Self Checking Network Protocols: A Monitor Based Approach

Gunjan Khanna, Padma Varadharajan, Saurabh Bagchi
Dependable Computing Systems Lab
School of Electrical & Computer Engineering, Purdue University
465 Northwestern Avenue, West Lafayette, IN 47907.
Email: {gkhanna, pvaradha, sbagchi }@purdue.edu

Abstract

The wide deployment of high-speed computer networks has made distributed systems ubiquitous in today's connected world The machines on which the distributed applications are hosted are heterogeneous in nature, the applications often run legacy code without the availability of their source code, the systems are of very large scales, and often have soft real-time guarantees. In this paper, we target the problem of online detection of disruptions through a generic external entity called Monitor that is able to observe the exchanged messages between the protocol participants and deduce any ongoing disruption by matching against a rule base composed of combinatorial and temporal rules. The Monitor architecture is application neutral, with the rule base making it specific to a protocol. To make the detection infrastructure scalable and dependable, we extend it to a hierarchical Monitor structure. The infrastructure is applied to a streaming video application running on a reliable multicast protocol called TRAM installed on the campus wide network. The evaluation brings out the scalability of the Monitor infrastructure and detection coverage under different kinds of faults for the single level and the hierarchical arrangements.

1. Introduction

The wide deployment of high-speed computer networks has made distributed systems ubiquitous in today's connected world. Distributed middleware, such as CORBA, DCOM, GLOBE, distributed file systems, such as NFS, XFS and distributed coordination based systems, such as publish-subscribe systems, distributed network protocols, such as reliable multicast, and above all, the distributed infrastructure of the world wide web form the backbone of much of the information technology infrastructure of the world today. The infrastructure, however, is increasingly facing the challenge of dependability outages. The outages result both from naturally occurring failures and malicious attacks, collectively referred to as disruptions in this paper. The potential causes of natural failures are hardware failures, software defects, operator failures, and misconfigurations, while the malicious attacks may be launched by external or internal users. The consequences of downtime of distributed systems are catastrophic. An extent of the financial losses can be gauged from a survey by Meta Group Inc. of 21 industrial sectors in 2000 [15], which found the mean loss of revenue due to an hour of computer system downtime to be $1.01M, with power companies at the top ($2.8M). 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 [16] and the trend becomes clear. Little wonder that distributed systems are called upon to provide always-available and trustworthy services. Failures of distributed systems employed in safety critical applications, such as, flight control, nuclear plant monitoring, and railway signaling, can lead to loss of human lives.

The challenges to building distributed systems capable of tolerating disruptions are manifold. The machines on which the applications are hosted are heterogeneous in nature, the applications often run legacy code without the availability of their source code, the systems are of very large scales, of the order of tens of thousands of protocol participants (such as, a system with DNS clients and servers), and the systems often have soft real-time guarantees. While it may be possible to devise very optimized and targeted solutions for individual distributed applications, such approaches are not very interesting from a research standpoint due to their limited applicability. In our earlier work, we have demonstrated the ability to make a reliable multicast protocol, TRAM, resilient to a class of natural failure and a class of malicious attacks through incremental protocol changes [17]. However, the changes are intrusive to the protocol, require thorough understanding and access to the source code of the protocol, and are effective only against the specific classes of disruptions.

In this paper, we propose a generic Monitor architecture for detection of disruptions in distributed applications, which pose all the challenges mentioned above. The solution approach employs a Monitor that snoops on the communication between the protocol participants and performs matching of the observed communication against a rule base that characterizes acceptable protocol behavior. The protocol participants are treated as black boxes and their internal state transitions are invisible to the Monitor. 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. We provide a specification syntax for the rules, in which a rule may be combinatorial (valid for all points in time in the lifetime of the application) or temporal (which have an associated time component). We provide fast rule matching algorithms that match the incoming messages against the rules. In order to make the infrastructure scalable, efficient, and accurate, we propose a hierarchical Monitor structure where the Local Monitors directly gather communication between the protocol participants that are geographically localized and the higher level Monitors get processed and filtered messages from the lower level Monitors. This allows a higher level Monitor to perform detection using observed behavior that may not be local. Further, in the Monitor approach, the combination of the output from the Monitor and the monitored protocol satisfies the definition of a self-checking system. This definition states that when subjected to the tolerated class of faults, the system either masks the fault, produces no output at all, or an output that falls outside the set of permissible outputs and can therefore be easily detected.

The Monitor based approach is demonstrated by applying it to a streaming video application running on top of a tree-based reliable multicast protocol called TRAM. Three different kinds of errors are injected into the application and the performance and the accuracy of the Monitor structure evaluated. The overall detection coverage is 84.4% and 91.0% for the single level and the two-level Monitor system, respectively. The false alarm rate is 4% and is the same in the single level and the two-level hierarchical cases since all the false alarms are generated by the Local Monitors.

The rest of the paper is organized as follows. Section 2 presents the design of the Monitor. Section 3 provides details about the workload, the deployed system, and the instantiated rule base. Section 4 gives the experiments and the results. Section 5 surveys related work. Section 6 concludes the paper with mention of future work.

2. Design

In this discussion, we will use the term Monitor to refer to the detection infrastructure and observed entities to refer to the protocol participants that are being monitored. The Monitor is said to *verify* the observed entities. Since the application is considered to be a black box, there is no access to the source code of the protocol participants or to their execution hosts. The specification of the protocol is, however, available to the Monitor. The Monitor has access to the messages exchanged between the monitored entities, and hence is able to examine the communication header and payload. The Monitor approach employs a stateful model for rule matching, preserving state across messages. The Monitor would typically not run on the host of the observed entity but in its network vicinity and should be able to observe the messages in and out of the observed entity.

2.1. Monitor Architecture

The Monitor architecture shown in Figure 1 consists of several components classified according to their functional roles.

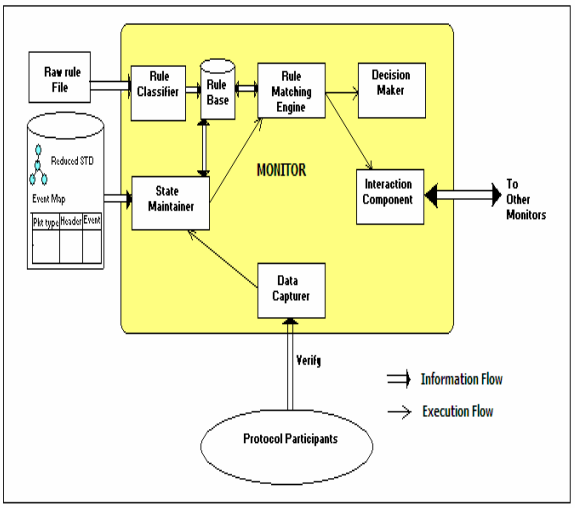

Figure 1. Monitor architecture with process flow and information flow among multiple components

2.1.1 Data Capturer. The Data Capturer snoops on the communication channel and captures messages exchanged by the protocol participants. The capture can be in either active or passive mode. In the active mode, the protocol participant itself sends a copy of the message to the Monitor. In the passive mode, there

is no direct cooperation from the participant and the Data Capturer snoops over packets exchanged in the communication medium.

This may be achieved either by having the Monitor placed in the same broadcast LAN as the monitored entities in a promiscuous mode, or with router support with the router mirroring packets to the port to which the Monitor is connected. In switched environments, port mirroring enables data capture, by sending to the port a copy of all messages received by the switch. This feature can be configured in many of the popular switches available today, including the CISCO Catalyst family, 3COM SuperStack, and Intel Express. Since the Monitor is considered a trusted entity in the system, we assume that cryptographic keys, if any, will be made available to it. Thus it can perform its processing after decrypting the message. The Data Capturer passes the message to the State Maintainer.

2.1.2 State Maintainer. During the setup phase of the Monitor, the State Maintainer is loaded with a set of event definitions and a reduced state transition diagram of each monitored entity. The event definition maps the values of header fields in a message to an event. A reduced state transition diagram is input to the Maintainer, in which the transitions are due to messages sent and received by a monitored entity, and not due to any internal state transitions. The state objects are maintained in a global hash table, indexed by the state name. Each state object contains the state variables in the particular state, and the expected incoming events and the corresponding next state. The combination of current state and incoming event determines the set of rules to be matched. The rule base is linked to the State Maintainer, but is not a part of it. At runtime, the State Maintainer gets an incoming message from the Data Capturer, maps it to an event, checks if the event is an expected event in the current state, and if so, triggers the appropriate state transition and invokes the Rule Matching Engine with the appropriate rule(s).

2.1.3 Rule Matching Engine. The Rule Matching Engine, which is triggered by the State Maintainer on the receipt of an appropriate event in an appropriate state, is the most resource intensive workhorse of the Monitor. In order to constrain the detection latency, it is crucial that the matching algorithms be optimized for speed. Due to the different nature of temporal and combinatorial rules (See Section 2.2), we provide separate matching algorithms for them.

Each combinatorial rule is translated into an expression tree which has Boolean operators in the intermediate nodes, and operands (state, state variable, or event) in the leaf nodes. Combinatorial rule matching is performed by traversing the expression tree. We use two optimizations in the combinatorial rule matching. The first is based on the observation that the same rule may be matched multiple times and not all the operands would change between successive rule matches. Hence, the algorithm should be incremental in its Boolean value computation. This is achieved by storing the previously computed value in each node of the expression tree and for a non-leaf node, the list of operands in the sub-tree underneath it. The algorithm is passed the set of operands that have changed and it re-evaluates only that part of the expression tree which may have changed. The second optimization is to have the nodes arranged in order of the frequency of change with the more frequently changing operands towards the left. The algorithm visits nodes left to right, and if a rule has no repeated operand and a single operand has changed, as is often the case, then once a match occurs, the algorithm does not need to explore the remainder of the expression tree.

Temporal rule matching handles appropriate alarm generation and rule matching according to temporal rule specifications. There are two time scales in the context of temporal rules, one is the time instant when the state variable is captured for matching, and the second is the time when the captured value is used. Since the two time points may be arbitrarily spread apart in time, two separate timers — Variable Copier and Rule Matcher — are used. After expiration of the Variable Copier timer, the Monitor copies the state variables to be examined in the rule into the rule object. After the expiration of the Rule Matcher timer, the rule is matched. The matching of temporal rules is thereby optimized for speed by having these two timers. Moreover, the variable copier ensures authenticity of the state variables captured for matching. Even if the rule matching itself is delayed, the state variables are captured at the precise moments they were meant to by the Variable Copier. The classification of the rules into four categories also adds efficiency in matching. This is because the rules are matched in the *tightest* category applicable, incurring the least possible overhead. The implementation involves creation of a thread pool so that matching multiple rules for a single or multiple monitored entities that are applicable in a particular state can occur concurrently.

2.2. Structure of Rule Base

Rules are specified in terms of current state,

incoming event, and corresponding state variables. The formal rule syntax is important because it determines the expressibility of the system and by extension, its ability to detect different classes of disruptions. 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. The rules are currently created manually by the administrator and fed in during the set up phase. 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 for real-world large scale distributed systems vulnerable to different kinds of disruptions. When choosing the rule base syntax, several existing approaches were evaluated. We use a formalism based on temporal logic actions [5],[6].

2.2.1 Combinatorial Rules. These are the rules expected to be valid for the entire period of execution of the system, except transients. A combinatorial rule has the operators '!' for logical NOT, 'V' for logical OR, and 'Λ' for logical AND. Each expression representing a rule finally yields a Boolean true or false. Although the combinatorial rules must be valid for the entire length of time, there could be transients, which cause temporary deviations. The rule matching algorithm, given an input length of transients in the system, has the intelligence not to flag the temporary deviations as errors.

2.2.2 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. While it is clearly impossible to claim that rules for detecting all kinds of disruptions in all networking protocols can be folded into these categories, a pragmatic design choice was made to come up with a classification that seem adequate for a large class of anticipated disruptions and protocols.

Type I:

$S_p = \text{true for } T \in (t_N, t_N + k) \Rightarrow S_q = \text{true for } T \in (t_1, t_1 + b)$

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 at t_1. The time t_N represents a time when some defined event E_1 takes place.

Type II: $S_t \neq S_{t+\Delta}$, where S_t is the state of an object at time t and if a defined event E_i takes place at t.
The state S_t will not remain constant for more than Δ time units if an event E_i takes place.

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

The state variable V_t in a particular state S_I will be bounded by L and U in some time k starting at time t_i when the defined event corresponding to the rule first occurs.

Type IV: $L \leq |V_t| \leq U$, for $t \in (t_i, t_i+k) \Rightarrow L' \leq |B_q| \leq U'$, for $q \in (t_n, t_n+b)$.
A state variable V_t being bounded by upper and lower bounds in time k will cause another state variable B_q to be within some lower and upper 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.

2.3. Hierarchical Monitor

2.3.1 Mutli-level Monitor Architecture. A single Monitor 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 entities may lead to reduced coverage. Therefore, in our system, we incorporate the idea of using a hierarchy of Monitors working interacting with one another to detect disruptions. The entire structure is divided into Local, Intermediate, and Global Monitors as shown in Figure 2. The functionality of the Local Monitor has been described earlier in the section on "Monitor Architecture" (Section 2.1). The Intermediate Monitor gathers information from several Local Monitors, each locally monitoring a set of entities. In addition, the Intermediate Monitor may also be monitoring entities directly. The Local and Intermediate Monitors perform filtering of messages which are not required for rule matching at the higher level, and processing, such as aggregation through counts of events. The Global Monitor has a global view of the protocol and is unique in the system. 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 Global Monitor enhances coverage and accuracy, performing detection across multiple hosts monitored

by different lower level Monitors. Each Monitor has the same architecture as described in Section 2.1 with the same rule base.

The hierarchical approach increases the accuracy and the coverage of detection. Redundancy exists through having a protocol entity being monitored by multiple monitors on the path from the Local Monitor to which the entity is bound, to the Global Monitor. The Interaction Component in Figure 1 acts as the gateway for communication between monitors placed at the same level or at different levels.

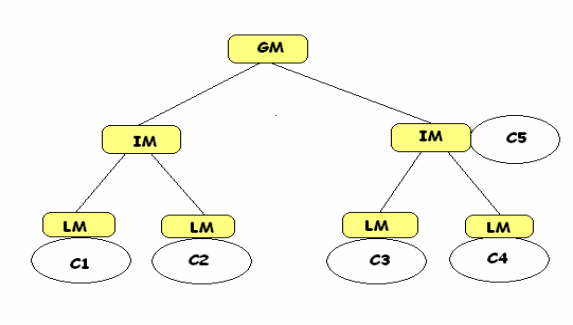

Figure 2. Example topology of Local (LM), Intermediate (IM), and Global (GM) Monitors. [C denotes a cluster]

2.3.2 Rule Classifier. Owing to the hierarchical architecture of the Monitor, there are situations where all rules are not relevant to a given Monitor. The *Rule Classifier* is responsible for deciding whether a rule is to be matched at this Monitor. The Rule Classifier partitions the rule base automatically into three classes – one which is not relevant to the Monitor (*Bypass Rule*), the second which is relevant to the Monitor alone (*Local Rule*) 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 (*Global Rule*). A rule does not enjoy a universal classification in the system. Rather, each rule classification is with respect to an individual Monitor. A global rule is further categorized as *non-processable* rule (message corresponding to the rule is simply forwarded), *match-and-pass* rule (message corresponding to the rule is matched and forwarded), and *match-process-and-pass* rule (message is matched, processed, and then forwarded). The algorithm for automatic rule classification can be found in another submission [18].

2.4. Design Requirements

It is essential that the Monitor does not become a performance bottleneck for the monitored application. The Monitor should be capable of scaling to thousands of protocol participants. The Monitor should be applicable to a large class of applications with minimal effort in moving from one application to another. Finally, the Monitor should have a low latency of detection so that propagation of the disruption is limited. Let us see how these design requirements are met.

First, the Monitor is not a performance bottleneck since it functions asynchronously to the application protocol and runs on independent hosts, not those of the participants. Second, the hierarchical Monitor structure addresses the issue of scalability. The Local Monitors oversee the local communication among the participants, while the higher level Monitors are invoked if the behavior to be monitored spans multiple local clusters. Since for a well-designed protocol, the local communication should be the common communication pattern, the proposed architecture can scale with the number of participants. Third, the Monitor is generic in its architecture and therefore, widely applicable to message passing based distributed protocols. The rule base which is specific to the application is also specified in an easily understandable temporal and combinatorial logic format. There are automated rule classification algorithms to take a unified rule base and partition it into local and global rules. Fourth, the challenge of providing low latency detection is addressed by partitioning the rules intelligently, thereby minimizing the number of rules to be matched at any Monitor. Also, highly speed optimized and distinct algorithms for matching the temporal and combinatorial rules are provided.

3. System Description

3.1. Workload

The Monitor architecture described in Section 2.1 is implemented in Java for portability reasons. We demonstrate the use of the Monitor on the running example protocol — a reliable multicast protocol called TRAM [19],[20] to make it self checking.

TRAM is a hierarchical tree based reliable multicast protocol for multicasting data from a single sender to multiple receivers. The receivers and the data source of a multicast session in TRAM interact with each other to dynamically form repair groups. Figure 3 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).

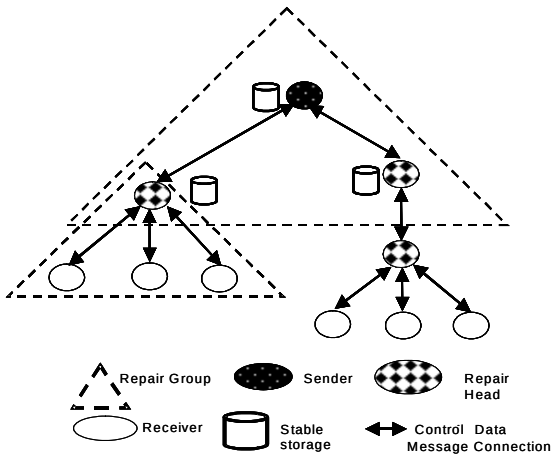

Figure 3. Hierarchical tree structure in TRAM

Within a repair group, recovery happens locally through the RH, which stores the data in stable store till all receivers in the group have acknowledged receipt. These repair groups are linked together in a hierarchical manner to form a tree with the sender at the root of the tree. The group members report lost and successfully received messages to the RH using a selective acknowledgement mechanism. A *nack* message is sent implicitly through the ack by setting a bit vector accompanying the ack message appropriately to indicate the missing packets. An ack message is sent after every *ack window* worth of packets has been received, or an *ack interval* timer goes off. The RHs aggregate acks from all its members and send an aggregate ack up to the higher level to avoid the problem of ack implosion.

The different kind of messages exchanged in TRAM and their interpretations are given in Table 1. The values in two header fields *Message Type* and *Sub-message Type* together determine the control message type.

3.2. Monitor State

In our implementation, the recipient of a packet does active forwarding of the packet to the Local Monitor. The Monitor reads the header fields, maps the packet to an event, performs the appropriate state transitions and finally, performs the relevant rule matching. The Monitor follows a reduced STD which is input by the system administrator. Forming the reduced STD is somewhat of an art that the administrator deploying the Monitor needs to master. We are currently exploring some methods for automating the reduced STD creation given some formal specification of the protocol, such as UML state diagram. Minimally, the states in the reduced STD must cover all the states mentioned in the rules in the rule base. Henceforth, in the description of the reduced STD and the rule base, we use the event IDs instead of the packet types. The mapping is given in Table 1.

The reduced STD of the receiver, which is stored in the State Maintainer, is shown in Figure 4. In state S2, the receiver checks the integrity of the data packet using the sequence number of the packet. From state S3, if an unexpected error event happens in the interaction between the receiver and its RH, it starts a counter and goes to state S4. If the counter value exceeds a threshold, the receiver disassociates itself with the current RH and re-affiliates with a different RH. The CNF form of the syntax for representing the STD is *STDSpec*⇒“:”*CurStateID*(*IncomingEventID NextStateID*)+ (“!” “:”*CurStateID* (*IncomingEventID NextStateID*)+)*

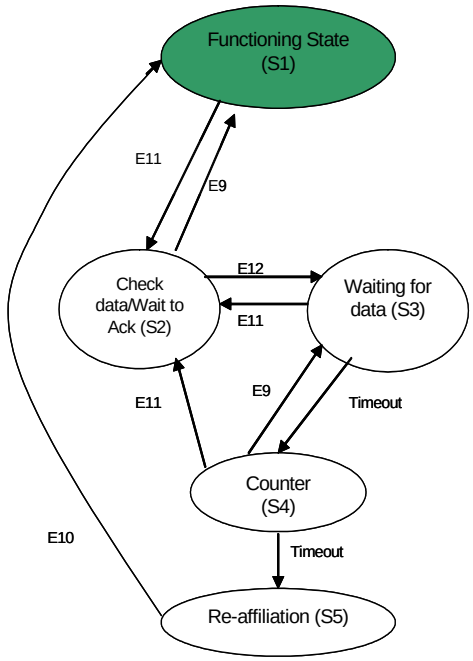

Figure 4. Reduced STD for the TRAM receiver

Table 1. Different control messages in TRAM

Message Name	(Source, Destination)	Interpretation	Event ID
Beacon	Sender, Anybody within set TTL distance	Initialization message for sender to query for interested entities to join in its multicast stream	E1
Head Adv.	Sender(RH), Receivers	Repair Heads send advertisement of the channel	E3
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	E10
Accept/Reject	Repair Head(Sender), Receiver(RH)	Acceptance or Rejection message sent by the repair head to the seeking receiver.	E6, E7
Ack Packet	Receiver, Repair Head(Sender)	Aggregate Acknowledgement sent by the receiver to the repair head.	E9
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.	E2, E8
Data Retransmission	Sender(RH), Receivers(RH)	Data is retransmitted if a nack is received	E12

3.3. Monitor Rule Base For TRAM

The rule base consists of anomaly-based rules governing the execution of the protocol as seen from the viewpoint of 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 3. Here we give examples of rules for the TRAM receiver used in the experiments.

1. ***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. The thresholds are calculated using the maximum and minimum data rate specified by the user.

2. ***T R4 S2 E11 30 500 5000 S2 E9 1 8 500 7000.*** 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. Again the numbers are based on the propagation delay of the packets and the fact that receiver acks at intervals of every 32 packets, the default ack window.

3. T R3 S4 E12 0 5 5000. We restrict the number of nacks to be a max of 5 within 5000 ms. This number is calculated based on the minimum acceptable data rate and the sliding window protocol followed by TRAM for data transmission. A subtle, yet important, technique is used to specify any rule that involves nacks. In TRAM, a nack is sent implicitly in the form of an ack by sending a bit vector and setting the corresponding bit in the vector to be 1. The Monitor only examines the packet header and therefore cannot observe the bit vector sent as payload. In order to observe a nack, the Monitor uses a resultant effect — the data retransmission which usually follows a nack.

4. *T R3 S1 E15 0 16 5000.* This is a *global rule* which restricts the total number of nacks seen globally. The Local Monitors, on receipt of a nack packet, process the rule, and pass the computed results of the number of nacks seen locally as a new event (E15) to the Global Monitor. In the deployment used in the experiments, the Global Monitor verifies two Local Monitors, each of which verifies two nodes. The aggregate number of nacks is restricted to 16, within 5 seconds. Note that this is a tighter bound than what would be indicated by a linear extrapolation of the number of nacks locally as in rule 3 (16 nacks instead of 20). The second global rule corresponds to the number of data packets seen at the global level which is also a rule of type 3. For the deployment, this is kept at 265 to 2500 in a 5 second period.

Several other rules are used in the experiments. A combinatorial rule is that the data rate should be between 20KBytes/sec and 40KBytes/sec. These are specified as configuration parameters for TRAM. Examples of further temporal rules are the state of a receiver should not be the initial state 50ms after receipt of a data packet at the initial state since it should move into the ack-ing state. The number of re-affiliations in a state is upper-bounded by 5 in 10 seconds. This rule prevents a malicious receiver exhausting resources by disconnecting and re-affiliating with different RHs or the sender in rapid succession. The number of unicast hello messages should be limited between 1 and 5 because it is sent only when a particular receiver is not sending hello replies. It is an indicator of the failure of the liveness of the receiver.

3.4. Example of Rule Matching

Consider Rule no. 3 in Section 3.3 – *T R3 S4 E12 0 5 5000.* Let us assume that the current state of the monitored entity is S4, and the first packet that is received after entering S4 is a data retransmission packet, denoted as E12. The Data Capturer captures this packet and passes it to the State Maintainer. The State Maintainer deduces the packet as an E12 event, and checks to see if this is a valid event in state S4, according to its reduced state transition diagram. As this is the case, it further checks if there are any rules corresponding to event E12 in state S4. Finding one, the State Maintainer makes the appropriate state transition and triggers the rule matching engine with the given rule. The state variable corresponding to

number of E12 events for the given rule is incremented. The rule matching engine instantiates a Rule III object, setting the lower bound as 0, the upper bound as 5, and the time period corresponding to Rule III as 5000, as given in the rule. It also sets the Variable Copier and Rule Matcher timers to generate alarms at the end of 5000 ms. Any subsequent packet corresponding to event E12 results in just an increment of the appropriate state variable. A new rule is not instantiated for the same master rule till the previous rule has expired. At the end of 5000 ms, the Variable Copier captures the state variable denoting number of E12 events, and stores the same in the instantiated rule. Following this, the Rule Matcher compares this value to the lower bound and upper bound, to decide if an error should be flagged. The actual rule matching done by the Rule Matcher is a simple comparison, and hence an O(1) operation.

4. Experiments and Results

4.1. Experimental Setup: Workload, Error Injection, Topology

A streaming video application with MPEG-2 video stream is used as the workload. The application is executed over TRAM, with the server running on the sender and multiple clients running one on each receiver. 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 data 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.

The errors are injected into the header of the TRAM packet before dispatching it to the receiver. The receiver actively forwards the injected packet 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 kept at 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.

(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 model.

(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.

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 (D_E), or no failure occurs (N_F). 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 (N_E; N_C; D_E) denotes no exception, no crash, but client flagged a data rate error. When only a single consequence occurs, the notation can be abbreviated, as (D_E) for the above case. Also, whenever an error is manifested in the protocol, the data rate ultimately drops leading to the data rate error (D_E). If data rate error is *not* the only consequence, D_E 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 D_E), this is a case of error propagation and is classified as a coverage miss. An error which does not lead to a failure but is flagged by the Monitor is categorized as a false alarm.

For the experiments we use a cluster of Linux machines, with the TRAM protocol entities run on 450 MHz Pentium II machines with 256 MB of memory and the Monitors run on Pentium 4 2.26 GHz processor machines with 1 GB memory, 533 FSB and 512 KB cache. The entities are connected among themselves by 1 Gbps links and the entities are connected to the Monitors with 100 Mbps links.

At the outset we conduct performance experiments on the single level Monitor system. It is seen that the latency of detection at one Monitor is less than 30 ms when it matches 50 packets/sec or less and beyond that it grows linearly. With varying the

number of receivers, the increase in latency is found to be linear with 6.25 ms for 1 receiver and growing to 16.71 ms for 26 receivers. This amount of latency is considered tolerable in many environments.

4.2. Single Level Monitor Results

Table 1 presents the effect of the three types of error injection on the protocol. 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). There are no experimental runs where the receiver crashes without any exception and hence this consequence is not shown in the table. This indicates that the exception flagging in TRAM is very extensive. Oftentimes, the receiver side code catches the exception through large try-catch block, but does not do any application specific processing. It prints out the exception stack and allows the receiver to crash.

Overall, Monitor accuracy for the single level case is 84.37%. If we look across the columns for various types of injections, we see that the Monitor's accuracy is high for DE, but drops for (E;NC). This is due to the fact that the exception, if raised by the protocol, is done very soon after the error, *before* the Monitor can flag it, which it does eventually. However, the case counts as a coverage miss for the Monitor. The Monitor's latency negatively affects its coverage as evidenced by the significant drop in coverage from the loose client to the tight client. Part of the latency is a fundamental property of the rule base and cannot be solved by faster processing at the Monitor. Several rules have the form of counting a particular event over a given time interval and checking bounds on the number of occurrences of the event. The length of the time interval forms a constraint in terms of the minimum granularity at which faults can be detected. Hence, a single error that causes an exception goes undetected and the coverage for the (E;NC) cases is considerably lower than for other categories. An alternate design of maintaining a sliding window of the events and a running count would eliminate this problem of higher latency but is not used due to the high state maintenance and processing overhead it entails on the Monitor.

Considering the error injection experiments with the loose random client (LR), overall there is 9.2% missed alarms and 8% false alarms. The main source of missed alarms is the case when there is an exception but no crash of the receiver. An example of this is when the protocol flags an exception because of a large message length and then continues to run. But this is missed by the Monitor. This happens when an injected error into the packet header converts a data packet to an ack packet whose length is greater than expected, leading to an exception. The Monitor is effective in catching the cases where the client flags a slow data error rate. For TR we see an identical number of DE as in LR. However, since the receiver is checking its observed data rate more frequently, it is able to find the slow data rate error much faster. The Monitor's detection latency, on the other hand, remains the same, causing the detection accuracy to go down to 90%. For the directed error injection for the *loose client* (LD) there are 18% DE errors out of which 83% are caught by the Monitor. Compared to LR, there is an increase in the cases where exceptions are raised ((E;NC) and (E;C)) and an equal decrease in DE errors. 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 messages. 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

Table 2. Results of error injection with single level 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	False Alarm	Coverage
Loose Random (LR)	57/58 (98%)	24/30 (80%)	4/4 (100%)	7/92(7.6%)	8%	85/92 (92.4%)
Tight Random (TR)	52/58 (90%)	17/26 (65%)	6/6 (100%)	15/90(16.67%)	10%	75/90 (83.3%)
Loose Directed (LD)	15/18 (83%)	51/64 (80%)	17/18 (94%)	17/100(17%)	0%	83/100 (83%)

Tight Directed (TD)	23/28 (82.1%)	50/62 (80.5%)	10/10 (100%)	17/100(17%)	0%	83/100 (83%)
Loose stuck at (LS)	43/44 (98%)	38/50 (76%)	2/2 (100%)	13/96(13.54%)	4%	83/96 (86.46%)
Tight stuck at (TS)	47/52 (90.38%)	23/32 (71.8%)	7/14 (50%)	21/98(21.4%)	2%	77/98 (79%)
				90/576(15.63%)	24/600 (4%)	486/576(84.37%)

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. The Monitor has an overall coverage of 83%, with greater effectiveness in capturing errors that lead to crashes. In the directed injections, the false alarms are eliminated because of increase in the number of *invalid* transitions, as argued earlier, leading to manifested errors.

In the stuck-at error injection, for LS, about 44% protocol responses are (DE) and 50% are (E;NC). The Monitor detects 98% and 76% of these cases, respectively, confirming the trend from LR that loose client data rate failures are easier for the Monitor to detect than exceptions. There is a sharp increase in the DE errors compared to directed injection. In stuck-at injection, if the injected packet is not expected in a particular state, it is discarded. This causes few state transitions in the protocol, but the data rate goes down ultimately leading to the receiver error. The number of false alarms is low (2%) indicating that the protocol is affected in most cases. Distinctly in TS, the number of (E;NC) cases drops to 32% because the *tight client* causes a DE to be flagged before an exception takes place. Monitor is only able to detect 72% of these cases. There is a sharp rise in the number of (E;C) to 14% half of which are missed by the Monitor. The increase is due to the fact that in stuck-at injection, the same packet is injected for the burst length and if it is a valid protocol packet but injected in a wrong state, it causes the protocol to crash and not simply throw exception. The reduction in coverage is due to the fact that since the same faulty packet is injected, it may cause internal state transitions at the receiver, while the Monitor only observes external messages. These internal state transitions may be preliminary to an exception or a crash which is missed by the Monitor.

4.3. Hierarchical Monitor Results

The setup for the hierarchical Monitor is shown in Figure 5. It is a two level hierarchy with each Local Monitor overseeing two receivers and a Global Monitor overseeing the two Local Monitors. 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.

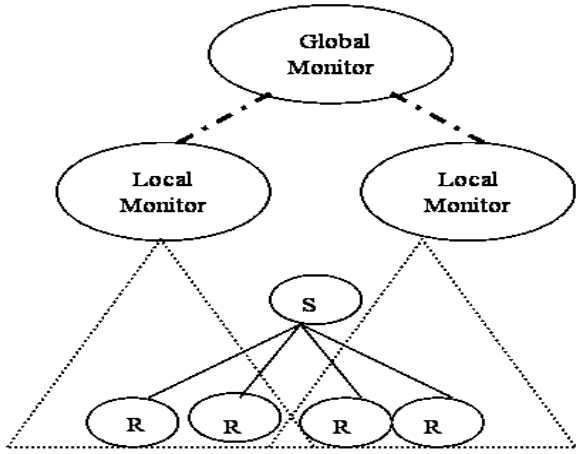

Figure 5. Two level Monitor hierarchy used for the experiments

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 2. 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. Note that these are a new set of experiments compared to the single level experiments presented in Table 1. The coverages for the single

level case differ only due to statistical differences caused by the two sets of experiments. 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 results corroborate the need for a hierarchical setup of Monitors. The increase in coverage is most significant for the loose directed case (12%).

Table 3. 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 System	False Alarm	Coverage By Hierarchical Monitor System	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%

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 due to the same reason as in the single level case (more number of invalid protocol transitions). However, the difference in coverage is not as sharp indicating that the global rules are able 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 [4], Temporal Logic Actions (TLA) [5],[6], and Petri net based models [7]. 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 [21]), building resilient distributed applications through fault tolerant algorithms built into the application (e.g., see [22],[23]). Their goals are considerably different from the work presented here and hence, not surveyed further. There is previous work [8],[9] 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 protocol entities themselves. There exist systems with the high-level goal of checking online system behavior against specifications [24]-[27]. However, they differ widely in approach, assumptions, or focus. For example, the MAC project

[27] is focused on bridging the gap between the high level specification of correctness and the low level events generated by the system implemented in Java.

A detection approach using event graphs is proposed in [10], 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 [11],[12] which have relied on participation by the protocol entities and the classes of faults have also been restricted.

Near identical goals, as in this paper, has motivated the work in [3] and [13]. 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 [14], but it requires a global state graph of the system which may be infeasible to build or verify at runtime for complex systems. In [13], 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.

6. Conclusion

In this paper, we have presented the Monitor architecture for detecting natural failures and malicious attacks, collectively termed disruptions, in large distributed systems. The Monitor is an external entity that observes exchanged messages between the protocol participants and deduces any ongoing disruption by matching the exchanges against a rule base of combinatorial and temporal rules. To make the detection infrastructure scalable and dependable, a hierarchical Monitor architecture is presented. The infrastructure is applied to a streaming video application running on a reliable multicast protocol called TRAM over the Purdue campus-wide network. The evaluation shows coverage of 84% and 91% for single level and two level Monitor deployments aggregated over three different classes of errors.

Current work is focusing on redundancy in the Monitor hierarchy with a protocol participant being monitored by multiple Monitors at the same level. We are also investigating making the environment dynamic where participants and Monitors may come and go, and the assignment of the protocol entities to the Monitors may change dynamically. We are extending our framework to deal with the sources of non determinism like delays in the network and difference in the sequence of observed events from the sequence of occurred events.

Acknowledgements

We have been enormously helped in the deployment and management of TRAM on Purdue's network by personnel at ITAP, Purdue, which is our information technology organization. Particular thanks go out to Casey Carlson and Dale Talcott.

References

[1] D. A. Anderson and G. Metze, "Design of totally self-checking circuits for m-out-of-n codes," IEEE Trans. on Computers, vol. 22, Mar 1973.

[2] M. Diaz, P. Azema, and J. M. Ayache, "Unified design of self-checking and fail safe combinational circuits and sequential machines," IEEE Trans. on Computers, vol. 28, no. 3, pp. 276-281, Mar 1979.

[3] 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.

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] I. Katzela and M. Schwartz, “Schemes for Fault Identification in Communication Networks,” IEEE/ACM Trans. on Networking, vol. 3 , no. 6 , pp. 753-764, Dec 1995.

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

[13] 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.

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

[15] 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

[16] FIND/SVP, 2003, "Costs of Computer Downtime to American Businesses," At: www.findsvp.com

[17] 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.

[18] P. Varadharajan, G. Khanna, and S. Bagchi, “Automated Online Monitoring of Distributed Applications through External Monitors,” Submitted to IEEE Trans. on Parallel and Distributed Systems. Available at:

www.ece.purdue.edu/~sbagchi/research.html

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] A. K. Mok and G. Liu, “Early Detection of Timing Constraint Violation at Runtime,” In Proceedings of the IEEE Real-Time Systems Symposium, December 1997.

[25] F. Jahanian and A. Goyal, “A Formalism for Monitoring Real-Time Constraints at Run-Time,” In Proceedings of the 20th International Symposium on Fault-Tolerant Computing Systems (FTCS-20), pp.148-155, 1990.

[26] K. Havelund and G. Rosu, "Monitoring Java programs with Java PathExplorer," in K. Havelund and G. Rosu (Eds.), Proceedings of Runtime Verification (RV'01), Vol. 55 of Electronic Notes in Theoretical Computer Science, Elsevier Science, 2001c.

[27] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, Mahesh Viswanathan, “Java-MaC: a Rigorous Run-time Assurance Tool for Java Programs,” In “Formal Methods in Systems Design”, vol. 24, no. 2, March 2004.