SELF CHECKING NETWORK PROTOCOLS: A MONITOR BASED APPROACH

A Thesis

Submitted to the Faculty

of

Purdue University

by

Gunjan Khanna

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

December 2003

ACKNOWLEDGMENTS

Special thanks to Prof. Saurabh Bagchi without whose efforts and input this thesis would not have made its way to the Graduate Office. He gave me the inspiration to strive , think and achieve what stands in the form of this thesis. I would also like to thank Dale Talcot and Casey Carlson from the ITAP department in Purdue whose help in installation and running of TRAM was priceless. A special thanks to Padma and John for their immense help in the project. The inputs from Prof. R. K. Iyer and Zbigniew Kalbarczyk of the University of Illinois at Urbana-Champaign was helpful in giving the right shape to the project and helped in avoiding some pitfalls. I would also like to thank Prof. Ness Shroff and Prof. Rudolf Eigenmann for taking time out and reading the thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Khanna, Gunjan. Masters of Science. Purdue University, Dec 2003. Self Checking Network Protocols : A Monitor Based Approach. Major Professor: Saurabh Bagchi.

The wide deployment of high-speed computer networks has made distributed systems ubiquitous in today's connected world. The systems are affected by disruption i.e. errors within the protocol or intrusions. This motivates the need for building distributed systems that are capable of tolerating disruptions and providing highly available and correctly functioning services. 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) 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 developing this thesis we have focused on Monitor based detection of disruptions in a distributed environment. Monitor detects the disruptions by looking at only the external message exchanges, without looking at the internal transitions of the monitored entity. It is made to run asynchronously to the application thus avoiding the performance bottleneck. We have chosen a black box Monitor approach suitable for any generic protocol. By developing the "Monitor Based Detection Approach", aim is to provide higher reliability and dependability.

We propose a Hierarchical Monitoring approach by placing a hierarchy of local and Global Monitors in the system. A Local Monitor only monitors a set of local nodes while a Global Monitor can have several local monitors reporting local interactions to it. This provides increased coverage and accuracy of detection. The Monitor consists of a Rule Classifier, Data Capture and Matching Engine as the main components. The rules are classified into Local and Global rules intelligently by the rule classifier. The Matching Engine consists of fast matching algorithms each for Temporal and Combinatorial rules. Testing of the Monitor is done on a Distributed Reliable Multicast Protocol called TRAM. The Monitor is tested by injecting faults into the running protocol using a Fault Injector.

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. The naturally occurring failures can be crash failures (server halts but works correctly till it halts), omission send or receive failures (server fails to send or receive incoming messages), timing failures (server's response falls outside the acceptable time bound), response failures (server's response is incorrect because of value failure or incorrect flow of control), or, the worst scenario, arbitrary or Byzantine failures (server may produce arbitrary responses at arbitrary times). The potential causes of downtime are manifold – hardware failures, system or application software failures, operational failures (or operator errors), maintenance (such as, backups and software upgrades), environmental problems (such as, power outages and communication lines being down).

An example from recent memory is AT&T's ATM network outage in February, 2001 which caused a downtime of 4 hours for 7% of all its ATM customers and was caused by a misconfiguration in its WAN switch resulting in a firestorm of system management messages [2]. The outages may also be caused by malicious intruders launching attacks against the infrastructure through sending viruses, worms, malformed network packets, etc. designed to exploit vulnerabilities in the hardware or software design of the systems. An example from recent times is the distributed denial of service (DDoS) attack that brought down 9 of the 13 root DNS servers that control the internet traffic. We refer to the combination of failures and intrusions as *disruptions* in the rest of this proposal. The consequences of downtime of distributed systems are catastrophic. A survey of 450 Fortune 1000 companies found the mean loss of revenue due to an hour of network outage was $82,500, with financial institutions being in the higher end of the curve with downtime costs of $6M/hour[32]. 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.

This motivates the need for building distributed systems that are capable of tolerating disruptions and providing highly available and correctly functioning services. The challenges to providing this in today's distributed systems 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 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 research, we propose a Generic Monitor approach to detect disruptions in the protocol. Disruptions is the term coined for collective set of errors and intrusions which cause discrepancies in the operating protocols. We propose a Monitor-based solution to detect disruptions that can be applied to a large class of distributed applications. The solution approach employs a Monitor that snoops on the communication between application modules (also referred to as, protocol participants) and performs matching of the observed communication against a rule base which characterizes acceptable protocol behavior. The research addresses the following critical issues:

The Monitor should not become a performance bottleneck. The Monitor functions asynchronously to the application protocol and runs on independent hosts, not those of the participants.

The Monitor should be scalable. We propose a Hierarchical Monitor structure. 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.

The Monitor should be generic and widely applicable. The design of the Monitor is generic and applicable to message passing based distributed protocols. The Rule Base is specific to the application. The rule base is specified in a commonly understandable

temporal and combinatorial logic format. The monitor is designed to take the rule base as input and partition it into local and global rules according to the deployment.

The Monitor should have low latency of detection. This is ensured by partitioning the rules intelligently into local and global rules, which reduces the number of rules to be matched at each monitor. Highly speed optimized matching algorithms are designed for matching the temporal and the combinatorial rules. The matching algorithm uses multiple threads and therefore can leverage any concurrency available in the host.

We extend the Monitor approach by developing Hierarchical Monitor based detection system. In the hierarchical structure there are several Monitors placed at different logical levels in the system making a hierarchy of local, intermediate and global monitors. The Local Monitors directly snoop on the messages exchanged between the protocol participants. The Intermediate and Global Monitors are invoked for interactions that span across the hosts monitored by a local monitor. The Intermediate and Global monitors observe only messages forwarded by the local monitors and therefore perform rule matching on a subset (hopefully, a small subset) of messages. Each of the Intermediate and Global Monitors have several Local Monitors working on a subset of entire nodes which they are monitoring respectively. Diving the entire space under several Local and Global Monitors reduces the number of rules matched at each level. It helps in reducing the detection of latency because of reduction of load and also improves in coverage of the detection system because not all interaction patterns in distributed systems are local.

The Monitor-based approach is demonstrated on two real world distributed applications - A reliable multicast protocol called *TRAM* and a control protocol for managing sessions called SIP. In the next chapter we explain the protocols TRAM and *SIP*, their characteristics and how these protocols suffer from attacks. In chapter we demonstrate an orthogonal way of making a protocol robust – by augmenting the protocol with carefully designed extensions and embedding the extensions within the protocol with non-trivial code additions. We demonstrate the methodology on TRAM and come up with a new protocol called TRAM++ which is resilient to malicious and slow receivers and reduces the buffer requirement of the system as well. Chapter 4 discusses the monitor based detection approach. It describes the Monitor architecture and rule classification. Implementation and results form chapter 5 with system details included. Chapter 6 discusses the related research. Finally we conclude in chapter 7 and provide directions for future work.

2 SYSTEM DESCRIPTION

The networked systems are forming an integral part of human lives causing increased reliance on distributed computing. Several support systems and even critical life systems rely on these distributed protocols. Hence the need to make these underlying protocols robust is imperative and not just necessary. We look at the two protocols that are in wide use in building information technology infrastructures, are deployed in critical environments, and have a distributed nature. The two protocols are the Session Initiation Protocol (SIP) and the reliable multicast protocol called TRAM. These are described below.

2.1 Session Initiation Protocol (SIP)

It is an application layer control protocol for creating, modifying and terminating sessions involving internet telephone calls, multimedia distribution and conferences between two or more participants. SIP is an initiation protocol which helps the clients to agree on the characterization of the session which will exist between the two. It is not a vertically integrated system nor does it have any network reservation capabilities. But it does provide some security methods like DoS prevention, authentication, encryption etc. SIP supports the five facets namely :

1. User location
2. User availability
3. User capabilities
4. Session setup
5. Session Management

SIP message or packet consists of a method name, request URI, protocol version. The method name is the type of the message carried by the packet. The various type of messages which exist in SIP are REGISTER, INVITE, ACK, CANCEL and BYE. The status codes represent the type of response and fall in 6 categories.

I. 1xx: Provisional -- request received, continuing to process the request;
II. 2xx: Success -- the action was successfully received, understood, and accepted;
III. 3xx: Redirection -- further action needs to be taken in order to complete the request;

IV. 4xx: Client Error -- the request contains bad syntax or cannot be fulfilled at this server;
V. 5xx: Server Error -- the server failed to fulfill an apparently valid request;
VI. 6xx: Global Failure -- the request cannot be fulfilled at any server.

SIP initiation causes a message to go to the server which looks for that particular client, if its available then the request is forwarded otherwise the request is forwarded to the proxy server which looks for the requested client. A detailed description of the protocol can be found at [34][35][36].

2.1.1 Potential Threats

The protocol is designed to initiate a session and manage it but it suffers from several vulnerabilities. Some of them are listed below along with the word description of the rule to detect the attack.

1. *Repeated Calling and Hanging Up:* The client repeatedly initiates a request with the proxy and then cancels it, thus launching a DoS attack against the proxy. Time between receipt of any 1XX response and sending of 'Cancel' message cannot be less than 't' where 't' is the Expire time specified in the invite header.
2. *Babbling:* A client tries to set up various sessions with the proxy with out killing the previous sessions. .The number N of invite messages in any time duration t with different *CSeq* numbers cannot be more than n, where n = floor(t/expire).

$$0<|M_i|<n \;\; t \in (t_i, t_{i+k}) \;\; k \text{ is } \infty \text{ and } i=0,$$ where M_i is number of invite messages

3. *Malicious session termination through SIP ID spoofing:* A malicious user can spoof the ID of a valid client and send a false Bye message. It can be prevented by ensuring that a 200OK/487 response to a false BYE message must not be initiated by that particular user agent indicating compromised IP.
4. *Compromised Clients:* A compromised client can send various message request with the same session id. Avoid it by ensuring that clients cannot send consecutive requests with different header/message body without changing the sequence number of the request.
5. *Repeated Connecting and Hanging up:* A client can repeatedly call and then terminate the session. A rule to avoid it should prevent a next transaction immediately after a 2XX receipt cannot be a BYE more than 'n' times in time duration t.
6. *Client Error:* A client can send the same request of the session once the termination message is received. So one should ensure that after receiving a 4XX

message from a particular domain, CUA should not resend the same request without modifying request and changing *CSeq*.

7. *Compromised Proxy Server*: A compromised proxy can send malicious responses to client instead of sending the original response forwarded by other proxy. Hence if a proxy server receives a 2XX response to a request that it had sent, it has to forward the same response upstream, and cannot generate a non-2XX response.

   $\forall T \in (t_N, t_N + k) \Rightarrow V_T \Rightarrow U_q \; q \in (t_I, t_I + b)$ where V_T stands for receipt of 2XX response, and U_q stands for passing on response upstream.
8. If proxy does not generate response to request received, it is faulty

   $T : Vt_i \neq Vt_{i+\Delta}$ where Δ is the expire time in the request header.
9. *Looping:* It could happen that the proxy generated too many responses which are circulating in the system. A proxy cannot generate 'n' 482 responses in time period t, where n = no. of requests serviced in time period t and this number is greater than a threshold over a reasonably wide time interval.

   $\forall t \in (t_i, t_{i+k}) \; L \leq |V_t| \;\; \Rightarrow \;\; L' \leq |B_q| \leq (n\text{-}1) \;\; \forall q \in (t_i, t_{i+k})$ where V_t is requests generated in time t_i to t_{i+k}, and B is 482 responses generated.

These are some of the vulnerabilities that SIP inherits because of a design which is not robust.

2.2 Tree Based reliable Multicast Protocol (TRAM)

IP Multicast is the basic multicasting framework which exists in the internet. This multicast is unreliable and earned a bad name because of initial problems like large network bandwidth usage and lack of underlying support mechanism. At the other front Reliable multicast protocols are important classes of protocols which reliably disseminating information from a sender to multiple receivers in the face of node and link failures. Guarantee of packet delivery make them important over the simple unreliable IP Multicast. A Tree-based Reliable Multicast Protocol (TRAM) provides scalable reliable multicast by grouping receivers in hierarchical repair groups and using a selective acknowledgment mechanism. The detailed description of TRAM can be found in [3][4]. TRAM is distributed as a part of the Java Reliable Multicast Service (JRMS) by Sun Microsystems [10]. JRMS is a set of libraries and services for building multicast-aware applications.

2.2.1 TRAM Protocol Features

TRAM ensures a reliability of packet delivery in case of network and node failures as long as the sender has sent that packet and received by at least one node. It ensures this reliability by placing Repair Heads (RH) at intermediate locations in each LAN for local repair. If a receiver in a particular LAN loses a packet then it asks for its local Repair Head for repair of that packet. The entire structure is formed like a TREE with the sender as root. RH's form the intermediate repair nodes with receivers or multiple RH below each. Each RH is responsible for local repair in its region. This makes the protocol scalable as each RH is only responsible for a small group of receivers. The RH's are dynamically chosen within each LAN called the LAN Head. In each LAN a node is chosen to serve as RH for the local nodes. If RH dies or goes dysfunctional then another node from the same LAN is made the RH.

2.2.1.1 Ack Implosion :

The local repair heads are also responsible for Ack processing of the nodes under them. Each repair head sends a cumulative ack for all the nodes under them to the RH up above and so on until it reaches the sender. The sender deletes the packet from buffer only if all the nodes below (one level below) have acked the packet. This policy is followed by all RH's as well i.e. until all the nodes ack the packet its not deleted from the RH's local buffer. Since all nodes are not responsible to send ack's to sender this prevents the Ack implosion problem and sender doesn't get bogged down by too many acks. TRAM removes the problem of Ack implosion.

2.2.1.2 Tree Formation:

TRAM has various options which can be set to do tree formation. One can do an optimized LAN formation in which if a sender is in some other LAN and the receivers are in another LAN then a node is chosen in that LAN to serve as repair head to the local receivers. TRAM tries to place RH's optimally so that inter LAN traffic is minimized. A suitable LAN can be chosen by looking at the TTL fields of the packets. TRAM has different tree formation for unidirectional multicast and bidirectional multicast. For our study we have chosen unidirectional multicast mechanism. It assumes that only sender can do a multicast to the receivers and not other way round. Sender initiates the process of Tree formation by sending a Beacon message. The nodes interested in the data respond by sending a Head Bind message. The sender (or repair head) responds by sending an

Accept member or reject member response. A receiver could be rejected because the sender might have got a large number of Head bind request which it is unable to acknowledge.

2.2.1.3 Ack Mechanism:

TRAM is designed for high scalability targeted towards multicasting streaming data from a single sender to a large number of receivers. TRAM ensures reliability by using a selective acknowledgement mechanism. An ack is sent in the form of an offset and a bit vector once every ack window (32 packets). Ack is only sent to the next level repair head (could be sender as well) which does ack accumulation. . Every ack message contains a start message number indicating the first missing message, and a bit vector, with a 1 denoting a missing packet and a 0 denoting a received packet. If no packets are missing, the message number indicates all messages prior to and including this one has been received and the bit vector is of zero length. An ack message is sent after every *ack window* worth of packets has been received, or an *ack interval* timer goes off.

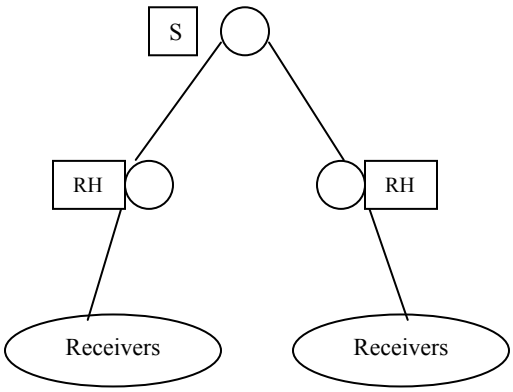


Figure 1:A simple TRAM build tree with sender at root, Repair Heads at the intermediate level and receivers as leaves.

The Figure 1 shows a simple TRAM tree with sender at root, RH at the middle level and receivers at the leaf level.

TRAM provides scalability by adopting a hierarchical tree-based repair mechanism. The receivers and the data source of a multicast session in TRAM interact with each other to dynamically form repair groups. These repair groups are linked together in a hierarchical manner to form a tree with the sender at the root of the tree. Figure 1 shows a typical TRAM repair tree. The nodes participating in TRAM play three roles, some nodes playing multiple roles – sender, receiver and repair head (RH). Every

repair group has a receiver that functions as a group head; the rest function as group members which are said to be affiliated with their head. All members receive data multicast by the sender. The group members report lost and successfully received messages to the group head using a selective acknowledgement mechanism. The RH's maintain a high and low water mark for monitoring cache occupancy. If the amount of buffer occupied by the packets goes beyond the high water mark, an attempt is made to purge the cache. Failure to do so is taken as an indication of congestion in the network. The RHs aggregate acks from all its members and send an aggregate ack up to the sender to avoid the problem of ack implosion. The data rate sent out by the sender is bounded by maximum and minimum rates configured at the sender. Receivers that cannot keep up with the minimum data rate can be pruned from the repair tree.

2.2.1.4 Flow Control

TRAM incorporates rate based flow control to control the flow of packets and prevent cascading effects. Each RH and sender has a high and low water mark. When the buffer value reaches the high water mark any new packets are discarded and acks are demanded from the receiver below to get the buffer empty. Entities below also report the congestion above to the sender by setting their congestion bit in the flags. Congestion is detected on the basis of missing packets.

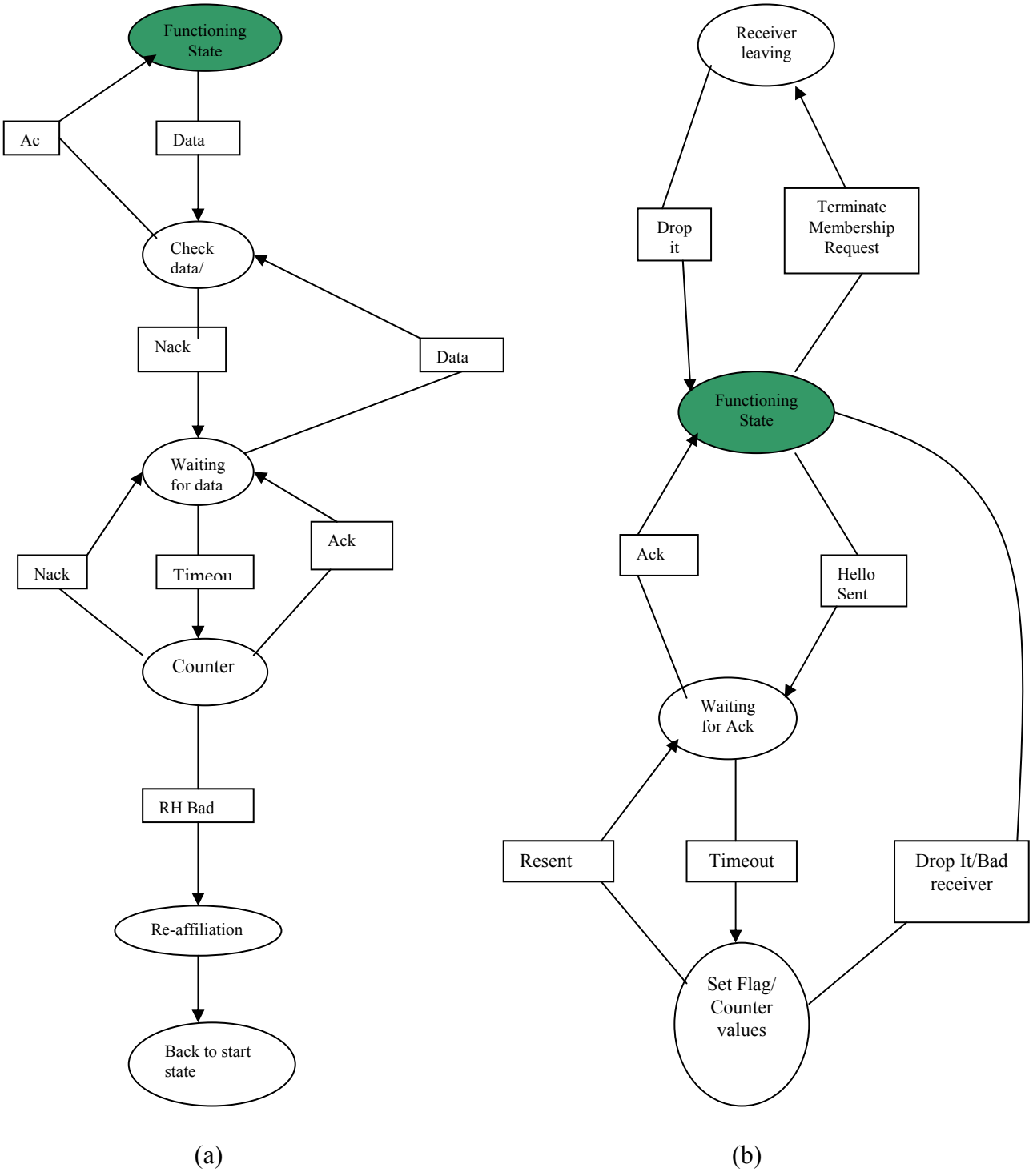


Figure 2: The above state diagrams represent (a) Receiving Module of a Repair Head (b) State Transitions caused by Hello Messages.

2.2.2 Lacking Security and Robustness in TRAM

TRAM is designed without security and with only simple crash failures in mind. It assumes that every receiver will be behaving according to the protocol and sending acks at regular rate. But in a real system and wide deployment different nodes have different real time constraints because they might be lying in completely different LAN's with different available bandwidths. In TRAM a single malicious receiver could withhold the acks. This will inhibit the repair head up above to send acks and the effect ripples up to the sender bringing the whole system's performance down. Also all the repair heads keep a copy of the data maintaining a huge buffer and it is maintained at all the RH's including sender. If a single node withholds the acks then buffer for all the RH's in the path to sender gets filled up to the high water mark. A simple application of video on demand using this protocol will make it inefficient if one receiver is slow because that will cause jitter in other receivers video as well.(because the entire tree data rate is governed by the slowest receiver.) We propose an efficient protocol called TRAM++ which is based on TRAM but is resilient to slow and malicious receivers and efficiently manages buffer in the system. Some of the other potential threats to the protocol are as follows :

1. Buffer Overflow can be caused by withholding the acks by a single receiver .
2. A malicious node can cause repeated LAN optimizations in TRAM by volunteering for being Repair head and then resigning later.
3. A receiver can malfunction by sending repeated Nacks.
    a. $0 \leq |N_{rate}| \leq N_{max}$ $t \in (t_i, t_i+k)$ k is an integral multiple of the Ack Windows. where $N_{max} << N_{individual\_max}$ * *Max number of members*
4. A malicious repair head could feign overloading by not accepting any more members under it.
5. A receiver could feign congestion by repeatedly sending packets with congestion bit set.
6. A bit flip could cause error in the height variable of each node which might lead to inappropriate affiliations by new members.

Items 1-5 can be caused by malicious entities or an unfortunate sequence of natural errors. It is impossible to determine intent, e.g., if there is a transient condition at the repair head which causes it to detect wrongly that there is overloading, or it is being malicious. In our system, there is no cause to distinguish between the two. Item 6 is caused by a natural error, but it is conceivable that incorrect tree height may have been sent in by an entity keen to disrupt the multicast tree formation.

All the above disruptions make TRAM non-dependable and motivates a generic solution to improve dependability.

3 Making Protocol Robust : TRAM++

Making a protocol reliable and fail safe requires knowing the vulnerabilities that a protocol suffers from. In the previous chapter we looked at two example protocols namely SIP and TRAM which are widely deployed and popular in the research community for the standardization and availability of source code. We described a few potential threats to the protocol which hamper their deployment for in critical applications. In this chapter we have analyzed TRAM for one vulnerability and one inefficiency and propose an augmented protocol called TRAM++[33].

3.1 TRAM++

TRAM++ builds upon TRAM with the following two goals

1. Handle slow or malicious receivers in the environment while localizing their effect on correctly functioning receivers.
2. Reduce the resource requirement at the repair heads, chiefly cache utilization, but also processing.

To achieve these goals, TRAM++ introduces the changes described below. A figure of the hierarchical structure in TRAM++ with a sender, receivers and repair heads is shown in Figure 3(b).

3.1.1 Buffer management at RH:

The design point in TRAM++ is that the RHs may be spread over a wide area and have constraints on available buffer, while the sender has higher, though not infinite, buffer capacity. TRAM++ optimizes the buffer requirement at the RHs by pruning old messages even if they have not been acknowledged by all its receivers. The advantage is that this frees up the buffer resources at the RH for accommodating new messages which are required for the well-behaved receivers to make progress. Consequently, a nack from a receiver may not always be satisfied locally at the immediate RH. A message is not discarded from the sender's storage till it has been acked by *all* the receivers. Therefore, a nack can always be satisfied by the sender. When a RH cannot satisfy a nack, it indicates to the receiver to initiate a *temporary re-affiliation* with a RH at a higher level. This is shown through the dotted arrow in Figure 3(b), where the receiver re-affiliates temporarily for recovering the messages its RH does not have. Reaffiliation is transient

and lasts for the duration of recovery of the single packet. This process is repeated recursively if recovery is not successful at the higher level, till the receiver reaffiliates with the sender at which point its nack is guaranteed to be satisfied.

3.1.2 Handling Slow or Malicious Receivers:

TRAM lets the data rate be driven by the slowest receiver. Therefore, the effect of a slow receiver is visible to the correctly functioning receivers all across the network. Even if the pruning feature of TRAM is turned on (which it is not for most deployments), the threshold minimum tolerable data rate is set quite conservatively and there are likely to be large periods of slowdown to the normal receivers. On the contrary, TRAM++ localizes the disruption to the part of the tree where the lagging receivers reside. In TRAM++, the RH uses two types of acks – a *greedy ack* and a *permanent ack*. The maximum sequence number of the packet that the sender should send down is sent piggybacked with acks. The greedy ack is sent by the RH upwards when its buffer reaches the low water mark. The purpose of sending the greedy ack is to indicate to the sender to send new data down even though all the receivers have not acked yet. The permanent ack is sent once all the receivers have acked. The role of this ack is to let the sender know that reclamation of storage is possible. In TRAM, the sequence number sent upwards is determined by the slowest receiver thus affecting the data rate observed by all the receivers, normal or laggard. In TRAM++, the sequence number is determined by the RH's available buffer capacity. Incorporating the additional ack requires additional computation at the sender and the RHs which is the same cost as for the basic ack determination in TRAM. But in a failure free system where all receivers are keeping up with the data rate, the greedy acks are not sent and therefore, the additional processing overhead is not observed. TRAM++ has the functionality to prune receivers which are considered lagging beyond an acceptable degree. The metric used for the pruning decision is the percentage of retransmission requests which cannot be locally satisfied as a fraction of the total number of packets. When the metric exceeds a tunable threshold parameter, the receiver is pruned. This is an effective means of removing malicious receivers which may increase the processing load in the system by requesting repeated retransmissions. This may serve as an indication to the receiver to disassociate from the current RH because of its resource constraints and reaffiliate with a more resource rich RH.

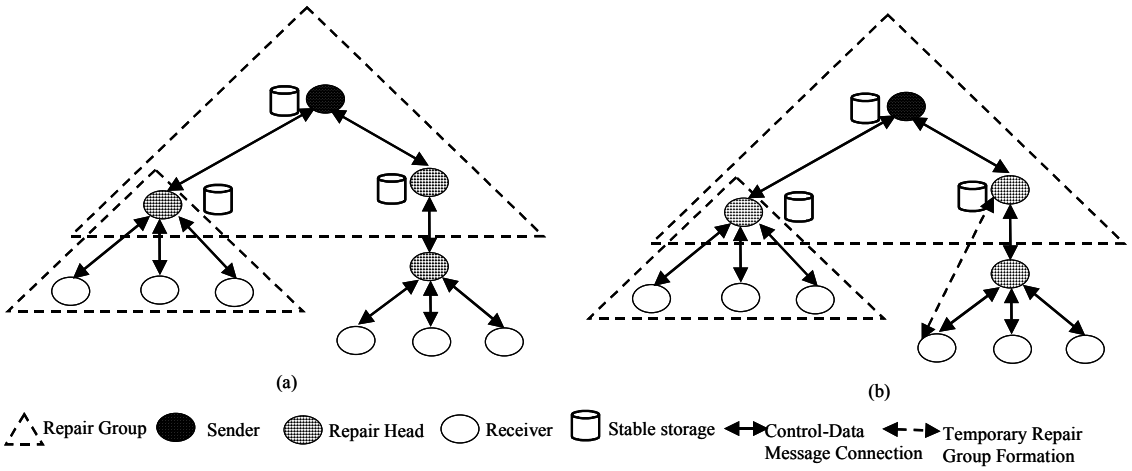


Figure 3: (a) Shows the TRAM interrelationships amongst the receiver, RH's and Sender; (b) Shows the interrelationship amongst the receivers, RH's and Sender in TRAM++

Figure 3(a) shows a TRAM deployment with a sender, two levels of RHs and multiple receivers connected through links over which bi-directional data and ack messages flow. Two examples of repair groups are shown, one involving the sender and the three RHs at the first level, and the second showing a RH and its receivers.

3.2 TRAM Implementation

The TRAM code is multi-threaded. These threads are responsible for carrying out the group management functions in addition to basic sending and receiving of data packets. *GroupMgmtThread* is the main thread which is responsible for starting up TRAM, initiation of the *beacon messages* by the sender and affiliation of the receivers to the senders or repair heads. The beacon messages are used to advertise the session and invite nodes to join the multicast session. This thread performs the task of sending periodic hello messages among the receivers and its head. Each receiver also maintains a backup list of heads which it can switch to if the current head resigns or fails. Once the data transmission phase starts, *InputDispThread* and *OutputDispThread* come into picture. *OutputDispThread* transmits the packets. *InputdispThread* gives the packet to all the listeners and hence, each entity calls its received packet method to get the desired packet. The sender and the repair head's sending functionality use *HeadAck* class to receive ack packets. The receivers use *MemberAck* class to receive data packets and to send acks. Repair head uses *MemberAck* class, as it is a receiver for the sender above, to send cumulative acks. Figure 4 shows pictorially the threads or methods which are used for upstream and downstream communication in TRAM. In the case of errors, the

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

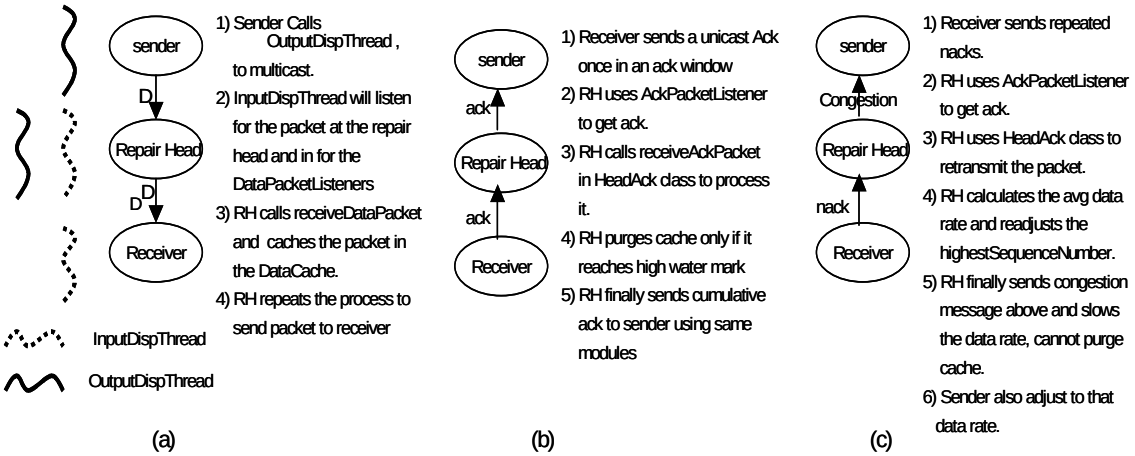


Figure 4.TRAM message processing (a) downstream with no errors, (b) upstream with no errors, (c) upstream with message, node or link errors

3.3 Modifications for TRAM++

To create TRAM++ from TRAM, we have added new message types to the existing message and sub-message types of TRAM. We have tried to minimize the changes to the basic TRAM structure and be able to add a separate layer of functionality which transforms TRAM to TRAM++. The processing of messages downstream and acks upstream in the no error scenario is shown in Figure 5

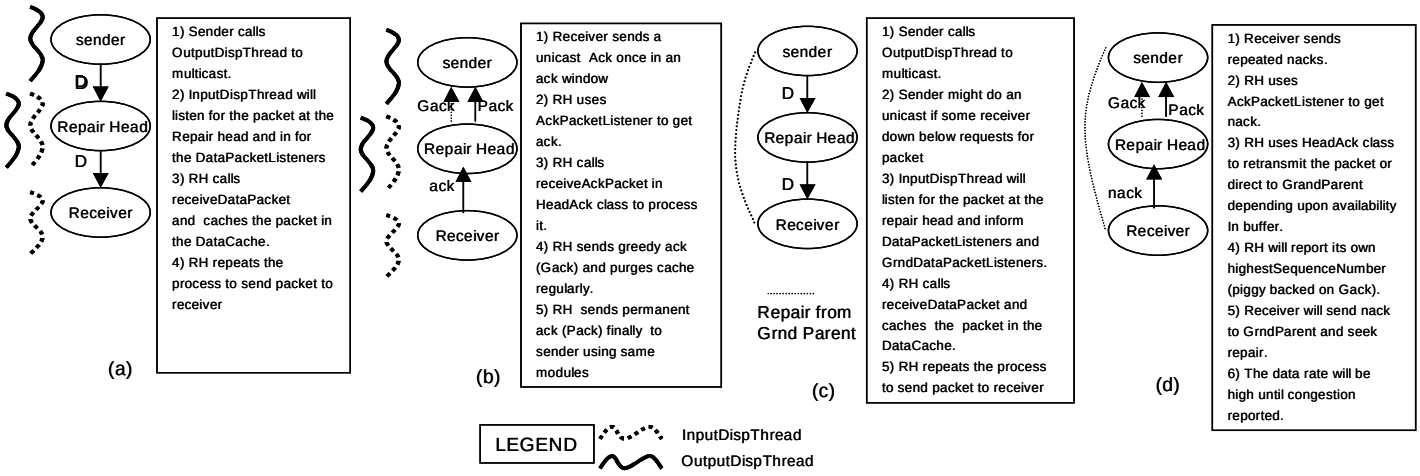


Figure 5.TRAM++ Implementation. Without errors: (a) Downstream. (b) Upstream. With errors: (c) Downstream. (d) Upstream.

Recollect that a RH may not be able to satisfy a receiver's nack. In order for the RH to indicate to the receiver that it doesn't have the packet in cache, we have introduced a sub-message type called NOT_REPAIRED. This sub-message type forces the receiver to go for *temporary reaffiliation*. The data packet sent by the RH in response to the above type of nack, has a payload which only contains information about the higher-level repair head, also called *grand repair head (GRH)*. The receiver uses this information and sends a request for the missing packet to the new GRH. The GRH unicasts the requested data packet to the receiver. In TRAM, only multicast data exists so there is no notion of sending data through unicast. To enable this, we added a message type called UCAST_DATA. A flag is added to the ack packet of TRAM to differentiate between greedy and permanent ack types.

The ack processing mechanism of TRAM is modified for TRAM++. In TRAM++ as the ack (implicit nack) is received by the repair head, it checks which packets the receiver has asked for retransmission. If the packet is not in the cache, the repair head sends a packet to the receiver giving it the information that the requested packet could not be repaired and provides it with the address and port of the GRH. The execution steps of TRAM++ with errors are shown in Figure 5. The downstream processing is identical to the TRAM case except that the sender may send a unicast if some receiver had directly requested a retransmission through the temporary reaffiliation process.

3.4 Experiments and Results

TRAM and TRAM++ are installed on a campus-wide WAN and experiments are conducted on the test-bed. The experiments have two broad goals:

- Evaluate the scalability and robustness of TRAM with respect to different types of message errors.
- Evaluate the improvement provided by TRAM++ in the event of failures and overhead incurred by TRAM++ under failure-free conditions.

3.4.1 Test-bed Setup

The test-bed has a sender, multiple repair heads and varying number of receivers. The computing machines are distributed across campus in the Mathematics (clusters *al-zn*), Electrical Engineering (named *pegasus* and *lyra*, together with its cluster) and Materials Science (*msee190*) buildings. The machine named msee190 is always used exclusively as the sender. All the machines run RedHat Linux. It is important to note that

the implementation is done and the experiments performed on a production campus-wide wide area network with normal traffic coexisting with the reliable multicast traffic.

MSEE machine (Sender)	Pentium IV 1.5 GHz machine with 1 GB of memory
EE machines	Pentium 4 2.26 GHz processor with 1 GB memory, 533 FSB and 512 KB cache
Math machines	450 MHz Pentium II machines with 256 MB of memory divided into 13 clusters with 48 machines each
Rtr_{MSEE}	Cisco 5505 switch
RtrMATH	Cisco 6509 switch
Links	Intra-cluster links 100 Mbps, inter-cluster links 1 Gbps

Table 1. Hardware configuration

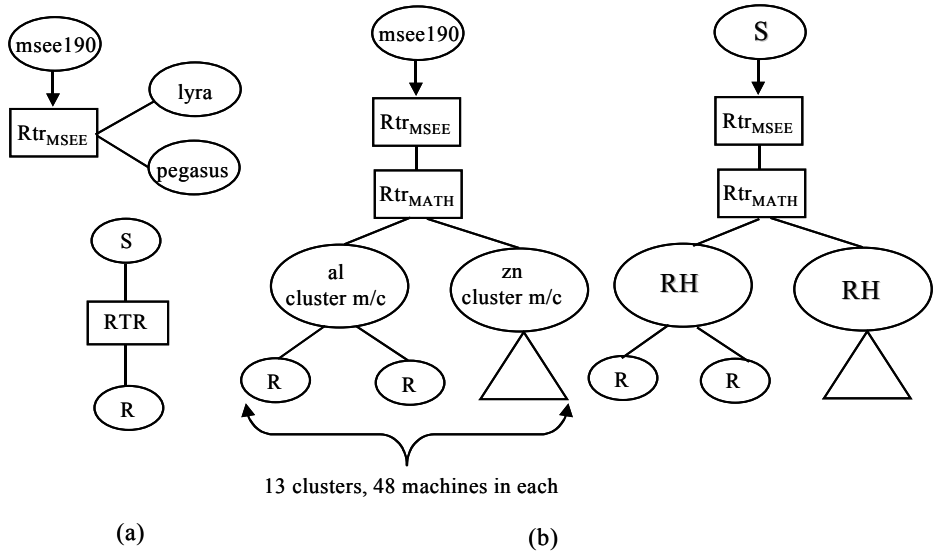


Figure 6. Sample physical and logical configuration for deployment with (a) 2 hops (b) 4 hops between sender & receivers.

Different experiments are conducted with the receivers at different hops distance from the sender. In Figure 6, we show the physical and logical views of two different hop topologies. The physical topologies show the routers as well as the protocol participants (sender, receiver or RH), while the logical topologies only show the participants. A *hop* is defined as a link in the physical topology. This is different from the traditional definition of TTL in networking. For example, traversal from a receiver within a Math cluster machine to its RH in the same cluster does not decrement the TTL, but is considered a hop. We felt this to be reasonable because there is a cost of processing both at protocol participants and at the routers.

We use the reliable multicast infrastructure to send a high bandwidth Mpeg-2 video data feed from the sender to the receivers. A 40 kbps feed is sent in 1316 bit payload packets, leaving space for the TRAM and the IP headers to fit within 1500 Ethernet MTU. The number of packets sent is at least 8000, with a larger number if the initial transients, which are discarded from our results, are longer. The parameters of the protocol set at the participants are shown in Table 2.

Parameter	**Value**	**Parameter**	**Value**
(Max, Min) data rate	(40 kbps, 1 kbps)	Stable storage size at RH (TRAM) Low water mark: High water mark (num packets)	1200 packets 400:800
Stable storage size at RH (TRAM++) Low water mark: High water mark (num packets)	200 packets 32:120	Pruning of receiver (% of packets asking for reaffiliation) (TRAM++)	0.08%

Table 2 TRAM and TRAM++ configuration settings

3.4.2 Output measures

The output metrics studied in the experiments are shown in Table 3 along with a brief interpretation of each.

Latency	End-to-end measure between sender and receiver. Clocks on the machines are synchronized using the *nntp* service.
Jitter	The difference in latencies between successive packets. This metric is important for a smooth video playback.
Data rate	The data rate computed at the sender which should lie between the max and the min data rates specified. If a slow receiver tends to reduce it beyond the limit, it will be pruned.
Buffer utilization	Stable storage used at the sender or the RH (which one it is will be clarified in the context).
Memory utilization	The amount of main memory used by the RH process. This is taken as an indicator of the processing resources needed at the RH since the processor utilization stays very close to zero.

Table 3. Output metrics used in the evaluation

3.4.3 Normal and Error Injection Runs

A single run of the experiment is defined as the transfer of at least 8,000 packets of the video feed. A normal run is one where no errors are injected, though the variability

of the environment may create congestion and transient instability such as spikes in latency. In the error injection runs, three kinds of message errors are simulated in the network – drops, delays and reordering. The first type of error is message drop where a message is dropped on the downward link between the RH and a single receiver. In one error injection run, a single receiver is identified as the faulty (or, malicious) one and the error injector works on its link. For simulating message delays, the ack packet from the faulty receiver is delayed on the upstream link to the RH. For injecting message reorderings, we vary the inter-message gap (M_d) between the messages which are to be reordered. Recollect that the ack window (W_a) is the number of packets for which one ack gets sent. If $M_d < W_a$, then TRAM buffers the out-of-order packet and delivers it when an ordered sequence of packets can be created. This results in behavior identical to the error free case since the element in the ack bit vector is reset to 0 when the message is delivered in-order. If $M_d > W_a$, then the receiver sends a nack in the next ack window for the message whose place was taken by the out-of-order packet. This is identical to the message drop case and hence we do not separately show results for the message reordering case.

3.5 Evaluation of TRAM

3.5.1 Error-free Case

In this experiment, we investigate how TRAM scales with the number of receivers. The latency curves in Figure 7 (a) show that the latency is substantially less for the single hop case since the message is on a direct connection and does not have to traverse a router. The latency for 2, 3 or 4 hops is comparable. None of the latencies become substantially worse with increasing number of receivers. The jitter for our environment is found to be below 1 ms on an average which is lower than the granularity of our timing mechanism.

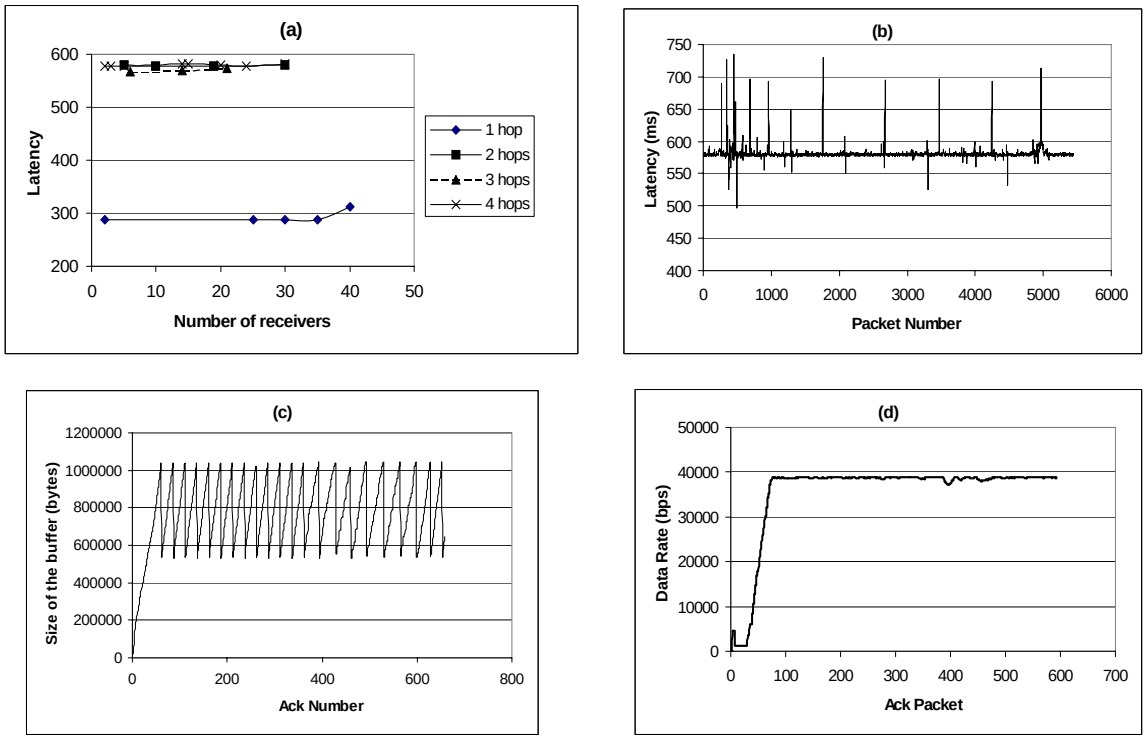


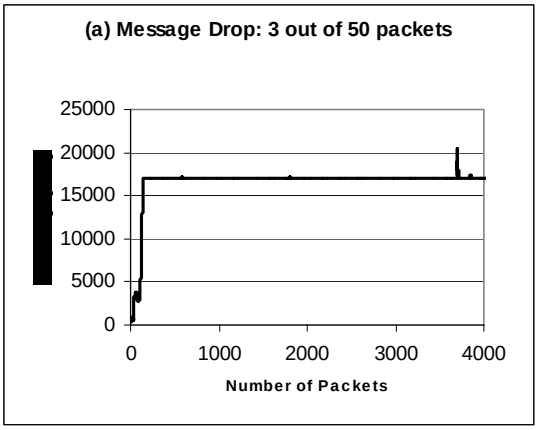
Figure 7: Latency and Buffer Utilization in TRAM for error free run.

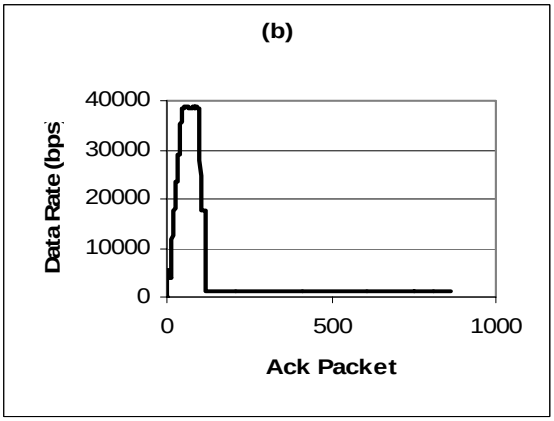
The next experiment investigates the buffer utilization at the RH and scalability of the protocol with the number of receivers shown in. TRAM is set up to use one-thirds of maximum buffer capacity (1200 packets) as the low water mark and two-thirds as the high water mark. Therefore, the buffer utilization oscillates between 400 and 800 packets, i.e., between 526.4 KB and 1.0528 MB Figure 9. This is found to be true as the number of receivers is increased and is also independent of whether the utilization is at the RH or the sender. When we measure the data rate against time (or equivalently the number of the ack packet), it is found that in the normal case, after the initial transient, the data rate picks up and tends towards the max data rate specified in the configuration.

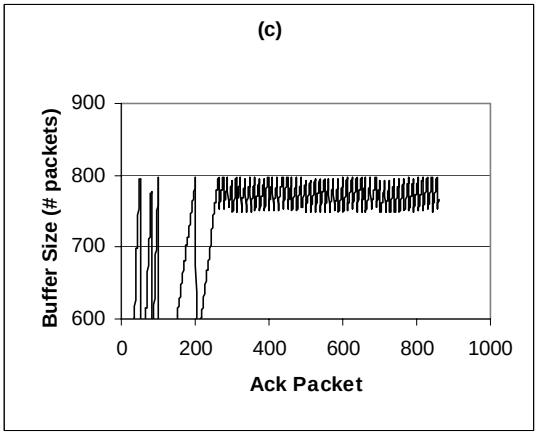
3.5.2 Error Injection Case

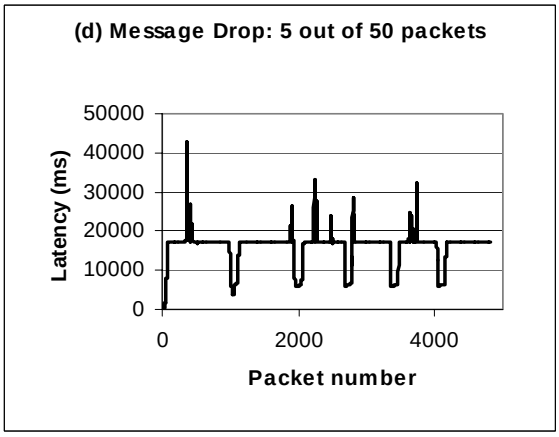
For TRAM, the effect of the error on the faulty receiver and the normal receiver will be identical, and therefore no distinction is made in the presentation. Figure 8 shows the variation of latency with the number of receivers for different message drop rates. The drop rates considered here are 1 every 50 packets, 3 every 50 and 5 every 50. The dropped packets are all consecutive. Figure 8(a), (d) show the variation of latency and Figure 8 (b), (e) show data rate variations against time for 2 different packet drop rates –

3 out of 50 and 5 out of 50. The buffer utilization at the RH (Figure 8(c),(f)) shows an interesting trend. The purging of the buffer begins when its size reaches the high water mark (800 packets), and while under error-free conditions, the purging would have been able to reduce the buffer utilization to 400 packets, here the reduction is only down to about 750 packets. If the drop rate is increased to 5 out of every 50 packets, the affected receiver is pruned. On pruning, the buffer is purged completely since the packets were being buffered to accommodate the receiver that just got pruned. Therefore the utilization comes down to zero. Immediately after pruning, the sender tries to increase the data rate and the utilization goes back to the usual oscillation between 400 and 800 packets till the pruning happens next.









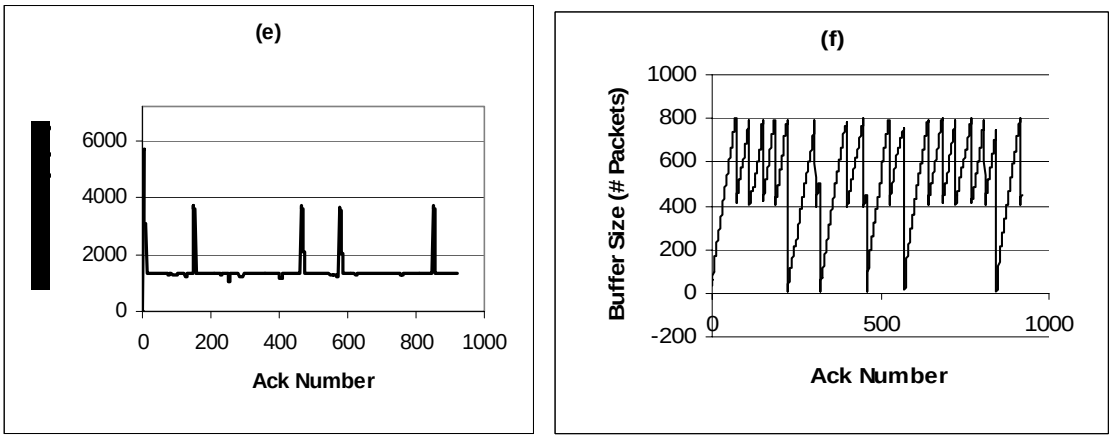


Figure 8. Effect of message drop on TRAM: (a)&(d) Latency for 3 and 5 packets dropped out of 50, (b)&(e)Respective Data Rates, (c)&(f)Respective Buffer Occupancy

In Figure 11, we show the effect of introducing message delays in TRAM. In the first experiment, a delay of 8,000 ms is introduced. The latency shows a regular spike of 8,000 ms and this spike repeats roughly every 32 packets since the ack window is set to 32 and a delay is introduced for every ack. The data rate shows congestion control at work. The sender tries to increase the data rate to the max rate (40 kbps), but is forced back because of the delayed ack. It reduces the data rate, but is able to sustain a rate greater than 1 kbps, and as a result, the receiver does not get pruned. The buffer utilization varies between the low and high water mark as in the error free case (Figure 7(c)) and is therefore not shown here again. It is found that a delay of 10,000 ms causes pruning of the slow receiver. Thus, it is seen that in TRAM, once pruning of misbehaving receivers happens, the remaining receivers continue to see performance as in the error free case.

3.6 Evaluation of TRAM++

3.6.1 Error-free Case

The results of the scalability test of TRAM++ are shown in Figure 10. It is observed that the protocol is scalable like TRAM in the range under consideration (5-30 receivers). The variation in latency in this range is about 1.01%. The average latency in the range is 584.44 ms, which gives a 3.2% overhead over TRAM. As the buffer constraint is varied in TRAM++, the relative overhead is found to remain constant in the error free case. This is expected since the buffer is not completely utilized in the absence

of errors. The overhead is ascribed to three main reasons: extra message schema matching due to the introduction of new message types, aggressive cache pruning to satisfy the storage constraints at the RH, and additional control messages – two kinds of acks being sent upstream by the RHs.

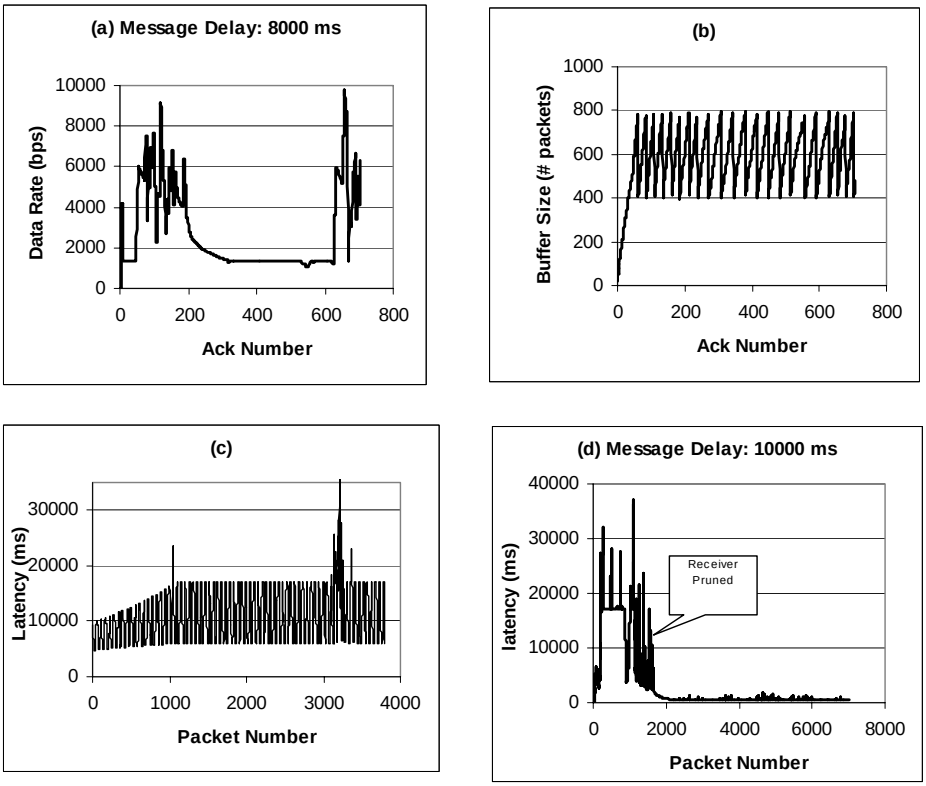


Figure 9. Effect of message delay on TRAM: (a)-(c) – 5 receivers, delay of 8000 ms, no pruning; (d) – 2 receivers, delay of 10000 ms, pruning

Regarding the comparative resource usage, the CPU utilization is very close to zero for the maximum data rate that the network infrastructure can support and therefore cannot form a meaningful point of comparison between the two protocols. The main memory utilization for both TRAM and TRAM++ vary between 7.5% and 30.0%. The sender buffer utilization curve shown in Figure 10(b) oscillates between 400 and 800 packets as in TRAM's buffer utilization. In TRAM++ a maximum of 5 reaffiliations are allowed per receiver. If a receiver tries to reaffiliate more than that, it is pruned assuming malicious nature. The RH buffer utilization varies between the maximum pruning level (32 kB) and the high water mark (120 kB) and is shown in Figure 10(c). In the no error case, this buffer utilization does not depend on the number of receivers. A system

designer can set the buffer upper bound knowing the stable storage constraints and TRAM++ will operate under the bound. The data rate supported by TRAM++ also approaches the max data rate parameter set in the system (40 kbps).

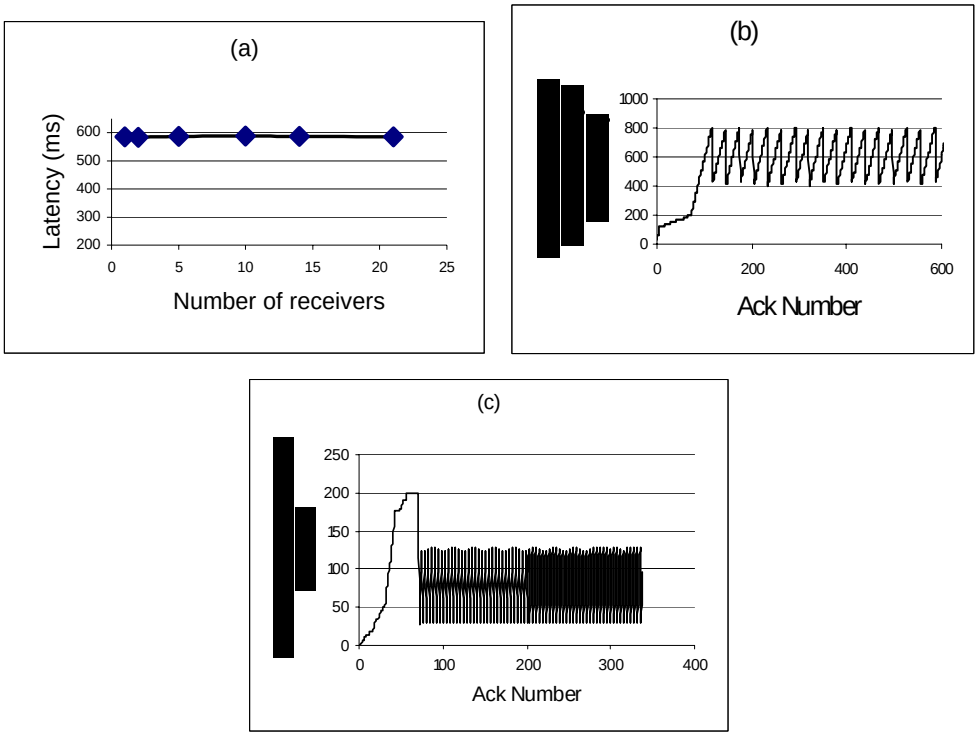


Figure 10. Evaluation of TRAM++ in error-free case: (a) Receiver latency (b) Sender buffer utilization and (c) Repair Head buffer utilization for 27 receivers

3.6.2 Error Injection

For the error scenarios in TRAM++, the metrics are expected to be improved for the normal receivers compared to the faulty receivers.

For message drops, we conduct two sets of experiments, one with 5 packets out of 50 being dropped, and the second with 2 out of 50. For the 5 of 50 case (Figure 11), it is observed that the malicious receiver gets pruned at around 650 packets, after which the system behaves as in the error free case. The sender's buffer utilization once drops to zero because of purging of entire cache at the time of pruning. Then the utilization oscillates as usual between 400 and 800 packets, and the data rate also tends towards the maximum. However, the buffer utilization at the RH goes up to the maximum buffer space, before purging occurs. For 2 out of 50 packets being dropped (Figure 12), the receiver is repeatedly pruned and rejoins the multicast group. The normal or non-faulty

node is not affected at all and its latency remains around the no error scenario value. This achieves the important design goal of TRAM++ of isolating the effect of a malfunctioning receiver to its part of the repair tree. Contrast this to the behavior in TRAM shown in Figure 8 where the latency of the normal receiver is shown to go above 10,000 ms for similar drop rates. We can see in Figure 12(b) that the malicious receiver latency shows spikes followed by drop to zero latency which indicates the point when the receiver is disconnected. We had the malicious receiver repeatedly reconnect to the multicast group to test the robustness of TRAM++ to this kind of malicious behavior. It can be argued that once a receiver has been detected as misbehaving, then its subsequent attempts to reconnect will be denied by keeping track of IP addresses of such receivers. However, it is well known that IP spoofing can defeat such a scheme. Therefore, we decided to use this error model to simulate a real-world malicious receiver. The sender data rate fluctuates between a rise towards the maximum and a steep descent to near zero as each pruning operation occurs. The sender buffer utilization remains as in the higher drop rate case.

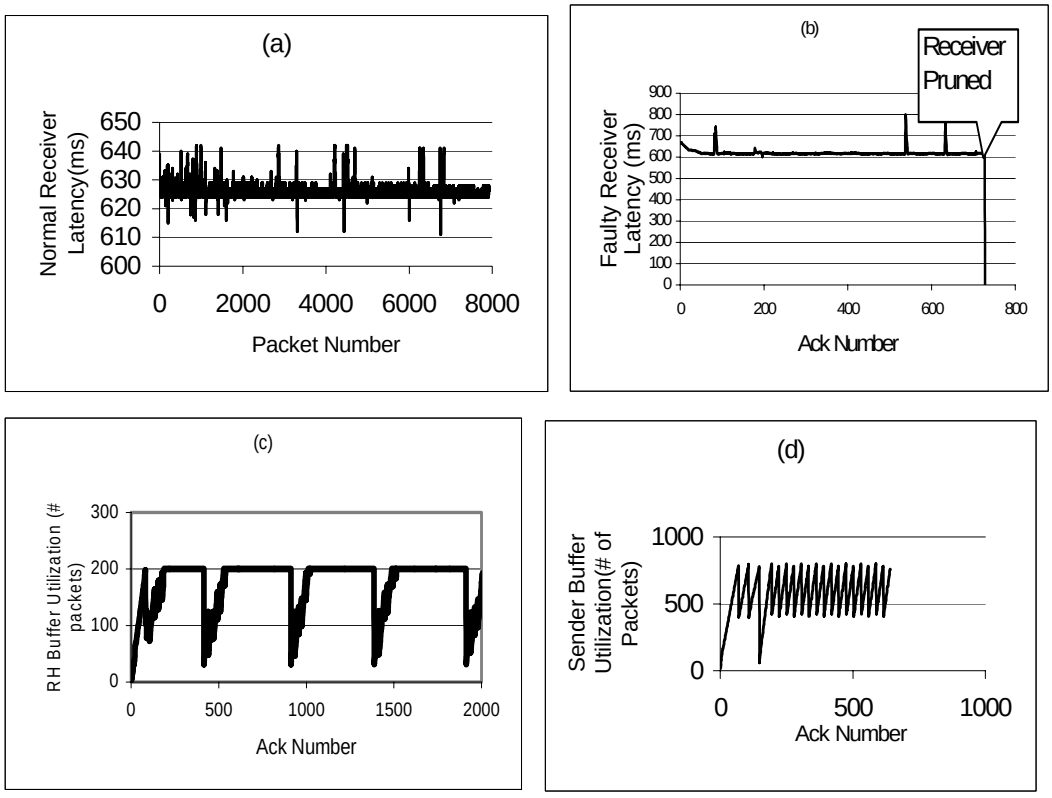


Figure 11. Evaluation of TRAM++ under message drop rate of 5 out of 50 packets.

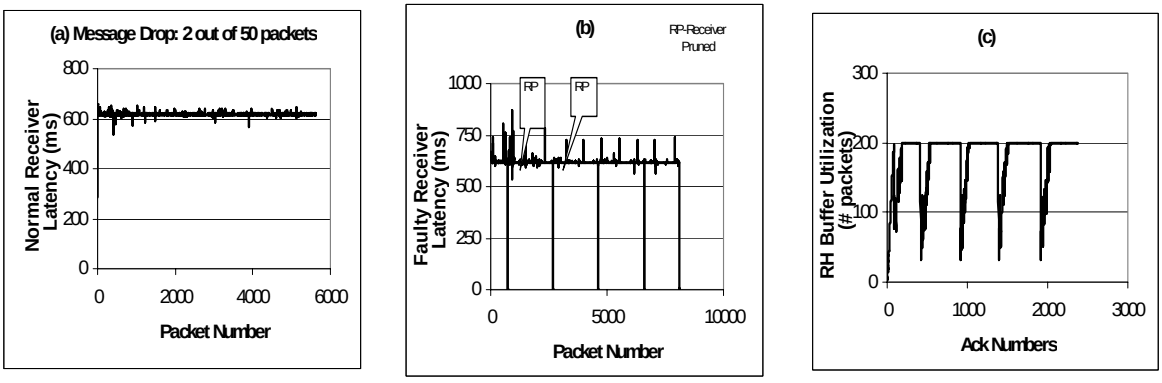


Figure 12. Evaluation of TRAM++ under message drop rate of 2 out of 50 packets

In the case of message delays, the pruning is found to happen for delays of 8000 ms and above. Figure 13 shows a scenario where pruning is not done (delay = 1000 ms) and Figure 14 shows a pruning scenario (delay = 8000 ms). For the no-pruning scenario, the sender data rate and buffer utilization behave as in the error free case (see (b) & Figure 10(b) respectively). The latency of the normal receiver remains unaffected though the sender detected congestion causes its latency to rise towards the end of the experimental run. The malicious receiver has a saw-tooth latency pattern with the peak separated from the base by the delay amount.

3.7 Highlights of Result

Some of the important results coming out of the study are summarized here.

1. Both TRAM and TRAM++ scale well with respect to latency as the number of receivers is increased, up to a total of 30. Also, the jitter is negligible even for 4 hops spread across campus. Under error-free conditions, both were able to maintain a streaming video of data rate 40 Kbps, but the next higher

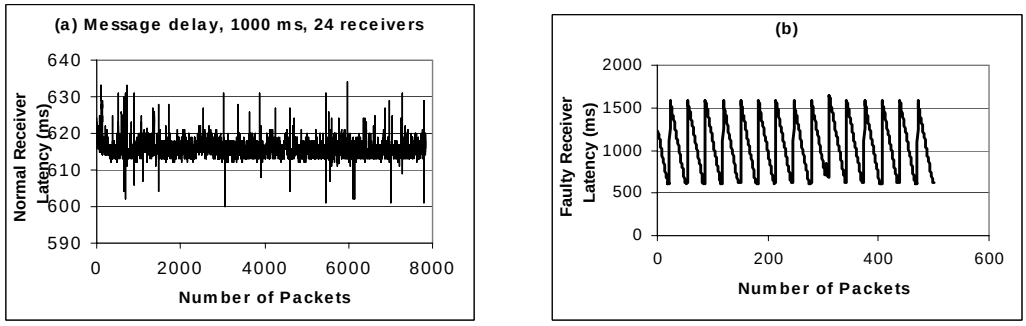


Figure 13. Effect of message delay on TRAM++ for 1000 ms delay

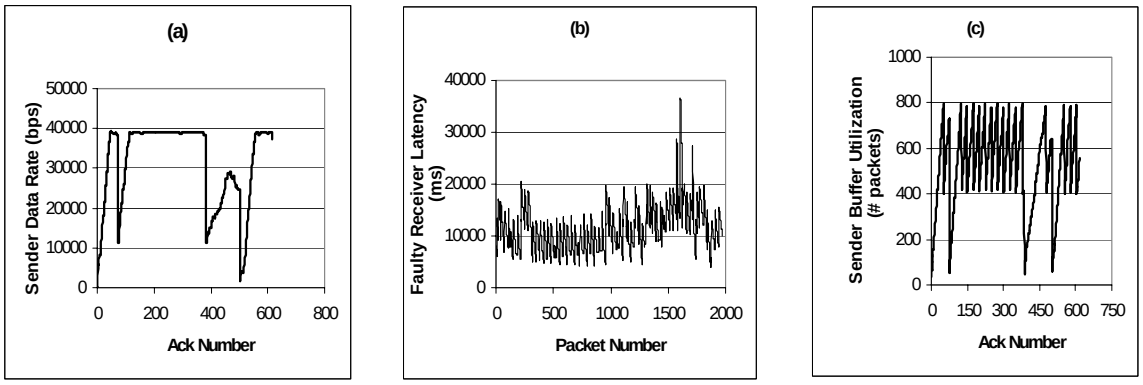


Figure 14. Effect of message delay on TRAM++ for 8000 ms delay (Faulty receiver is pruned)

step of 60 Kbps that we tried was not stable. This is because the testbed network was shared and though the rated bandwidth was 100 Mbps, it could not support jitter free 60 Kbps traffic.

2. TRAM is not successful in isolating the normal receivers from the effect of faulty or malicious ones. TRAM++ is able to achieve this through its protocol of differentiated acks and buffer management.

3. TRAM++ under a constraint of 16% of the TRAM buffer availability at the RH is able to maintain the latency within an overhead of 3.2% in the error free case. TRAM++ achieves this without any additional memory overhead. In cases with errors, the latency of normal receivers in TRAM++ is better by a factor of up to 30. TRAM++ is also able to prune malicious receivers faster because of local decision-making ability at the RH based on temporary reaffiliations.

The comparison brings out that both protocols are scalable in latency in the range in which the experiments are done. TRAM++ incurs a minor latency penalty in failure-free conditions. If constraints are enforced on the intermediate nodes, then TRAM++ can enforce the conditions and yet localize the disruption to the system due to a few slow or malicious nodes. TRAM++ also includes an algorithm for pruning nodes deemed too slow or suspected to be malicious.

3.8 Why do we Need a Generic Method ?

In this chapter we highlighted an approach to make an existing protocol TRAM robust to malicious and slow receivers and also optimize resource utilization at the local repair heads. This process is highly labor intensive. It involves detailed understanding of the current protocol – not just the specification, but also the implementation. The effort

described here does not generalize. Even for TRAM, it handles only one vulnerability and one source of inefficiency. To handle a more comprehensive set of disruptions, and for a wide class of applications through a method similar to this will be practically infeasible. Also the method assumes the availability of the source code. All of this motivates the need for coming up with a generic solution which is widely applicable and independent of the underlying protocol.

4 Monitor Based Detection Approach

We propose to provide a detection infrastructure to message passing based distributed protocols through a monitor based approach. We assume a black box model for the applications to be monitored meaning no access to the source code or the execution hosts. The specification of the protocol is, however, available to the system. The monitor has access to the external interactions of the protocol participants, or in other words the messages exchanged by the monitored entities. The monitor design makes the solution independent of the underlying protocol and hence generic. We assume that the monitor is able to examine the communication header and payload. Since the monitor is considered a *friendly* entity in the system, we assume any cryptographic keys will be made available to it.

4.1 Monitor Placement

Monitor is like another entity running somewhere in vicinity (not constrained to run on the monitored node) and detecting invalid transitions through snooping over the message exchanges. Depending upon resource constraints sometimes it might not be feasible and advisable to run monitor on the same node because it might slow down the entity, hindering the protocol's overall performance. Monitor is also made to run asynchronously to the application thus not constraining protocol operations to wait for monitor to match instruction by instruction. Monitor is provided with the protocol information and rule base against which the matching has to be done. Following subsections will build up on this approach and describe the whole architecture.

4.2 Monitor Architecture

The Monitor architecture consists of several components classified according to their functional role. These components are the Rule Classifier (which is invoked at initialization time or when rules are updated) and the remaining ones, in the order in which they are invoked, the Data Capturer, the State Maintainer, the Matching Engine, and the Decision Maker. Figure 15 gives a pictorial representation of the Monitor components. Each component is described in the detail in the next subsections.

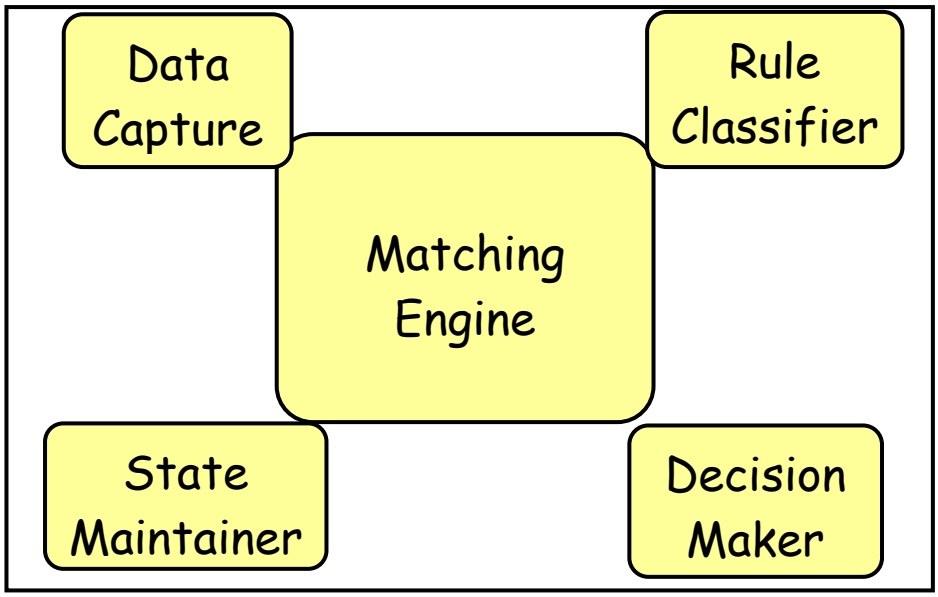


Figure 15: Monitor Components.

4.2.1 Data Capture

This component has the responsibility of getting the messages exchanged between the monitored entities from the communication channel for processing by the monitor. The capture can be carried out in two ways, either *Active* or *Passive*. In the active mode the protocol participant itself sends a copy of the message to the monitor. In case of passive mode the participant keeps silent and the monitor snoops over the packet in the communication medium. The obtained message is provided to the State Maintainer for processing.

4.2.2 State Maintainer

This component comes into play for the first time during the setup phase of the monitor. This part basically is the store house for the monitor. Information about the monitored protocol, including the state transitions, the event space protocol headers, etc. are stored in the State Maintainer. The Rule Base is linked to the state maintainer but is not a part of it. When the input is provided, the state space (i.e. all possible states for an entity) are stored in a global hash table, which is indexed by the state name. Hash table has state objects stored in it which store information about the state like state variables, rules associated with a state etc. Each state has a list of events associated with it i.e. all the possible events which can take place in that state. For each event corresponding next

state information is also stored in the a hash table within the state object, which is indexed by the event name. The global event space of the system gets stored in another hash table. The global event space of the system is all the possible events which are linked with the entire state transition diagram. The state variables are a part of the state object itself. The State Maintainer is responsible for checking the incoming messages for any event occurrence, performing the appropriate state transitions, and triggering the rule matching engine accordingly.

4.2.3 Rule Classifier

The Rule Classifier has the important task of deciding whether a particular rule is to be matched at this monitor or not. It is invoked at initialization time and whenever edits are made to the Rule Base. It does the stratification of the rules into two major classes: Local and Global. A rule is local to a Monitor if the matching has to be carried out at the current level otherwise the rule is global. The rule classification is carried out automatically by the classifier. The basic classification of a rule is done into following classes:

1. *Local Rules*: Rules pertaining to my nodes (nodes which 'this' monitor is monitoring.
    a. Processable: Rules for which processing will be required at local level only and the message will not be passed up. Example: The Number of Nacks in a certain given interval (interval should be defined in terms of number of ack windows) should be restricted by $N_{max} << N_{individual\_max}$ * Max number of members.
    b. Non-Processable: Rules which would be matched and the message passed up above. Example: the total number of children in the tree should be less than N_{global}.

2. *Global Rules*: Rules which will be required by other monitors. Basically these rules involve nodes outside the monitored area of 'this' monitor. Example: The average data rate should be > D_{min} based on a set of Ack windows. This is global with respect to RH so only the global monitor will match it.

The rules are specified in terms of state and their variables. Monitor contains a mapping to specify which state variables correspond to which nodes. Monitor has a list of entries which it is monitoring. Monitor carries out processing for each input rule and matches each rule variable against the set of variables which correspond to the node

which it is monitoring. If the rule contains only state variables which correspond to the monitored node then that rule is considered local. Monitor outputs the rules into local processable, local non processable and global rules. The pseudo code for the algorithm is given below.

```
For each rule in input file
do
        GetRule();
        inGlobal = false;
        inState= false;
        For each variable (v) in the rule
        do
                if(v.isInStateVariableList)
                        inState = true;
                else
                        inGlobal = true;
        if((inState=false)&&(inGlobal==true))
                rule.IsGlobal;
        elseif((instate==true)&&(inGlobal==true))
                rule.IsLocalNonProcessable;
        elseif(instate==true)&&(inGlobal==false))
                rule.isLocalProcessable;
end
```

4.3 Types of rules

So far we have not formally defined the syntax for the rules. The syntax is of importance in the system because it captures the expressibility of the system, and determines the speed of matching that can be achieved. The ***rules*** as defined in our system could be specifications of the protocol or QoS restrictions placed by the administrator or the application. The rules are entered by the user or administrator who is running the monitor during the set up phase. Another important characteristic of the system is that rules defined must be misuse based and not anomaly based. A primary reason for that is because the space of anomaly based rules could be potentially infinite for real-world large scale distributed systems.

When defining the syntax of the rule base, several competing approaches are to be evaluated. Previous researchers have tried to use Petri Nets[23], some with only combinatorial state equations[25] and others using temporal logic[30]. The next subsection details on our rule classification and underlying assumptions.

4.3.1 Temporal Rules

After carefully studying the properties of the rules[27][30][31]. exhaustively we came up with following classification for the Temporal rules.

1. Type I
   $\forall T \in (t_N, t_N + k) \Rightarrow V_T \Rightarrow U_q \; q \in (t_I, t_I + b)$
   The above rule represents the fact that if for some time t_N a state is true then it will cause the state U_q to be true for some time b starting from t_I.

2. Type II
   V_t is the state of an object at time t : $V_t \neq V_{t+\Delta}$
   The state V_t will not remain constant for more than Δ time units if an event E_l takes place.

3. Type III
   $L \leq |Vt| \leq U \;\; t \in (t_i, t_{i+k})$
   A state (variable)will be bounded by L and U for some time k.

4. Type IV
   $\forall t \in (t_i, t_i + k)\; L \leq |V_t| \leq U \Rightarrow \; L' \leq |B_q| \leq U' \;\; \forall q \in (t_n, t_n + b)$
   A state (variable) being bounded by upper and lower bounds will cause another state (variable) to be bounded by some bounds and will hold true for some time interval b. *If one look at the rule carefully it is actually the parent rule or the master rule. Basically all the first 3 rules can be derived from this rule. But we still need the above because matching this rule requires more variables to be matched instead of the above cases and increases the latency of detection.*

The type of the rule is easily identified by matching the template to the input rule. Also a Temporal rule can be distinguished easily from the Combinatorial rule by looking for the time factor i.e. 't' (or 'T') variable in the input rule. We restrict the variable 't'(or 'T') to be used only for time and not for representing any state, state variable or event.

4.3.2 Combinatorial Rule

These are the rules independent of time and expected to be valid for all times except transients. An example of the combinatorial rule is take a simplex protocol in

which only one node can send at one time. So if ‘s’ and ‘r’ represent sending and receiving (listening) states, then for nodes A and B we can say that

((A= =s) && (B= =r)) | | ((A= =r) && (B= =s)) :

It states that either A or B is sending assuming infinite data to send. We can see that a combinatorial rule might consist of states, state variables and events. But each of the expressions will finally yield a Boolean true or. An arbitrary combinatorial rule has the operators: ‘¬’ stands for logical NOT, ‘V’ stands for logical OR and ‘Λ’ stands for logical AND operation. We cannot really subdivide these rules further as there is no distinct feature to look for. All the rules will be some combination of these boolean operators only.

Although the combinatorial rules must be valid for all the time, there could be fluctuations and transients which cause the value to flip to an incorrect value and then return to the normal correct value. If the rule matching takes place at one of such occasions then the decision maker will flag an error. We must not flag an error for such cases because these are transients which do not last for long enough to cause errors. The rule matching algorithm also handles this case.

4.4 Matching Engine

The Rule Matching Engine is invoked by the State Maintainer when an incoming event triggers a rule to be checked. The Rule Matching Engine is the most resource intensive workhorse of the Monitor. In order to bound the detection latency, it is crucial that the matching algorithms be optimized for speed. Due to the different nature of temporal and combinatorial rules, we provide separate matching algorithms for them.

4.4.1 Combinatorial Rule Matching

Since Combinatorial rules are expressed with Boolean variables and matching requires simple calculation of the expression, the algorithm takes `O(n)` time to evaluate it given an expression with *n* literals. This is assuming the uniform cost model. In a uniform cost model we assume that each mathematical computation takes the same amount of time irrespective of the size of the input in bits. Even if we use a logarithmic cost model still we would not achieve much gain because each input is only 1 bit (boolean input).

We propose to optimize the matching by converting the combinatorial rule into an expression tree. An expression tree has the operators in the intermediate nodes and the operands at the leaf levels. An observation for our system is that the same rule may be matched many times. For each invocation of the matching algorithm, not all the variables in the expression would have changed. Hence, the algorithm needs to be incremental in its output Boolean value computation. The expression tree stores at each node two sets of variables, one corresponding to the left child sub-tree and the other corresponding to the right child sub-tree. Each node also stores its value from the previous matching of the rule.

When a rule is triggered, it is passed a list of variables that have changed since the last invocation. Only the parts of the tree which contain the modified variables are re-evaluated and for the other parts, the previous values are used. In this way only part of the tree is recalculated. The matching time is minimized by using hashing. We can maintain a hash table at each node of the variable in the left and right sub-tree. Searching if a literal exists in a has table or not takes a constant amount of time.

The second optimization used in the data structure is to keep the tree ordered. The order is in terms of frequency of change of the variables and the arrangement is decreasing from left to right. Thus, the variable which changes most frequently is the leftmost node in the expression tree. The matching algorithm explores the tree from left to right. If the rule is such that no variable is repeated, then when a match occurs, the algorithm terminates and does not need to search the tree any further. Since the more frequently changing variables are kept towards the left, the matching can terminate early.

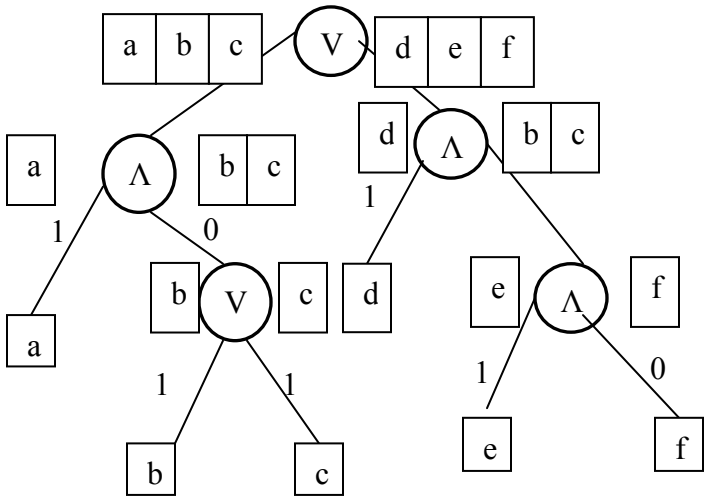


Figure 16: Expression Tree used for Combinatorial Matching

Example:

Assume an event occurs which changes the value of the variable b in Figure 16. The procedure followed according to the proposed algorithm is as follows

1. The algorithm checks at the root for the variable b, detects that it lies in left sub-tree so moves to the left sub-tree.
2. It recursively arrives at the node 'V' of depth 2 and knows that this has to be evaluated.
3. It evaluates the new value for bVc and pushes that value (v_1=1) up the tree.
4. Now at node 'Λ' the new value (v_1) arrives and it uses previous '1' as the value for the left sub-tree for calculation of the expression.
5. This way the new value is propagated up to the root to get the final expression.

Handling transients: As mentioned in Section 4.3.2, there may be transients in the system that cause violation of the system properties for a short period of time. The period of the transient can be input by the system administrator, say Δ time units. If the combinatorial rule matching flags an error, it is re-executed after Δ time units. If the re-execution also flags an error, it is taken as a valid detection. The pseudo code for the expression tree evaluation algorithm is given below :

```
evaluateExpTree(TreeNode, Variable a)
        If(a.existsInLeftSubTree)
                r = evaluateExpTree(Treenode.left,a);
```

```
            Treenode.previousRightValue=r;
            l=node.previousLeftValue;
        else(a.existsInRightSubTree)
            r = evaluateExpTree(Treenode.right,a);
            Treenode.previousRightValue=r;
            l=node.previousLeftValue;
        Op = TreeNode.operator;
        return (r.Op.l)
end
```

4.4.2 Temporal Rule Matching

There are four kinds of temporal rules as described in Section 4.3.1. The temporal rule matching algorithm handles all the four types of rules.

The algorithm translates the rules that are mentioned in the rule base into the master rules associated with events that are maintained in the *State Maintainer* module. The master rule consists of the basic parameters necessary for rule checking and rule specification, and is used as a template to spawn off new rules on occurrence of suitable events. On narrowing down arrival of a packet to a particular event in a particular state, all master rules associated with that event are triggered. This leads to the creation of a new rule object with the same parameters as the master rule, which is added to a list, and the invocation of a *Timer* object, which handles alarm generation and the actual rule checking.

To deal with the inevitable latency that might creep up in an asynchronous matching system, the design incorporates two *Timer* objects. One *Timer* object deals with storing the appropriate state variables in the rule itself on generation of alarms, and the other *Timer* object deals with checking the state variables stored against the values expected, to do the actual rule checking. A thread pool has been implemented to ensure parallel checking of multiple rules.

4.5 Decision Maker

This component has the task of flagging an error. Once the matching algorithm returns the match the decision maker flags an appropriate flag reporting the error. The part of the decision maker currently is very limited because we are restricting our system to only detection of disruptions. Motivation of making another logical entity such as decision maker is to provide scope for addition of features of not only detection but

recovery action depending upon the type of disruptions. Separation of the functions provides easy access to the part to be modified for additional changes. If the rules are all not completely accurate and multiple matches are weighted by the confidence in the rules, the decision maker may come up with an aggregate decision and assign it a confidence metric[37].

4.6 Hierarchical Monitor

The previous discussion of the monitor may give the impression that it is a single point of failure and a potential performance bottleneck in the system. Also if a Monitor is trying to monitor all the protocol participants then it will be overwhelmed by the computation making the latency of detection high. On the other hand, if the Monitor is only monitoring selected nodes out of the entire set, then a failure in some other part will go undetected.

Critics might think that Global and Local monitors only make sense for Tree based protocols but not in general but that thought is not true. In this Monitor based detection, we incorporate the idea of using not a single but a hierarchy of monitors, working in conjunction at multiple levels to ensure failure and intrusion free operation of the protocol. The entire structure is divided into Local, Intermediate and Global Monitor. The Local Monitor are concerned with the a local set of nodes. They monitor and do rule matching at the local level and eliminate the major part of rules which are governing with the operation of the protocol. The Intermediate Monitor might have several Local Monitors which send information to it regarding the operation of their local set of nodes. It can also have some nodes directly beneath it. Finally the Global Monitor looks at the global view of the protocol. It does not have to do a lot of matching because message filtering takes place at the local and intermediate monitors.

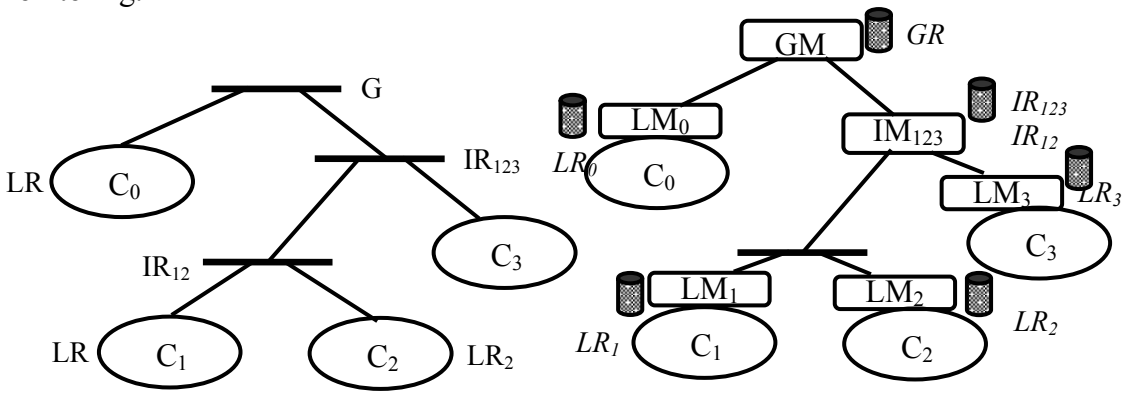
Dividing the single Monitor into several monitors has multiple advantages namely:

1. Task gets evenly (more or less) distributed among the various monitors. At each level message filtering is carried out so that the higher level only sees the messages which are relevant.
2. This hierarchical structuring works the best for well designed distributed protocols because most interaction is local and therefore substantial filtering can happen at local monitors.

3. It increases the accuracy and coverage of the detection system for example: If a Local Monitor fails to detect an error in a local region then the monitor up above will detect the error and prevent it from causing a complete shutdown of the protocol. Also if local nodes in a region are malfunctioning then a Local Monitor can easily prune it from the logical set of protocol participants.
4. There are several features which an administrator would like to look at collectively instead of individual behavior. The rules governing the collective behavior might be different from the rules governing individual behavior and so we need something more than a Local Monitor to capture it.

A second thought might lead one to a more basic application where this Hierarchical Monitor based system could be used. Take an example of Purdue Network. We can keep a Local Monitor to monitor the nodes in MSEE while another one for Computer Science. And we can keep a Global Monitor which can collective look at the behavior of the two LANs and coordinate with the Local Monitors to flag any disparities.

A significant point to be noted is that although we are placing multiple monitors in the system at various levels, each Monitor has the same implementation but monitors either completely disjoint or overlapping set of nodes. Each Monitor has the identical rule base which is derived from the protocol specification. The rule base is split into the different kinds of rules by each monitor based on its placement and the entities it is monitoring.



C: Cluster; LR: Local Rule; IR: Intermediate Rule; GR: Global Rule; LM: Local Monitor; IM: Intermediate Monitor; GM: Global Monitor; : Rule repository

Figure 17: Example of Hierarchical Monitor Placement with respect to the protocol TRAM.

The hierarchical structure is an important design decision in view of any protocol. Figure 17 illustrates the hierarchical nature of the monitor in respect to the target

application protocol TRAM. One can see that Local Monitors are monitoring nodes under local repair heads, intermediate monitors for intermediate repair heads and a Global Monitor has a global view of the operation of the protocol.

5 Experiments and Results

5.1 Systems Details

The Monitor system is implemented in JAVA as a separate entity which is multi-threaded. Currently the system is only being tested on a single protocol i. e. TRAM. The monitor code is running on Pentium 4 2.26 GHz processor with 1 GB memory, 533 FSB and 512 KB cache (EE machines) for the initial testing. TRAM is run with the same settings as done previously for the TRAM testing and comparison with TRAM++. All the configurations for the TRAM remain the same as in the previous experiments. In this case again implementation is done and the experiments performed on a production campus-wide wide area network with normal traffic coexisting with the reliable multicast traffic.

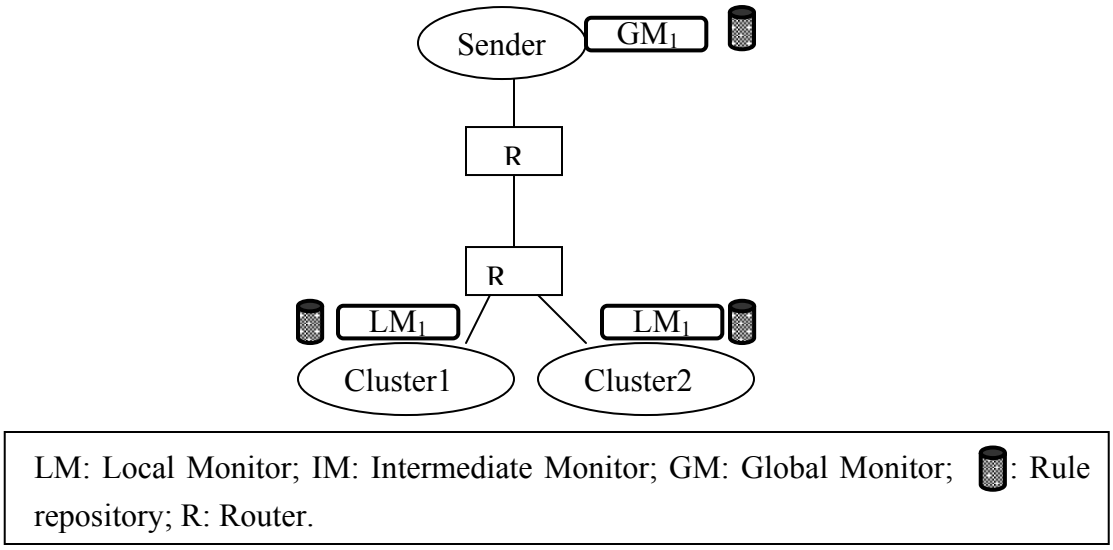


Figure 18 Monitor Placements in the actual testing on TRAM protocol

The Figure 18 above gives a sample configuration of placement of the monitors in respect to the TRAM protocol. Monitor is set up by inputting the rule base, State transitions diagrams, event information and packet header information. A sample rule file is given below:

T R1 S1 E1 t1 S2 t2 t3

T R2 S1 E1 t1 S2

T R3 S1 E1 L U

T R4 S1 E1 L U S2 E2 L U
C (S1ΛS2)
C (¬ (S1ΛS2)V(E1ΛS2))

Each line represents a rule. The first alphabet informs the monitor if it is reading a temporal or combinatorial rule. This information is also deduced automatically by the monitor by simply looking for the variable 'T' in the rule equation. For the Temporal rules the second letter or argument provides the type of temporal rule followed by its parameters. We explain the rule file in detail when we specify the exact rule file for TRAM used in the experiments(section 5.4).

5.2 Fault Injection

A normal run of the protocol would not really test out a monitor which is designed for detecting errors. Hence fault injection is necessary to bring out the performance and latency of detection of the Monitor. We have adopted the method of random error injection. We are injecting non fail silent errors. We randomly inject bit flips in the TRAM header. If the application keeps running then we try to measure the latency of detection by the monitor if the bit flip has caused an error. The method of causing random bit flip can cause the application to crash because it might inject error in a field which causes application to abort. We measure the accuracy and latency of the monitor only for the successful runs of the application (TRAM). We have also adopted specific error injection into only specified fields in the header.

The header field of TRAM consist of the Flags, *MessageType*, *SubMessageType*, *InetAddress* etc. The first three fields is flags *MessageType* and *SubMessageType* are each of one byte only. Each of the three fields is 1 byte in length. For error injection in a general experiment we pick a bit out of the total bits in the header randomly and flip it.

5.3 Instantiation of the RuleBase

The rule base is given as in input to the Monitor. Since currently testing is carried out on TRAM we have rule base and STD for TRAM which are given as input to the Monitor. A sample rule and state transition file is shown below.

RuleBase.txt

T R2 S3 E3 20

T R4 S1 E1 0 32 10 S2 E2 0 5 8 15

T R3 S1 E4 0 32 10

T R3 S1 E3 0 10 20

C ((S1 V E1) V (S2 Λ E4))

StateTransition.txt

: S1 E1 S2 ! : S2 E3 S3 E2 S1 ! : S3 E1 S2 E3 S4 ! : S4 E4 S1 E1 S1	E1 2 1 1 ! $ E2 2 1 4 ! $ E3 2 1 2 ! $ E4 2 1 5

Explanation : Here the symbols S1–S4 and E1-E4 correspond to states and events respectively. The special characters are only used as delimiters for file reading. The event definition is given below :

E1 : Message received is Data Type

E2 : Message is ACK

E3 : Message is a NACK

E4 :Message is a Head Bind (Reaffiliation with a new Repair Head)

The states represented above are a small subset of the complete state transition diagram of TRAM. The one advantage which we get by specifying the state transition diagram is that the Monitor will only be monitoring these states only and you can give reduced set of states to be monitored.

5.4 Preliminary Results

The initial testing of the monitor is being carried out on the TRAM's receiver state transitions diagram. We allow the monitor to look into the receiver's state transition

diagram. Monitor is monitoring the packet which are getting into the receiver and sent out of the receiver. For the initial measurements we allow monitor to monitor the header fields of the TRAM packet. The input to the state monitor consists of the state transition diagram for the receiver. The description about the state transition depicted in Figure 2 diagram which is fed to the Monitor is given in Table 4.

: S1 E1 S2 E4 S1 E3 S1 ! : S2 E3 S3 E2 S1 E1 S2 ! : S3 E1 S3 E3 S4 ! : S4 E4 S1 E1 S4 E3 S4 !	S1 1 1 1 ! S2 2 1 1 ! S3 2 1 1 ! E1 2 1 1 ! E2 2 1 4 ! E3 2 1 2 ! E4 2 1 5

Table 4 STD of receiver as given to the monitor.

The Table 4 gives the STD which is given as input to the monitor. The states are given in separate lines followed by the events and corresponding states. The description about the state and the events is given in the second column. The second column contains the information about the events with respect to their location in the header fields of the packets of that protocol and their corresponding values. The rules which are inputted to the Monitor are given below.

```
T R2 S1 E1 200
T R3 S4 E3 0 5 5000
T R3 S2 E2 0 31 1000
T R4 S1 E1 0 2 30 S2 E2 0 2 1000
T R2 S2 E2 200
```

Above are the rules used for the experiment of the monitor on a TRAM receiver. If we see the STD's we can easily comprehend that the first rule expects that if the state is S1 and the event E1 occurs then the state must not be the same after 20 ms i.e. it must

change. Hence it is a 2 temporal rule. The second rule states that in state S4 a receiver should not be sending more than 5 nacks (event E3) within 5000 ms. The rule gets instantiated only if the monitor sees at least one nack in state S4.

Fault Rate (per packet)	Injected Faults	Error Flagged	Total Time in Matching	Latency
1/8	1007	49	2.52	4902.92
1/16	541	49	1.84	4902.24
1/32	296	49	1.705	4902.12

Table 5 Results for Monitor detection monitoring Receiver

Explanation : The Monitor is monitoring the state transition diagram for the receiver part. The error free operation expected behavior of the state diagram is that the receiver will be getting 32 data packets after which it sends an ack. A receiver might send a nack depending upon the missing packets for which data retransmission takes place.
The faults are injected into the packets header by doing a specific fault injection for the above case as a first experiment. Later we inject faults by doing a random bit flip randomly in the header. We are varying the fault rate by injecting at varying rates of 1 per 8, 16 or 32 packets. Fault is injected in the form of sending nacks instead of acks by the receiver. We use the rule base which we have developed and input it to TRAM. The rule file is shown above.

As we can see that since the rule base has rule which governs the transitions of the receiver in an orderly fashion. It has a rule which ensures that once a data packet is received then a receiver must respond with a ack or nack depending upon the packets received. Another rule is governing the number of nacks received in a particular interval. The number of nacks in a particular state must be less than 5 (current setting for the above system) for the receiver within a time interval set to 5000 ms. The data packets are sent in at 1pkt/20 ms. In the table the first column represents the rate at which the faults are injected. The second column contains the number of faults injected for that run of the experiment. The last 3 columns represent the results in the form of errors flagged, the latency in detection and the total average time taken in rule matching which includes the time for which a rule waits after the rule is instantiated. We can see that latency of matching is very low and is varying from 1.7 to 2.5 ms. We see a slight increase in the latency of the system as the workload increases in this round of experiments. The numbers of errors flagged remain the same because we are simulated for a fixed numbers of error detection. We are keeping this parameter fixed. One can see that although the

number of errors injected in system are varying for the same number of the errors detected. A close inspection of the state transition diagram and the rule base explains the results. We can see that in the STD's there are several states which expect nacks and are taken as normal, so of course all the nacks which are injected in the system are not treated as faults but the initial few nacks are taken to be correct operation of protocol. Now the rule states that the number of nacks must be less than 5 for the entity in state S_4 within 5000 ms. Once a nack is received in state S_4 , a rule is created which sets an alarm time for 5000ms. After the alarm time the thread wakes and checks for the number of nacks. If they are more than 5 then an error is flagged.

If we inject more errors also the number of nacks will keep on building in that state but since the matching is done only after 5000 ms are over hence only a single error will be flagged. It is the primary reason that although the injected faults are different for the three cases but the detected are the same. The average matching time is close to 4900 ms because the thread has to wait for 5000 ms before a matching is carried out. As the injected faults are more than the errors flagged so we measured the number of packets which actually invoke a transition in the state transition diagram. Since the packets are random they might be carrying events which are not expected in a particular state and hence will be discarded by the Monitor based on the STD which given as input to monitor. The percentage of packets which cause the transition in STD are found to be close to 22 %.

In another set of experiments we withhold the acks to simulate a malicious receiver. The Monitor only sees the data packet. One can see that the number of errors flagged are close to the errors injected because the rule base requires the protocol to send an ack after every 32 packets and the errors injected are via withholding those acks. The number of injected faults are 34 and errors flagged by the monitor is 32. The average Latency of detection is 1.67ms and the average matching time (including the wait time) measured was 221.24ms.

We also do a loadability test on the Monitor. We increase the load on the Monitor in terms of increased packets to be matched and measure the total time for matching (including the waiting time).

In these set of experiments when we do a random injection its important to note that there could be packets which might not even cause a message transition. So if an event is there is a randomly injected packet but even in the current state of the entity that event is not expected then the packet would be dropped by the Monitor after verifying fro

STD that it is not a valid packet. After the STD is invoked by the packets there could be certain events which only cause the state variable to be incremented and not generate a new rule. This is the primary reason for the disparity in the number of injected faults and the numbers of errors flagged. If we do a very specific injection of a particular kind of fault for which there is a rule in rule base , then that would be testing the implementation of the Monitor's Matching Engine because the accuracy would be 100%. It would not be a very good measure of the monitor's capabilities.

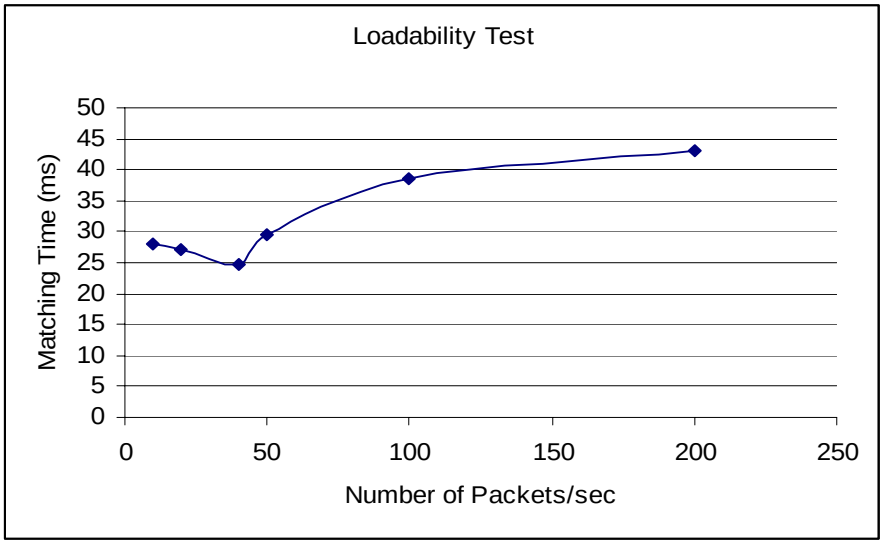


Figure 19 Loadability Test on Monitor

Figure 19 shows the behavior of the Monitor with respect to increasing load. We can see that for 50 packets the Monitor's latency remains within 30 ms. As the number of packets are increased the latency increases but it tends to saturate hinting that Monitor is scalable with respect to the load of the system.

6 Related Research

Error detection is an important field of study for dependable and reliable community. Trying to come up with a system which is reliable involves a lot of thought process into detecting and identifying the errors in the system and taking recovery actions. Traditionally focus on reliable systems has been divided into two major areas

1. Developing specialized systems which are fault tolerant and reliable. Primarily reliability has been provided using replication in hardware and software. for eg. Tandem systems.
2. Developing specific error and fault tolerant techniques for existing applications.

Several researchers have come up with specialized solutions for some specific protocols but an important challenge is to come up with solutions which uses Cheap off the shelf (COTS) equipments and use them to provide enhanced reliability and dependability to the system.

There is a need to come up with systems which are self checking because self checking components tend to have lower latency of detection compared to remote detection. Also these methods try to inhibit any rippling effect by containing the problem to a much smaller domain. The work by Madeira[15] shows that on an example Z68 microprocessor by using signature monitoring scheme the latency for detecting fail-silence violations come down to 0.4msec compared to an Error Capturing Technique which causes the latency of 47.8msec. Previously work has been carried out in developing fault tolerant systems like the MARS architecture, which uses specialized hardware and software to provide fail silence guarantees. It aims to provide 98.7% coverage using the specialized hardware and software techniques [18]. There has been further studies and testing on the system as well [17][16].

Recently there has been work in developing self checking systems like the Chameleon which is a software implemented fault tolerant middleware[12]. Through interactions among the entities called ARMOR's they make an application fault tolerant. ARMOR's are **a**daptive **r**econfigurable **m**obile **o**bjects for **r**eliability. ARMOR's are processes which run on the node and can send messages to other ARMOR's [13]. These ARMOR's could be embedded in the application or can be made to run independently but in synchrony to the application. They have specialized managers which can have several ARMOR's underneath it but they don't have the same architecture as that of the ARMOR. Also each ARMOR only checks for a specialized error in only one entity. The

system is not designed for distributed systems. Results have shown that with Chameleon in place a system can achieve an availability of 0.9720 when the error rate is 0.5/hour. The availability is more if the error rate is reduced. Techniques also exist [14] which use pre-emptive control flow signature (PECOS) to provide error detection. This signature embedding prevents the program from taking an incorrect control flow path. Since it is pre-emptive it reduces the number of process crashes and lead to graceful termination of the program. It is able to detect 87% of all the generated errors and fail silence coverage is improved by a factor of 36. Such a technique requires access to the application's source code and not feasible in all practical situations.

The concept of self checking in the form of Observer architecture is mentioned in [12]. An observer is a formal observer i.e. it implements exactly the formal specification of the system. The task of the observer is to do verification of the output of the underlying worker. Worker is the normal existing running system. The approach is based on detection by matching against a formal and verified model of the system. They have described formally what a self checking system comprises of and its behavior with respect to set of faults. An observer is like a replicated entity which will be checking the output of the worker against the formal specification of the system. Diaz *et al* [11] discuss how an observer can be used for a doing an online validation of MAC and OSI protocol.

Since the thesis aims to develop a self checking protocol one must have a method of defining the system or in other words formal specification of the system. There have been several approaches to designing formal models and specification of the system[22][23][24][25]. As described by [11] a formal model must be able to express the specifications of the system, support verification procedures and should be worth matching against in case of an observer model. The formal specification of the protocol is just one part of making the system reliable. In order to be able to tolerate disruptions the formal specification language must also have tools to handle intrusions.

Lot of research has been carried out in finding ways of specifying the distributed systems. It is important to specify the system in as precise ways possible because specification of a distributed system is used for its implementation. If the specification is inexact then there are bound to be errors in the implementation of the system. Lamport describes what is called *transitional axiom*[27]. It gives the conceptual foundation for developing a formal specification for a system. It defines what are the differences if a system and its environment and how to handle interactions between the two. It describes

the temporal logic but with respect to *eventual states* and not intermittent changes. The paper provides an in-depth knowledge of the concept but fails to provide and specification language which can be used for specifying the system.

Time forms an important part of the specification of the distributed systems. Several researchers have worked on what is called Temporal Logic [30][29] . In temporal logic specifications the behavior pf the protocol is captured with time as one of the variable. Thomesse *et al* [38] have described the system characteristics by specifying Timing Constraints. They describe the time characteristic as a physical greatness expressed in time units. It introduces the concept of Deadline constraint and time window constraint. It discusses the interactions between the entities in Cooperation based model like producer/consumer and client/server models. We have used a similar concept in developing our own Temporal Rules to specify the systems. We have kept the interactions of the various entities and dependent structure in mind before coming up with the generic temporal rules to specify the systems.

Rainer Schlor and Werner Damm [27] have come up with timing diagrams to specify and do formal verification of distributed systems. They demonstrate their graphical model on hardware designs. Their method is a hybrid of Temporal logic specification and graphical approach. The approach seems to be more valid towards the system which have graphical interface and CAD environments. Although the Timing diagram approach provides a more lucid picture to a reader who is trying to understand the distributed framework but from a point of view of person who has to implement the system, the approach lacks the precision. The diagrams have to be again interpreted and transformed into temporal equations for them to be implemented. Temporal Methodology has also been developed by Manna and Pnueli in [29].

For testing the monitor we have chosen TRAM because it forms a good representative of the reliable multicast protocols. Also TRAM is being developed by the research group at Sun which makes the source code available. It makes it easy to install and debug and provides deeper insight into the protocol. TRAM was first presented by Chiu *et al* in [2]. This study evaluated some of the parameters considered here, namely, rate, loss and cache occupancy, but using a simulation model. The simulation model made several simplifications – all RHs were at a fixed distance from the sender, no node failures were simulated and only a small subset of the links (between the RH and the receivers) were injected with failures. The congestion control mechanism in TRAM was studied in a recent paper by Chiu *et al* [4]. The two aspects of congestion control – receiver feedback based windowing and server data rate based traffic shaper – are studied

using an implementation for a LAN and an emulator for a WAN. The study showed how to dynamically adjust the data rate used to schedule packet transmission at the sender to smooth the transmission. The authors in [3] examine the issue of pruning decisions in multicast transport protocols. The decision boils down to choosing a minimum data rate and pruning receivers that fail to meet the minimum rate. It is mentioned that the minimum rate has to be chosen carefully so as not to prune genuine receivers experiencing occasional network bottlenecks. Determining an optimal data rate for the system is dependent on the kind of network and its traffic conditions, and is a complex decision. The rate of the entire group is controlled by the slowest receiver in the un-pruned set of receivers. This results in slowdown observed by normal receivers even in completely disjoint parts of the multicast tree. The scheme in [3] is a simplification of the more general scheme called *optimal pruning* described in [6]. In that paper, the authors propose multiple subgroups of receivers with a utility function for individual nodes in the subgroup and one for the subgroup as a whole. The algorithm presented aims to pick the subgroup that maximizes the sum of node utility and subgroup utility. We believe that the algorithm can work if it is possible to assign utility functions for all possible subgroups. In the practical scenario considered in this thesis, it was not possible to come up with a utility assignment. Also, the pruning decision needs to be rapid to isolate malicious receivers and the choice algorithm in Jiang's work runs in exponential time. The design point of providing stable storage only at end points has been proposed and implemented in the context of publish-subscribe systems for a system called Gryphon to provide reliable exactly-once message delivery in the face of node failures [5]. The work assumed the extreme design point of no stable storage available at the intermediate nodes whereas in our study this is a parameter that can be tuned. At least a decade of research has gone into designing and improving group communication protocols to provide membership, multicast and ordering services. However, such protocols are overkill for our goal. First, the members of a group are considered homogeneous and group joins and leaves are made visible to all the group members. Second, the design principle in this thesis is not to enforce synchronicity among the group members which is what the group communication protocols are designed to achieve. Finally, group communication paradigm is particularly suited to local area networks with tight bounds on end-to-end latency which may not be achievable in the wide area network considered here.

By providing generic method and a specific method for improving the protocol's reliability in this thesis we have tried to draw out the differences between the two approaches highlighting the importance of coming up with a generic monitor.

7 Conclusions and Future Work

In this thesis we have presented a Hierarchical Monitor based detection approach which is applicable to a large class of message exchanging protocols. We show that although it is feasible to make a particular system robust but its not feasible to do it for every existing protocol . We demonstrated the approach of making a system robust via TRAM++. We showed how TRAM++ performs better than TRAM in resource constrained situations. It is also robust to malicious and slow receivers. The Monitor based approach presents a unique method to detect disruptions in the protocols. We have tried to make a system which is automatic and performs matching based on a comprehensive rule base given as input. We test the system on TRAM measure its performance. We have shown that the monitor is scalable with respect to number of input packets. By providing generic method and a specific method for improving the protocol's reliability in this thesis we have tried to draw out the differences between the two approaches highlighting the importance of coming up with a Generic Monitor.

The system as stands is only being tested on a single protocol. As a future work we plan to test the monitor on SIP and other protocols. The method of rule classification can be improved to do a better classification for rules lying in grey region. A further insight into the temporal rules is needed to determine their *completeness*.

LIST OF REFERENCES

LIST OF REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[25] A. S. Danthine, “Protocol Representation with Finite-State Models,” IEEE Transactions on communications, vol COM 28, no. 4, pp 632-643, 1980.

[26] G. V. Bochmann, “A General Transition Model for protocols and Communication services,” *IEEE Transactions on Communications*, vol COM 28, no. 4, pp. 643-650, Apr 1980.

[27] L. Lamport “ A Simple Approach to Specifying Concurrent Systems,” *Communcations of the ACM*, January 1989, vol 32, No. 1.

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

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

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

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

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

[33] G. Khanna, S. Bagchi, J.S. Rogers “Failure resilient buffer management protocol in Java Reliable Multicast Service”, to be published in Pacific Rim Dependable Computing (PRDC), 2004.

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

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

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

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

[38] P. Lorenz, Z. Mammeri, J.-P Thomesse, "A state-machine for temporal qualification of time-critical communication," System Theory, 1994., Proceedings of the 26th Southeastern Symposium on , 20-22 March 1994 Page(s): 654 -658.