Automated Online Monitoring of Distributed Applications through External Monitors

Gunjan Khanna, Padma Varadharajan, and Saurabh Bagchi, *Member, IEEE*
School of Electrical and Computer Engineering, Purdue University
Email: *{gkhanna, pvaradha, sbagchi}@purdue.edu*

Abstract— It is a challenge to provide detection facilities for large scale distributed systems running legacy code on hosts that may not allow fault tolerant functions to execute on them. It is tempting to structure the detection in an observer system that is kept separate from the observed system of protocol entities, with the former only having access to the latter's external message exchanges. In this paper we propose an autonomous self checking *Monitor* system, which is used to provide fast detection to underlying network protocols. The Monitor architecture is application neutral and therefore lends itself to deployment for different protocols, with the rulebase against which the observed interactions are matched making it specific to a protocol. To make the detection infrastructure scalable and dependable, we extend it to a hierarchical Monitor structure. The Monitor structure is made dynamic and reconfigurable by designing different interactions to cope with failures, load changes, or mobility. The latency of the Monitor system is evaluated under fault free conditions, while its coverage is evaluated under simulated error injections.

Index Terms—Error detection, Blackbox detection, Monitor system, Temporal and combinatorial rules, Reliable multicast.

1. INTRODUCTION

Increased deployment of high-speed computer networks has made distributed systems ubiquitous in today's connected world forming the backbone of much of the information technology infrastructure of the world today. 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, space flight control, banking, air traffic control, and railways signaling could be catastrophic. It is therefore imperative to build low latency detection systems that can subsequently trigger the diagnosis and recovery phases leading to dependable systems that are robust to failures.

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. A common requirement is for the detection system to be non-intrusive to the payload system 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 generalizability. Trying to make changes to a particular protocol also requires in-depth understanding of the code which may be unavailable.

In this paper we present a generic Monitor architecture for detection 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) that has been executed by the PEs, which is necessarily less complete than the actual STD for the PE. 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 to meet QoS requirements. The system provides a rich syntax for the rule specifications, categorized into combinatorial and temporal rules (valid only for specific times in the protocol operation) and optimized matching algorithms for each class of rules.

The Monitor is designed using a set of objectives as guidelines. First, the Monitor should not become a performance bottleneck for the payload system. This is achieved by making the Monitor operation asynchronous to the payload system's operation and removing any requirement for co-locating the Monitors with the PEs. Second, the Monitor should be scalable to a system with thousands of verified PEs. The design of the Monitor components, the hierarchical Monitor structure, and the fast rule matching algorithms help in the scalability goal. 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 of failures in different applications. Finally, the Monitor should have a low latency of detection. This is critical since the Monitor functions asynchronously to the application and a high latency will make any subsequent diagnosis, containment, or recovery complicated. This is achieved by the same design features that support scalability.

In order to make the Monitor infrastructure scalable, accurate, and efficient, we propose a hierarchical structure, where the Local Monitors (LMs) directly gather the information from the PEs and send processed and filtered messages to the higher level Monitors. Higher level Monitors detect errors through correlating protocol behavior that spans multiple LMs. Since a scalable distributed application should have more local interactions than remote or global ones (e.g., closely spaced streaming media consumers and servers interacting more frequently), it is expected that most messages will be filtered at the LMs. Each Monitor is fed the entire rulebase for the payload system and automatic rule segmentation is performed to classify the rulebase into applicable and non-applicable rules at each Monitor. Automatic rule classification eases the task of deploying the Monitor hierarchy. The Monitor hierarchy can be reconfigured to deal with dynamism such as failures, joins, and leaves of PEs as well as Monitors, always satisfying the property that each PE is verified by at least one Monitor.

The Monitor system is applied to a streaming video application running on a reliable multicast protocol called TRAM installed on the Purdue campus-wide network. We perform scalability tests on the Monitor to evaluate the latency of rule matching under varying load. We evaluate the detection coverage of the Monitor by running it with three different kinds of message errors injected into the application for the single level and hierarchical Monitor deployments.

The rest of the paper is organized as follows. Section 2 presents the design of the single level Monitor. Section 3 introduces the hierarchical Monitor design. Section 4 provides details about the workload

– a distance learning application, and the configuration of the Monitor system for the particular workload. Section 4 also gives the experiments and the results. This is divided into Section 4.4 on performance results in the error free case and Section 4.5 on coverage results with error injection. Section 5 surveys related work. Section 6 concludes the paper with mention of future work.

2. SINGLE LEVEL MONITOR DESIGN

The Monitor is a generic entity that snoops on the communication between the protocol participants for which the Monitor is responsible, and matches the observed communication against a given rulebase which characterizes acceptable protocol behavior. Since the application is considered to be a black box, there is no access to the execution hosts or to the source code of the protocol entities. The specification of the protocol is, however, available to the Monitor. The Monitor has access to the messages exchanged between the PEs, and hence is able to examine the communication header and payload. The detection infrastructure is independent of the underlying protocol being checked, while the rulebase against which the behavior is matched, is not.

The Monitor approach employs a stateful model for rule matching by preserving 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 partial order reduction [20] or symbolic state space reduction [19]. Partial order methods exploit interleaving of concurrent events while state space exploration techniques calculate the possible reachable states. Authors in [18] provide a hybrid approach using the underlying benefits of both approaches. The Monitor would typically run in the network vicinity of the PEs it is verifying and should be able to observe the messages in and out of the PEs. In this work we assume Monitor has perfect observability of the monitored system, which has been relaxed in our more recent work [39].

2.1 Fault Model

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 data 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 rulebase. 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, in such a case, 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.

2.2 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 1 gives a pictorial representation of the Monitor components.

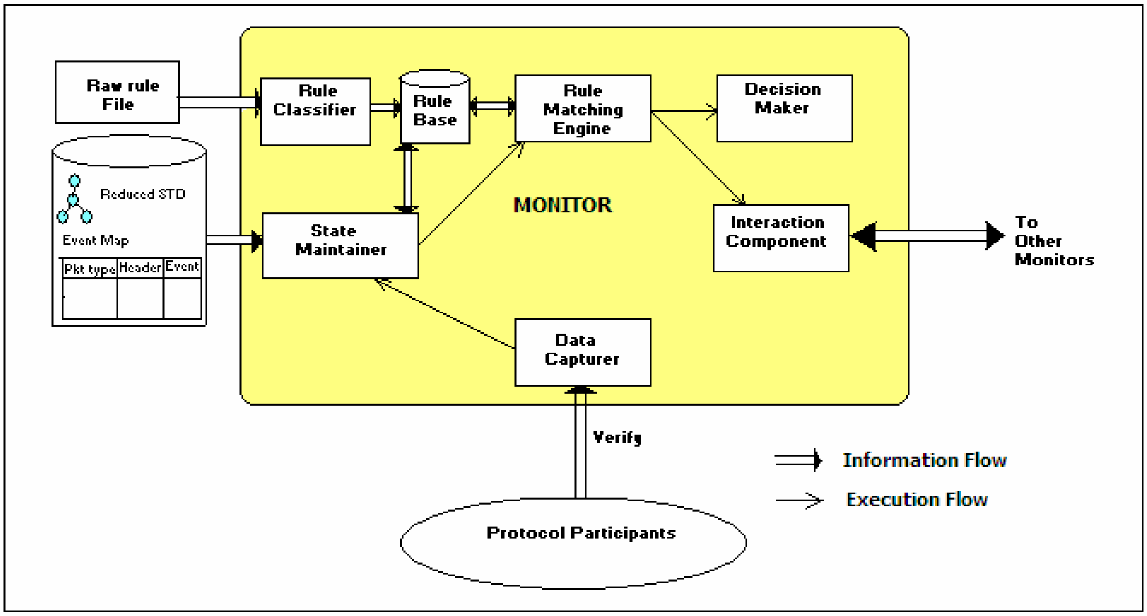


Figure 1: Monitor architecture with process and information flow among multiple components

The details of the structural and functional roles of the components have been described in [17] 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 them 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 (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. 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. Unlike temporal rules which are valid for only some time interval from the occurrence of an event, combinatorial rules need to hold all through the system lifetime. Therefore the testing for temporal rules only occurs over a time window from the occurrence of an event while the testing for combinatorial rules occurs at all times, of course with a certain periodicity. 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.

2.3 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.

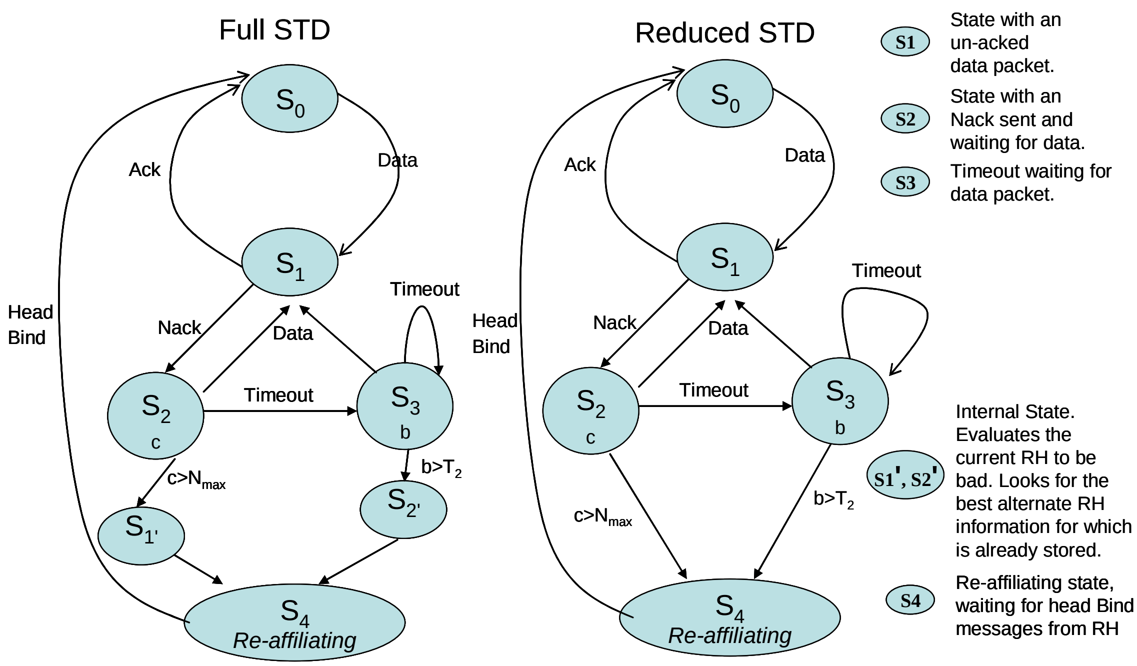


Figure 2: State Transition Diagram for a receiver in TRAM protocol

The running protocol we use as example is the TRAM protocol for reliable multicast of data from a single sender to multiple receivers through intermediate routing nodes called repair heads (RHs). 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, it joins a different RH assuming the old RH is not functioning any more. The STD in Figure 2 is for a receiver receiving data from the sender or an RH – the reduced STD is derived by removing internal states from the full STD. 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, 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 $\forall t \in (t_i, t_i+a)$, $S_2 \wedge \neg P \Rightarrow S_3$ (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 rulebase with additional 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 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.

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. Examples of each rule type for the running application TRAM is provided in Section 4.2 and an exhaustive set of rules in the Appendix (Section 7).

1. Type I (R1): (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 event or state variable.

2. Type II (R2): $S_t \neq S_{t+\Delta}$, where S_t is the state of an object at time t and event E_k takes place at t

This rule implies the state S_t will not remain constant for Δ time units from t, the occurrence of an event E_k.

3. Type III (R3): $L \leq |V(t)| \leq U$, $t\in(t_i,t_i+k)$, t_i is the time of event E_k and E_k occurs in state S_k

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

4. Type IV (R4): $L \leq |V(t)| \leq U$, $t\in(t_i,t_i+k)$, t_i is the time of event $E_k \Rightarrow L' \leq |B(q)| \leq U'$, $q\in(t_n,t_n+b)$, $t_n > t_i$ and k, $b \geq 0$.

If a state variable V in a particular state S_k has a bounded count from above and below over a time window k, it will cause another state variable B to be bounded for a time window b starting from t_n.

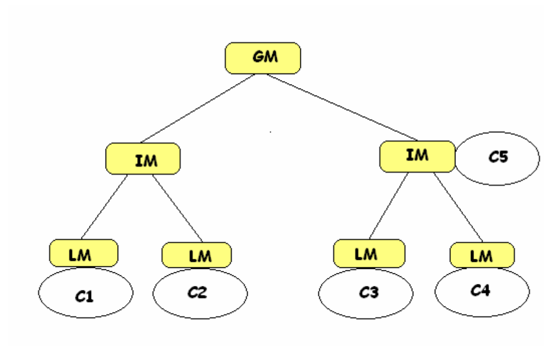
Two important considerations underlying the capability of the system are how expressive the rule language is and how feasible is it to specify rules in this language. The first question can be answered by the fact that our rule syntax is derived from Temporal Logic Actions (TLA) and therefore preserves its expressivity. However, we see that deciding on a completely satisfactory rule syntax will be an ongoing process. Regarding the second consideration, there are two kinds of rules that can be automatically derived from a timed state automaton – those corresponding to unexpected events in a particular state and those which violate the timing properties represented in the automaton. We have developed tools for reducing a given timed automaton for a particular application to one containing only states observable through external messages and relevant to at least one rule in the rulebase. The two other kinds of rules – QoS specifications to the application and vulnerability checking in the application – have to be specific manually, as it is in current systems (e.g., a network IDS).

3. Hierarchical Monitor Design

3.1 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.



C: Clusters; LM: Local Monitor; IM: Intermediate Monitor; GM: Global Monitor

Figure 3: Example topology of local, intermediate, and global monitors

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 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.

3.2 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.

3.2.1 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. This is enabled by having the Monitor resident on the same broadcast medium as the PEs (such as, a LAN) or through router support (e.g., by enabling port mirroring on the routers). 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 variables 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.
   (i) 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.
   (ii) *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, a rule of type 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.

(iii) *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 $L_1 < |E_1| < U_1$, $L_2 < |E_2| < U_2$, …, $LG < |E_G| < UG$, where E_1 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.

3.2.2 Automatic Rule Segmentation

The automatic rule segmentation algorithm classifies the rules into one of the three categories according to the rules defined below. This algorithm runs offline, checking the state variables and events specified in each event. All states and events corresponding to entities verified by a Monitor are called local variables.

1. *Local Rule*. A rule that contains solely local variables is classified as a Local Rule.
2. *Global* Rule. A global rule is one that is based on local and non-local states. Within a global rule, the algorithm for classifying a rule puts it into one of two possible buckets – PNF and PFF. The PNF rules are of two kinds: (i) A rule of type IV ($\forall t \in (t_i, t_i + k)\ L \leq |V_t| \leq U \Rightarrow L' \leq |B_q| \leq U'$ $\forall q \in (t_n, t_n + b)$), where the antecedent clause has local state or event and the consequent clause has a global state or event; (ii) a rule of Disjunctive Rule III type. The PFF rule consists of one type - a rule of Conjunctive Rule III type. Any other rule which has global and local variables is classified as Forward only rule.

3. *Bypass Rule*. A rule with *all* its states and events non-local to the Monitor is classified as bypass.

It is acknowledged that an incorrect rule segmentation can cause missed alarms (local rule put in bypass bucket), false alarms (a stricter global rule put in the local bucket), or performance degradation (a bypass rule put in the local or global bucket).

3.3 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 protocol runs for an extended 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 during the discussion of rule classification (Section 3.2), 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:** We borrow the heartbeat mechanism to use in our system whereby the liveness of the child Monitor is verified by parent Monitor on receipt of a packet from it. A heartbeat is only sent to the child Monitor if no packets are received for the last θ seconds. The parent Monitor must receive the *HeartBeatReply* within Φ seconds of sending the request. The time θ is determined by the rule specification and the round trip time RTT, while the time Φ is determined by 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 $T_i = (1/R_i + D_i + c)$, where *c* incorporates

rule matching latency and network delays. The median T_i over all the rules at the child Monitor is denoted by T_{med}. Consider θ' as $\max\{\Phi, T_{med}\}$. We bound it by θ_{ub} which is the maximum value θ can take, for situations where T_{med} is unbounded. Hence, θ is $\min\{\theta', \theta_{ub}\}$.

2. **Load Balancing :** Figure 5 in Section 4.4.2 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 $\eta/2$. So the desirable range of operation, called the *Target Load Range* (TLR), is between $\eta/2$ 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. Each Monitor maintains a sliding window based measure of the rule matching rate. In case it exceeds η or falls below $\eta/2$, the Monitor sends an *Overload* or an *Underload* message to its parent Monitor. If a Monitor is operating at a load λ and $\lambda < \eta/2$, then it is an under-loaded Monitor which can increase its load by $f(\eta/2-\lambda)$, which is called the *residual capacity*, where $1 \leq f \leq (\eta-\lambda)/(\eta/2-\lambda)$. The upper bound represents the maximum load before the latency of matching exceeds the neck. Similarly if $\lambda > \eta$, the Monitor needs to reduce its capacity by $f(\lambda-\eta)$, called the *overload capacity*, where $1 \leq f \leq (\lambda-\eta/2)/(\lambda-\eta)$.

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* message contains the list of nodes which the Monitor feels should be removed from its domain. Each *Underload* message contains the additional number of rule matching 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 message, 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* message 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 that only a minimum number of entities is 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 messages.

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. In the Self-Evolving case, NM broadcasts its call for a parent Monitor through a beacon message with increasing values of TTL(Time To Live). 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. All the IMs that have the need for a new Moni-

tor, due to one or more overloaded child Monitors, send a Need message to NM indicating the number of PEs to be transferred. NM collects the Need messages 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 message, JM 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* message with the IP address of the PEs to be transferred to Monitor *B*. Monitor *B* snoops over the LAN to ensure that it can actually receive the messages for the PEs which are being transferred. Monitor B sends an *acceptTransfer* message 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. We address failures in the Monitors in recent work in [38].

3.4 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:

I. *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.

II. *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.

III. *Monitor Crash:* Monitor crash is detected by three successive failures to receive *HeartBeatReply* messages. 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 to when the parent Monitor should get a reply, and $\theta \leq \theta_{ub}$. Therefore, $T \leq 3(\theta_{ub} + \Phi)$. The minimum value that θ can take is Φ, as θ' is max(Φ, 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 I. Hence the property holds during Monitor crash.

4. WORKLOAD

4.1 TRAM

We demonstrate the use of the Monitor on the running example protocol — a reliable multicast protocol called TRAM [11][12]. 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 packets. 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 packets has been received, or an ack interval timer goes off. The RHs aggregate acks from all its members and send an aggregate ack up to the higher level to avoid the problem of flooding the sender with a large number of acks with increasing number of receivers. .

4.2 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 rulebase and can only depend on externally visible message exchanges. We do not address the issue of how the reduced STD is created from the protocol specification. For this, one of several existing approaches may be used [35],[36], though currently it is manually generated. The rulebase 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 2.3.1. The total number of rules in the rulebase for the experiments is 11 with 8 rules matched at the Local Monitor and 3 matched at the Global Monitor. Forming the reduced STD is somewhat of an art that the administrator deploying the Monitor needs to master.

A representative set of rules from the rulebase is given below. For each rule, the formal syntax, the explanation, and the objective of the rule – whether derived from a correctness specification or a QoS specification – are provided.

1) T R1 S4 0 0 !S4 0 3000

   a. **Explanation of Rule:** A receiver in its re-affiliation state (S4) should not come back to the re-affiliation state for at least 3 seconds.

   b. **Objective:** This is to protect the system from a faulty receiver that disconnects and re-affiliates with multiple RHs in quick succession, thereby wasting system resources in terms of bandwidth and processing.

2) T R2 S0 E11 50

a. **Explanation of Rule:** The state of a PE should not be the initial state (S0) 50ms after it receives a data packet (E11) as it should have moved into the ACK-ing state (S1).

b. **Objective:** This rule specifies correctness of the state transitions done by the PEs. For correct operation, the PE should move to the state where it can send ACK packets on receipt of data packets.

3) T R3 S0 E3 30 150 5000

a. **Explanation of Rule:** Total number of head advertisement messages (E3) in the initial state S0 should be between 30 and 150 in 5 seconds.

b. **Objective:** Head Advertisement messages are sent out by affiliated repair heads at periodic intervals of time. There is an associated QoS requirement that the bandwidth for these messages should consume only a small portion of the total data bandwidth on a global scale (12Kbps in the experiments).

4) T R3 S1 E11 30 500 5000

a. **Explanation of Rule:** Total number of data packets (E11) in the data send/receive state S1 should not exceed 500 in 5 seconds.

b. **Objective:** This rule is derived from the QoS requirement in the experimental environment that the data rate should be below 40 kbps.

5) T R4 S1 E11 30 500 5000 S2 E9 1 8 500 7000

a. **Explanation of Rule:** If there are between 30 and 500 data packets (E11) in the data send/receive state (S1) in 5 seconds, which is the necessary range of data rate as per the previous rule, there should be between 1 to 8 ACK packets (E9) sent from the same state (S1) in the time range 0.5 to 7 seconds following the first data packet (E11).

b. **Objective:** This rule regulates the flow of ACK packets in accordance with the flow of data packets in the TRAM system. That is, given an optimal data rate, the rule specifies an optimal rate for ACK packets.

6) Combinatorial rules, as opposed to temporal rules, are required to hold true for the entire session of the protocol, with exceptions being made only for transients. An example of a combinatorial rule dictated from the QoS requirement specified earlier is that the data rate at any receiver should always be between 20 Kbps and 40 Kbps.

4.3 Experimental Fault Model

The experiments conducted for the Monitor system fall into two categories – for measuring performance and for measuring coverage. Faults are injected for the coverage experiments. The faults are injected into the header of the packet, and the accuracy of the Monitor setup in detecting the resulting failures is studied. It must be noted that the faults are considered to be natural faults, which may be of arbitrary nature. Malicious nodes launching deliberate attacks on the system are not used in the experiments. The Monitor inspects only the header of the packet as it is protocol-independent and is not aware of the contents of the payload. Hence faults are only injected into the packet header. The fault is *emulated* by changing bits in the header after the PE has sent the message. The emulated faults may be symptomatic of a broader class of faults including 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 errors in the protocol itself which are clearly outside the purview of checksum.

4.4 Performance Results

4.4.1 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 from when the packet comes into the Monitor to 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.

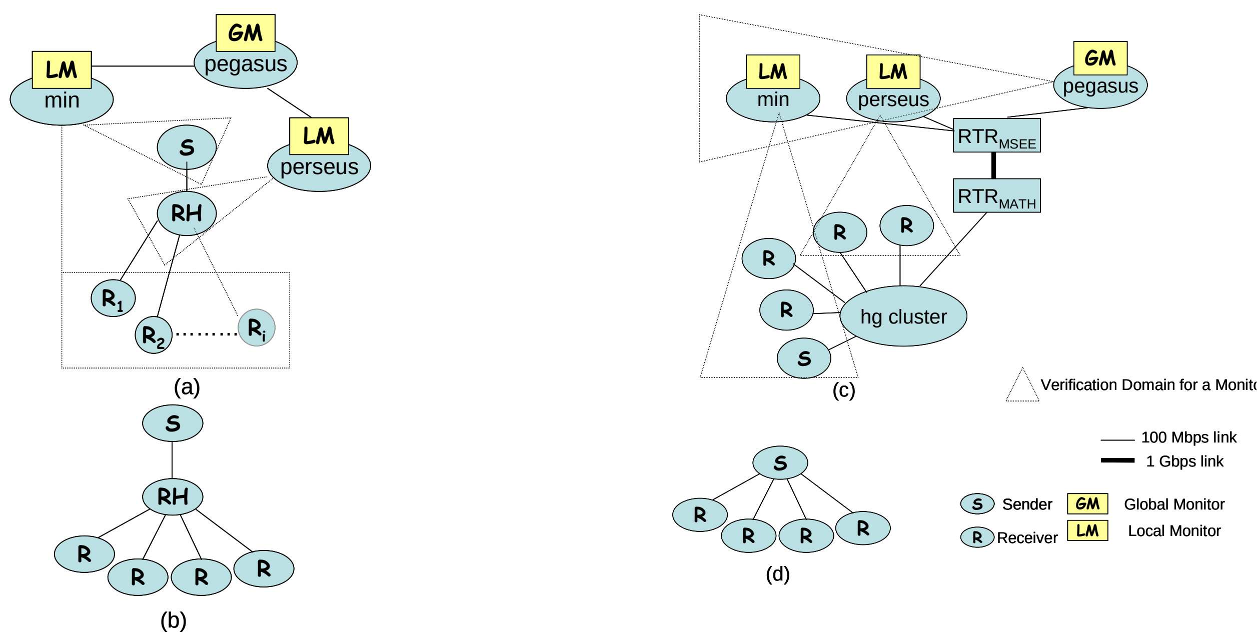
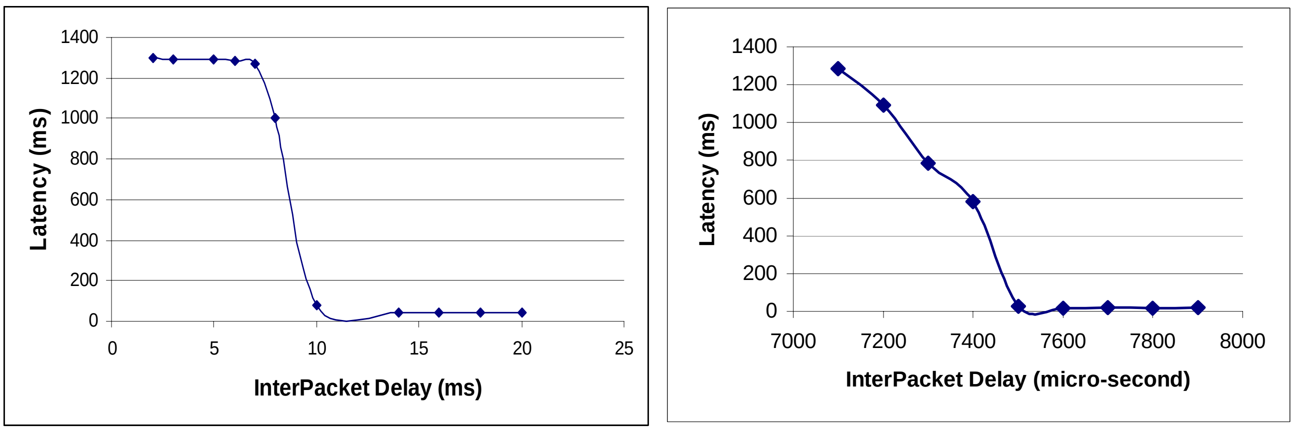


Figure 4: Physical Configuration of the testbed and logical setup of TRAM for respectively the Packet Generator experiments [(a),(b)] and the coverage experiments [(c),(d)]

The scalability experiments would ideally be conducted in the TRAM configuration on the campus-wide network as shown in Figure 4(c). 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 [10]. Hence, a simple *Packet Generator* module is used, which emulates the receiver in TRAM (Figure 4(a),(b)). 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.

4.4.2 Experimental Results

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



(a) (b)

Figure 5: (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 5 that when the inter-packet delay is small the latency of matching is high because the packets are coming in at higher data rate. As the inter-packet 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 inter-packet 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 inter-packet 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). The latency stabilizes at a constant value for inter packet delays lesser than 7 ms. This is because the Monitor starts dropping packets because of the high incoming data rate, thereby causing the number of rule matching to stabilize around this region. To better characterize the rapid change between 7 ms and 8 ms, we zoom in to the data in the range of interest and show it in Figure 5(b) where the increments are in multiples of μs. 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.

4.4.3 Latency with Varying Rule Base

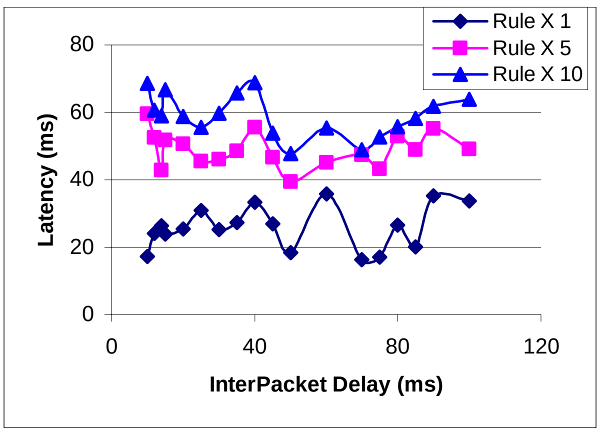


Figure 6: Variation of Latency with Rulebase Size

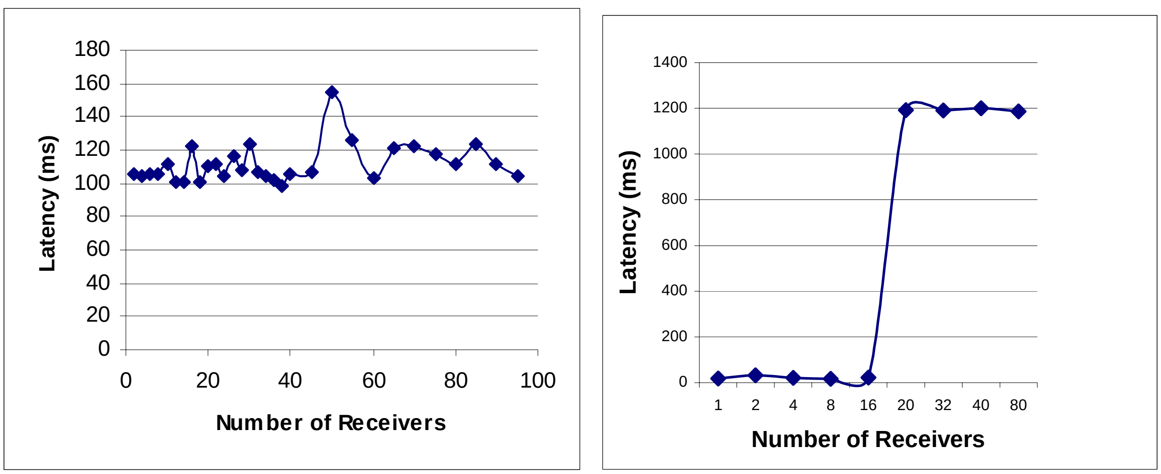
We measure the effect of size of the rulebase on the latency of matching. We vary the rulebase size by replicating the identical rulebase, once, 5 times, and 10 times for the experimental results shown here. Figure 6 shows the latency curves for different rulebase sizes.

We see that latency curves for each rulebase size follows 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 inter-packet delay. This can be explained by the fact that for a certain value of the inter-packet delay, say D, multiple rule matching 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.

4.4.4 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 7: (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 7(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. The higher is 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 an R2-Rule. Since the sum of the concurrently active rules of A and R2 type is approximately constant, we see a flat latency curve.

In the second test (Figure 7(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. The latency stabilizes for very high number of receivers following the neck owing to the dropping of packets at the Monitor, thereby decreasing the number of required rule matching and also reducing coverage. A load rebalancing interaction as described in Section 3.3 can be invoked if the load on a single Monitor increases beyond the cutoff point. We perform tests on sensitivity of Monitor with the thread pool 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.

4.5 Coverage Results

4.5.1 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. Coverage is defined as the Monitor correctly detecting an error *before* there is a manifested failure of the protocol, including the client detecting it and raising an exception. The minimum and the maximum date rate specified by the client to TRAM are 20 kbps and 40 kbps. 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 [17]. The configuration used for our experiments is depicted in Figure 4(c),(d).

4.5.2 Error Injection

Errors are injected into the protocol to cause invalid state transitions which should be detected by the Monitor. The errors 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.

(1) *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. Example: An error can cause the sender to send 32 HELLO packets instead of 32 data packets.

(2) *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, but the changed header value is still valid, as in the stuck at fault.

(3) *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. Similar error injection techniques have been employed in [37].

We carry out two sets of runs 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 the events – 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 an entry 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.

4.5.3 Hierarchical Monitor Results

The set up for hierarchical Monitor is shown in Figure 4(c),(d). 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.

Table 1: Results of error injection with hierarchical Monitor

	No Exception No Crash Slow data rate (DE)	Exception No Crash (E;NC;DE)	Exception Crash Slow data rate (E;C;DE)	Missed Alarms by Hierarchical Monitor	False Alarm	Coverage By Hierarchical Monitor	Coverage by Single Level Monitor	Improvement 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%

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. This improvement is achieved by adding just two rules at the Global Monitor. The two rules correspond to aggregate data rate and aggregate nack rate respectively being above and below thresholds, which are tighter than would be given by summing the local thresholds. 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.

5. RELATED WORK

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 [1] , Temporal Logic Actions (TLA) [2],[3], and Petri net based models [4]. 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.

There is a volume of work on detecting crash failures through heartbeats, failure detectors, etc. (e.g., see [13]), building resilient distributed applications through fault tolerant algorithms built into the application (e.g., see [14],[16]). Their goals are considerably different from the work presented here and hence, not surveyed further. There is previous work [5],[6] 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 [7], 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.

There are other research efforts that have provided the separation into an observer and an observed system and performed online monitoring ([8],[15],[23]-[27]). They differ from our approach in one or more of the following ways – the fault models supported are more restrictive [24], the level of observability or controllability required from the application is higher [27], their approach involves both static and runtime components [26], or their scalability and detection coverage have not been evaluated with real world applications [15].

Near identical goals, as in this paper, have motivated the work in [8] and [15]. 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 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 [9], but it requires a global state graph of the system which may be infeasible to build or verify at runtime for complex systems. In [8], 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 though the local rules did not. Finally, the effectiveness of the approach has not been demonstrated through any error injection based experiments.

The work done in the SNMP protocol [28], Big Brother [29], and HP OpenView [30],[31] deal with system and network monitoring. BigBrother is a simplified web-based system and network monitoring tool which has a client-server architecture with mechanisms to push and pull data. SNMP is an extension of the TCP/IP protocol suite and enables transfer of management information between network nodes. HP Openview monitors, controls and reports on the status of the IT environment using various management technologies like SNMP, ARM etc. All these protocols are mostly used for crash failure detection, and are generally intrusive as they require agents on the monitored nodes.

IDSs are used in inspection of inbound and outbound traffic to detect known patterns that indicate attempts to attack the system. The IDSs are used to observe traffic with respect to a specific host or a specific part of the network. They do not operate at the application level as the Monitor does. The proxy firewalls [32] are capable of performing simple checks on application level traffic, such as, is the web request properly formatted. However, their scope is limited compared to the Monitor. Also, the usage of the Monitor is primarily for detecting natural errors in contrast to these technologies which are used for detecting security attacks.

Distributed information management systems like Astrolabe [21] collect system state and permit aggregation and updates. Astrolabe is a peer-to-peer protocol and requires the application to be aware of it and requires to be installed on the application hosts. Finally, Ganglia from Berkeley [22] is a distributed monitoring system for clusters and grids, which uses multicast for listening and announcing state and a tree-based aggregation mechanism. It uses heartbeats for maintaining membership, and can hence only detect crash failures. Its focus is scalably gathering state from machines in systems of widely varying scales and not on detection of failures.

There are several efforts aimed at debugging, primarily focused on performance problems [33],[34]. The central theme is collecting message traces or event logs at runtime and using *offline* analysis to pinpoint services (or a set of services) that may be contributory to an observed performance problem. The level of observability is assumed high, and some of the sources of information used can be fed to the Monitor to aid in detection. Also the problem solved is diagnosis, not detection, which is the focus of this paper.

6. CONCLUSIONS

In this paper, we have presented a dynamic Monitor architecture for detecting failures in large distributed systems. The Monitor is an external entity independent of the underlying protocol and observes exchanged messages between the protocol participants. It deduces any runtime failure by matching the exchanges against a rulebase of combinatorial and temporal rules. To make the detection infrastructure scalable and dependable, a hierarchical Monitor architecture is presented. Automatic rule segmentation, inter-monitor interactions, and load balancing are incorporated in the Monitor architecture, making it adaptable to the dynamics of today's real networks. Each Monitor can handle a high data rate of 1.49 Mbps with a small number of threads providing a low latency of matching. The performance evaluation using a TRAM testbed running on a production network shows coverage of 84% and 91% for single level and two level Monitor deployments.

Current work is focusing on automatically creating geographically optimal placements of the Monitors, redundancy in the Monitor hierarchy with a protocol participant being monitored by multiple Monitors at the same level. We are extending the current work to incorporate Byzantine failures in PEs and Monitors as well. In an effort to build a complete reliable distributed system we are exploring the mechanisms of doing probabilistic diagnosis once a fault is detected.

References

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

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] G. Khanna, J. S. Rogers, and S. Bagchi, "Failure Handling in a Reliable Multicast Protocol for Improving Buffer Utilization and Accommodating Heterogeneous Receivers," In IEEE Pacific Rim Dependable Computing Conference (PRDC '04), pp. 15-24, March 2004.

[11] 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.

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

[13] 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.

[14] 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.

[15] M. Diaz, G. Juanole, and J.-P. Courtiat, "Observer-A Concept for Formal On-Line Validation of Distributed Systems," IEEE Trans. on Software Engineering, vol. 20, no. 12, pp. 900-913, Dec 1994.

[16] 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.

[17] 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.

[18] R. Alur, R.K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "Partial-Order Reduction in Symbolic State-Space Exploration, " Journal of Formal Methods in System Design, 2001.

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

[20] A. W. Mazurkiewicz, "Basic notions of trace theory," in J.W. de Bakker, W.-P. de Roever, and G. Rozenberg (Eds.), Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency. Lecture Notes in Computer Science, Vol. 354, Springer-Verlag, Berlin, 1989, pp. 285–363.

[21] R. V. Renesse, K. P. Birman, and W. Vogels, "Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining," ACM Transactions on Computer Systems, Vol. 21, No. 2, pp. 164-206, May 2003.

[22] M. L. Massie, B. N. Chun, and D. E. Culler, "The Ganglia Distributed Monitoring System: Design, Implementation, and Experience," Parallel Computing, Vol. 30, Issue 7, July 2004.

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

[24] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M.Viswanathan, "Runtime assurance based on formal specifications," In Proceedings International Conference on Parallel and Distributed Processing Techniques and Applications, 1999.

[25] V. Paxson, "Automated packet trace analysis of TCP implementations," Computer Communication Review, 27(4), October 1997.

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

[27] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem Determination in Large, Dynamic Internet Services," Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN), Pages: 595 - 604 , 2002.

[28] SNMP Research International Inc., "Simple Network Management Protocol," At: http://www.snmp.com/protocol/

[29] Quest Software, "Big Brother System and Network Monitor," At: http://www.bb4.org/

[30] P. Mason, "Turning IT Overhead into Business Value by Improving Infrastructure Management", IDC White Paper, May 2002.

[31] Hewlett Packard, "HP OpenView Management Solutions for your Adaptive Enterprise," At: www.openview.hp.com.

[32] E. Skoudis, "Counter Hack," Chapter 2, Prentice-Hall Inc., 2002.

[33] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem Determination in Large, Dynamic Internet Services," Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN), Pages: 595 - 604 , 2002.

[34] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, “Performance Debugging for Distributed Systems of Black Boxes,” In the Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP), 2003.

[35] I. Rouvellou and G. W. Hart, “Automatic Alarm Correlation for Fault Identification,” Infocom, pp. 553-561, 1995.

[36] I. Katzela and M. Schwartz, “Schemes for fault identification in communication networks,” IEEE/ACM Transactions on Networking, Volume 3, Issue 6, pp.753-764, Dec. 1995.

[37] S. Bagchi, Y. Liu, K. Whisnant, Z. Kalbarczyk, R. K. Iyer, Y. Levendel, L. G. Votta, “A Framework for Database Audit and Control Flow Checking for a Wireless Telephone Network Controller,” DSN 2001, pp. 225-234.

[38] 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.

[39] G. Khanna, M. Y. Cheng, J. Dyaberi, S. Bagchi, M. P. Correia, P. Vérissimo, “Probabilistic Diagnosis through Non-Intrusive Monitoring in Distributed Applications,” In submission to IEEE International Symposium on Dependable Systems and Networks. Available as Purdue ECE Technical Report 05-19.

7. APPEDIX

The exhaustive set of rules that form the Normal Rule Base is given here. The prefix S, R, and RH denote respectively the sender, the receiver, and the Repair Head (RH). These are the entities at which the corresponding rule applies.

Temporal Rules

```
S)      T R3 S0 E3 30 150 5000
```
The sender should send data between the minimum and the maximum rate (30 to 150 packets in 5000sec).

```
S,R)    T R2 S0 E11 50
```
The PE should change from S0 (initialization state) after sending/receiving a data packet. The change of state is confirmed by the next event which transpires at that PE.

```
S,R)    T R4 S1 E11 30 500 5000 S2 E9 1 8 500 7000
Receiver must send ack after receiving data packets.
```

```
S,R,RH) T R3 S1 E4 1 4 500
```
This restricts the number of Hello messages, to between 1 and 4 in 500 seconds.

```
S,R,RH) T R3 S2 E4 1 5 500 S5 E5 1 5 500
```
If an entity receives a Hello message, then it should send a reply.

```
R)      T R1 S4 0 0 !S4 0 3000
```
If receiver is in the re-affiliation state, then it should get re-affiliated within 3000 seconds.

```
R)      T R3 S4 E2 1 3 500
```
The receiver must not send more than 3 head-bind messages within 500 seconds in the re-affiliation state.

```
R)      T R3 S4 E3 1 3 500
```
The receiver must not send more than 3 re-affiliation requests within 500 seconds.

```
RH)     T R3 S0 E3 1 30 5000 S1 E3 1 30 5000
```
If an RH gets data packets from the sender(or a higher RH), then it should send data to its child PEs.

```
RH)     T R3 S2 E11 1 5 5000 S3 E11 1 5 5000
```
If an RH gets acks from receivers then it should send an ack out to the sender (or to a higher RH).

Combinatorial Rules

Notation :
&: Logical AND
V: Logical OR

```
S) S0 & !E9
   S1 & !E11
```
A sender should not send out any Ack packet.

```
S,RH,R) (S0 & E4 & E5)
```
In the start state each PE must send a Hello message and receive replies.

```
R) S4 & !E11
```
Re-affiliation state should not accept a data packet.

```
R) (S4 & -E3) & (E2 V E3 V E14)
```
A receiver should either send a Head-Bind, Re-affiliation Request or leaving message (E14) in the re-affiliating state.