}$ and $l_{r(B,D)}$ are high (to ensure that the error must have propagated). The increment $\Delta(EMC)$ is thus

$$\Delta(EMC) = EMC_{\text{prev}} \frac{c(A) \bullet l_{r(A,B)} \bullet l_{r(B,D)}}{c(B)} \text{ and } \mathrm{EMC}_{\text{new}} = \mathrm{EMC}_{\text{old}} + \Delta(EMC)$$

We decrease the EMC for every intermediate node residing on a path which is finally diagnosed to have caused the error propagation.

4.1.7. Distributed PPEP

The PEs may be spanning several networks and even organizational boundaries and be verified by different Monitors each of which constructs a part of the AG and the CG obtained from its local information. During diagnosis it is quite likely that the DT contains PEs which are verified by some other Local Monitors. This entails the requirement of distributed diagnosis. Complete transfer of the local AGs and CGs to construct global information at higher level Monitors is not scalable. Instead, we leverage the fact that due to the multiplicative form of the PPEP computation, the PPEP value can be computed incrementally by each Monitor for the part of the DT under its verification domain. Assume in Figure 4.1, that nodes B and C are monitored by LM_1 and C and D by LM_2 and the diagnosis is performed by LM_2. In order to calculate PPEP(B, D) for the path (B-C-D) LM_2 needs n_r(B) , $l_{r(B,C)}$, e_m(C), and $l_{r(C,D)}$ of which the first two are not available locally. Therefore, LM_1 sends PPEP(B,C) for a failure at C to LM_2. In general, for a path A, X_1 X_2 ….X_n, ,B, PPEP(B, A) can be recursively written as: $PPEP(B, A) = l_r(A,X1) \cdot e_m (X_1) \cdot PPEP(B, X1)$.

4.2. Choice of Applications for Experiments

In this thesis we have applied the detection and diagnosis framework i.e., the Monitor to a few distributed applications. Before choosing a distributed application for verification by the Monitor, we scrutinized a multitude of applications looking for following features:

– The application should have multiple entities, especially distributed deployment.
– The entities should interact amongst each other using some kind of messages which are externally observable. This is necessary because Monitor looks at the externally observable messages only.
– The application protocol should have a defined behavior which represents the correct/normal behavior. This could be specified in terms of a state transition diagram, UML or Petri Net. A precisely defined protocol is a necessary condition.
– The application entities individually should have a notion of states and transitions amongst these states. This state diagram should be relatively complex. The complexity here refers to not only the number of states but also to the transitions amongst the states.
– The overall state transition diagram for the entire application protocol should be complex. The overall state transition diagram for the protocol comprises of a combination of the individual state transition diagrams of the entities.
– The applications should be of practical importance and used in providing important services.

In trying to meet the above requirements we choose to perform extensive experiments on TRAM [9] and a 3-tier e-commerce system. Beside we also studied Session Initiation Protocol (SIP) and applied our rule-base to form detection rules for verification of SIP.

All three protocols are used extensively in today's internet.

Section 2.9 provides details of TRAM describing the variety of messages which are exchanged between the entities of TRAM (sender, RHs, and receivers). The TRAM protocol is also more challenging because of both unicast and multicast messages being exchanged amongst the entities. This makes the detection process of the Monitor to be competent enough to encompass both kinds of messages. We studied the TRAM protocol especially for its availability of code to be deployable and provide extensive measurements for our study. As the source code and write-up on TRAM were readily available and it met the criteria of our selection process, we chose TRAM as one of the protocols for our experimental study. Because of the rich set of features which TRAM has it can be closely compared to other protocols thus making our Monitor scheme generalizable. For example: TRAM has congestion control mechanisms which are similar to congestion control mechanisms in other internet protocols such as TCP. Due to close interactions between the entities of TRAM, there are a lot of situations where errors can propagate from sender down to the receivers and vice versa. An example of error propagation is sender sending down data at a slow rate. This will cause the RH to send slow data down to the receiver. Thus causing error propagation from sender to the receiver. This makes the diagnosis challenging and interesting to study how Monitor's performance is affected in scenarios of error propagation.

In using TRAM for our experiments, we did study the literature in-depth to build the state transition diagrams and rule base. The only changes made to the source code of TRAM are for the purposes of measurements and no simplification is made for the Monitor.

The second application which we have used extensively in our experiments is a 3-tier e-commerce application. E-commerce applications form the backbone of the plethora of services which one finds on the internet. From e-banking to buying books on Amazon, e-commerce applications are ubiquitous today. It is commercially one of the most important distributed systems which we find in use in everyday practical life. A typical e-commerce system in its simplest form consists of 3 tiers: *web-tier*, middle or *application tier* and back-end data base or the so called *data base tier*. The tiers are usually deployed in a distributed setting. The e-commerce systems supporting the services on the internet are much more complex in architecture. Each of the tiers is further sub-divided into components which might be replicated for higher availability. There is a *sprayer* which re-routes the incoming transactions to one of the replicas thus providing load balancing. Each of the tiers interacts with each other through exchange of messages.

A primary reason for choosing the e-commerce test-bed is because of there is a large industrial research community interested in providing reliable e-commerce system. There are multiple groups from both IBM research and HP labs working on trying to find solutions for improving the reliability of e-commerce systems for over a decade. Some of the research work found in [65][111] addresses development of reliability primitives in e-commerce systems. The reason why understanding e-commerce systems is challenging is the complex interactions which occur between the multiple tiers. Besides the inter communications between the tiers, there is a lot of intra communications which goes on inside a tier. For example in an application server like JBoss, there are messages which are exchanged between enterprise java beans (EJBs) and servlets. Each EJB and servlet can have its own state transitions and state variables (for example see section 6.3.1). Interaction between multiple EJBs causes the entire state transition diagram to be a composite of each individual state transition diagram.

In choosing a state of the art diagnosis approach for comparing Monitor's diagnosis process we chose Pinpoint (see section 6.2). Pinpoint also demonstrates the efficacy of its generic diagnosis approach on an e-commerce testbed. In order to provide a fair comparison between Pinpoint and Monitor's diagnosis approach we chose the same e-commerce test-bed as used by Pinpoint for its experiments. Thus 3-tier e-commerce test-bed was chosen for our experiments because of its complex interactions, ease of availability and wide research interest. We have used a distributed setting of the 3-tier framework. The front user interface comprises of PetStore application which is similar to an online store like Amazon. The middle tier consists of JBoss application server running on the back-end Cloudscape database[108]. The PetStore application consists of over 50 different components (like EJBs and servlets) each having a different state transition diagram. There is a rich set of interactions between the EJBs and servlets which can cause error propagation. For example: A servlet function calls a function in EJB which returns null. This null return can cause servlet to crash and hence causing error propagation. This makes the 3-tier system interesting and challenging to study under Monitor's approach to detection and diagnosis. The only modification which is made to the source code is inserting message traps for making the communication observable to the Monitor. There is no special change which is made to the application in order to achieve better diagnosis.

4.3. Experiments and Results

4.3.1. Application

Similar to previous experiments, we deploy the Monitor system across the Purdue campus-wide network to monitor the reliable multicast protocol (TRAM).

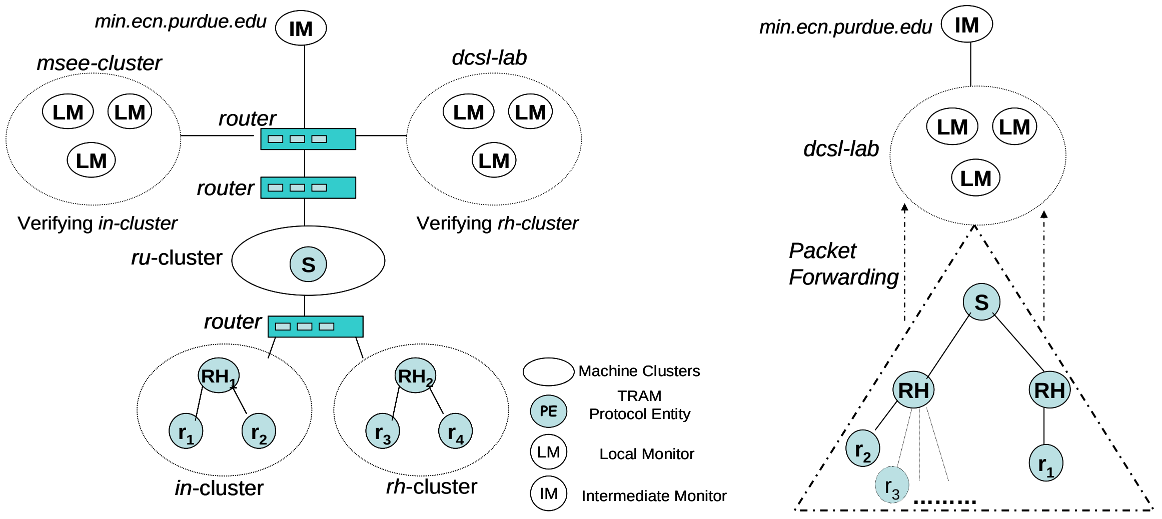


Figure 4.2: (a) Physical topology of test-bed (TRAM-D) (b) The emulated TRAM configuration (TRAM-L)

Figure 4.2(a) illustrates the topology used for the diagnosis experiments on TRAM with components distributed over the campus network (as previously called TRAM-D), while Figure 4.2(b) shows the topology for an emulated local deployment of TRAM (TRAM-L) used in these set of experiments. 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).

Running a larger configuration of receivers on TRAM-D was not possible because of a synchronization problem in the vanilla TRAM code obtained from [70] which causes receivers to disconnect unpredictably. The TRAM entities are verified by replicated Local Monitors with one Intermediate Monitor above. The TRAM entities do active forwarding of the messages to the respective LMs. The routers are interconnected through 1Gbps links and each cluster to a router through a 100Mbps link. Each cluster machine is Pentium III 930.33 MHz with 256 MB of RAM.

4.3.2. Rule Base

The Normal Rule Base (NRB) and the Strict Rule Base (SRB) for TRAM are input to the Monitor. The exhaustive enumeration of rules in the rulebases for the experiments are in Appendix B. Recollect that the SRB verifies the messages sent by the PEs over the *phase* interval which is a much smaller window compared to that of NRB. A few examples of the SRB rules used for our experiments are: "*O S1 E11 1 S3 E11 30 1*" This rule states that if there is one data message (E11) in state S1, then at least 30 more E11 links should be present in the CG. The last value "1" is the weight assigned to this rule. For these experiments all SRB rules are assigned equal weights. "*HO S6 E1 1 S6 E9 1 1*" This hybrid outgoing rule (HO) verifies that on receiving a *hello* message(E1) in state S6, the receiving entity must send a *hello-reply* (E9) within the same phase for liveness.

4.3.3. Fault Injection

We perform *fault injection* in the header of the TRAM packet to induce failures. We choose the header since the Monitor's current implementation only examines the header. A PE to inject is chosen (TRAM sender or receiver) and a burst length worth of faults is injected. The fault is *injected* by changing bits in the header for both incoming and outgoing messages. A burst length is chosen since TRAM is robust to isolated faults. The burst may cause multiple detections and consequently multiple concurrent diagnoses. Note that the emulated errors are not simply message errors, but may be symptomatic of protocol faults in the PEs. Errors in message transmission can indeed be detected by checksum computed on the header but protocol errors cannot. Three types of fault injection as used previously(explained again below) are employed:

(a) *Random Injection*: A header field is chosen randomly and is changed to a random value, valid or invalid *w.r.t.* the protocol. If the injected value is not valid, then a robust application check may drop the packet without processing it.

(b) *Directed Injection*: A randomly chosen header field is changed to another randomly chosen value, which is valid for the protocol.

(c) *Specific injection*: This injection is carefully crafted and emulates rogue or selfish behavior at some PE.

4.4. Experiments and Results

Accuracy of diagnosis is defined as the ratio of number of correct diagnosis to the total number of diagnosis performed. Correct diagnosis is when the PE flagged as faulty, i.e. the PE with highest PPEP, is the PE that was injected with faults. The latency is the time measured from the point to detection to the end of diagnosis.

4.4.1. Latency and Accuracy for TRAM-D

4.4.1.1. Random Injection at Sender

Effect of Buffer Size: The fault injection causes error propagation to the RH and the receivers causing independent diagnoses at each entity. Figure 4.3 shows the latency and accuracy of diagnosis with increasing maximum buffer size for the CG. TRAM sender's data rate is kept at a low value of 15 Kbits/sec to avoid any congestion effect. Each data point is averaged over 300 diagnosis instances. Latency of diagnosis increases with buffer size since on an average the CG will be storing more links, leading to more nodes in the DT and hence higher processing for calculating the PPEP. The fundamental factor that determines the latency is the size of the DT, which depends on how full the CG was when the diagnosis was triggered, which is bounded by the CG buffer size. Diagnosis has a low accuracy for low CG buffer sizes as it is likely that the link connecting the faulty PE is purged from the CG during the CG-AG conversion. Higher CG size increases the accuracy because there are more links in the CG which increases the probability of SRB rules detecting errors leading to a high value of PPEP. Accuracy decreases with very high CG because several diagnoses do not complete as the size of the DT is large leading to an unacceptably high load on the Monitor.

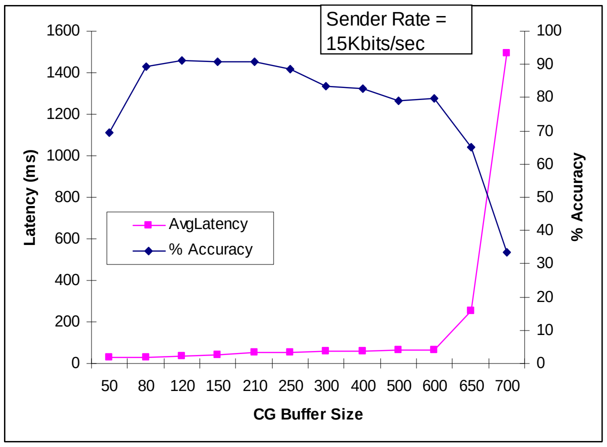
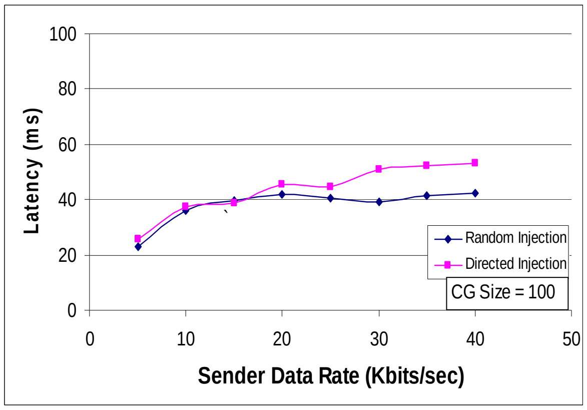
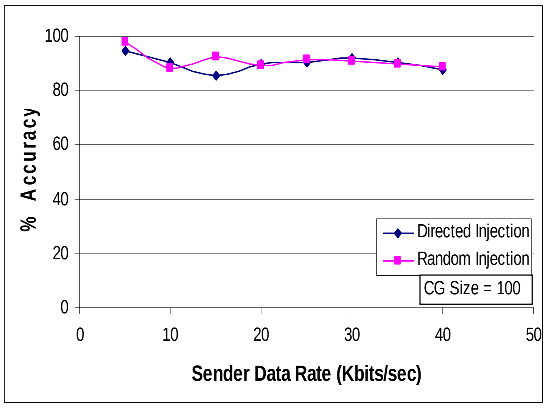


Figure 4.3: Latency and Accuracy for TRAM-D with random injection at sender

The increase in load with increasing CG buffer size is a direct consequence of the probabilistic diagnosis and was not present in the deterministic diagnosis. In probabilistic diagnosis, the entire CG is explored for faulty entities while with deterministic diagnosis, as soon as an entity is (deterministically) flagged as faulty, the process is halted. It is to be noted that during the diagnosis process, the TRAM entities are still sending packets to the Monitors leading to an additional detection overhead at the Monitor.

Effect of data rate: In this experiment, the buffer size at CG is fixed at 100 links, the data rate from the sender is varied and random injection is performed at the sender. Figure 4.4(a) shows the latency with increasing data rate. As the sender's data rate increases, the incoming packet rate at the Monitors increases by a multiplicative factor since each LM is verifying multiple PEs. Theoretically, till the Monitor's capacity is overrun, the sender data rate is expected to have no effect on latency since the CG buffer size is fixed and therefore the size of the DT being explored is fixed. It is not possible to see the breaking point (i.e. the "*knee*") in TRAM-D with the sender data rate since TRAM is robust and throttles the sending rate when the network starts to get congested.





(a) (b)

Figure 4.4: Latency and Accuracy with increasing data rate for random and directed injection

4.4.1.2. Directed Injection at Sender

We repeat the above experiments with directed injection at the sender. From Figure 4.4(a), we discern that the latency is higher compared to random injection for the same data rate. This is because directed injection causes more valid but faulty packets. This leads to a higher number of state transitions in the Monitor's STD causing more diagnosis procedures to run resulting in an increased load on the Monitor. With random injection on the contrary, a larger fraction of packets is discarded by the Monitor and therefore resulting in a lighter load. The diagnosis accuracy follows a similar trend as the random injection scenario.

4.4.2. Latency and Accuracy for TRAM-L

4.4.3. Fault Injection at Sender

We emulate the topology depicted in Figure 3.5(b) on a local network to investigate the performance of Monitor in high data rate scenarios and use more receivers under an RH. First we fix the CG size at 100 links and vary the incoming data rate from 200 Kbits/sec to 1.9 Mbits/sec. Figure 4.4 depicts the latency and the accuracy variations with increasing data rate. For low data rates the latency is about 300 ms and remains almost constant till a data rate of 1 Mbits/sec. Further increase in data rate causes the latency to rise exponentially because the Monitor's servicing rate is not able to keep up with the large CG size and incoming packet rate. We can see that this "knee" occurs at a lower

CG size compared to TRAM-D (Figure 4.3) because of higher data rate. Accuracy is near constant for data rates up to 1 Mbits/sec and breaks beyond that because of incomplete diagnoses attributed to higher load on the system.

4.4.4. Specific Injection

We perform specific injection in TRAM-D to observe the effect of a rogue receiver and to precipitate error propagation to varying degrees. Receiver R4 is modified to send ack at a slower rate (similar to [38]).Since in TRAM a cumulative ack is sent up the tree, R4's misbehavior prevents RH2 from sending the ack This forces the sender to reduce the data rate because the previous buffer cannot be purged, causing a slow data rate across the entire system. Thus error propagation occurs across the entire protocol system. The detection engine reports detection at several PEs as shown in Figure 4.5. The diagnosis algorithm is able to diagnose the correct node (R4) as faulty in all the cases since its PPEP value was highest in each DT.

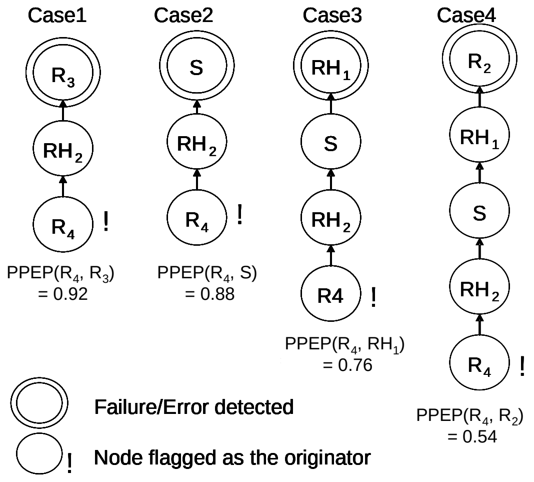


Figure 4.5: Parts of DTs formed during specific injection scenario in TRAM-D

4.5. Discussion

This chapter on diagnosis concludes the probabilistic diagnosis framework employed by the Monitor to perform diagnosis. Now, the Monitor provides detection and diagnosis in tandem re-using the efficiencies achieved in one domain to benefit the other. This is the first reliability framework which models the underlying protocol as black-box and provides both detection and diagnosis in unity. Currently the Monitor is still susceptible to commonly known *state space explosion*. Because of thousands of state in distributed systems, the Monitor's performance (accuracy & latency) can suffer in such scenarios. The next chapter explains how the Monitor stands against large number of states.

5. STD REDUCTION

At an abstract level, the fault tolerance infrastructure reasons about the distributed application and one way of doing this is through tracking the state transitions of the protocol entities in the application. Complex distributed applications in use today have a large state space due to the complexity of the protocol entities and the large scale of the application. This leads to the well known problem of state space explosion commonly occurring during the process of formal verification of hardware design or distributed protocols. This is true for any representation of the protocol whether State transition Diagrams, PetriNets or UML. This hampers the role of the fault tolerance system that reasons about the state of the application, especially if the system provides its services at runtime. A popular example is stateful firewall technology that reasons about application state by observing network packets. Other examples are intrusion detection systems (IDS) and information management systems, such as IBM's Tivoli suite of products.

5.1. Motivation for STD Reduction

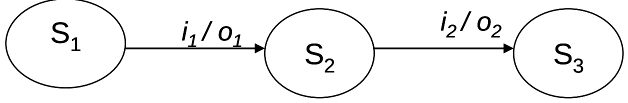
The fundamental premise that can be exploited to reduce the problem of state space explosion is that not all states are equally valuable from the point of view of fault tolerance services. The less valuable states can be ignored and the more valuable ones tracked reducing the total number of states that needs to be tracked. This classification into valuable and not valuable states may change frequently over the execution of the system, e.g., a new rule being installed in a firewall that requires a different state to be verified. The reduction in the number of states that needs to be verified reduces the latency of the service and increases the coverage of the Monitor's detection and accuracy process.

5.2. System Model

A state transition diagram (STD) is a 6-tuple $G = [S, s_0, T, I, O, \Gamma]$ where the element is described in Figure 5.1(a).

- S is the set of states
- s_0 is the set of the initial state(s).
- T is the state transition function $S \times I \rightarrow S$
- I is the set of input variables/alphabet
- O is the output set/alphabet
- Γ is the output mapping function $S \times I \rightarrow O$

(a)



$$\begin{tikzcd} S_1 \arrow[r, "i_1/o_1"] & S_2 \arrow[r, "i_2/o_2"] & S_3 \end{tikzcd}$$

(b)

Figure 5.1: (a) STD tuple; (b) An example STD

Figure 5.1(b) depicts an example STD for a protocol entity. The STD consists of 3 states $\{S_1, S_2, S_3\}$. Each label of the edge (also called as transition edge) represents an input (output) pair. For example in state S_1, an input message i_1 causes an output message o_1 to be generated. Simultaneously the protocol entity (PE) transitions from state S1 to S2. From the purview of a monitoring system, set I and O consist of the messages exchanged between the PEs. Further we divide the set I and O into two classes based on the visibility of these transitions to the monitoring system. Sets I_{int} (O_{int}) consists of inputs (outputs) which are internal to the PEs and sets I_{ext} (O_{ext}) consist of transitions which are visible to the monitoring system. Besides the state transition diagram, a set of rules R is specified which needs to be verified at runtime. Each rule in the set is formed from a sub-set of elements from each S, I, and O. The verification rules could have some timing constraints on the transitions within the graph. We denote S_R, I_R, O_R as the set of elements of G which have rules associated with them. The goal of the state reduction is to possibly remove the states in the set S with internal incoming transitions. And also remove the set I_{int} completely. We define *predecessor*(s_i) as the set of states which have an outgoing transition edge to s_i. And similarly *successor*(s_i) is the set of states which have an incoming transition edge from state s_i. Figure 5.2 depicts the abstraction considered in this thesis where there are some monitoring elements (M_1 and M_2) which are monitoring a protocol consisting of some communicating PEs. Each PE has some state transition diagram which is shared by the corresponding monitoring system(s) verifying that particular PE. For example in Figure 5.2, monitoring system M_1 is verifying PE_1 and PE_3 and hence contains the STDs for both of them.

5.3. Solution Approach

We propose a two step process to reduce the STD of each PE. Consequently, the state space of the entire protocol is also reduced. In the first step, the STD is reduced to STD′ based on the external transitions i.e. we remove all the internal transition edges (possibly including some states) of the STD. Formally the goal of the first step is to come up with a STD $G_{ext} = [S_{ext}, s_0, T_{ext}, I_{ext}, O_{ext}, \Gamma_{ext}]$ where the set of input symbols is reduced from I to I_{ext} and O to O_{ext}, correspondingly changing other elements of the tuple. During the process of reduction, some states (which have all input transitions as internal) can also be removed, such that $|S_{ext}| \leq |S|$. Removals of internal states help a monitoring system to track the STD of the PEs. The reduction also reduces the memory footprint of the monitoring system.

The second step reduction from STD′ to STD″ is done based on the rules defined for the different states in STD′. The approach is to reduce the states which do not have rules associated with them. Formally, we come up with a STD $G_e = [S_r, s_r, T_r, I_r, O_r, \Gamma_r]$. The property that should be satisfied is $|S_r| \leq |S_{ext}|$. Removal of states from the set $S_{ext} \setminus S_r$ can help improve the efficiency of the detection and diagnostic process by reducing the state search space for the monitoring system. The 2-step reduction can be summarized as: STD$\rightarrow$External Messages STD′$\rightarrow$rule base STD″.

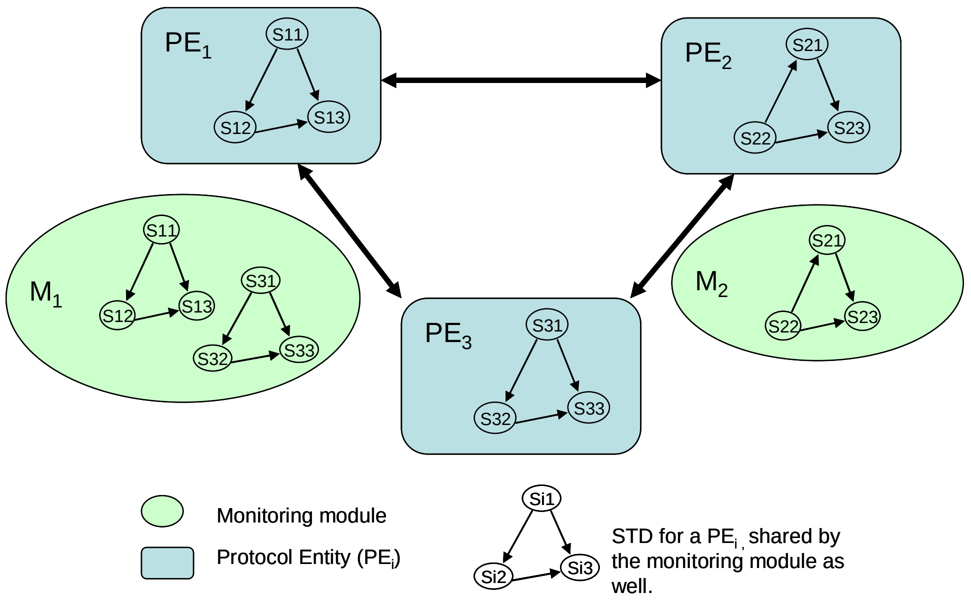


Figure 5.2: An example illustration of monitoring system(s) M_1 and M_2 monitoring a protocol consisting of 3 entities $\{PE_1, PE_2, PE_3\}$.

5.4. Removing internal transitions (STD→STD′)

The sets I_{int} (O_{int}) denote the internal transition edges in the STD. These transitions are not visible to the monitoring system because of the following reasons:

1. The transition is internal to the PE and does not depend on any externally visible message.
2. The monitoring system is placed in a network location where the observation of the PE is not perfect.
3. There are firewall rules that block the monitoring system from observing this kind of transition.

In this step, we try and remove all such transition edges that do not have any external inputs or outputs. During this process some states are also likely to be removed. A state which is removed has the property that all incoming transition edges to the state are internal. Since all the transitions leading to the state are internal, a monitoring system cannot determine if the PE has moved to that state. We mark a transition to be internal (external) if both symbols in the input (output) pair are internal (external). We represent k^{th} input (output) pair as i^k_{ext}/o^k_{ext}, if it is external and as i^k_{int}/o^k_{int}, if it is internal. Example: Consider an example STD depicted in Figure 5.3. For each state, all incoming transition edges (input (output) pairs) are checked to see if they are internal. The first rule of reduction is that

R1:*"If a state does not have any external transition edge, then remove that state and re-assign all the external outgoing transition edges of the reduced state*".

If state s_i is reduced, then transition edges from s_i to set of states in *successor*(s_i) are re-assigned to *predecessor*(s_i). If α is the set of external transition edges from s_i to *successor*(s_i) i.e. $s_i \rightarrow_\alpha$ *successor*(s_i), then re-assignment causes *predecessor*(s_i) $\rightarrow_\alpha$ *successor*(s_i). In Figure 5.3, state S_2 has a single incoming transition edge with label i^1_{int}/o^1_{int} which is internal. Following **R1**, state S_2 is removed and outgoing transition edge of state S_1 = *predecessor*(S_2) is re-assigned as i^2_{ext}/o^2_{ext} leading to S_3 = *successor*(S_2). The reduced STD (Figure 5.3(b)) has no state S_2 and only external transition edge(s).

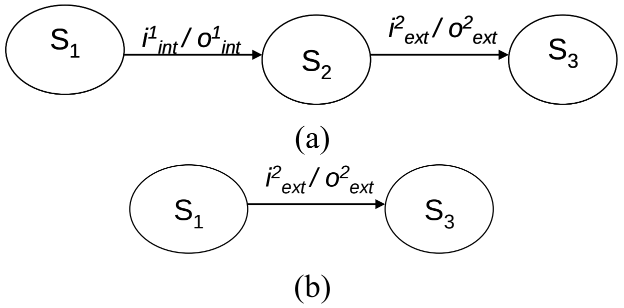
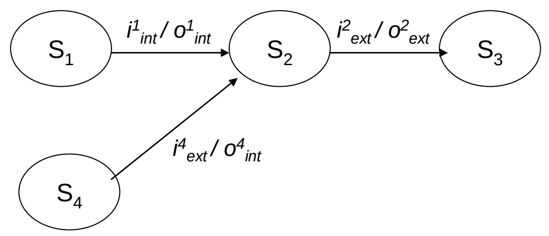


Figure 5.3: Example Reduction Process; (a) Original STD;(b) Reduced STD.

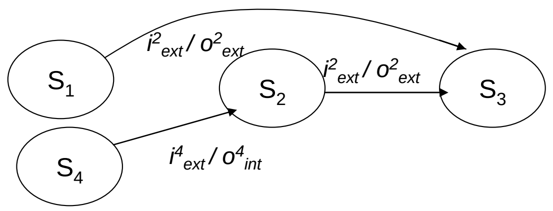
The second rule of reduction states that

R2:"*If a state exists which has both internal and external transition edges, we need to remove the internal messages and re-assign the external incoming and outgoing transition edges of that state".*

Re-assignment is performed in the same manner as for the rule 1 of reduction. In Figure 5.4 state S_2 has both internal input transition edge from S_1 and external input transition edge from S_4. S_2 cannot be reduced, but the STD is changed so that S_1 points to S_3 as i^2_{ext}/ o^2_{ext} is the only transition to signal a state transition from state S_1. Finally, the reduced STD will consist of only external transition edges.



(a)

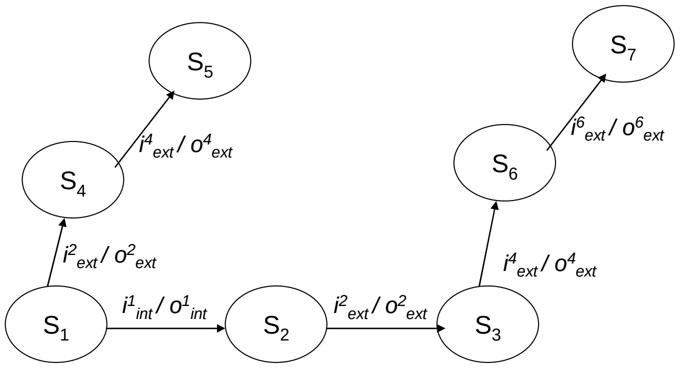


(b)

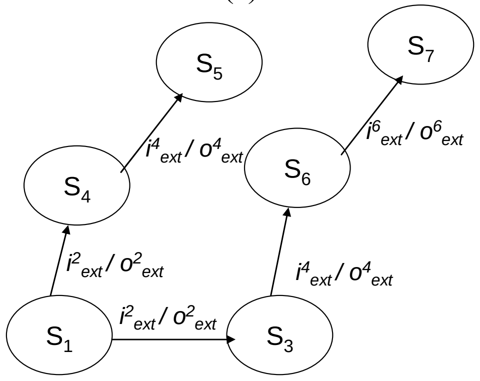
Figure 5.4: Example of STD→STD′ reduction where a state has mixed transitions (internal and external), (a) Original STD, (b) Reduced STD.

Let V be the number of states and E be the number of transitions. The operation of step 1 reduction considers all the incoming edges of all the states (total number of operations is O(E)) and for each incoming edge it performs a constant order operation. Therefore the entire running time is O(E) = O(V^2), where the upper bound is reached for a fully connected graph.

During the reduction process, while re-assigning the transitions it can happen that a reduced state might have two different transitions for the same input (output) pair leading to a conflict. Consider the example STD depicted in Figure 5.5(a) whose reduced STD is given by Figure 5.5(b). State S_1 is non-deterministic on receipt of input i^2_{ext} as two transitions are possible, either to state S_4 or state S_3. A monitoring system at this point cannot decide which of the two possible states the PE is in. Such a reduced STD can cause the monitoring system to lose track of the current state of PE leading to missed alarms. The monitoring system is forced to follow both the possible paths until some input transition is received which resolves the ambiguity. For the example in Figure 5.5, i^2_{ext} and i^4_{ext} will cause the monitoring system to keep two active paths but on receipt of i^6_{ext} the ambiguity of PEs current state is resolved, revealing that the PE traversed through states S_1-S_3-S_6-S_7.



(a)



(b)

Figure 5.5: (a) An example STD whose reduced STD, (b) will have non-deterministic transitions.

5.5. STD reduction due to ruleless states (STD′→STD″): Motivation

In this section we examine the motivation of reducing the STD based on the specified rules. The treatment is in the context of an existing detection and diagnosis framework called the Monitor because we use its rule syntax which is described in Section 5.6. Further onwards we will refer to the specific monitoring system i.e., the Monitor. The Monitor(s) takes as input the STD of the protocol to be verified. It performs state transitions based on the observed messages exchanged by the distributed protocol processes. For each state it verifies a set of rules specific to the state and some transition(s) in that state. Optionally a rule has precondition and time components.

The states that are not useful to the Monitor are those without any rule associated with them. A rule is defined as a condition check set up by the system administrator to check the correctness of the system by verifying whether the state transition is correct. The goal of STD reduction is to lower the latency of the diagnostic process for the Monitor. For details of the diagnostic process in the Monitor, the interested reader is referred to[9]. Briefly, when a failure is detected, a Suspicion Tree (ST) is created with the PE at which the failure was detected being the root of the tree (say R). A node in the ST represents jointly the PE and the state in which it sent the message. A node at depth 1 in the tree is the sender of a message to R (depth 0). Generally, depth *i* consists of all nodes which have sent messages to nodes in depth *i*-1. Note that a PE may appear multiple times in the ST because it was in different states at these times. This reflects the fact that a PE may appear multiple times in a chain of error propagation. Tests are executed on the state represented by the nodes in the ST to determine the faulty entity (or entities). Note that these are not active tests with which the PE is exercised. Instead these tests are executed on the state of the PE that has been built up by the Monitor by its observation. The reduction in the number of states reduces the execution time of the diagnosis process due to the following factors:

1. The Monitor system is hierarchical and any two nodes may be under the purview of two physically separated Monitors. Therefore testing an additional node incurs the cost of running the tests on that state plus possible network communication. Reducing the number of states reduces the number of nodes that need to be traversed during the diagnosis process.
2. The Monitor system tracks the possible causal chain of error propagation by tracking the message exchanges between the PEs. When an edge is removed from the STD, it means a message need not be tracked, which reduces the amount of storage and computation needed at the Monitor.

5.6. Rule Format used by Monitor

Monitor contains a rule base which is used for verification of the protocol behavior. For the completeness of this chapter we will explain the rule format used by the Monitor again. The rules in the Monitor can consist of some combination of *state*(S_I), *transition*(T_I) and *time*(t). The transition as defined before consists of a pair $I_I(O_I)$

denoting the input(output) message which characterize the transition. A rule could also simply verify single message of the transition by making the other message don't care i.e., with a transition as $I_I(d)$, the Monitor will only observer the input I_I. Monitor rules consist of the following 5 types:

Type I: $S_p = \text{true for } T \in (t_N, t_N + k) \Rightarrow S_q = \text{true for } T \in (t_I, t_I + b)$

The above rule represents the fact that if for some time k starting at t_N, a state S_p is true, then it will cause the state S_q to be true for some time b starting from t_I. The time t_N represents a time when some defined transition T_I takes place.

Type II: S_t is the state of an object at time t: $S_t \neq S_{t+\Delta}$, if T_I = true at t. The state S_t will not remain constant for more than Δ time units if a transition T_I occurs.

Type III: $L \leq |V_t| \leq U \; t \in (t_i, t_i+k)$

The number of transitions $\underline{V_t}$ in a particular state S_I will be bounded by L and U in some time k starting at time t_i.

Type IV: $\forall t \in (t_i, t_i +k) \; L \leq |V_t| \leq U \Rightarrow L' \leq |B_q| \leq U' \; \forall q \in (t_n, t_n+b)$. The number of transitions V_t being bounded by upper and lower bounds in time k will cause another transitions B_q to be within some bounds and will hold true for some time interval b. This rule is in fact the master rule and the three previous rule types are special cases. But we still need the first three rule types because matching this class of rule entails matching more variables, which incurs higher latency than the first three classes.

Type V: $s = S_i \; \forall t \in (t_0, t_0 + \alpha) \Rightarrow s \neq S_i \; \forall t \in (t_0 + \alpha, t_0 + \beta); \; s.t. \beta > \alpha$ This rule prevents a state transition from S_i back to the same state within time β of first arriving at S_i.

5.7. STD reduction due to ruleless states (STD′→STD″): How?

In this reduction step, we need to ensure that a reduction should not change the result of the verification of any rule by the Monitor. This is especially the case with Rule V which can be violated if the Monitor makes a transition to a state prematurely. Therefore, in order to prevent erroneous flagging of this rule, not all ruleless states can be removed. Also, if there is a rule corresponding to a transition, then the transition cannot be removed.

Broadly we classify the procedure into two steps which should be performed for the reduction to happen.

1. *Identifying the nodes for reduction:*

Case 1: Rule of Type V is not present in the *predecessor* state of a ruleless state:

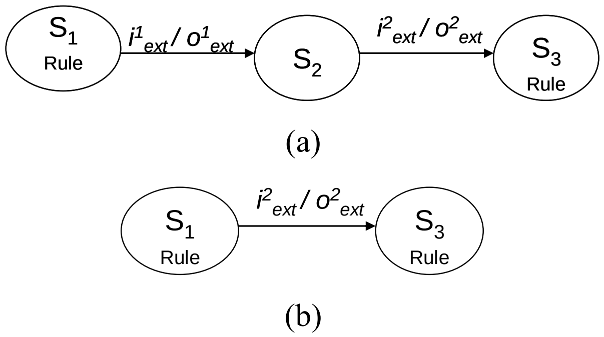


Figure 5.6: (a) Example input STD; (b) Reduced STD. Each state with a "Rule" has some rule associated with it which is verified by the monitoring system.

R3: A state S_i has no rule $\wedge$ ($\forall S_j \in$ *predecessor*(S_i)) S_j has no rule of type V $\wedge$ S_j has no rule on transition edge $i^1_{ext}/o^1_{ext} \Rightarrow$ Remove S_i and incoming transitions of S_i.

In the example shown in Figure 5.6, since S_2 has no rule and S_1 does not have rule V, S_2 can be reduced.

Case 2: Rule of Type V is present in a predecessor state of the ruleless state:

R3-1: A state S_i has no rule and ($\exists S_j \in$ predecessor(S_i)) *s.t.* S_j has a rule of type V $\Rightarrow$ Do not remove S_i.

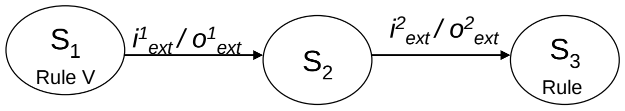


Figure 5.7: An example STD where *predecessor*(S_2)=S_1 has a rule of type V.

In Figure 5.7, S$_1$ contains rule V, so if we reduce S$_2$, thereby creating a direct transition from S$_1$ to S$_3$, then the timing properties of rule V associated with S$_1$ can get violated. This can happen because transition from state S1 is delayed because of removal of edge i^1_{ext}/o^1_{ext} thereby causing the time spent in state S$_1$ beyond Δ time bound and violating the rule. S$_2$ can only be reduced if there is another state with no rules adjacent to it. Consider the example in Figure 5.8**,** when S$_2$ and S$_3$ have no rules but S$_4$ has rule V. In this case, either S$_2$ or S$_3$ can be reduced and combined into a single state without rules.

<u>Case 3: Rule of Type II is present which runs on a self-loop in a predecessor state of the ruleless state:</u>

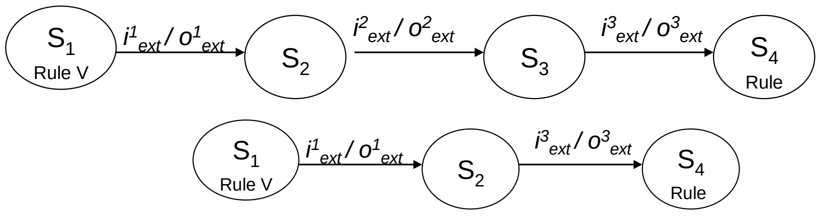


Figure 5.8: (a) Original STD (b) Possible Reduced STDs

R3-2: A state S_i has no rule and ($\exists S_j \in$ predecessor(S_i)) *s.t.* S_j has a rule of type II $\wedge$ T_j is self-loop $\Rightarrow$ Do not remove S_i.

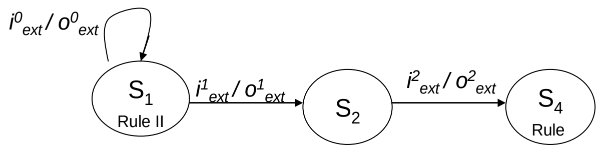


Figure 5.9: An example STD where a rule of Type II exists in the predecessor state of a ruleless state.

Consider a type II rule (see Figure 5.9) exists which is verifying the time spent in state S$_1$ once a transition i^0_{ext}/o^0_{ext} happens. If we remove the transition edge i^1_{ext}/o^1_{ext} because S$_2$ is ruleless, this will be incorrect since this can cause an increase in the time spent in state S_1 thus violating the type II rule.

2. *Reduction Step:*

If two or more states are identified for reduction, they are merged to form a single state. The transitions between the merged states are discarded. Consider a state that needs to be removed, namely S_2 in Figure 5.10. The state S_2 is removed and its outgoing transitions are re-assigned to the *predecessor*(S_2) states. In this example, the transition edge i^2_{ext}/o^2_{ext} is re-assigned to S_1.

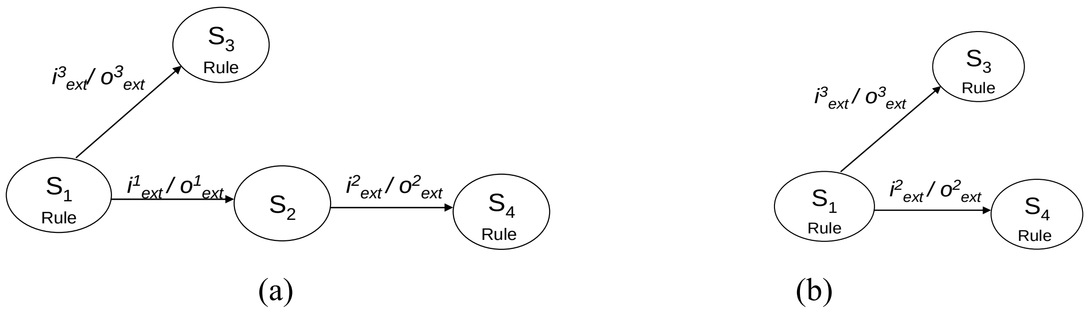
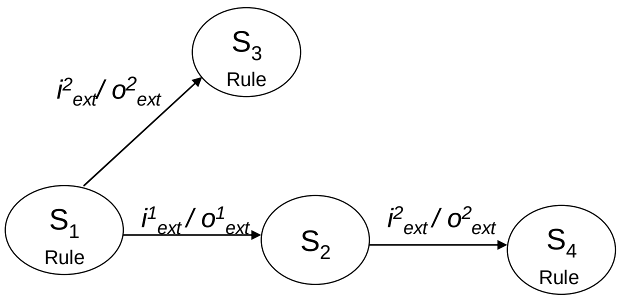


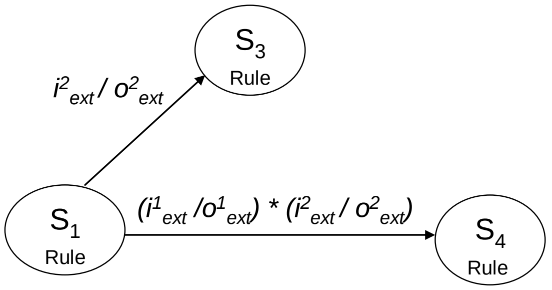
Figure 5.10: (a) Original STD;(b) Reduced STD

3. *Building Complex Edges:*

If the state being reduced has the same transition edge i.e. the same input (output) pair as one of its *predecessor* states, then the reduced state can now have multiple transitions for a single input making the STD non-deterministic. To ensure that the reduced STD is deterministic, we propose maintaining multiple pairs of input-output labels for a transition edge, called a *complex edge*. We define a complex edge to be of the form $t_1 *t_2 *t_3 ..... *t_k$ where t_j is pair i^j_{ext} (o^j_{ext}). If a complex edge represents a transition i.e., $s_j \rightarrow_{t1*t2} si$ then both t_1 and t_2 must be observed before the transition to state s_i is performed. For example in Figure 5.11, the edge between states S1 and S4 in the reduced STD has two pairs of transitions. When the conflicting transition occurs, we check if the prefix pairs had occurred to disambiguate, e.g., if the Monitor observed $i^2_{ext}(o^2_{ext})$it will check if $i^1_{ext}(o^1_{ext})$ had been seen prior, to disambiguate.



Original State



Reduced State

Figure 5.11: Example for introduction of complex label for a transition edge

With V as the number of states and E as the number of transitions, the running time for reassigning an edge is $O(1)$, for checking if the previous state has rule of type V is $O(V)$, and the total number of steps over which this has to be done is $O(E)$, thereby giving a total complexity of stage 2 reduction of $O(VE)$.

5.8. Correctness property

For the purpose of discussion, we consider the property from the point of view of an arbitrary state s_i. Let α_1 and α_2 be the set of transition edges from *predecessor*(s_i) and *successor*(s_i) to the state s_i. Assume $\exists s_j \in predecessor(s_i) and \; \exists s_k \in successor(s_i)\; s.t.$ $s_j \rightarrow_{tji} s_i \rightarrow_{tik} s_k$ where $t_{ji} \in \alpha_1$ and $t_{ik} \in \alpha_2$ in the original STD. Further we assume that PE goes through *predecessor*(s_i) $\rightarrow_{\alpha 1}$ s_i $\rightarrow_{\alpha 2}$ *successor*(s_i) trace unless otherwise stated. Since s_i is arbitrary, the proof holds in general for any state.

Theorem: *The reduction of the STD does not reduce the coverage of the Monitor.*
For this, we first prove three lemmas.

Lemma 1*: The reduction steps do not alter the time of entering a particular state but may delay the time of leaving a particular state as perceived by the Monitor*. ("Alter" implies changing from the actual transition of the PE to when the Monitor performs the transition.)

Consider the trace $s_j \rightarrow_{\text{tji}} s_i \rightarrow_{\text{tik}} s_k$. Here there could be two scenarios; 1) State s_i has all incoming transition edges to be internal; 2) Some incoming transitions which are external are also present. If t_{ji}.*type* = *internal* then reduction step 1 will remove this transition edge and reduce the STD trace to $s_j \rightarrow_{\text{tik}} s_k$.(using **R1** or **R2** depending upon the two scenarios). The Monitor will correspondingly perform transition from s_j to s_k only after receiving t_{ik}. But the PE may have left the state s_j some time before and moved to state s_i. This implies the reduction can postpone the time at which Monitor leaves a particular state (by amount *time*(s_i) in this example). However, since the reduced STD at the Monitor contains transition edge t_{ik}, the Monitor will make a transition to s_k at the correct time, thus causing no error in the time of entering a particular state. Thus, the Monitor's estimate of duration spent in a state can be longer than what the actual duration a PE spent.

Lemma 2*: Monitor correctly determines the PEs current state.*

The proof follows the premise that if all messages in the STD are visible to the Monitor then on observing each and every message, the Monitor will make correct transitions.

Lemma 3*: Outcome of verification of a PE against a rule base using a reduced STD remains unchanged from the complete STD.*

The verification by the Monitor should yield the same performance on the reduced STD as on the original STD, i.e., there should not be an increase in missed alarms or false alarms during verification of the rule set. Since we have eliminated all internal transitions by the first step of reduction, $e_1.type$ = *external* and $e_2.type$ = *external* $\forall e_1 \in \alpha_1$ and $\forall e_2 \in \alpha_2$. Some of the states on this path may or may not have rules. We will consider each scenario for every rule type and show how reduction step 2 affects the rule verification. In all cases state s_i has no rule associated with it and we consider if the reduction of s_i according to the rules defined in section 5.5, satisfies the lemma.

Type I: Assume both s_j and s_k have rule of type I associated with them present in the rule base for verification. Using **R3** s_i will be removed and re-assignment will yield a mapping between predecessor and successor sets given by: *predecessor*(s_i) $\rightarrow_{\alpha 2}$ *successor*(s_i). If PE traverses the path $s_j \rightarrow_{\mathrm{tji}} s_i \rightarrow_{\mathrm{tik}} s_k$, then correspondingly the Monitor will perform $s_j \rightarrow_{\mathrm{tjk}} s_k$. According to the format of type I rule, both the pre-condition and post-condition verify for a state being true for *atleast* some time period (*k* or *b* from definition of type I rule). It is important to note that both pre-condition and post-condition require the time of entrance to a particular state to be accurate and the time spent in that state by the Monitor should be at least the time the PE spends in that state. Using *Lemma 1* both conditions are satisfied. Hence, the reduction process does not affect the verification of type I rule. This argument is independent of whether a single type I rule encompasses both s_j and s_k and whether one or both of s_j_or s_k has a rule of type I.

***Type II*:** Assume either or both s_j and s_k have rule of type II associated with them. It is important to note that unlike type I rule, type II rule require accurate knowledge of time of leaving a state. Hence the line of reasoning from type I cannot be simply carried over. Below we analyze the specific cases which arise during handling of type II rule.

Case 1: Let's consider state s_j has a type II rule associated with the transition edge t_{ji}. In this scenario even if the state s_i does not have any rule, using **R3** the reduction step will not remove state s_i (and transition edge t_{ji}) because a valid rule over that transition edge exists in the rule base. Same reasoning holds if s_k has a type II rule associated with the transition edge t_{ik}.

Case 2: Let's consider state s_j has a type II rule associated with a transition edge $t_r \neq t_{ji} \wedge$ (t_r is not a self-loop). In this scenario, using **R3** the reduction step will remove the state s_i and perform the following re-assignment $s_j \rightarrow_{\alpha 2} successor(s_i)$. Although s_j has a type II rule present, since it is associated with some other transition edge ($\neq t_{ji}$), even if the PE traverses this particular trace (i.e. $s_j \rightarrow_{\mathrm{tji}} s_i \rightarrow_{\mathrm{tik}} s_k$) rule verification would not be triggered because the rule is not based on the transition edge t_{ji}. On the other hand if transition edge t_r transpires, Monitor will correctly estimate the time of departure from state (s_j) because t_r would not be removed during the reduction step (from Case I).

Case 3: If state s_j has a type II rule associated with transition edge_$t_r \neq t_{ji} \wedge$ (t_r is a self-loop for s_j). Using **R3-2** the state s_i will not be removed. This is because if s_i (and correspondingly t_{ji}) is removed, this will delay the amount of time spent in state s_j using *Lemma 1.* This can cause invalid response during verification of the type II rule verification. Since **R3-2** prevents a state removal in such scenario(s), reduction process does not interfere with the verification process.

Hence in all cases, the reduction step does not violate a type II rule if present.

***Type III*:** Assume state s_j has rules of type III. A type III rule measures a state variable and since a rule is specific to a state, this state variable is a function of the number of transitions following a self-loop to the same state (since a transition out of the state will no longer cause the rule to hold and a state variable changes only when there is a transition). The rule is dependent on accurately determining the time of entering a state and the duration spent in that state. Using *Lemma 1* we know that first condition is true. According to the rules, an edge re-assignment only happens if it is not present in any rule. If state s_j has a rule of type III associated with the self loop transition edge t_r, then t_r will not be removed. During the reduction step, state s_i is removed and transition edges re-assigned as $predecessor(s_i) \rightarrow_{\alpha 2} successor(s_i)$ using **R1**. If $t_r \in \alpha_2$, then a complex edge must be created which will help Monitor determine the correct time of leaving the state s_j. This satisfies the second condition of accurately determining the time spent in s_j. If state s_k has rules of type III instead of s_j, the exact same reasoning applies.

***Type IV*:** A rule of this type comprises a precondition and a postcondition, each of which looks like the condition in a rule of type III. As proved before, the condition for type III rule is not violated by the reduction. Therefore neither the precondition nor the postcondition is violated and therefore the rule of type IV is satisfied.

Type V: Assume type V rule is associated with s_j, and some rule is associated with s_k. One can see that a rule of type V prevents a state transition from s_j back to itself within a time bound. If s_i does not have any rule associated with it then it is a candidate for reduction (using **R3-1**). Since we have already proved that re-assignment can never cause inconsistencies in determining the time of arrival into a state but can delay the time of departure from a state. Hence if the reduction step reduces state s_i and the transition out of s_j is delayed, this can cause a violation of the type V rule associated with s_j but **R3-1** avoids this scenario.

Hence we prove that the protocol semantics are unchanged with respect to the Monitor during STD reduction. Putting *lemmas 1, 2, and 3* together, we assert that the STD reduction steps 1 and 2 combined do not change the semantics of the application protocol *w.r.t.* the Monitor which completes the proof for Theorem 1.

5.9. Discussion

By the end of this chapter we have developed a Monitor framework which provides detection diagnosis and tackles large number of states. At this point it is necessary to compare the strengths of the Monitor with other similar frameworks. This forms the basis of the next chapter of this thesis.

6. PINPOINT AND APPLICATION TO *e*-COMMERCE

6.1. Motivation

E-commerce today is an important backbone used by multiple everyday applications like customer survey's, online shopping, e-banking etc. Providing reliability in such an environment is very important and almost inevitable because of the high financial cost associated with the downtime. We apply the diagnosis protocol (from chapter 3)to a three tier e-commerce system consisting of the Pet Store application deployed on the JBoss application server with the Tomcat web server as the front end and the MySQL database server at the backend. The application supports multiple kinds of browse and buys transactions with each involving interactions between many components, where components are defined as servlets and EJBs. Through a modification to the JBoss containers, messages between the components are trapped and forwarded to the Monitor. We compare our approach to Pinpoint[106] in terms of accuracy and precision of diagnosis. We inject errors in the application where the errors may be due to a single component or interactions between multiple components. The accuracy and precision, correspondingly roughly to the complement of false negative and false positive, are measured for different kinds of faults. Our approach outperforms Pinpoint in 1, 2 and 3 component faults. The accuracy of the diagnosis gains between 20% to 100% over Pinpoint's approach for comparable precision values.

6.2. Pinpoint's Approach to Diagnosis

Pinpoint serves as a valid point of comparison with the Monitor since both systems have the same focused goal (diagnosis, as opposed to say performance debugging as in

[65] with diagnosis being a side issue) and have the same target application model (black-box or gray-box application and passive observation of the application for diagnosis). Importantly, Pinpoint represents a recent state-of-the-art development ([106]) and has been well explained and demonstrated on an open source application (compare to say Magpie [67] where the application is not available to us), and its algorithms are not dependent on a huge set of parameters whose settings are left mysterious in the publication (compare to the machine learning approach in [98] where several statistical distributions would have to be assumed).

We implement the Pinpoint algorithm (as explained in [106]) for comparison with our Monitor's diagnosis approach. Pinpoint requires as input a dependency table —a mapping of which components each transaction depends on. This is in contrast to the Monitor approach, where such dependency information does not have to be determined a priori and fed into the system before execution. Instead the Monitor deduces the dependencies through runtime observations as described in Section II.B. For Pinpoint, when transactions are executed, their failure status is determined by the failure detectors. A table (called the input matrix) is then created with the rows being the transactions, the first column being the failure status, and the other columns being the different components. If a cell T(i, 1) is 1, it indicates transaction i has failed. If a cell T(i, j) is 1, it indicates transaction i uses the component j. Pinpoint correlates the failures of transactions to the components that are most likely to be the cause of the failure. The input matrix is fed as input to the data clustering engine. The transpose of this binary input matrix is used by the data analysis engine. The data analyzer computes the dissimilarity between the rows of transposed matrix, which is represented by Jaccard`s distance. This matrix containing the distances between the components is fed to a clustering algorithm called the *Unweighted Pair Group Method with Arithmetic Mean* (UPGMA) [106]. The algorithm forms clusters. What is significant is which components fall in the cluster having the failure row. These components are diagnosed by Pinpoint to be faulty.

A crucial point for the accurate operation of Pinpoint is that the transactions should be diverse enough, i.e., use distinct non-overlapping components. Two transactions T_1 and T_2 are called distinct with respect to a set of components $\{C_1, C_2, \ldots, C_k\}$ if there is no overlap between these columns for T_1 and T_2, i.e., when T_1's row has a 1 in any of these columns, T_2's row has a zero, and vice-versa. Pinpoint as described by the authors in [106] is an offline approach. For comparison with the Monitor, we convert it into an online protocol. We incrementally feed the transactions and their corresponding failure status as they occur in the application, rather than waiting for all the transactions in a round to be completed before executing Pinpoint. The performance improves as the number of transactions increases (and consequently, the number of distinct transactions increases) and this is quantified through the experiment described in Section 6.3.

To provide a comparable platform between the Monitor and Pinpoint, we keep the testbed identical to that in [106]—same client, web server, application server (with identical components), and database server. Since the performance of the Monitor and Pinpoint are sensitive to the transactions used, we would have liked to use the same set of transactions as used by Pinpoint in [106]. However, the paper is silent on the issue—it does not even provide the total number of transactions used. We contacted the authors of Pinpoint but they were unable to provide us with the transactions either.

6.3. Implementation and Experimental Test-Bed

6.3.1. Experiment Test-bed

6.3.1.1. Application

We use PetStore (version 1.4), a sample J2EE application developed under the Java BluePrints program at Sun Microsystems[107]. It runs on top of the JBoss application server [108] with MySQL database [109] as the back-end for the example 3-tier environment. Figure 6.1 depicts the application topology for the experiments. The

PetStore application is driven by a web client emulator which generates client transactions based on sample traces. The web client emulator is written in Perl using lynx as the web browser. For the mix of client transactions, we mimic the TPC-WIPSo [110] distribution with equal percentage of browse and buy interactions. The servlets and the EJBs are considered as components in our experiments and these serve as the granularity level at which diagnosis is done. This design choice is based partly on the fact that in JBoss a faulty servlet or an EJB can be switched out at runtime for a correct one. We identified a total of 56 components in the application.

We consider a web interaction to be a complete cycle of communication between the client emulator and the application, as it is defined by the TPC Benchmark W specification [110]. This cycle starts when the client emulator initiates web request and it is completed when the last byte of data from the response page has been received by the client emulator. Examples of web interactions could be entering the Welcome page or executing a Search. A transaction is a sequence of web interactions. An example of a transaction by a user who is searching and viewing information about a particular product is: *Welcome page* → *Search* → *View Item details*. For our experiments we created a total of 55 different transactions. A round is a permutation of these 55 transactions modeling different user activities that occur on the web store. Within a round, transactions are executed one at a time. Two transactions are considered to be non-unique if they use exactly the same components, neglecting the order in which the components are used. Thus, a transaction that comprises: *Welcome*, *Search*, *Search* is not unique with respect to another that comprises: *Welcome*, *Search*. There are 41 unique transactions in the set of 55 transactions that we use. Although 55 is not an exhaustive set of possible transactions in the application, the chosen set exercised a wide variety of web-interactions and between them, touched all the components of PetStore. We note that the results presented here depend on the exact set of transactions used to exercise the system.

We instrumented the JBoss application server to snoop over the message communication between PetStore components. JBoss has a layered architecture and each communication traverses multiple interceptors. We modify the SecurityInterceptor to forward messages to the Monitor for updating the causal graph. Thus, the PetStore application is left unchanged.

6.3.1.2. Monitor configuration

The diagnosis algorithm in the Monitor is implemented in Java. The Monitor is provided an input of state transition diagrams (STDs) for the components verified and causal tests used during calculation of PPEP values. The size of the causal graph is bounded at 100 links.

Figure 33 shows an example STD for CreditCardEJB used by the Monitor in our experiments. A start state S_0 signifies a no request state. If a request for processing is received from another component, the state of the EJB moves from S_0 accordingly. Right below the STD we have some simple causal tests which can be derived from the STD itself. As explained in section 6.2, causal tests are dependent on the state and event of the component. For example, if the EJB is requested for getData() then in state S_1 there must be a return from getData() to ensure correct operation of the EJB. This is verified using the first rule in Figure 33.

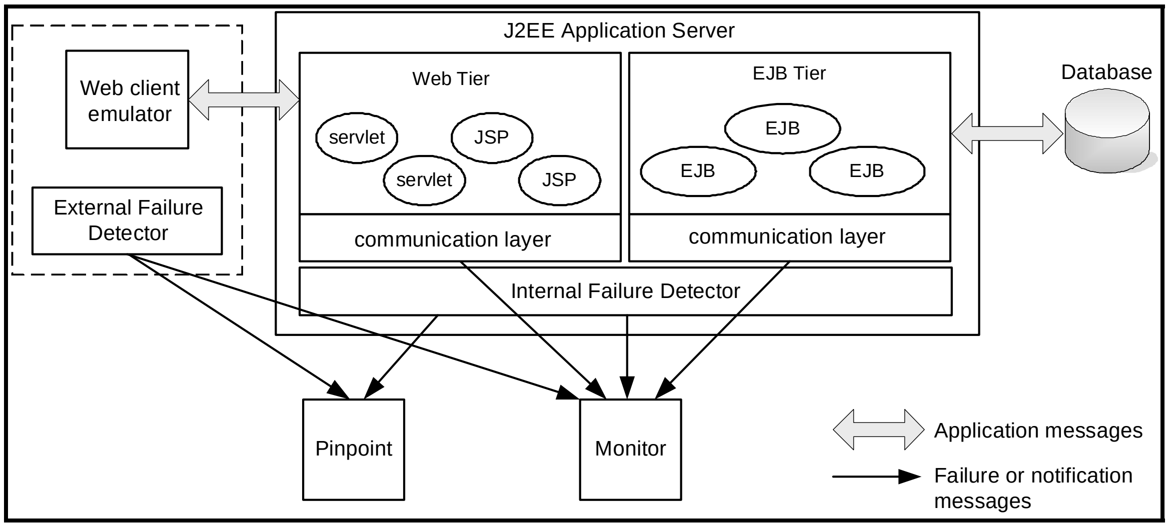
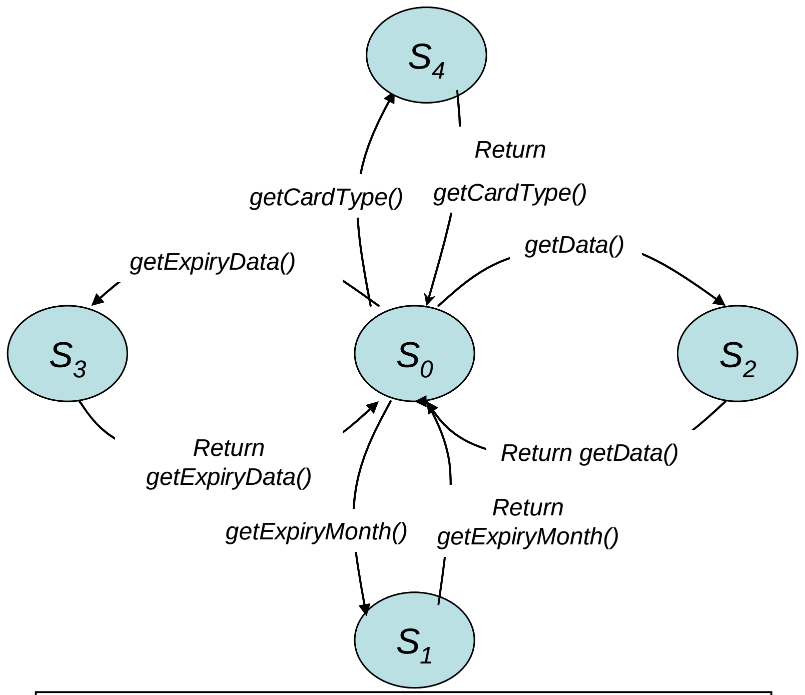


Figure 6.1: Logical Topology of the Client and Server for the Experiments



S0 getData 1 S2 return getData1 1
S0 getExpiryMonth 1 S1 return getExpiryMonth1 1
S0 getExpiryData 1 S3 return getExpiryData 1 1
S0 cardType 1 S4 return cardType1 1

Figure 6.2: An example STD for *CreditCardEJB* along with some illustration of Causal Tests.

6.3.1.3. Detectors

We create the same detectors as in [106]. An internal and an external failure detector are built which provide failure status of transactions to Pinpoint and the Monitor. The external detector detects failures that will be visible to the user, such as application-specific failures, machine crashes or complete service failures [106]. We implemented this external detector as part of the client emulator. It examines the output error log of lynx and flags a failure if an HTTP error is observed. Alternately, if a transaction does not complete within 20 seconds, timeout occurs and the detector flags a failure. An internal detector is used to detect a failure that may not immediately manifest itself to users. The internal detector is built to catch Java exception in the application and is embedded in each component.

6.3.2. Fault Injection

We perform fault injection into the components of the PetStore application (i.e., Servlets and EJBs). PetStore has about 56 components including the EJBs, and servlets. We choose a set of 9 components called target components consisting of 6 EJBs and 3 servlets for fault injection. The names of the components are AddressEJB, AsyncSenderEJB, CatalogEJB, CreditCardEJB, ContactInfoEJB, SupplierClientLocalFacadeEJB, enter_order_information.screen, order.do, and item.screen. We use four different kinds of fault injections similar to Pinpoint.

- Declared Exception: We inject IOException as the representative declared exception.
- Undeclared Exception: This is a Runtime Exception not caught in the application.
- Endless call: The target component has an infinite while loop.
- Null call: Instead of returning the appropriate value, a method returns a null object.

The internal detector is more likely to detect the declared and the undeclared exceptions, and the null calls while the external detector is more likely to detect the endless call. For a given round only one target component is injected. We use 1-component, 2-component and 3-component triggers. In a 1-component trigger, every time the target component is touched by a transaction, the fault in injected in that component. In a 2-component trigger, a sequence of 2-components is determined and whenever the sequence is touched during a transaction, the last component in the transaction is injected. This mimics an interaction fault between two components and in the correct operation of a diagnosis protocol, both components should be flagged as faulty. The 3-component fault is defined similarly.

6.4. Results

6.4.1. Performance Metrics

We use precision and accuracy as output metrics as in the Pinpoint work to enable a comparison. A result is *accurate* when all components causing a fault are correctly identified. For example, if two components, A and B, are interacting to cause a failure, identifying both would be accurate. Identifying only one or neither would not be accurate. However, if the predicted fault set (by the diagnosis algorithm) is {A, B, C, D, E} and the fault was in components {A, B} then the accuracy is still 100%. *Precision* captures the non-idealness in this case. Precision is the ratio of the number of faulty components to the total number of entities in the predicted fault set. In the above example, the precision is 40%. Components {C, D, E} are false positives. Lower precision implies high false positives. There is a tension between accuracy and precision in most diagnosis algorithms. When the algorithm is sensitive, it generates highly accurate results, but also causes a large number of false alerts reducing precision. Pinpoint uses the UPGMA clustering algorithm and varying the size of the faulty cluster varies the precision and accuracy. In the Monitor, after the diagnosis algorithm terminates, an ordered list of components is produced in decreasing order of PPEP. We define the predicted fault set as the top k components in the ordered output list. We vary k to obtain different accuracy and precision values.

6.4.2. Single Component Faults

In single component faults, the fault injection trigger consists of a single component. If a transaction touches the target component then one of the four kinds of faults chosen randomly, is injected and the injection remains permanent for the remainder of the round. First, let us consider the effect of varying cluster size on the performance of Pinpoint. The total number of injections for these results is 36—9 target components for injection and all 4 types of injection done on each component. The averaged results for accuracy and precision are plotted in Figure 6.3 (the bars show 90% confidence interval). As the size of the cluster increases, we see an increase in the accuracy which is intuitive because at some point the failure cluster includes all the components that are actually faulty. Beyond that, increase in cluster size does not impact the accuracy. As the cluster size increases, the precision increases to a maximum value and then decreases thereafter. The increase occurs till all the faulty components are included in the failure cluster. Thereafter, increasing the cluster size includes other non-faulty components and thus brings down the precision. The maximum value of precision occurs when all the faulty components are included in the failure cluster. However the precision is still poor (less than 10%). This is explained by the observation that for the transactions in the application, there is tight coupling between multiple components. Whenever the entire set of tightly coupled components does not appear together as a fault trigger, which is the overwhelming majority of the injections, the precision suffers. The amount of tight coupling between the components is quantified through the experiment in Section 6.4.6. We emphasize that if we were to hand pick transactions such that they are distinguishable with respect to the target components, then the performance of Pinpoint would improve. Two transactions T_i and T_j are indistinguishable with respect to a set of components $\{C_1, C_2, \ldots, C_k\}$ if the columns of T_i in the input matrix corresponding these components are identical to that of T_j.

Figure 6.3(a) shows the variation of Accuracy with False Positives for Pinpoint and the Monitor. For the Monitor, a given value of *k* (the top *k* elements from the ordered PPEP list are diagnosed) gives one value of accuracy and precision. This is averaged across the 36 injections for the presented results. For 1-component faults, Pinpoint has high false positives rates but the accuracy eventually reaches 1. In contrast the Monitor has a much higher accuracy keeping a low false positive rate. Monitor's accuracy also reaches 1 but at a much lower value of false positives (0.6) as compared to Pinpoint (> 0.9). The latency of detection in our system is very low. Thus, the faulty component is often at the root of the Diagnosis Tree in the Monitor. Since error propagation is thus minimized, the PPEP value for the faulty entity is high causing it to be diagnosed by the Monitor. This explains the high accuracy for the Monitor. However, Pinpoint's algorithm does not take advantage of the temporal information—the temporal proximity between the component where detection occurs and the component that is faulty. As a consequence its accuracy suffers relative to that of the Monitor.

Notice that in Pinpoint, for a given value of false positives, two different accuracy values are achieved since a given precision value is achieved for two different cluster sizes (Figure 6.3(b)). Since accuracy is a monotonically increasing plot with cluster size (Figure 6.3(a)), the different cluster sizes give two different accuracy values.

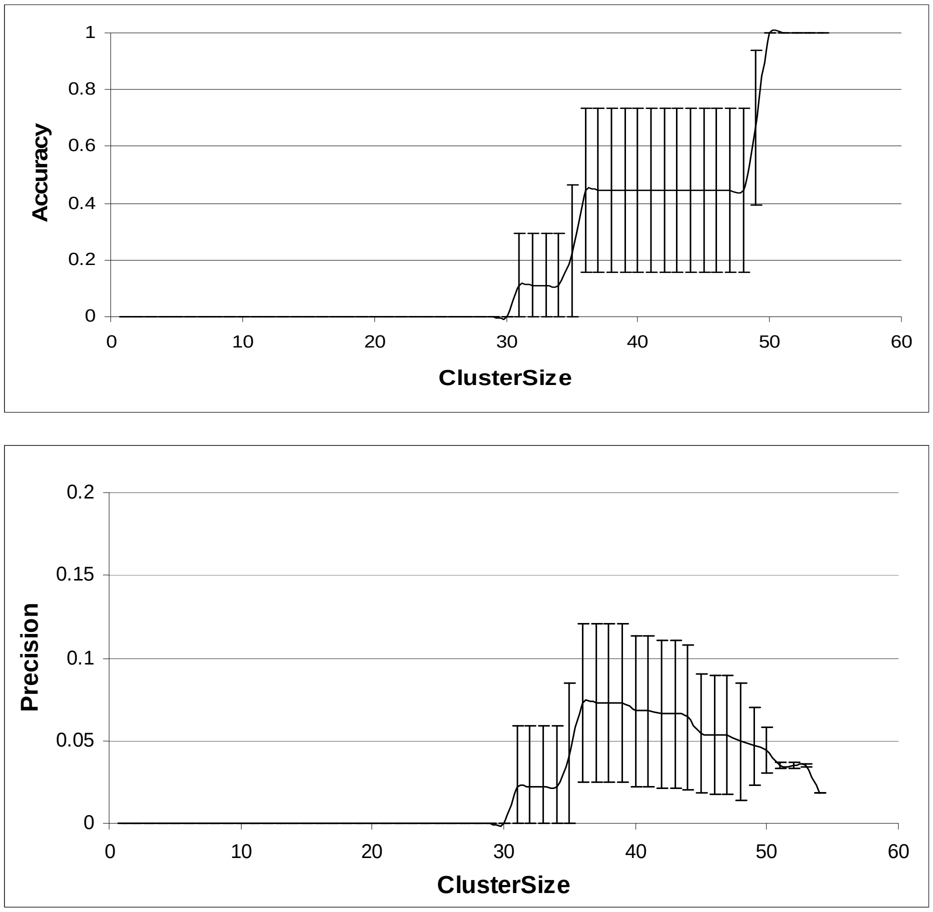


Figure 6.3: 1-component fault injection: Variation of Accuracy and precision with cluster size in Pinpoint.

6.4.3. Two Component Faults

The 2-component fault injection results are shown in Figure 6.5. Pinpoint results improve in terms of the false positives implying higher precision. This is attributed to the fact that Pinpoint's clustering method works better if the failing transactions are better distinguishable from the successful transactions. Recollect distinguishable is discussed in the context of components. A 2-component fault includes two components as the trigger and going from one component to two components increases the distinguish-ability of transactions.

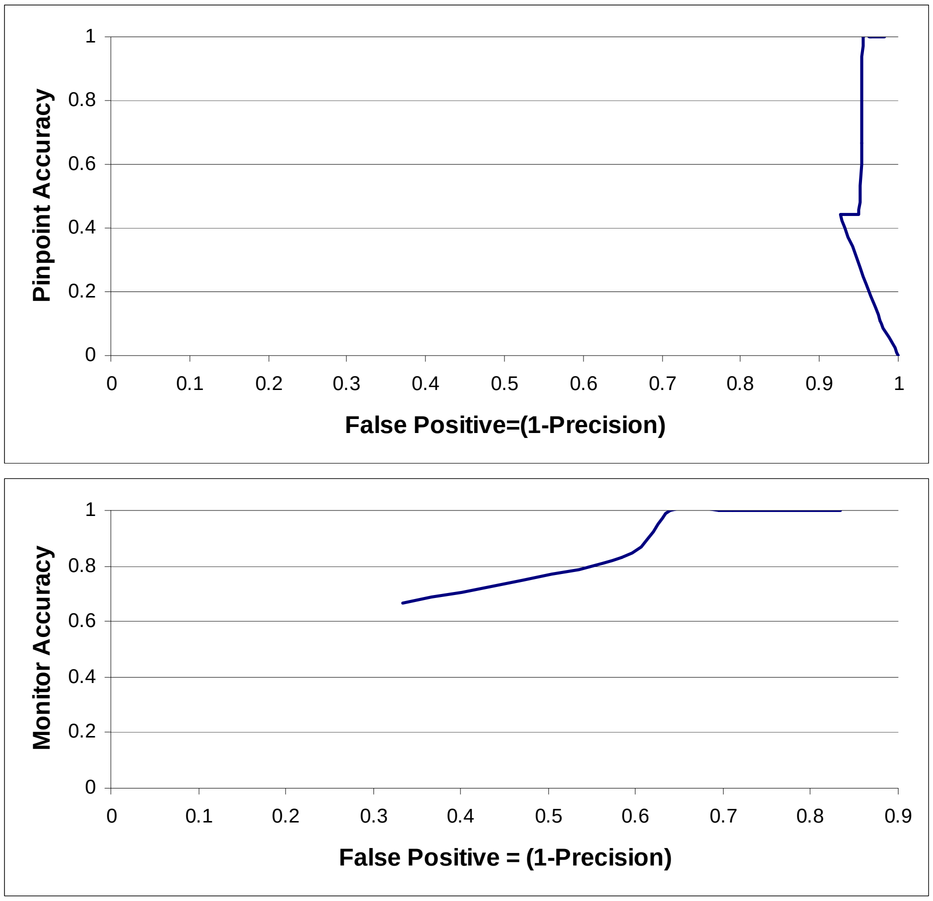


Figure 6.4: Single component fault injection: Performance of Pinpoint and Monitor. Both can achieve high accuracy but Pinpoint suffers from high false positive rates.

Consider transaction T_1 and T_2 both of which use component C_1 (the trigger in a single component fault injection). However, for a two component fault injection with trigger as $\{C_1, C_2\}$, the transactions T_1 and T_2 will be distinguishable as long as both T_1 and T_2 do not use C_2. Thus, say T_1 uses $\{C_1, C_2\}$ and T_2 does not use C_2. Then only T_1 will fail and T_2 will not, leading to the diagnosis (considering simplistically that these are the only transactions and components) of C_1-C_2 as the faulty entities.

In contrast, the Monitor results although still significantly better than Pinpoint suffer in the 2-component fault injection. One can see that accuracy reaches a maximum of only 0.83 compared to 1.00 in 1-component injection. The number of times in a round the trigger for the 2-component fault is hit is lower than for the single component fault. Each detection causes an execution of the diagnosis process and each execution of the diagnosis process updates the parameters of the causal graph away from an arbitrary initial setting toward an accurate set of values. Thus, for the 2-component faults, the Monitor gets less opportunity for refining the parameter values and consequently the PPEP calculation is not as accurate as for the single component faults. This explains the decline in performance of the Monitor for the 2-component faults.

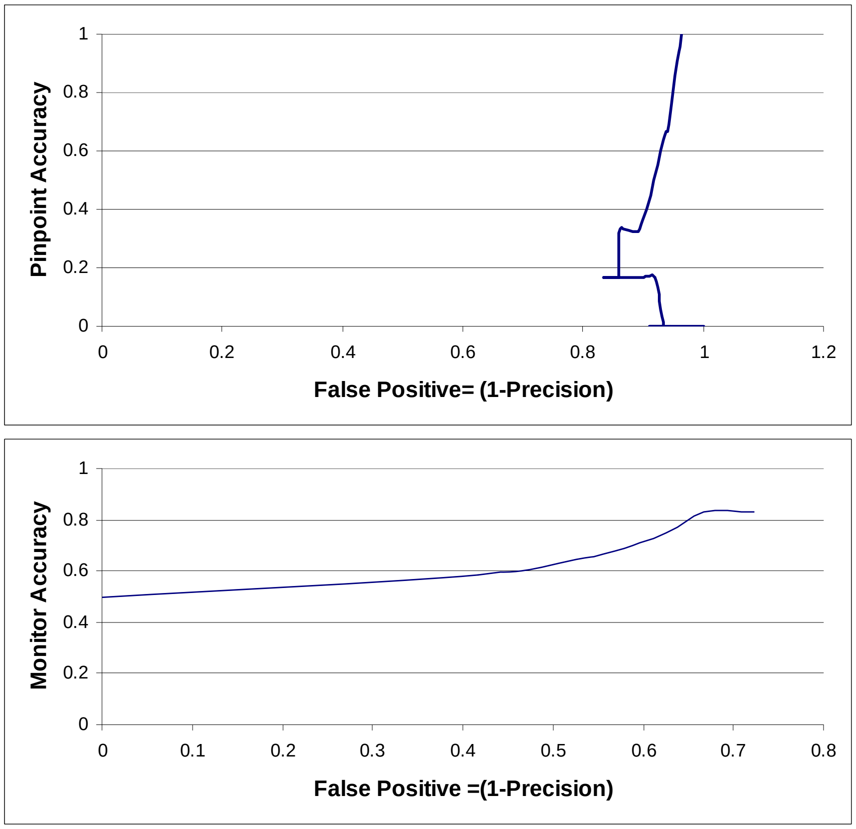
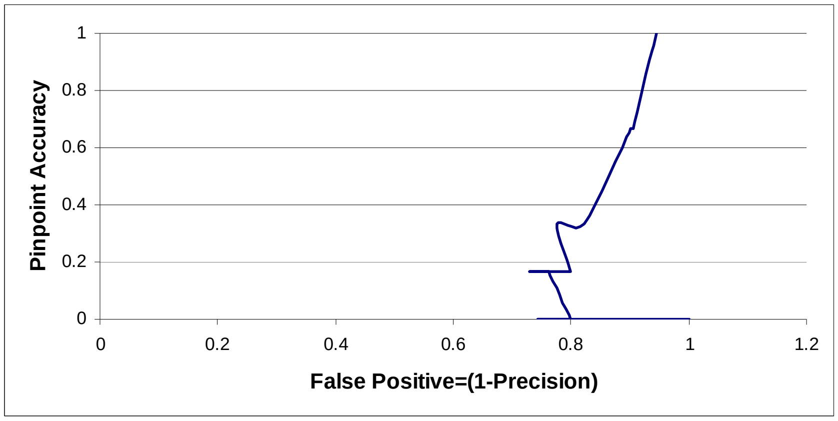


Figure 6.5: 2-component fault injection: Performance of Pinpoint and Monitor. Performance of Monitor declines and Pinpoint improves from the single component fault, but Monitor still outperforms Pinpoint.

6.4.4. Three Component Faults

Three 3-component fault injections show even better results for Pinpoint with the maximum average precision value touching 27%. This is again attributed to the fact that more number of components causes selected transactions to fail leading to a better performance by the clustering algorithm. The Monitor again outperforms Pinpoint by achieving higher accuracy at much lower false positives. The Monitor's performance again declines compared to the 2-component faults due to the same reason pointed in the previous section (the number of diagnoses for the 3-component trigger is less than that for the 2-component trigger).



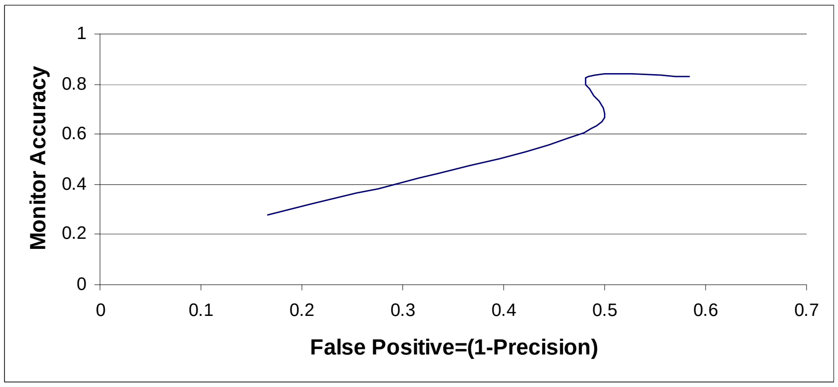
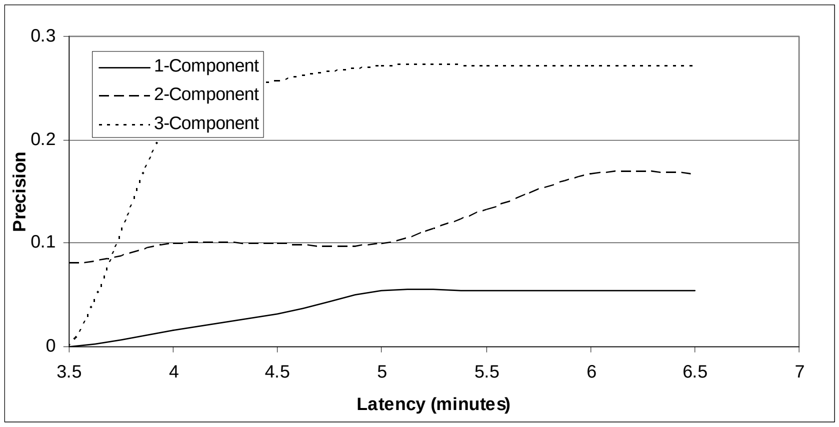


Figure 6.6: 3-component fault injection: Performance of Pinpoint and Monitor. Performance of Monitor declines and Pinpoint improves from the single and two component fault, but Monitor still outperforms Pinpoint.

6.4.5. Latency

In its online incarnation, Pinpoint takes as input the transactions and corresponding failure status every 30 seconds during a round. It runs the diagnosis for each of these snapshots taken at 30 second intervals, terminating when the round is complete and Pinpoint executes on the entire input matrix corresponding to all the 55 transactions. Pinpoint's performance only becomes reasonable at 3.5 minutes and above and hence we report only this part of the plot. Arguably this is a subjective decision, but we find the meaningful insights are only possible when Pinpoint has data worth 3.5 minutes or more. The latency plots show that after 3.5 minutes the accuracy and precision increase monotonically with latency.



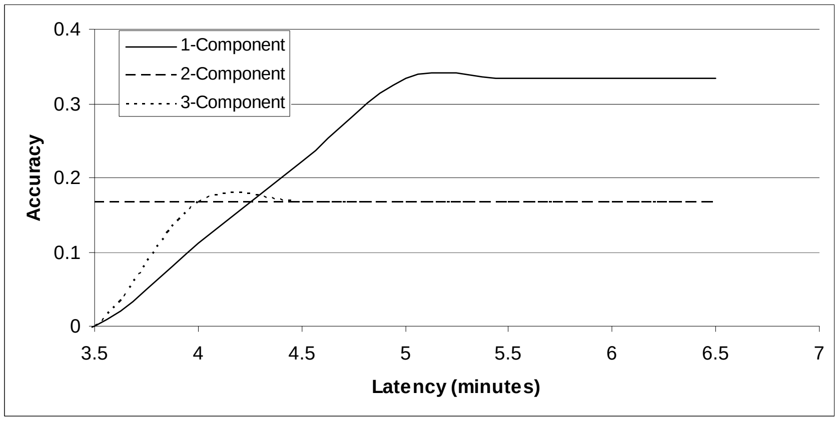


Figure 6.7: Single component fault injection: Variation of accuracy and precision with latency for Pinpoint. Higher latency means higher number of transaction data points and Pinpoint's performance improves monotonically.

We define the latency of diagnosis for the Monitor as the time delay from the receipt of the detector alert which marks the beginning of the diagnosis till the PPEP ordered list is generated. The Monitor has an average latency of 58.32 ms with a variance of 14.35ms, aggregated across all three fault injection campaigns.

6.4.6. Behavior of Components

The PetStore application has some components which are tightly coupled, i.e., they tend to be invoked together for the different transactions supported by the application. We have noted earlier that tight coupling negatively impacts Pinpoint's clustering algorithm. For our experiments, we inject 9 components and here we consider how tightly coupled these components are with the other components in PetStore. AddressEJB is tightly coupled with 4 components implying that AddressEJB always occurs with these 4 components in all the 55 transactions in our experimental setup. Pinpoint cannot distinguish between sets of components that are tightly coupled and thus reports the super set of the actual faulty components. This is the fundamental reason why its precision is found to be low in all our experiments. To counter this problem, one can synthetically create transactions that independently use different components (as noted by the authors themselves in [65]). However, for an application like PetStore, components are naturally tightly coupled and thus generating such synthetic transactions is a difficult task. Also even if we could devise such "unnatural" transactions that would make components distinguishable, it cannot be assumed that such transactions will be created with regular users in the system. Therefore, the premise of being able to diagnose failures by observing the transaction traffic generated by the normal users would be violated.

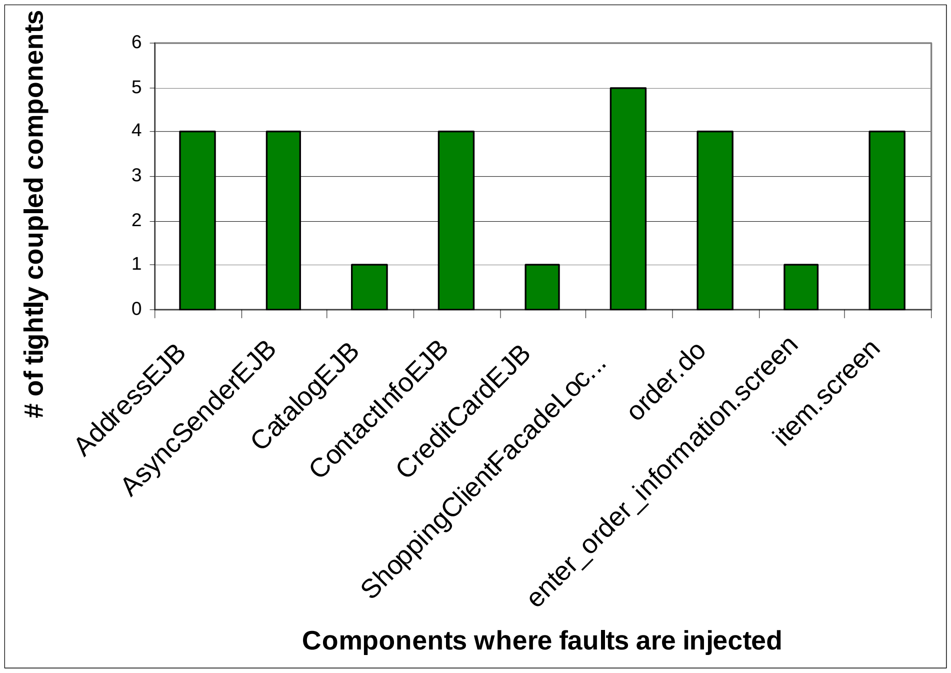


Figure 6.8: Tightly connected components

6.5. Discussion

In this chapter we compared the Monitor to the state-of –the-art diagnosis framework called Pinpoint. We tested the two systems on a 3-tier Java-based e-commerce system called PetStore. Extensive fault injection experiments were performed to evaluate the accuracy and precision of the two schemes. The Monitor outperformed Pinpoint particularly in precision, though its advantage narrowed for interaction faults. Because the Monitor might be operating in high throughput streams, it is necessary to maintain reasonable accuracy. This forms the basis of the next chapter of this Thesis where we develop a novel approach for detection in high throughput scenarios by modifying the existing detection algorithm of the Monitor.

7. STATEFUL DETECTION

7.1. Introduction

The proliferation of high bandwidth applications and the increase in the number of consumers of distributed applications have caused them to operate at increasingly high data rates. Many of these distributed systems form parts of critical infrastructures, with real-time requirements. Hence it is imperative to provide error detection functionality to the applications. Error detection can broadly be classified as stateless detection and stateful detection. In the former, detection is done on individual messages by matching certain characteristics of the message, such as the length of the payload of the message. A more powerful approach for error detection is the stateful approach, in which the error detection system builds up state related to the application by aggregating multiple messages. The rules are then based on the state, thus on aggregated information rather than instantaneous information. Stateful detection is looked upon as a powerful mechanism for building dependable distributed systems [103][104]. The stateful detection models can be specified using various formalisms, such as, State Transition Diagrams, PetriNets or UML. Though the merits of stateful detection seem to be well accepted, scaling a stateful detection system with increasing application entities or data rate is a challenge. This is due to the increased processing load of tracking application state and rule matching based on the state. This problem has been documented for stateful firewalls that are matching rules on state spread across multiple, possibly distant, messages [104]. The stateful error detection system has to be designed without increasing the footprint of the system. Thus throwing hardware or memory at the problem is not enough because the application system also scales up and demands more from the detection system.

In chapter 2, we developed the Monitor detection system (also see [39]) which provides detection by only observing the messages exchanged between the protocol entities (PEs). The Monitor is said to verify a set of PEs when it is monitoring them. The Monitor is provided a representation of the protocol behavior (using a state transition diagram i.e., STD) of the PEs being verified along with a set of stateful anomaly based rules. The Monitor uses an observer model whereby it does not have any information about the internal state of the PEs. The Monitor performs two primary tasks on observing a message. First, it performs the state transition corresponding to the PE based on the observed message. Note that the state of the PE estimated by the Monitor may differ from the real state of the entity since not all messages related to state changes are necessarily observable at the Monitor. Second, it performs rule matching for the rules associated with the particular state and message combination. We observe that the Monitor has a breaking point in terms of the incoming message rate or the number of entities that it can verify beyond which the accuracy and latency of its detection suffer([105], Figure 7.1). The drop in accuracy or rise in latency is very sharp beyond the breaking point. We observe through a test-bed experiment that as the incoming packet rate into a single Monitor is increased beyond 100 pkt/s, the Monitor system breaks down on a standard Linux box. In other words, its latency becomes exceedingly high and accuracy of detection tends to zero. This effect is shown in Figure 7.1. This breakdown is caused by the processing capacity at the Monitor being exhausted. Hence, messages see long waiting times and on the buffer becoming full, the messages also get dropped. Thus, for reasonable operation, the Monitor can only support data rates below the breaking point.

In the current work, we devise a stateful detection approach which scales with the increasing data rate of applications, or equivalently, the number of PEs being verified. We observe that in order to make stateful detection feasible; firstly the processing of each message must be made extremely efficient and secondly the system must reduce the total processing workload (e.g., by selectively dropping incoming messages). The amount of work at the Monitor per unit time can be conceived as the rate of messages being processed for detection × the amount of work performed for each message. Our approach optimizes both these terms. The goal is to provide an error detection system for high throughput distributed streams and correspondingly push the knee to the right (Figure 7.1). Existing detection systems like [101][100] which aim at handling high data rate provide detection of changes in high rate streams using mean and higher order moments. This approach cannot capture the richness in the error detection rules that is needed for specifying verifiable behavior.

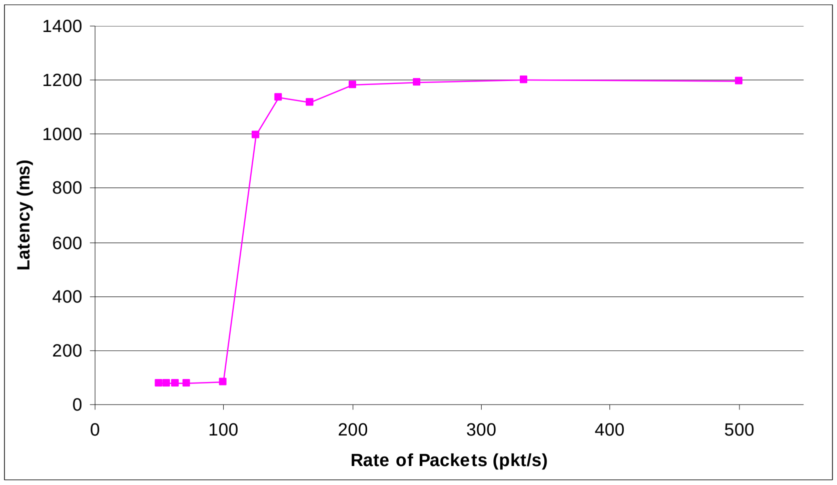


Figure 7.1: Latency variation with increasing inter-packet delay. The graph depicts the breaking of the Monitor system at an incoming rate of 100 pkt/s.

As a first aspect, we minimize the processing cost of an individual incoming message into the Monitor. We do this by using multistage hash tables for look ups when a state transition needs to be performed at the Monitor. We observe that for realistic systems, multiple rules will be active concurrently. The rules take the form of verifying values of some state variables or counts of messages (events) lying within a range. There exists significant overlap in the state variables or counts being referred to in the rules. Since processing for an incoming message most often involves updating these counts, we optimize this operation by compact representation of the state variables.

In the second aspect, we optimize the incoming message rate the Monitor has to process. We set a threshold for the incoming rate guided by the breaking point of the

Monitor. Sampling the incoming stream to reduce the rate of messages is a logical start. However, since the Monitor provides stateful detection, dropping messages can cause the Monitor to lose track of the PE's current state with resultant decrease in accuracy of rule matching. This phenomenon is called state non-determinism, whereby to the Monitor it is non-deterministic which state the PE is in. In our approach the Monitor tracks the set of possible states the application could have reached given that a sequence of messages is dropped. The Monitor aggressively pre-computes information about the states for possible sequences of messages to reduce the cost of computing the non-deterministic state set. While the cost of processing each (sampled) message now increases over the baseline case, through careful design the Monitor's total amount of work is reduced by reducing the rate of messages that it needs to process. The sampling is made adaptive to tolerate fluctuations in the message rate generated by the PEs. Also, the sampling scheme necessitates changes in the rules to prevent false detections due to the sampling.

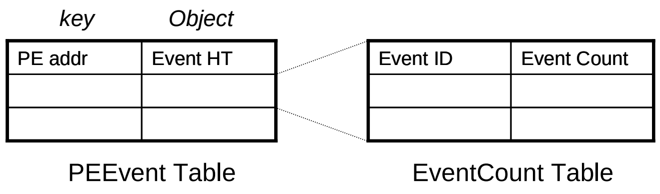
We implement the two aspects of efficient stateful detection in the Monitor and use it to detect errors in a reliable multicast protocol (TRAM). TRAM provides a motivating application since it is at the core of many e-learning applications which feed high bandwidth streams to a large set of receivers. We inject errors into the TRAM PEs and compare the accuracy and latency to the baseline system. The sharp decrease in performance beyond the breaking point is no longer observed; in fact, a sharp breaking point is completely eliminated and a gradual decrease in performance with increasing message rates is observed instead.

7.2. Scalable Stateful Detection

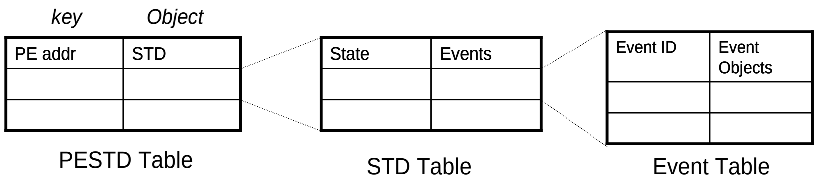
In developing a suitable approach for stateful detection we carefully study the tasks performed by Monitor-Baseline for error detection. Thus, the main steps on the receipt of a message are: 1) perform the state transition; 2) instantiate any rule corresponding to the state and event combination. Upon expiry of the time specified in a rule, the Monitor checks the value of the variable(s) mentioned in the rule to verify that they lie in the permissible range. It is observed for Monitor-Baseline that as the number of incoming messages increases, the latency of detection breaks down beyond a threshold. We attribute this problem quite intuitively to two root causes – 1) High cost of processing per message, and 2) High rate of incoming messages. We target both these causes and solutions to them are described respectively in Sections 7.2.1 and 7.3.

7.2.1. Making Rule Matching Efficient

In the modified approach, henceforth called *Monitor-HT* (for Hash Table, due to its widespread use in the redesign), we perform several modifications to Monitor-Baseline data structure to achieve efficient per message processing. Figure 7.2(b) depicts the logical organization of multi-level hashtables used in Monitor-HT. These hashtables are organized by carefully observing the processing path a message takes after being received by Monitor-Baseline. We designed the data structure consisting of multi-level hashtables to provide constant order look-up. The STDs of the PEs are organized as multi-level hashtables to provide constant order lookup. PE address is used in PESTD table to obtain the STD for that PE. The STD table is indexed using a state s_i which provides a list of events possible in that state (again organized as a hashtable). In the Event table each event ID maps to an event object, which contains information like event ID, event Name and rules pertinent to that event.



(a)



(b)

Figure 7.2: Data Structure used in Monitor-HT for (a) Storing Incoming Event Counts; (b) Storing the STDs. The first column represents the key of the hash table.

Next, in Monitor-Baseline, for every rule instantiation, its own copy of state variables is created. When a message arrives, active rules that depend on the message (through a state variable) are searched and every rule's local copy of the state variable is updated. This process is expensive because for every message, a long list is traversed. We observe that there exists significant sharing of state variables between the different rules and this makes the design of separate copy for each active rule inefficient. As an example, consider that multiple rules are tracking the data rate around different events, say within 5 seconds of a Nack being sent. All the rules would be counting the number of data messages (the state variable) received over different time intervals.

Monitor-HT removes the above-mentioned source of inefficiency by having a central store of the state variables. Monitor-HT keeps a hashtable to store the updates for a given message (see EventCount table in Figure 7.2(a)). We use a multi-level hashtable where PEEvent indexes all the PEs in the system and the EventCount table contains all the events corresponding to the given PE. The incoming messages can be thought of as a tuple as (a_i, e_i), where a_i is the PE address (IP address or some logical address) and e_i is the event ID. The value a_i is used to look up PEEvent table for the events. The e_i is used to index in EventCount table and increment the event count for e_i (currently all increments are by a value of 1). Because of this organization every unique $PE \times Event\ ID$ symbol is only incremented once.

Regarding the rule matching procedure, instead of having every active rule use local variables, every rule instance reads the value of the associated state variable from the hashtable. When a new rule is created it reads the value of the current event count from the EventCount table to see the current value of the state variable referenced in the rule, call it $v_{init.}$ Later, at the time of rule matching, the Monitor-HT again reads the value of the state variable, call it v_{final}. Thus, the EventCount table is read from the rule instances only twice, and written by a separate thread which handles the incoming messages from the PEs. The advantage of Monitor-HT over Monitor-Baseline, quantified in the experiments, is dominated by the effect of this design choice.

7.3. Handling high rate streams: Sampling

Even with the modifications made in Monitor-HT, a constant amount of work is performed for every incoming message. In the next optimization, not all messages are processed; instead messages are sampled and only the sample set is processed. This version is called Monitor-Sampling, or *Monitor-S.* Sampling raises a few obvious questions:

- How and what sampling approach should be taken?
- How are the rules modified due to sampling?
- How does Monitor-S track the PE's STD in the presence of sampling?

The first two questions are answered in Section 7.3.1 and the third one in Section 7.3.2.

7.3.1. Design of Sampling

We propose uniform sampling approach which is agnostic to the kind of messages coming in. This prevents Monitor-S from having to deduce the type of the incoming message before deciding to drop it or keep it. This would have imposed the per message processing overhead on Monitor-S and defeated the purpose of the design. With sampling, the corresponding parameters in the detection rules have to be re-adjusted for matching. Assume that the Monitor gives a desired latency and accuracy of matching for an incoming rate of upto R_{th}. Any rate $R > R_{th}$ the Monitor chooses to drop the messages uniformly with a rate of 1 in every $R\ /(R\ -\ R_{th})$ messages. Figure 7.3 illustrates the behavior of Monitor which switches from Monitor-HT to Monitor-S because sampling kicks in after R_{th}. Since the messages being processed by Monitor-S are a sample of the entire set of messages, the rules originally specified by the system administrator are not valid on the sampled stream.

Once a new sampling rate is chosen based on the incoming traffic rate, the rules are also modified. We keep the rule type the same but the constants get scaled according to the sampling rate. This is necessary because rules are defined according the normal operation of the PEs but because of sampling, Monitor-S is viewing an alternate sampled view of the operation of PEs. If the incoming rate is R and the threshold rate is R_{th} then the constants in the rules must be scaled by a factor of R_{th}/R. For example: if a rule states "*receive 10 Acks in 100 sec*" then because of sampling the rule is modified to "*receive 10.*(R_{th} / R) *Acks in 100 sec*". This rate will be changed as and when the incoming rate is changed. We measure the incoming rate over non-overlapping time windows of length Δ by counting the number of incoming messages in the window. At each rate computation, the new rate is compared with R_{th} and if it exceeds R_{th} then a new sampling rate is determined based on this new incoming message rate. To reduce the overhead of rate computation Δ is kept higher than the time period over which a rule is matched.

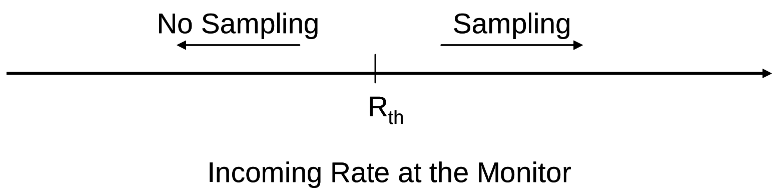


Figure 7.3: Change in Monitor's algorithm beyond a threshold rate of packet (R_{th}).

7.3.2. STD Transition with Sampling

If all incoming messages are not processed, this will cause the Monitor-S to lose track of the current state of the PE. We modify the approach of STD transitioning at Monitor-S such that instead of tracking the current state, Monitor-S keeps a state vector $\vec{S}$ which contains all the possible states the given PE can be in $\vec{S} = \{S_1, S_2 \ldots S_K\}$. The reason for having multiple possible states is that Monitor-S does not know which of several possible paths the PE has taken given a start state S_{start}.

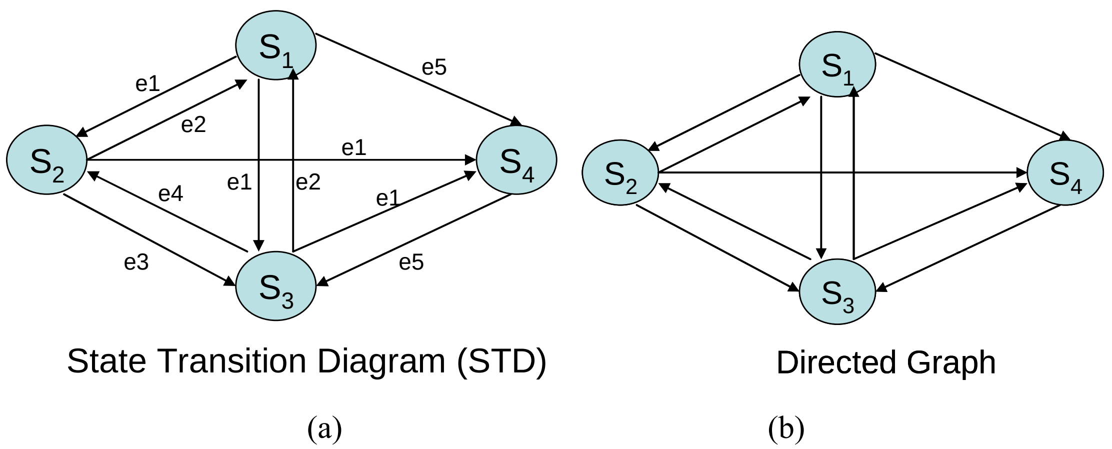


Figure 7.4: A sample STD which is converted to a directed graph by removing the event labels.

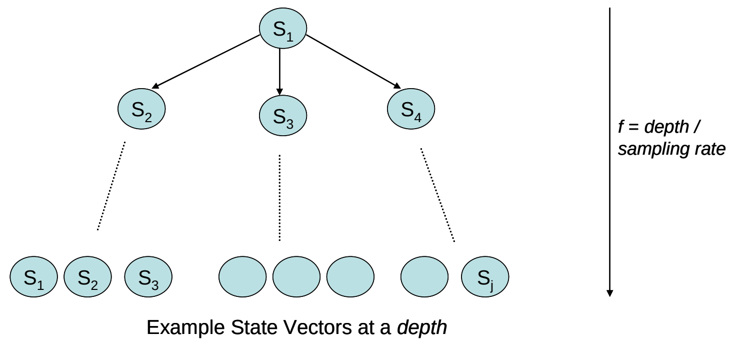
As a result of sampling, instead of knowing exactly which state the PE is in, Monitor-S will know a possible set of states the PE is in (based on the transition edges outgoing from the current state). For example: In Figure 7.4(a) if the current state is S_1 and a packet is dropped then the next possible state is one of $\{S_2, S_3, S_4\}$. To determine this set, Monitor-S pre-computes the possible states which can be reached in steps of size 1, 2, 3 and so on. Each set of these states form the state vector $\vec{s}$ if 1, 2, 3 and so on messages are dropped. In other words if a single message is dropped starting from the start state S_{start}, then $\vec{s}_1$ will consist of all the states S_i such that S_i has an incoming edge from S_{start} in the graph. $\vec{s}_i$ vector starting from state S_{start} gives the state vector if i packets are dropped. Now given the rate of sampling one can transform one state vector $\vec{s}_1$ to another state vector $\vec{s}_2$. Let us say $\vec{s}_0 = \{S_i \mid i \in (1, g)$; g is the number of nodes in the initial state vector$\}$ be the initial state vector. If Monitor-S dropped one message then the new state vector $\vec{s}_1 = \{S_j \mid S_i \rightarrow S_j$ is reachable using a single edge AND $S_i \in \vec{s}_0\}$. Similarly if 2 messages are dropped then $\vec{s}_2 = \{S_m \mid S_j \rightarrow S_m$ is reachable using a single edge AND $S_j \in \vec{s}_1\}$.

The state vectors ($\vec{s}_1$ and $\vec{s}_2$) are created offline because the STD is already known to Monitor-S. Figure 7.4 (a) illustrates for the STD in Figure 7.4, a tree structure for maintaining the state vectors after different numbers of messages are dropped. Nodes at the depth h form the state vector $\vec{s}_h$ and represents the states after h messages are dropped starting from S_1. At runtime, Monitor-S tracks how many messages are dropped and looks up the appropriate state vector.

7.3.3. Error Detection with Sampling

Figure 7.4 (b) represents the flow of detection in Monitor-S when sampling is taking place. If the incoming rate is below R_{th} then no sampling occurs and Monitor-S simply runs as Monitor-HT. During sampling, the state transition is performed between various state vectors $\vec{S}$ which have been computed offline. When a message is sampled, all detection rules corresponding to that event ID and states in the current $\vec{S}$ are instantiated for matching. When messages are being dropped, the size of the state vector ($|\vec{S}|$) increases. Once a message is sampled, the state vector is pruned since the message may not be valid for all the states in the state vector. Consider that the state vector is $\vec{S}^{a-}$ just before sampling and $\vec{S}^{a+}$ just after sampling message M. Then $\vec{S}^{a+} = \{S_i| S_i \in \vec{S}^{a-}$ and M is a valid message in state S_i according to the PE's STD$\}$. Qualitatively, the sampling scheme will be beneficial only if the pruning in the size of the state vector is significant compared to the growth due to message drops. For example: let $\vec{S}$ initially consists of $\{S_1, S_2, S_3\}$ and the sampled message be e_2. Then from Figure 7.4 we can see that only S_2 and S_3 can have a valid event e_2 and therefore the state vector becomes $\{S_2, S_3\}$.

This ambiguity about which state the PE is in and the design of using the entire state vector may give rise to false alarms since Monitor-S may match some rules that are not applicable to the actual state the PE is in. Computing the state vectors offline imposes a memory requirement on the system. If we assume that at most τ messages will be dropped by Monitor-S then the offline computation should have state vectors upto $\vec{S}_\tau$. The total number of states in this state vector tree is given by $k(k^\tau-1)/(k-1)$ assuming a k-regular structure of connectivity between the states. Thus the space required to store these state vectors is proportional to $k(k^\tau-1)/(k-1)$. However the total number of states in the STD also imposes a cap on the size of the state vectors and prevents further increase in $|\vec{S}|$. If there exists a ω *s. t.* $k^\omega > N$ (total states in STD), then space required to store the state vectors is proportional to $k(k^{\omega\text{-}1}\text{-}1)/(k\text{-}1)+(\tau\text{-}\omega+1)N$. The exact memory required is dependent on the data structure used to store these state vectors. Bit vector representation for storing them is an efficient option to reduce the overall memory used.



(a)

1. Input Rules and STD for the PEs for detection by Monitor-S
2. Construct the State Vectors offline
3. Run the Monitor and start verifying the PEs
4. If $R_{incoming} < R_{th}$ operate in Monitor-HT mode else operate as Monitor-S
5. If sampling, then perform state transition using the state vectors
6. For every sampled message instantiate rules for all states in the state vector

(b)

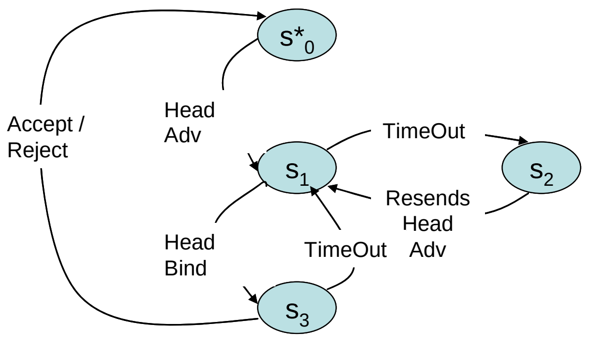
Figure 7.4. Union of nodes present at depth *h* represents the nodes in set $\vec{S}_h$ if *h* messages are dropped starting with S_1. (b) Flow of detection in Monitor-S.

7.4. Experimental Setup

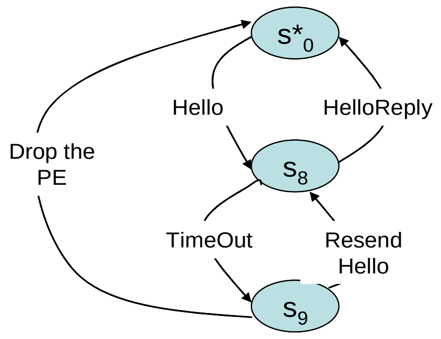
7.4.1. Application: TRAM

We demonstrate the use of the Monitor on the running example protocol — a reliable multicast protocol called TRAM [5]. For the sake of completeness, we will recapitulate the basic features of TRAM. TRAM is a tree based reliable multicast protocol consisting of a single sender, multiple repair heads (RH), and receivers. Data is multicast by the sender to the receivers with an RH being responsible for local repairs of lost messages. The reliability guarantee implies that a continuous media stream is to be received by each receiver in spite of failures of some intermediate nodes and links. An Ack message is sent by a receiver after every Ack window worth of messages 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.

The multicast tree is formed via sender sending Head Advertisement messages and new nodes joining using the Head Bind message (see Figure 7.5(a)). Nodes ensure liveness of other neighbor nodes by periodically sending Hello messages as depicted in the STD shown in Figure 7.5(b).



(a)



(b)

Figure 7.5: Example State Transition Diagrams (STDs); (a) TRAM sender adding new receivers in TRAM; (b) TRAM entities (sender, receiver, RH) sending *liveness* messages (Hello).

The detection approach is provided with a rule base for detection which is derived from the STDs (shown in Figure 7.5). Some example of rules are as follows: *R4 S4 E11*

30 500 5000 S4 E2 1 8 4000 7000 If a *Data* message is seen then the Monitor must see an Ack message following it; *T R4 S1 E9 1 2 1000 S1 E8 1 2 2000 3000*: If the entity is in state S_1 then it the Monitor should observe one or more Head Bind messages followed by Accept message; *T R3 S0 E14 10 30 5000*: The number of Hello message within a time window should be bounded to prevent Hello flooding.. It is evident from the set of rules that several of them verify the message count for the same message type (such as, Data, Hello, Ack). Therefore the redesign of Monitor-HT of keeping only a shared writable copy of the state variables is likely to be beneficial. More rules used in our experiments are listed in Appendix B.

7.4.2. Emulator

In order to be able to study the performance of the Monitor under high data rate conditions, we emulate the TRAM protocol [3][5]. This is necessary because operating multicast protocol across Purdue's shared wide area network at a high data rate causes multiple switches to crash. The extra beacon messages sent out for advertising the multicast channel causes an overload of the LAN switches leading them to crash. In order to avoid this problem and to have the ability to perform experiments in a controlled environment, we emulate the topology of TRAM depicted in Figure 7.6. The emulated messages following the STDs in Figure 7.5 are forwarded to the Monitor.

7.4.3. Fault Injection

We perform *fault injection* in the header of the emulated TRAM messages to induce failures. We choose the header since the current detection mechanism only examines the header. In general a PE to inject is chosen (sender, RH or receiver) and faults are injected for a burst length. We use a burst length of 500ms and inject the burst length of faults after every 5 minutes during each experimental run. For these experiments we inject only the sender with faults because of high probability of error propagation down the multicast tree. A burst length is chosen since TRAM is robust to isolated faults and to mimic faults close to reality. The rules in the rule base typically run over a window of messages and are likely to not get violate because of an isolated faulty message.

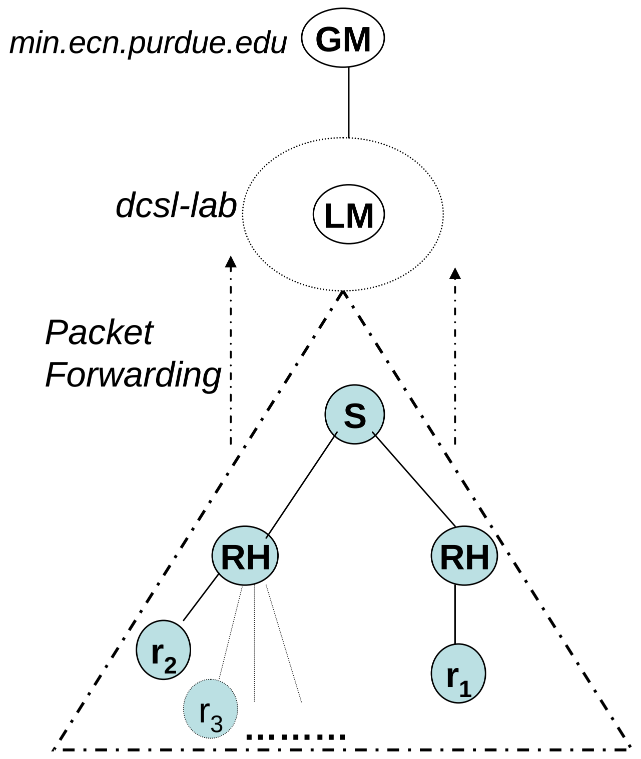


Figure 7.6: Physical Topology of the TRAM emulator and the Monitor in the experiments

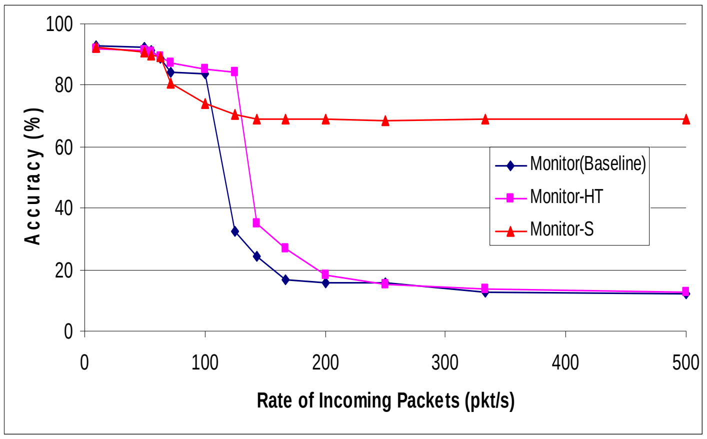
The burst can cause multiple rules to be instantiated simultaneously for each of sender, RH and receiver. Note that the emulated faults are not simply message errors, but may be symptomatic of protocol faults in the PEs. Errors in message transmission can indeed be detected by checksum computed on the header but these protocol errors cannot. We perform *random injection* where a header field is chosen randomly and changed to a random value, valid or invalid *w.r.t.* the protocol. If the injected value is not valid, then the message is dropped without processing. An alternate mode of error injection used in our earlier work [39] is directed injection whereby messages are transformed to a valid protocol value. Experimentally, we find that the performance of Monitor-HT and Monitor-S relative to Monitor-Baseline is not affected by this choice.

7.5. Experiments and Results

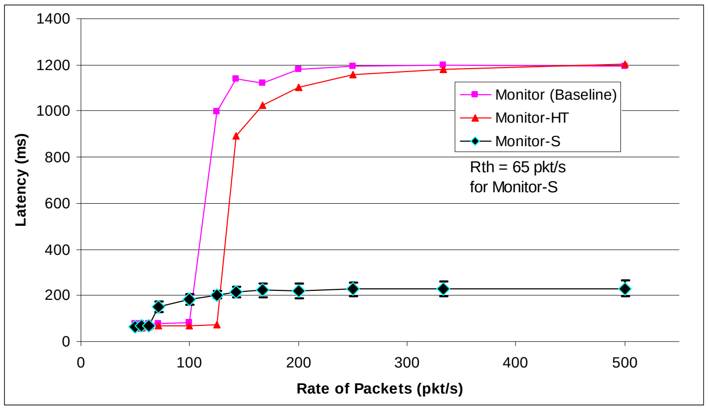
Experiments are performed on the topology shown in Figure 7.6. The Monitor system and the TRAM emulator are executed on separate desktop PCs with a 2.4GHz processor and 1GB RAM. We use TRAM sender and receiver (Figure 7.6) as the PEs being verified by the Monitor in all the experiments. We measure the *accuracy* and *latency* of detection procedure for the Monitor. Accuracy is defined as (1-missed detections). We characterize the fault injections which affect the PEs but are undetected by the Monitor as missed detections. A PE is said to be affected if it crashes or raises an exception. False detections are defined as the errors which are flagged by the Monitor but do not affect the TRAM entities. Latency is measured as the time from the instantiation of a rule to the time when the rule matching is completed, subtracting the time for which the rule is dormant. For example, if a rule states "*Observe 32 data messages in 5 sec*" then 5 sec is the time during which the there is no Monitor-related processing. This time needs to be subtracted since it is not an index of the Monitor's performance; rather it is a feature of the rule itself. The value of Δ in our experiments is set to 30 seconds.

7.5.1. Accuracy and Latency Results

We vary the incoming data rate for the Monitor by varying the inter-packet delay from the sender. The emulator sends packets at a low rate of 20 pkt/s for the first 30 seconds and then increases it to the required rate. Each experiment run lasts for 20 minutes. Every latency and accuracy value is averaged over at least 60 data points. The experiment is repeated for three different systems i.e., Monitor-Baseline, Monitor-HT, and Monitor-S. Every packet is forwarded to the Monitor from the TRAM PEs. The rate of packets is varied between 10 pkt/s and 500 pkt/s. Figure 7.7(a) shows the variation of accuracy with packet rate. The 95% confidence interval is plotted for Monitor-S and is seen to be very small indicating that the variance in the results is small. We can see that with an improved data structure Monitor-HT's knee, i.e., the breaking point, occurs around 125 pkt/s compared to 100 pkt/s for Monitor-Baseline. Let us denote the breaking point for the incoming message rate as R_{bp}. The improvement of 25% is due to the sharing of the state variables and the efficient hash table lookup. The false alarms vary between 0-6% for both Monitor-HT and Monitor-Baseline. For extremely high packet rates, Monitor-HT and Monitor-Baseline have a drop in false alarms because the number of rule matches itself is reduced.



(a)



(b)

Figure 7.7: Variation of (a) Accuracy and (b) Latency with increasing rate of packets.

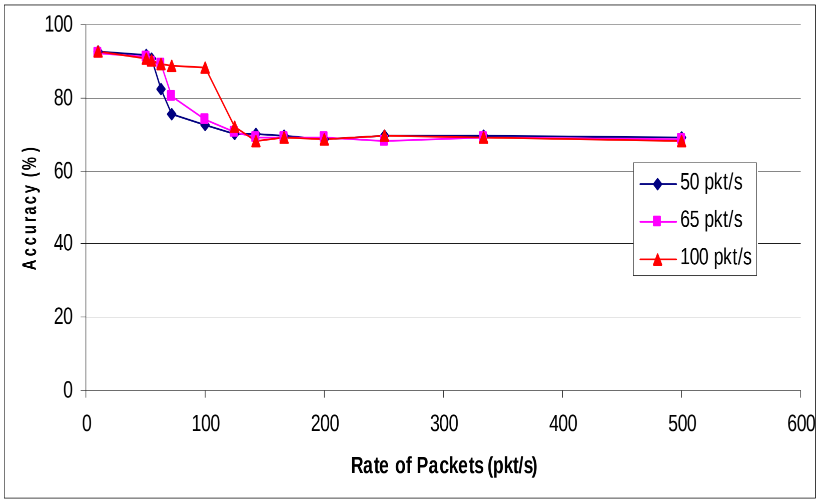
We can see that beyond 125 pkt/s even with efficient per packet processing, the accuracy drops below 40% because of the increased rate of incoming messages which causes the processing capacity of Monitor-HT to be exhausted. In comparison, with sampling, the accuracy drops gradually as the Monitor-S drops increasingly more packets with increasing data rate to maintain the rate below R_{bp}. We can observe from Figure 7.7(a) that with increasing packet rate Monitor-S has a small decrease in accuracy but it still maintains accuracy at approximately 70% compared to Monitor-HT's 16% accuracy. Monitor-S has a marginal increase in the rate of false alarms due to the knowing of the state vector rather than the precise state. The false alarms vary between 0-9%. At high data rates we observe lower false alarm rates for Monitor-S compared to low data rates.

An example of a rule which does not get violated due to sampling resulting in loss of accuracy is *R1 S0 E1 1000 S8 1500 2500.* This rule verifies that for a TRAM PE (sender, receiver) the state has successfully changed to *S8* from *S1* after receiving *E1* (Hello message). At high data rates if a large number of packets is getting dropped, it happens that $\vec{S}$ still contains state *S8* causing this rule not be violated and hence decreasing the accuracy.

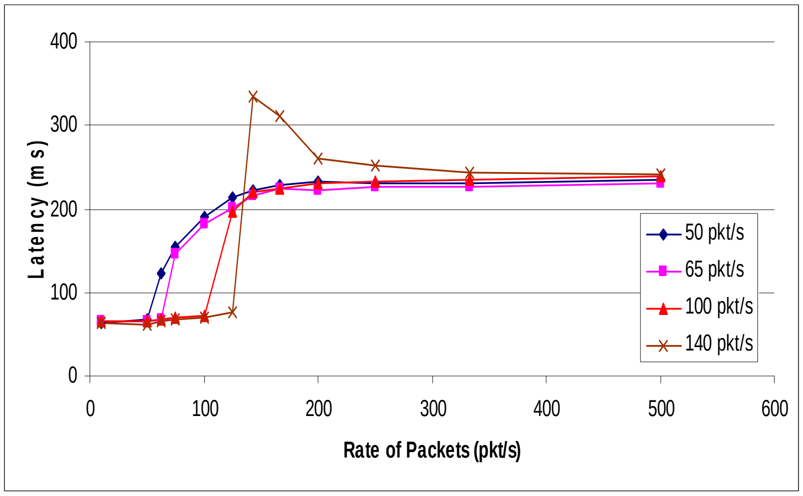
The latency plot in Figure 7.7(b) provides a similar picture. The breaking points for Monitor-Baseline and Monitor-HT are the same as in the accuracy plot – 100 pkt/s and 125 pkt/s respectively. For Monitor-S, we can see a small jump in latency around 65 pkt/s (R_{th} in this experiment) because the algorithm switches to sampling and the probability of dropping a packet increases (being zero previously). This results in a higher overhead for processing each packet and the attendant marginal increase in latency. The processing done by Monitor-S is proportional to $|\vec{S}|$ times the number of detection invocations. Increasing data rate causes higher $|\vec{S}|$ leading to higher latency of rule matching. However, the growth of $|\vec{S}|$ slows down with increasing packet rate causing the latency to saturate. We observe that even at high packet rates Monitor-S maintains a low latency of rule matching (~200ms) because of effective adjustment to the sampling rate reducing the rate of packets that are processed. This provides an 83.3% decrease in latency compared to the latency of 1200ms for Monitor-Baseline.

For a fixed R_{th}, as the data rate is increased, the size of the state vector ($|\vec{S}|$) increases but it saturates at higher packet rates. The processing for the rule matching is directly proportional to $|\vec{S}|$. Also, as the data rate is increased beyond R_{th}, the number of rule invocations of Monitor-S stays constant. The latency is proportional to the total work done by Monitor-S, which is given by: *processing for the rule matching* $\times$ *number of rule invocations of Monitor-S*. Therefore, initially when the data rate is increased beyond R_{th}, the latency increases, but beyond a point, it saturates.

7.5.2. Effects of Varying R_{th}



(a)



(b)

Figure 7.8: Effect of R_{th} on the (a) Accuracy and (b) Latency.

Figure 7.8(a) depicts the behavior of accuracy and latency for different values of R_{th} in Monitor-S. Recollect that when the incoming message rate goes above R_{th}, the Monitor switches to the sampling mode. For all cases the accuracy is almost the same at high data rates and low data rates. Let us consider a single curve (say R_{th} = 50 pkt/s). For data rates below 50 pkt/s there is no sampling and since this threshold is much below the breaking point (125 pkt/s from Section 7.5.1) the latency remains quite low (~65ms). As the data rate increases beyond 50 pkt/s, sampling starts and with increasing data rate an increasing number of packets is dropped. Difference in characteristics of the curve around R_{th}

provides the system administrator a useful tuning parameter to choose a suitable latency value for the requirements of the distributed application. Clearly picking $R_{th} > R_{bp}$ is unsuitable due to the spike in latency (see the 140 pkt/s curve). It is tempting to choose R_{th} as close to R_{bp} as possible (notice the delayed increase in latency for R_{th} = 100 pkt/s compared to R_{th} = 50 pkt/s). However, in practice the breaking point cannot be exactly determined since it depends on the kinds of messages (and hence, the kinds of rules) that are coming into the Monitor. Thus the system administrator has to choose a R_{th} suitably below R_{bp}. For our experimental setup, if a latency of less than 100 ms is desired for data rates up to 100 pkt/s, then R_{th} of 100 pkt/s is an appropriate choice.

When R_{th} is 140 pkt/s, i.e., greater than the breaking point (125 pkt/s), it causes a heavy load and higher latency of matching for the region (125 pkt/s, 140 pkt/s). But as the run of experiment continues, sampling starts and this brings down the average latency to just over 300ms. The jump in the latency is because the incoming rate is close to the R_{th} because of which the Monitor switches between sampling and non-sampling modes. However in the non-sampling mode, since incoming rate is greater than R_{bp} Monitor-S incurs a high latency. This oscillation between the modes happens when the rate is close to R_{th} which explains the high latency (275-330 ms) around the incoming message rate of R_{th}.

7.5.3. Variation of State Vector Size ($|\vec{S}|$)

As described before, the amount of processing done by Monitor-S is dependent on size of state vector i.e., $|\vec{S}|$. We investigate the variation of $|\vec{S}|$ with time in an experimental run. In this experiment we keep the R_{th} fixed at 65 pkt/s and run the emulator to provide an incoming rate of 250 pkt/s. This experiment is targeted at bringing out the dynamics of Monitor-S when the incoming message rate is higher than the breaking point, forcing sampling to kick in. For this configuration, approximately one in four packets is sampled. Figure 7.9 shows the variation of $|\vec{S}|$ with time. We measure the size of state vector once every 2 packets. Instead of displaying the entire run of 20 minutes, we pick a representative 100 contiguous samples of $|\vec{S}|$. We can see the large fluctuations of $|\vec{S}|$ due to the sampling. We can see that $|\vec{S}|$ grows to as large as 10, multiple times during the experimental run. The number of rules which get instantiated for each packet is proportional to $|\vec{S}|$. However the rules get instantiated *after* a message is sampled. When a message is sampled, it will likely cause $|\vec{S}|$ to decrease because all the states in $\vec{S}$ do not have the message as a valid message in that state. Thus the rule instantiations take place at the troughs and not at the peaks of the plot in Figure 7.9. We can see that in Region 1, $|\vec{S}|$ drops in steps from 9 to 6 and finally to 1. The drop in $|\vec{S}|$ is because of the unique possibility of the sampled event in only some of the states.

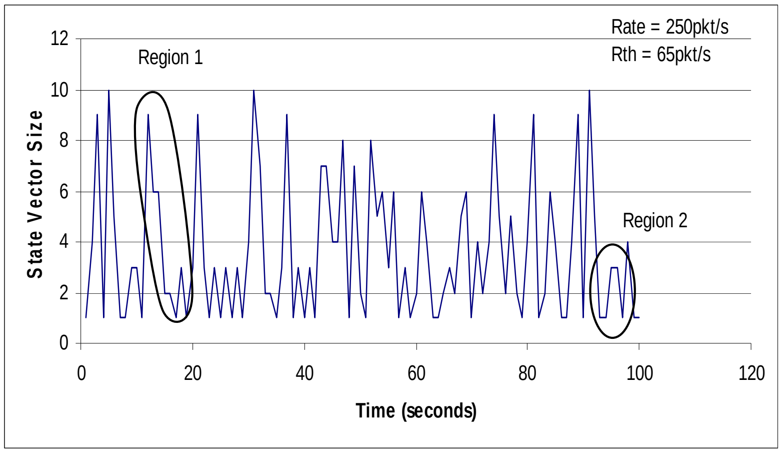


Figure 7.9: Variation of State size $\vec{S}$ in a sample run.

$|\vec{S}|$ can also remain the same if the dropped event corresponds to some self-loops. This explains the small plateaus in Region 2. In Region 2, $|\vec{S}|$ increases from 1 to 3 because of a message drop. It stays at 3 even with further message drops and then reduces to 1 with a newly sampled message.

7.6. Discussion

This chapter addressed one of the important pieces of the Monitor framework, *scalability*. We observe that through incorporating sampling, one can prevent the drastic fall in the Monitor's accuracy. In other words we avoid the breaking of the Monitor. One can see that through this approach Monitor achieves a much higher accuracy of detection. With addition of sampling, the framework truly stands to the claim of scalability with increasing data rate or number of entities to be verified. Although sampling helps us in maintaining a reasonable accuracy and latency, there are some drawbacks of the approach in certain scenarios.

7.6.1. Losing track of PE's state

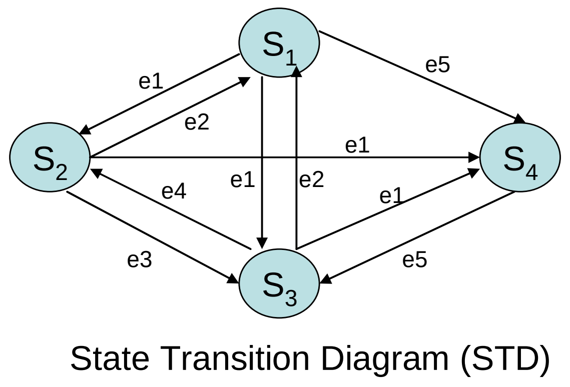


Figure 7.10: An example STD

The sampling approach tries to reduce the workload by dropping packets. Simultaneously in an effort to provide stateful matching and track the state of the PE the Monitor tries to use a state vector of possible states. However there can be scenarios in which Monitor can actually lose track of PE's state. Consider the sample STD shown in Figure 7.10. Assume that initially the state vector equals $\{S_1\}$. Now the next incoming message is dropped which causes the new state vector to be $\{S_2, S_3, S_4\}$. Assume that the next incoming message received at the Monitor is e_3. However e_3 was received because of an error in the communication channel and actually the message which was sent out was e_5. Now because of this error of receiving e_3, the new state vector becomes $\{S_2\}$ however the actual state vector should be $\{S_4\}$. In this scenario now the Monitor has actually lost track of the state of the PE. This actually is possible because now the state vector has more than one state and correspondingly a lot more events are plausible events (as in e_3 in this case). Further on if the Monitor tries to perform state transitions or expand state vectors, either it will start jumping through erroneous states or it will cause false alarms. This is an important drawback which occurs because of the way sampling approach handles the state tracking.

The important thing here is that Monitor should try to latch back to the correct state so as to be able to provide correct detection. One way to do that is to sample the next α messages. If the STD is deterministic and holds the α-distinguishable property then Monitor should be able to deduce the correct state. The α-distinguishable property ensures that for a deterministic STD if α consecutive messages are observed then one can make a conclusion about the starting state before the α messages were observed. And hence one also knows the final state of the STD after α messages. A comprehensive guide on this approach and software testing can be found at [126]. This approach can be used by the Monitor to latch back to the correct state if it loses track of the state. However, answering the question "How to determine that Monitor has lost track of state" is challenging. Some starting heuristics can be used for example: if Monitor sees a high rate of false alarms within a certain time window then it should try to check if the PE state is valid.

8. RELATED WORK

8.1. Detection

Formal Specification: Preliminary to building self-checking protocols, the application behavior has to be specified formally. Different formalisms exist for distributed systems, the most common ones being Extended State Machines [79], Temporal Logic Actions (TLA) [80], [81], UML [96] and Petri net based models [82]. Our approach is derived from the TLA model where the valid actions are represented as logical formulas. The formulas can be augmented with the notion of lower and upper time bounds to capture the temporal properties of protocols. We employ State Transition Diagrams for protocol representation because it provides all the needed functionalities to express most of the protocols and it is faster to do operations on it compared to other approaches for e.g., PetriNets.

There is a volume of work on detecting crash failures through heartbeats, failure detectors, etc. (e.g., see [83]), building resilient distributed applications through fault tolerant algorithms built into the application (e.g., see [84][85]). Their goals are considerably different from the work presented here and hence, not surveyed further. There is previous work [86][87] that has approached the problem of detection and diagnosis in distributed applications modeled as communicating finite state machines(CFSMs). The designs have looked at a restricted set of errors (such as, livelocks) or depended on alerts from the protocol entities themselves. A detection approach using event graphs is proposed in [87], where the only property being verified is whether the number of usages of a resource, executions of a critical section, or some other event globally lies within an acceptable range. The problem of diagnosis in distributed systems has been studied in [46][88] which have relied on participation by the protocol entities and the classes of faults have also been restricted. Near identical goals, as in this thesis, have motivated the work in [12]and [89]. In the first work, the approach is to structure the system as two distinct sub systems — worker and observer. The worker is the traditional system implementation, while the observer is the redundant system implementation whose outputs are comparable to the worker outputs. The observer can only spy on interactions, without any worker support. The observer is made highly reliable through formally specifying and verifying it. Some unanswered questions are that the observer is a monolithic entity and is not shown to be able to operate outside a broadcast medium, how the subset of worker functionalities for observing is determined, and the independent verification of layers of the worker are apt to miss out misbehaviors that span multiple layers. An extension to use multiple observers is proposed in [42], but it requires a global state graph of the system which may be infeasible to build or verify at runtime for complex systems. In [89], the authors propose a compositional approach to automatic monitoring of distributed systems specified using CFSMs. The fundamental contribution is to show how to monitor a complex system by monitoring individual components, thereby eliminating the state space explosion problem. This work assumes some internal states are visible to the monitor through program instrumentation, etc. It assumes that if local interactions are correct, the system execution is globally correct. This is in contrast to our system, where we allow for the possibility of a global rule flagging an error where the local rules missed it. Finally, the effectiveness of the approach has not been demonstrated through any error injection based experiments.

8.2. Diagnosis

White box systems: The problem of diagnosis in distributed systems can be classified according to the nature of the payload system being monitored – white box where the system is observable and, optionally, controllable; and black box where the system is neither. White box diagnostic systems often use event correlation where every managed device is instrumented to emit an alarm when its state changes [45][53]-[55]. By correlating the received alarms, a centralized manager is able to diagnose the problem.

Obviously, this depends on access to the internals of the application components. Also it raises the concern whether a failing component's embedded detector can generate the alert. This model does not fit our problem description since the target system for the Monitor comprises of COTS components, which have to be treated as black-box. A number of white box diagnostic systems that correlate alarms have been proposed in the intrusion detection area [49][50]. An alternative diagnostic approach is to use end-to-end probing [56]-[58]. A probe is a test transaction whose outcome depends on some of the system's components; diagnosis is performed by appropriately selecting the probes and analyzing the results. Probe selection is typically an offline, inexact, and computationally heavy process. Probing is an intrusive mechanism because it stresses the system with new requests. Also it is not guaranteed that the state of the system with respect to the failure being diagnosed has stayed constant till the time of the probe. Monitoring approaches have also been proposed in [38], [42]for distributed systems.

Multiprocessor system diagnosis: The traditional field of diagnosis has developed around multiprocessor systems, first addressed in a seminal paper by Preparata et al. [46] known as the PMC method. The PMC approach, along with several other deterministic models [59], assumed tests to be perfect and mandated that each entity be tested a fixed number of times. Probabilistic diagnosis, on the other hand, diagnoses 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[48]. Follow up work focused on multiple syndrome testing [47] 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 authors in [60] propose a fully distributed algorithm that allows every fault-free node to achieve diagnosis in at most, $(log\text{N})^2$ testing rounds. More recently, in [61] the authors extend traditional multiprocessor diagnosis to handle change of failure state during the diagnostic process. All of these approaches are fundamentally different from ours since there is no separation between the payload and the monitor system. This implies the payload system has to be observable and controllable (to generate the tests and analyze them).

Embedded system diagnosis: There has also been considerable work in the area of diagnosis of embedded systems, particularly in automotive electronic systems. In [62], the authors target the detection and shut down of faulty actuators in embedded distributed systems employed in automotives. The work does not consider the fallout of any imperfection in the analytical model of the actuator that gives desired behavior. The authors in [63] use assertions to correlate anomalies from the components to determine if a component is malfunctioning. The technique has some shared goals with the Monitor system – ability to trace correlated failures of nodes in a distributed system and handle non binary results from tests. The approach uses assertions that can examine internal state of the components. The papers in this domain do not consider imperfect observability of the sensor input or the actuator output, possibly because of tight coupling between the components. They are focused on scheduling monitor processes under processing resource constraints while we do not have such constraints.

Debugging in distributed applications: There has been a spurt of work in providing tools for debugging problems in distributed applications – performance problems [64]-[66], misconfigurations [67], etc. In [64] authors propose two offline algorithms to draw causal relationships between communicating processes. The method relies on RPC communications and stops at deriving causal relations. MagPie[67] is a system which performs extensive instrumentation in the application to obtain extremely accurate tracing of message calls. The general flavor of these approaches is that the tool collects trace information at different levels of granularity (line of code to process) and the collected traces are automatically analyzed, often offline, to determine the possible root causes of the problem [51]. For example, in [64], the debugging system performs analysis of message traces to determine the causes of long latencies. The goal of these efforts is to deduce dependencies in distributed applications and flag possible root causes to aid the programmer in a manual debug process, and not to produce automated diagnosis. Irina *et. al.* in [98] address the problem of optimal test selection so as to minimize the number diagnostic tests which need to be used. Their approach is complementary to ours and can be used in tandem.

Automated diagnosis in COTS systems: Automated diagnosis for blackbox distributed

COTS components is addressed in [68][69]. The system model has replicated COTS application components, whose outputs are voted on and the minority replicas are considered suspect. This work takes the restricted view that all application components are replicated and failures manifest as divergences from the majority. In [52], the authors present a combined model for automated detection, diagnosis, and recovery with the goal of automating the recovery process. However, the failures are all fail-silent and no error propagation happens in the system, the results of any test can be instantaneously observed, and the monitor accuracy is predictable.

In none of the existing work that we are aware of, there exists a rigorous treatment of the impact of the monitoring system's constraints and limited observability of the payload system on the accuracy of the diagnosis process.

9. CONCLUSIONS AND FUTURE WORK

In this Thesis we have described a scalable approach for non-intrusive monitoring of distributed systems. We have addressed detection, diagnosis, state space explosion, and scalability issues faced by a detection and diagnosis framework in high throughput distributed systems. The developed hierarchical Monitor architecture is deployed and tested across Purdue for detection and diagnosis.

9.1. Future Work

Future research directions include a multitude of extensions on theoretical and practical front.

Virtual Machine Management: Virtual Machine are emerging as a new paradigm for distributed computing. Adding a *hypervisor* (VMware[112], Xen, UML) layer and stacking multiple Operating Systems together on the same physical box has multiple advantages. Given the cost effectiveness of this solution, it is getting wide acceptance and replacing the way distributed applications currently run. Virtualization, in its microcosm, brings a whole new challenge to system management. The increased layer causes increased complexity and makes it harder for a system administrator to find and resolve problems. Management of distributed systems is a huge practical problem which is plagued with human errors and is in dire needs of autonomic solutions. I worked with IBM T. J. Watson Research Lab to help their system management group tackle this problem. I proposed several novel measures including the Monitor architecture to be applied to virtualized server scenarios for problem detection and diagnosis. The work presented in this PhD thesis can be used to provide efficient and scalable virtual machine management.

Autonomic Recovery: An important of the puzzle in improving reliability is in providing recovery and response in the event of failures. Also recovery is important after one has performed diagnosis to prevent recurrence of failures. Fault Tolerance community currently lacks models for providing recovery. Existing literature and tools about providing recovery mere rely on reboot or are theoretical with little to no feasibility. After having spent my time in graduate research and talking with people in industry, I am motivated to research on how to provide automatic recovery? Is recovery going to be specific to every applications or can we abstract some properties and make some generalizations? These questions are important and require in-depth study before one can develop a completely autonomous system.

Modeling the Monitor System: The presented PhD thesis has focused mainly on the experimental evaluation of the Monitor framework. Research efforts in developing good mathematical models can provide useful insights into the behavior of the Monitor Framework. The thesis provides modeling of some aspects of the framework which can be used as a starting point. Theoretical bounds can provide important boundaries for system administrator while using the Monitor framework.

LIST OF REFERENCES

LIST OF REFERENCES

[1] H. Eriksson, “MBone: The Multicast Backbone”, *Communications of the ACM*, pp.54-60, August 1994.

[2] Network World Fusion, February 23, 2001, "Can one rogue switch buckle AT&T's ATM network?," At: http://www.nwfusion.com/news/2001/0223attupdate.html.

[3] D. M. Chiu, S. Hurst, M. Kadansky and J. Wesley, “TRAM: A Tree-based Reliable Multicast Protocol”, *Sun Technical Report TR 98-66*, July 1998.

[4] D. M. Chiu, M. Kadansky, J. Provino, J. Wesley and H. Zhu, “Pruning algorithms for multicast flow control”, *Proceedings of NGC 2000 on Networked group communication*, pp. 83-92, 2000.

[5] D. M. Chiu, M. Kadansky, J. Provino, J. Wesley, H. Bischof, H. Zhu, “A congestion control algorithm for tree-based reliable multicast protocols”, *Proceedings of INFOCOMM ’02*, pp.1209-1217, 2002.

[6] S. Bhola, R. Strom, S. Bagchi, Y. Zhao and J. Auerbach, "Exactly-once Delivery in a Content-based Publish-Subscribe System". *In Proceedings of the International Conference on Dependable Systems and Networks (DSN'02)*, pp. 7-16, June 2002.

[7] T.Jiang, M.Ammar, and E.Zegura, "On the Use of Destination Set Grouping to Improve Inter-receiver Fairness for Multicast ABR Sessions", *in Proceedings of INFOCOMM'00, 2000.*

[8] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang, “A reliable multicast framework for lightweight session and application layer framing,” *IEEE/ACM Trans. Networking*, vol. 5, Volume 5, Number 6, pp. 784-803.

[9] Sanjoy Paul and John C. Lin, “RMTP: A Reliable Multicast Transport Protocol, ” *INFOCOMM 1996*, pp.1414-1424.

[10] Java Reliable Multicast Service, http://www.experimentalstuff.com/Technologies /JRMS/

[11] R. Yavatkar, J. Griffioen, and M. Sudan, “A Reliable Dissemination Protocol for Interactive Collaborative Applications,” *In Proceedings of the ACM Multimedia '95 Conference*, November 1995.

[12] M. Diaz, G. Juanole, J. P. Courtiat, “Observer-a concept for formal on-line validation of distributed systems”, *IEEE Transactions on Software Engineering*, Volume: 20 Issue: 12 , Dec. 1994, pp. 900 -913.

[13] S. Bagchi, B. Srinivasan, Z. Kalbarczyk, R. K. Iyer "Hierarchical Error Detection and Recovery in a SIFT Environment," *Fault Tolerance Computer Symposium-29*, 1999.

[14] S. Bagchi, "Hierarchical Error Detection Protocols in a Software Implemented Fault Tolerance (SIFT) Environment," *PhD Thesis* , Advisor: R. K. Iyer, Center for Reliability and High Performance Computing, Univ. of Illinois, 2000.

[15] S. Bagchi, Z. Kalbarczyk, R. K. Iyer, Y. Levendel, "Design and Evaluation of Preemptive Control Signature (PECOS) Checking for Distributed Applications," *IEEE Transactions on Computers,* 2002.

[16] H. Madeira, J. G. Silva, "Experimental Evaluation of the Fail Silence Behaviour in Computers Without Error Masking," *Proceeding of 24th International Symposium on Fault Tolerant Computing(FTCS-24)*, pp. 350-359, July 1994.

[17] E. Fuchs, " An Evaluation of the Error Detection Mechanisms in MARS Using Software Implemented Fault Injection," *European Dependable Computing Conference(EDCC)* 1996, pp. 73-90.

[18] J. Karlsson, P. Folkesson, J.Arlat and G. Leber, " Application of three Physical Fault Injection Techniques to Experimental Assessment of the MARS architecture," in *Proc. of the 5th IFIP Working Conf. on Dependable Computing for Critical Applications (DCCA-5)*, pp. 267-287, March 1996.

[19] H. Kopetz, H. Kantz, G. Gruensteidl, P. Pushner, J. Reisinger, "Tolerating Transient Faults in MARS," *Proc. 20th International Symposium on Fault Tolerant Computing*, pp. 466-473, June 1990.

[20] D.Powell, "DELTA-4: A Generic Architecture for Dependable Distributed Systems," *Springer-Verlag*, October 1991.

[21] D. Powell " Lessons Learned from Delta-4", *IEEE Micro*, vol 4, No.4, 1994, pp. 36-47.

[22] R. Chillarage, R. K. Iyer, "An experimental study of Memory Fault Latency," *IEEE Transactions on Computers*, Volume:38, Issue:6, pp. 869-874, June 1989.

[23] M. Diaz "Specification and Validation of Communication and Cooperation Protocols using Petri Nets based models," *Computer Networks* , Dec 1982.

[24] G. V. Bochmann and C. A. Sunshine, "Formal Methods in Communication Protocol Design," IEEE Transactions on Communications, vol COM-28, no. 4, pp 624-631, 1980.

[25] G. V. Bochmann, "A General Transition Model for protocols and Communication services," *IEEE Transactions on Communications*, vol COM 28, no. 4, pp. 643-650, Apr

1980.

[26] L. Lamport " A Simple Approach to Specifying Concurrent Systems," *Communications of the ACM*, January 1989, vol 32, No. 1.

[27] R.Schlor,; W. Damm, "Specification and verification of system-level hardware designs using time diagrams," *Proceedings. [4th] European Conference on Design Automation 1993, with the European Event in ASIC Design*., 22-25 Feb. 1993 Page(s): 518 -524.

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

[29] 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.

[30] G. Westerman, J.R. Heath, Stroud, C.E ., "Delay fault testability modeling with temporal logic," *IEEE Autotestcon Proceedings*, AUTOTESTCON '97. 1997, 22-25 Sept. 1997, Page(s): 376 -382.

[31] Z. Manna, A. Pnueli, "A temporal proof methodology for reactive systems," *Proceedings of the 5th Jerusalem Conference on Information Technology*, 1990. 'Next Decade in Information Technology', (Cat. No.90TH0326-9) , 22-25 Oct. 1990 Page(s): 757 -773.

[32] J. S. Ostroff, "Deciding Properties of Timed Transition Models," IEEE Transactions on Parallel and Distributed Systems, vol 1, No 2, April 1990.

[33] FIND/SVP, 1993, "Costs of Computer Downtime to American Businesses," At: www.findsvp.com.

[34] P. Loshin. M. Kaufmann, "Big Book of IP Telephony RFCs,"2001. ISBN 0-124-558550 .

[35] H. Schulzrinne, E. Wedlund, "Application-Layer Mobility using SIP," *Mobile Computing and Communications Review* (MC^{2}R), Volume 4, Number 3, July 2000.

[36] http://www.sipforum.org.

[37] Y. S. Wu, B. Foo, B. Matheny, Tyler, S. Bagchi, "ADTS: Disruption tolerance in e-commerce environments" , *to be published in ACSAC 2003*.

[38] P. Lorenz, Z. Mammeri, J.-P Thomesse, "A state-machine for temporal

qualification of time-critical communication," System Theory, 1994., Proceedings of the 26th Southeastern Symposium on , 20-22 March 1994 Page(s): 654 -658.

[39] 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 '04), pp. 15-24, March 2004.

[40] G. Khanna, P. Varadharajan, and S. Bagchi, "Self Checking Network Protocols: A Monitor Based Approach," In Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems (SRDS '04), pp. 18-30, October 2004.

[41] META Group, Inc., "Quantifying Performance Loss: IT Performance Engineering and Measurement Strategies," November 22, 2000. Available at: http://www.metagroup.com/cgi-bin/inetcgi/jsp/displayArticle.do?oid=18750.

[42] M. Zulkernine and R. E. Seviora, "A Compositional Approach to Monitoring Distributed Systems," IEEE International Conference on Dependable Systems and Networks (DSN'02), pp. 763-772, Jun 2002.

[43] M. Castro, and B. Liskov,, "Practical Byzantine fault tolerance and proactive recovery," ACM Transactions on Computer Systems, vol. 20, no.4, 398–461, Nov. 2002.

[44] M. Correia, N. F. Neves, and P. Veríssimo, "How to tolerate half less one Byzantine nodes in practical distributed systems, " In Proceedings of 23rd International Symposium of Reliable and Distributed Systems, pp. 174–183, Oct. 2004.

[45] M. Correia N. F. Neves, and P. Veríssimo, "The design of a COTS real-time distributed security kernel," In Proceedings of the Fourth European Dependable Computing Conference, pp. 234–252, Oct.2002.

[46] I. Katzela and M. Schwartz, "Schemes for Fault Identification in Communication Networks," In IEEE/ACM Trans. On Networking, vol. 3, no. 6, pp. 753-764, Dec 1995.

[47] F.P. Preparata, G. Metze, R.T. Chien. "On the Connection Assignment Problem of Diagnosable Systems"., " In IEEE Transactions on Electronic Computers, vol. 16, no. 6, pp. 848-854, Dec. 1967.

[48] D. Fussel and S. Rangarajan, "Probabilistic Diagnosis of Multiprocessor Systems with Arbitrary Connectivity," 19th Int. IEEE Symp. on Fault-Tolerant Computing, pp. 560-565, 1989.

[49] S. Lee and K. Shin, "Probabilistic diagnosis of multiprocessor systems," ACM Computing Surveys (CSUR), vol. 26, Issue 1, 1994.

[50] F. Cuppens and A. Miege, "Alert correlation in a cooperative intrusion detection

framework," Proceedings of the 2002 IEEE Symposium on Security and Privacy, May 12-15, 2002.

[51] H. Debar and A. Wespi, "Aggregation and Correlation of Intrusion Detection Alerts," Proceedings of the 4th Symposium on Recent Advances in Intrusion Detection (RAID 2001), Davis, CA, USA, Springer LNCS 2212, pages 85-103, October 2001.

[52] X. Y. Wang, D. S. Reeves and S. F. Wu, "Tracing Based Active Intrusion Response," In Journal of Information Warefare, vol. 1, no. 1, September 2001.

[53] K. R. Joshi, W. H. Sanders, M. A. Hiltunen, R. D. Schlichting, "Automatic Model-Driven Recovery in Distributed Systems," At the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05), pp. 25-38, 2005.

[54] A. T. Bouloutas, S. Calo, and A. Finkel, "Alarm correlation and fault identification in communication networks," IEEE Transactions on Communications, vol. 42, pp. 523--533, 1994.

[55] B. Gruschke, "Integrated Event Management: Event Correlation Using Dependency Graphs," at the 10th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM), pp. 130-141, 1998.

[56] S. Kliger, S. Yemini, Y. Yemini, D. Ohsie, and S. Stolfo, "A coding approach to event correlation," Intelligent Network Management, pp. 266-277, 1997.

[57] A. Frenkiel and H. Lee, "EPP: A framework for measuring the end-to-end performance of distributed applications," in Proc. Performance Engineering Best Practices Conference, 1999.

[58] I. Rish, M. Brodie, and S. Ma, "Intelligent probing: A cost-efficient approach to fault diagnosis in computer networks," IBM Systems Journal, vol. 41, no. 3, pp. 372-385, 2002.

[59] I. Rish, M. Brodie, M. Sheng, N. Odintsova, A. Beygelzimer, G. Grabarnik, and K. Hernandez, "Adaptive diagnosis in distributed systems," IEEE Transactions on Neural Networks, vol. 16, no. 5, pp. 1088-1109, 2005.

[60] R. W. Buskens and R. P. Bianchini, Jr., "Distributed on-line diagnosis in the presence of arbitrary faults," at the The Twenty-Third International Symposium on Fault-Tolerant Computing (FTCS-23), pp. 470-479, 1993.

[61] E. P. Duarte, Jr. and T. Nanya, "A hierarchical adaptive distributed system-level diagnosis algorithm," IEEE Transactions on Computers, vol. 47, no. 1, pp. 34-45, 1998.

[62] S. Arun and D. M. Blough, "Distributed diagnosis in dynamic fault environments," IEEE Transactions on Parallel and Distributed Systems, vol. 15, no. 5, pp. 453-467,

2004.

[63] N. Kandasamy, J. P. Hayes, and B. T. Murray, "Time-constrained failure diagnosis in distributed embedded systems," at the International Conference on Dependable Systems and Networks (DSN), pp. 449-458, 2002.

[64] P. Peti, R. Obermaisser, and H. Kopetz, "Out-of-norm assertions [diagnostic mechanism]," at the 11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS), pp. 280-291, 2005.

[65] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," at the Proceedings of the nineteenth ACM symposium on Operating systems principles, Bolton Landing, NY, USA, pp. 74-89, 2003.

[66] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: problem determination in large, dynamic Internet services," at the International Conference on Dependable Systems and Networks (DSN), pp. 595-604, 2002.

[67] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, "Magpie: online modelling and performance-aware systems " at the 9th Workshop on Hot Topics in Operating Systems (HotOS IX), pp. 85-90, 2003.

[68] H. J. Wang, J. Platt, Y. Chen, R. Zhang, and Y.-M. Wang, "PeerPressure for automatic troubleshooting," at the Proceedings of the joint international conference on Measurement and modeling of computer systems, New York, NY, USA, pp. 398-399, 2004.

[69] A. Bondavalli, S. Chiaradonna, D. Cotroneo, and L. Romano, "Effective fault treatment for improving the dependability of COTS and legacy-based applications," IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 4, pp. 223-237, 2004.

[70] L. Romano, A. Bondavalli, S. Chiaradonna, and D. Cotroneo, "Implementation of threshold-based diagnostic mechanisms for COTS-based applications," at the 21st IEEE Symposium on Reliable Distributed Systems, pp. 296-303, 2002.

[71] http://www.experimentalstuff.com/Technologies/JRMS/

[72] InternetWeek 4/3/2000 and "Fibre Channel: A Comprehensive Introduction," R. Kembel 2000, p.8. Based on a survey done by Contingency Planning Research.

[73] A. Brown and D. A. Patterson, "Embracing Failure: A Case for Recovery-Oriented Computing (ROC)," 2001 High Performance Transaction Processing Symposium, Asilomar, CA, October 2001.

[74] G. Khanna, P. Varadharajan, M. Cheng, and S. Bagchi, “Automated Monitor Based Diagnosis in Distributed Systems,” Purdue ECE Technical Report 05-13, August 2005. Also in submission to IEEE Trans. on Dependable and Secure Computing.

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

[76] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, July 1978, 21(7):558-565.

[77] M. Castro and B. Liskov “Proactive Recovery in a Byzantine-Fault-Tolerant System,” Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI '00), San Diego, USA, October 2000.

[78] 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.

[79] A. S. Danthine, “Protocol representation with finite state models,” IEEE Trans. on Communications, vol. 28, no. 4, pp. 632-643, Apr 1980.

[80] L. Lamport, "The temporal logic of actions," ACM Transactions on Programming Languages and Systems, 16(3):872–923, 1994.

[81] Z. Liu and M. Joseph, "Specification and Verification of Fault-Tolerance, Timing, and Scheduling," ACM Transactions on Programming Languages and Systems, 21(1):46-89, 1999.

[82] B. Berthomieu and M. Diaz, “Modeling and Verification of Time Dependent Systems using Time Petri Nets,” IEEE Trans. on Software Engineering, vol. 17 , no. 3 , pp. 259-273, Mar 1991.

[83] W. Chen, S. Toueg, and M. K. Aguilera, “On the Quality of Service of Failure Detectors,” In IEEE International Conference on Dependable Systems and Networks (DSN'00), pp. 191-201, Jun 2000.

[84] R. Baldoni, J.-M. Helary, and M. Raynal, “From Crash Fault-Tolerance to Arbitrary-Fault Tolerance: Towards a Modular Approach,” In IEEE International Conference on Dependable Systems and Networks (DSN'00), pp. 273-282, Jun 2000.

[85] S. Krishna, T. Diamond, and V. S. S. Nair, “Hierarchical Object Oriented Approach to Fault Tolerance in Distributed Systems,” In Proceedings of IEEE International Symposium on Software Reliability Engineering (ISSRE ’93), pp. 168-177, Nov 1993.

[86] W. Peng, “Deadlock Detection in Communicating Finite State Machines by Even Reachability Analysis,” IEEE Conference on Computer Communications and Networks (ICCCN), pp. 656-662, Sep 1995.

[87] A. Agarwal and J. W. Atwood, "A Unified Approach to Fault-Tolerance in Communication Protocols based on Recovery Procedures," IEEE/ACM Trans. on Networking, , vol. 4 , no. 5 , pp. 785-795, Oct 1996.

[88] L.-B. Chen and I-C. Wu, "Detection of Summative Global Predicates," IEEE Conference on Parallel and Distributed Systems (ICPADS '97), pp. 466-473, Dec 1997.

[89] M. A. Hiltunen, "Membership and System Diagnosis," In 14th IEEE Symposium on Reliable Distributed Systems (SRDS '95), pp. 208-217, Sep 1995.

[90] C. Wang and M. Schwartz, "Fault Detection with Multiple Observers," IEEE/ACM Trans. on Networking, vol. 1, no. 1, pp. 48-55, February 1993.

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

[92] R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S.K. Rajamani. "Partial-order reduction in symbolic state-space exploration," In Proc. of CAV 97: Computer-Aided Verification, LNCS 1254, pages 340--351. Springer-Verlag, 1997.

[93] K. Ravi and F. Somenzi. "High–Density Reachability Analysis," In Proc. IEEE/ACM ICCAD'95, pages 154–158, San Jose, California, November 1995.

[94] J.R. Burch, E.M. Clarke, and D.E. Long, "Symbolic model checking with partitioned transition relations," in DAC 91: Design Automation Conference, 1991, pp. 403–407.

[95] K.L. McMillan, Symbolic Model Checking: An Approach to the State-Explosion Problem. Kluwer Academic Publishers, Dordrecht, 1993.

[96] Evgeni Dimitrov, Andreas Schmietendorf, Reiner Dumke. "UML-Based Performance Engineering Possibilities and Techniques," *IEEE Software*, vol. 19, no. 1, pp. 74-83, January/February, 2002.

[97] G. Cormode and S. Muthukrishna, " What's New: Finding Significant Differences in Network Data Streams", INFOCOM 2004.

[98] I. Rish, M. Brodie, S. Ma, N. Odintsova, A. Beygelzimer, G. Grabarnik, K. Hernandez. "Adaptive Diagnosis in Distributed Systems," in IEEE Transactions on Neural Networks (special issue on Adaptive Learning Systems in Communication Networks), vol. 16, no. 5, pp. 1088-1109, September 2005.

[99] G. S. Manku and R. Motwani, " Approximate frequency counts over data streams", In Proceedings of the 28th International Conference on Very Large Data Bases,Hong Kong, China, August 2002.

[100] R. Schweller, Y. Chen, E. Parsons, A. Gupta, G. Memik, and Y. Zhang, “Reverse Hashing for Sketch-based Change Detection on High-speed Networks,” In INFOCOM 2006, pp1-12, April 2006.

[101] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, “Sketch-based Change Detection,” In ACM Internet Measurement Conference, IMC, 2003.

[102] N. Alon, Y. Matias, and M. Szegedy, “The space complexity of approximating the frequency moments,” In Proceedings of the twenty-eighth annual ACM symposium on Theory of computing, p.20-29, May 22-24, 1996.

[103] S. Bagchi, Y. Wu, Sachin Garg, N. Singh, and T. Tsai, “SCIDIVE: A Stateful and Cross Protocol Intrusion Detection Architecture for Voice-over-IP Environments,” At IEEE Dependable Systems and Networks (DSN 2004), June 28-July 1, 2004, Florence, Italy.

[104] G. Vigna, W. Robertson, V. Kher, and R.A. Kemmerer, “A Stateful Intrusion Detection System for World-Wide Web Servers,” In Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC ’03), December ‘03.

[105] G. Khanna, P. Varadharajan, and S. Bagchi, “Automated Online Monitoring of Distributed Applications through External Monitors,” In the IEEE Transactions on Dependable and Secure Computing (TDSC), vol. 3, no. 2, pp. 115-129, Apr-Jun, 2006.

[106] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: problem determination in large, dynamic Internet services," at the International Conference on Dependable Systems and Networks (DSN), pp. 595-604, 2002.

[107] PetStore J2EE Application: http://java.sun.com/blueprints/code/index.html.

[108] JBoss Application Server: http://labs.jboss.com.

[109] MySQL: Open Source Database: www.mysql.com.

[110] TPC- Benchmark: http://www.tpc.org/tpcw.

[111] M. K. Agarwal, K. Appleby, M. Gupta, G. Kar, A. Neogi, and Anca Sailer: Problem Determination Using Dependency Graphs and Run-Time Behavior Models, pp. 171-182, DSOM 2004.

[112] VMware: www.vmware.com

[113] Power Outage Darkens West: http://www.zetatalk.com/theword/tword07i.htm

[114] S. Bagchi, Y. Wu, Sachin Garg, N. Singh, and T. Tsai, “SCIDIVE: A Stateful and Cross Protocol Intrusion Detection Architecture for Voice-over-IP Environments,” At

IEEE Dependable Systems and Networks (DSN 2004), June 28-July 1, 2004, Florence, Italy.

[115] G. Vigna, W. Robertson, V. Kher, and R.A. Kemmerer, “A Stateful Intrusion Detection System for World-Wide Web Servers,” In Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC ’03), December ‘03.

[116] D. Geels, G. Altekar, S. Shenker, and I. Stoica, "Replay Debugging for Distributed Applications," USENIX Annual Technical Conference, pp. 289-300, 2006.

[117] X. Zhang, R. Gupta, and Y. Zhang, "Precise dynamic slicing algorithms," ICSE, pp. 319-329, 2003.

[118] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging operating systems with time-traveling virtual machines," USENIX Annual Technical Conference, pp. 1-15, 2005.

[119] S. Narayanasamy, G. Pokam, and B. Calder, "BugNet: Continuously recording program execution for deterministic replay debugging," ISCA, pp. 284-295, 2005.

[120] W. Chen, S. Toueg, and M. K. Aguilera, “On the Quality of Service of Failure Detectors,” In IEEE International Conference on Dependable Systems and Networks (DSN'00), pp. 191-201, Jun 2000.

[121] R. Baldoni, J.-M. Helary, and M. Raynal, “From Crash Fault-Tolerance to Arbitrary-Fault Tolerance: Towards a Modular Approach,” In IEEE International Conference on Dependable Systems and Networks (DSN'00), pp. 273-282, Jun 2000.

[122] S. Krishna, T. Diamond, and V. S. S. Nair, “Hierarchical Object Oriented Approach to Fault Tolerance in Distributed Systems,” In Proceedings of IEEE International Symposium on Software Reliability Engineering (ISSRE ’93), pp. 168-177, Nov 1993.

[123] B. Berthomieu and M. Diaz, “Modeling and Verification of Time Dependent Systems using Time Petri Nets,” IEEE Trans. on Software Engineering, vol. 17 , no. 3 , pp. 259-273, Mar 1991.

[124] W. Peng, “Deadlock Detection in Communicating Finite State Machines by Even Reachability Analysis,” IEEE Conference on Computer Communications and Networks (ICCCN), pp. 656-662, Sep 1995.

[125] L. B. Chen and I-C. Wu, “Detection of Summative Global Predicates,” IEEE Conference on Parallel and Distributed Systems (ICPADS '97), pp. 466-473, Dec 1997.

[126] Software Testing: www.cs.purdue.edu/homes/apm/slides/foundations-book-slides/draftV1.0-slides

[127] N. Kandasamy, J. P. Hayes, and B. T. Murray, "Time-constrained failure diagnosis in distributed embedded systems," at the International Conference on Dependable Systems and Networks (DSN) pp. 449-458, 2002.

[128] S. Cabuk, C. E. Brodley, and C. Shields, “IP covert timing channels: design and detection,” In Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS), pp. 178-187, 2004.

[129] S. H. Sellke, C-C. Wang, N. B. Shroff, and S. Bagchi, “Capacity Bounds on Timing Channels with Bounded Service Times,” Accepted to appear in the IEEE International Symposium on Information Theory (ISIT), Nice, France, June 24-29, 2007.

APPENDIX

APPENDIX

A. Abbreviations

CG	Causal Graph	PPEP	Path Probability of Error Propagation
AG	Aggregate Graph	PE	Protocol Entity
IM	Intermediate Monitor	LM	Local Monitor
GM	Global Monitor	TL	Temporary Links
DT	Diagnosis Tree	SRB	Strict Rule Base (Diagnosis)
LC	Logical Clock	NRB	Normal Rule Base (Detection)
SS	Suspicion Set	TTCB	Trusted Timely Computing Base
EMC	Error Masking Capability	TRAM-D, TRAM-L	Tram Distributed and TRAM Local.

B. Rule Base

Here we provide some sample rules used in our experiments with TRAM.

a) Normal Rule Base (NRB)

1. *T R4 S3 E11 30 500 5000 S3 E2 1 8 4000 7000*: This rule of type 4 has a precondition to check data packets (*E11*) arrival within 5000msec. It verifies in the pre-condition that between 30-300 data packets are received within 5000ms. Pre-condition on being satisfied checks the post condition that at least one ack(*E2*) (between 1 and 8) must be sent. The timing constraints, e.g., 5000, 4000, and 7000 are derived from the ack and data rate as specified by the user.

2. *T R1 S0 E11 10 1000 S8 3000 3500:* This rule of type 1 is ensuring if the state

transitions are taking place properly. It says that if the entity is currently in state S0 and receives data packet (E11) then it should move to state S8 in the next 3000-3500 ms. The state S8 corresponds to the ack sending state.

3. *T R3 S6 E1 10 30 5000*: This rule of type 3 checks for the *hello* packet(*E1*) rate. The hello (E1) message count should be between 10 and 30 for the next 5000 msec which is stated by the rule through the constants “10”, “30”, and “5000”.

4. *T R3 S7 E13 0 2 5000*: This rule ensures that the number of *re-affiliation* packets (E13) are no more than 2 within 5000ms in state S7. This prevents a receiver from causing frequent joins and leaves.

5. *T R2 S0 E10 50:* This rule verifies that state of the receiver changes should change from S0 to some other state once the message E10 is received. The time limit on this change should be within 50 ms. The message E10 corresponds to a Head Advertisement Message.

6. *T R4 S0 E10 1 4 1000 S1 E9 1 2 2000 3000*: This is a rule of type 4 as indicated by “R4”. This has a pre-condition which states that in state S0, if the number of messages E10 received is between 1-4 in 1000 ms, then the post condition verifies the number of head bind (E9) sent out between 2000-3000ms. This rule basically tries to ensure that Head Advertisement messages should be followed by Head Bind

7. *T R4 S1 E9 1 2 1000 S1 E8 1 2 2000 3000*: This is again another rule of type 4. The precondition verifies the number of head bind messages (between 1-2) within next 1000ms in state S1. The post condition verifies if the number of Accept (E8) messages are atleast 1 within the next 2000-3000ms. This rule is making sure that the causal relationship between head bind and accept messages is followed by the protocol.

8. *T R3 S1 E9 1 10 5000*: This rule is simply verifiying that in state S1 the number of E9 messages i.e. head bind messages should be between 1-10 in 5000ms. This ensures a cap on the frequency of messages.

9. *T R3 S2 E8 1 2 10000*: This rule is trying to verify the rate of messages E8 in the state S2. It ensures that the number of E8 messages i.e. accept message should be between 1-2 in 10,000ms. This ensures that the repair head is not accepting a lot of new re-affiliation requests.

b) Strict Rule Base (SRB)

There are 4 different types of strict rules names ‘O’, ‘I’, ‘HI’, and ‘HO’. ‘O’ & ‘I’ represents rules which checks the outgoing and incoming links from a node respectively. ‘HI’ & ‘HO’ check combination of links in both outgoing and incoming links. The weight assigned to each rule is given by the number in the last column of the rule specification.

O S1 E11 1 S3 E11 30 1: If in state *S1*, receiver has received a data packet (*E11*), then in the state *S3*, a data packet (*E11*) must have been detected at least 30 times.

O S3 E3 1 S3 E11 0 1: If in state *S3*, receiver has received beacon packet (*E3*) at least once, then in the state *S3*, a data packet (*E11*) must not have been detected. This is because the data transmission has not started yet.

O S1 E3 1 S1 E1 1 1: In state *S1*, receiver has received a beacon packet (*E3*) at least once, then in the state *S1*, a Hello packet (*E1*) must be been detected.

O S6 E2 1 S33 E2 10 1: If the receiver has sent an ack packet (*E2*) then there should be at least 10 more acks (*E2*) sent out within the phase. This maintains a minimum ack-rate from the receivers.

I S1 E2 1 S3 E11 30 1: If in state *S1*, receiver has received an ack packet (*E2*) at least once, then in the state *S3*, a data packet (*E11*) must have been detected at least 30 times. This maintains a minimum data rate.

HI S6 E1 20 S6 E9 1 1 : *T*he rule refers to the fact that if in state *S6*, receiver has received a Hello packet (*E1*) at least once then in the state *S6*, a *Hello-Reply* packet (*E9*) must have been detected at least once.

HO S1 E11 1 S2 E11 1 1: This rule verifies that if Repair Head (RH) gets a data packet (*E11*), then it should send a data packet to the receivers.

HI S0 E15 1 S1 E14 1 1: If a receiver sends a Head-Bind (*E15*) then it must receive multiple Head-Ack (*E14*) in state S1.

c) Messages in TRAM

Message Name	**(Source, Destination)**	**Interpretation**	**Event ID**
Head Adv.	Sender(RH), Receivers	Repair Heads send advertisement of the channel	E10
Data	Sender(RH), Receivers(RH)	Multicast Data sent from head to group members	E11
Head Bind	Receiver, Repair Head(Sender)	Receiver sends a request to join group in the form of Head Bind	E9
Accept/Reject	Repair Head(Sender), Receiver(RH)	Acceptance or Rejection message sent by the repair head to the seeking receiver.	E7, E8
Ack Packet (Nack Packet)	Receiver, Repair Head(Sender)	Aggregate Acknowledgement sent by the receiver to the repair head.	E2 (E15)
Member Solicitation	Receiver, RH(Sender)	Message sent by a receiver seeking to join a group when group formation is started by receiver.	E4
Hello Messages (Reply)	RH(Receiver), Receiver(RH)	Indication of *Liveliness* of the members.	E1, E14
Re-affiliation	Sender(RH), Receivers(RH)	Join a new Repair Head; sent by the receiver	E13

VITA

VITA

Gunjan Khanna received his B.Tech degree in Electrical and Computer Engineering from Indian Institute of Technology, Delhi in August 2002. He obtained his Masters degree from Purdue University in December 2003. He started working on his PhD in January 2003 with Professor Saurabh Bagchi in Electrical and Computer Engineering. He worked as a research assistant in Dependable Computing Systems Lab under Professor Saurabh Bagchi. His research interests include Fault Tolerance, Dependable Computing, Sensor Networks, System Management and Virtual Machine Management. He spent several summers at IBM T. J. Watson Research Center working with Systems Management Group under Dr. Gautam Kar.

LIST OF PUBLICATIONS

LIST OF PUBLICATIONS

Journal

[1] G. Khanna, P. Varadharajan, Y. Cheng, S. Bagchi, M. Correia, and P. Verissimo, "Automated Monitor Based Diagnosis in Distributed Systems," accepted in IEEE Transactions on Dependable and Secure Computing (TDSC).

[2] G. Khanna**,** P. Varadarajan, and S. Bagchi, "Automated Online Monitoring of Distributed Applications Through External Monitors," in IEEE Transactions on Dependable and Secure Computing (TDSC), 2006.

Conference and Workshops

[3] G. Khanna, I. Laguna, F. Arshad, and S. Bagchi, "Stateful Detection in High Throughput Detection Systems," in submission to Symposium of Reliable and Distributed Systems, 2007.

[4] G. Khanna, I. Laguna, F. Arshad, and S. Bagchi, "Distributed Diagnosis of Failures in a Three Tier E-commerce System" in submission to Symposium of Reliable and Distributed Systems, 2007.

[5] G. Khanna, Y. Cheng, S. Bagchi, "State Space Reduction for efficient Detection and Diagnosis in Distributed Systems," in *submission*.

[6] R. Khosla, X. Zhong, G. Khanna, S. Bagchi, and E. J. Coyle, "Performance comparison of SPIN based Push-Pull Protocols" in Wireless Communications and Networking Conference (WCNC), 2007.

[7] R. Khosla, X. Zhong, G. Khanna, S. Bagchi, and E. J. Coyle, "Data Centric Routing in Sensor Networks: Single-hop broadcast or Multi-hop unicast?," in Vehicular Technology Conference (VTC), 2007.

[8] G. Khanna, K. Beaty, A. Kochut, and G. Kar, "Dynamic Application Management to address SLAs in a Virtualized Server Environment," in Network Operations and Management (NOMS), 2006.

[9] G. Khanna, S. Bagchi, K. Beaty, A. Kochut, N. Bobroff, and G. Kar, "Providing Automated Detection of Problems in Virtualized Servers using Monitor Framework," in Workshop on Applier Software Reliability (WASR) held in conjunction with DSN, 2006.

[10] G. Khanna, A. Masood, and C. N. Rotaru, "Synchronization Attacks Against 802.11," in Network and Distributed System Security Symposium (NDSS) Workshop, Feb 2-4, San Diego, 2005.

[11] G. Khanna, Y. Cheng, and S. Bagchi, “Modeling Probabilistic Diagnosis Parameters,” Fast Abstract in Dependable Systems and Networks (DSN), 2006.

[12] 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 ’04), pp. 15-24, March 2004.

[13] G. Khanna, S. Bagchi, and Y. S. Wu, “Fault Tolerant Energy Aware Data Dissemination Protocol in Sensor Networks,” In Dependable Systems and Networks (DSN), pp. 795-804, Florence, Italy, 2004.

[14] G. Khanna, P. Varadharajan, and S. Bagchi, “Self Checking Network Protocols: A Monitor Based Approach,” In Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems (SRDS ’04), pp. 18-30, October 2004.

[15] G. Khanna, S. Bagchi, and Y. S. Wu, “Data Dissemination Protocol to account for Node and Link Failures in Sensor Networks,” Fast Abstract Dependable Systems and Networks (DSN), 2003.