PERFORMABILITY MODELING AND ANALYSIS OF FAULT TOLERANCE SUPPORT IN COMMUNICATION PROTOCOLS

BY SAMIAN KAUR

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Professor Manish Parashar

and approved by

_____

_____

_____

_____

_____

New Brunswick, New Jersey

October, 2000

ABSTRACT OF THE THESIS

Performability Modeling and Analysis of Fault Tolerance Support in Communication Protocols

by Samian Kaur

Thesis Director: Professor Manish Parashar

There has been much research in assessing the performance of different messaging systems, but often messaging systems cannot be completely expressed by performance metrics alone. For an emerging class of large-scale distributed servers, robustness is at least as as important as performance. Three factors make protocol robustness critical: *(i)* these servers have very high availability requirements (e.g., minutes of down-time per year), implying that even occasional message loss cannot be catastrophic; *(ii)* intra-server communication depends on external client service demands, making it extremely difficult to exert enough control over the system "by design" to avoid message loss; and *(iii)* many commodity LANs do not implement sufficient hardware flow control to always prevent loss inside the network under arbitrarily adverse communication patterns. Most of the current paradigms of reliable communication either provide strong consistency semantics with high overhead (e.g. transactional RPC) or reliability with indeterminate failure states using retransmissions(e.g.,TCP/IP).

This work aims at building a new messaging layer that provides additional recovery states for applications to allow designers to reason about the cause of the error and to build customized recovery mechanisms. We present the design and implementation of a high performance Active Message (AM) layer over the Virtual Interface Architecture

(VIA) library as such a messaging infrastructure. Its performance is evaluated to ensure that the additional recovery states are achieved at a reasonable overhead. We then present a queuing model to allow in the analysis and evaluation of robustness of this messaging protocol by computing its performance as a function of dependability in the presence of component and overall failures.

Acknowledgements

I am grateful to my advisors Professor Rich Martin, Professor Thu Nguyen and Professor Manish Parashar for their immense patience, invaluable guidance, encouragement and support. I am also thankful to Professors Michael Hsiao and Ivan Marsic for their valuable advice and suggestions regarding my thesis. I would also like to thank my colleagues in DISCOLAB and TASSL for their support and cooperation. Finally, I am ever thankful to my family and most wonderful friends for their love and support during my graduate life here at Rutgers.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

"A distributed computing system is one in which your machine can be rendered unusable by the failure of some other machine you didn't even know existed" - Leslie Lamport

The ongoing expansion of the world wide web has brought about the emergence of a new class of high availability Internet services like auction sites, online trading, etc. Along with performance, these applications have additional requirements compared to traditional applications: *(i)* High availability – these applications should be able to maintain low down time in the order of minutes per year *(ii)* Scalability – these applications should potentially be able to support millions of users simultaneously, if required; and *(iii)* Fault tolerance – these applications should be able to tolerate message loses due to bursty traffic. Due to these considerations, there have been continuing efforts to build these applications on commodity clusters of computers. Clusters are an ideal platform for these services because of their expandability, potential for fault-tolerance and high performance vs. cost ratio. A robust, high performance messaging system is critical to successfully leverage the potential of the cluster. The messaging system thus has to maintain availability under the additional constraints such as: *(i)* occasional message loss; and *(ii)* difficulty in exerting enough control over the system by "design" to avoid message loss; *(iii)* insufficient hardware flow control in many commodity LANs implementations. These factors demonstrate the need to explore new design considerations for building robust and high performance messaging architectures that will allow applications to *efficiently* recover from system failures.

The performance of such layers has been studied intensively over the last decade. However, not as much attention has been paid to robustness. This is understandable

because the work evolved in the context of parallel programming, where the latency of the underlying communication layer is the only important factor affecting the performance [27]. Furthermore, parallel programs and machines are carefully designed so that messages are lost only very rarely [4, 14]; a message loss is typically treated as a catastrophic event, either the program or the system crashes and needs to be restarted. Thus, most of the previous research has concentrated on studying means of enhancing and evaluating the performance of the communication layers. The diverging spectrum of applications requires extension of this evaluation model to simultaneously evaluate both the performance as well as the robustness of messaging protocols and hence allow a two-dimensional assessment of the design and applicability of the various messaging libraries.

Performability, introduced by Meyer [21], is a metric to simultaneously evaluate the performance and dependability of systems, i.e., to evaluate the performance of systems in the presence of increasing failures. Performance can be evaluated as the effectiveness (throughput) and efficiency (resource utilization) of the system. Dependability encompasses the system's reliability, availability, safety and security. Using the metric of performability, messaging protocols can be chosen based not only on their performance, but also their availability, failure recovery semantics and performance in the presence of various failures.

1.1 Contributions

In this thesis, we explore the requirements and design considerations of building a high performance robust messaging protocol and present a methodology for the quantitative, albeit approximate, evaluation of its performability. It makes the following contributions:

1. Exploring the design considerations of building high performance messaging protocols that provides fault tolerance support for high availability systems.

2. Presenting the design and implementation of *Active Messages* over *Virtual Interface Architecture* (AMVIA) with these considerations and contrasting its performance and resource allocation against an existing Active Messages over User Datagram Protocol (AMUDP) implementation.

3. Formulating a performability model for evaluating messaging protocols based on performance in the presence of failures

4. Using our model to analyze the performability of the two protocols, AMVIA and AMUDP against:

   - component failures like NIC, wire, switch, kernel, library and application
   - overall failure (Bit Error Rate).

1.2 Outline

Our work attempts to specify the various dimensions of comparing messaging protocols and formulate a model to weigh these metrics according to different application requirements. We first study the requirements and the design considerations of a fault tolerant messaging system which should be able to provide feedback to the application about the transient and/or permanent failures. We then develop AMVIA, a messaging layer based on these considerations using the Active Messages paradigm over the Virtual Interface Architecture for highly available systems in a high performance networked cluster environment. Active Messages (AM) is a essentially a lightweight remote procedure call for message-based communication in parallel and distributed computing systems[8]. The Virtual Interface Architecture (VIA) defines an Application Programming Interface (API) for connecting high-performance network and computer systems. The main idea behind VIA is that user applications are provided with a protected, direct channel to the underlying network, thus providing the ability to bypass the operating system and the standard networking protocol (TCP/UDP, for example) stack. We then evaluate its performance against an existing Active Messages implementation over UDP (AMUDP) and analyze the overhead of the additional error semantics in AMVIA. We present the

performability modeling of these messaging systems using Markov reward chains and show how it can be used as tool to contrast their *degradability*, i.e., the degradation of performance in the presence of faults (malfunctions). The messaging system is broken down into its underlying components like the NIC, wire, switch, etc. and the degradation of the performance against increasing failures in each of these components is derived. We further validate it against experimental results obtained by injecting faults into the network.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 presents related research for this work. Chapter 3 presents the design and implementation of AMVIA as a state based messaging system to support failure recovery. Its messaging protocol and architecture are compared with AMUDP and their recovery states are presented. In Chapter 4, we derive an analytical model to quantify and demonstrate the sensitivity of the response time of a messaging protocol with increasing *Bit Error Rate*. The model also allows evaluating the messaging layers against failures in one component only, like the NIC, wire, switch etc. and determine the response time of each in view of increasing failure rates. This model is then validated against results obtained by fault injection experiments for the AMVIA and AMUDP protocols. Chapter 5 presents conclusions and directions for future work.

Chapter 2

Related Work

Other work in building high performance messaging systems includes Fast Messages over Myrinet [23] for optimizing the software messaging layer that resides between lower-level communication services and the hardware. Previous research on developing the Active Messages semantics over VIA [2] was built without considering fault tolerance and robustness. Specific to the VIA architecture, [19] demonstrated the feasibility of layering the distributed component object model (DCOM) over VI primitives. However, the costs associated with implementing the DCOM abstraction at user-level dwarfed the impact of mapping to the VI architecture per se. Our work extends this by using a much lighter-weight starting point (Active Messages) to inherit the characteristic low-latency of VI based communication.

The Horus project seeks to develop a communication system addressing the requirements of a wide variety of distributed applications. Horus implements the group communications model providing (among others) unreliable or reliable FIFO, causal, or total group multicasts. This architecture enables groups with different communication needs to coexist in a single system. The approach permits experimentation with new communication properties and incremental extension of the system, and allows the layer to support varying application consistency requirements. The underlying abstraction of Horus is based on two-phase commit protocol and thus has a high overhead for applications have weak consistency requirements.

Dependability evaluation involves the study of faults, errors, failures, and recovery. Most dependability studies in literature focus on hardware faults or software defects. Only a few studies [9, 12] have concentrated on communication faults, such as corrupted or lost messages. NEST [12] simulated the effects of connection faults in messages.

Orchestra [9] injected faults between protocol layers to study their effect on outgoing and incoming messages. One study [5] measured the effect of communication faults by corrupting message headers. An issue that has not been addressed is how to determine the network dependability for different network components, of the interface hardware, the control software, or the software that performs the data transfer.

There also has been research efforts in characterizing the reliability of communication networks in terms of the major design issues such as resource placement [25], [22] and topological optimization [28], [24], [17]. These studies use stochastic techniques like Petri Nets or analytical evaluation techniques like graph theory to derive optimal design of reliable networks in view of node and link failures.

There has been research in modeling the reliability of a communication network in terms of failure of the end nodes using Markov Chains [13]. The authors present the model which can be used to compare alternative strategies for managing two copies of a database in a distributed computing system. They present the response time in both models with failures from the point of view of a user request.

Our work differs from the above research in using the underlying messaging components as building blocks for a Markov chain model for reliability analysis of different messaging protocols. All of the research done so far in network modeling have only quantified reliability against failures of nodes and links as a whole, but do not break down the performance sensitivity into individual component failures.

Chapter 3

Active Messages over VIA - A High Performance Messaging Substrate

High availability systems need to export strong recovery semantics and ensure application consistency in presence of failures. One way this can be accomplished is by providing feedback to applications so that they can recover from intermittent errors such as link and network failures and export consistent error states in the case of permanent failures. Reliable communication paradigms that currently exist are broadly classified into two groups. Messaging systems like transactional remote procedure call [10] provide at most once semantics and a strong consistency model using two-phase commit protocol. Some applications may not want transactional semantics because the commit protocol has a high overhead in terms of increased message complexity and increased latency with the need for stable log to allow recovery through rollback. On the other hand, protocols like TCP/IP build reliability over underlying best effort network using retransmission as the primary means of recovery. Thus, the protocol does not guarantee absolute consistency and the cause and location of the failure in the byte stream is often indeterminate.

The motivation for designing Active Messages [20] over Virtual Interface Architecture (AMVIA) [15] library was to build a high performance messaging layer with clean semantics for applications to recover from errors without explicit fault recovery support.

In this chapter, we first present the design and implementation of AMVIA - the Active Message over Virtual Interface Architecture messaging layer for highly available systems in a high performance networked cluster environment. An earlier work [2] implements a similar library over VIA, but our concentration has been to extend the

VIA error model and provide additional information at end nodes to recover from failures. We then compare the performance and recovery semantics of AMVIA with an existing protocol also based on the Active Message paradigm - AMUDP (Active Messages over UDP).

3.1 Active Messages and the Virtual Interface Architecture

3.1.1 Active Messages

Active Messages (AM) is a request/reply model like asynchronous remote procedure call for message-based communication in parallel and distributed computing systems[8]. Each message contains the name of a user-level handler to invoke on a target node and a data payload to hold the arguments. The handler function serves the high-level purpose of extracting the message from the network, integrating the data into the computation and sending a response message.

We choose to base our work on Active Messages because this model has been shown to perform well and give the applications semantics for building Internet services.

3.1.2 Virtual Interface Architecture

The Virtual Interface Architecture (VIA) is a standardized user level networking API to allow user-process to user-process communication without kernel intervention. The main idea behind VIA is that user applications are provided with a protected, direct channel called Virtual Interface to the Network Interface Controller (NIC), thus providing the ability to bypass the operating system and the standard networking protocol stack e.g., TCP/UDP, if the application so desires. This is accomplished by registering user memory with the NIC directly before use. This allows messages to be sent and received without going through the OS kernel so that the NIC can perform the necessary virtual memory to physical memory address translation. The VI architecture can be broken down into the following components:

- *Registered Memory* - A portion of a user's virtual address space that has been pinned into physical memory and made known to a VI NIC. Registered memory

functions as the principle communications buffer for network operations. A unique name of Memory handle is associated with each region and used in conjunction with a user virtual address to access a buffer.

- *Descriptor* - A data object recognized by the VI NIC that describes a network transfer request to be performed. Descriptors reside in registered memory.
- *Work Queue* - A FIFO list of Descriptors to be processed by a VI NIC.
- *Doorbell* - A mechanism for a user process to notify the VI NIC that outstanding descriptors have been posted to a work queue. Each doorbell is a protected resource, typically mapped into a user's address space, which is unique to a particular VI/user pair.

A VI consists of *send* and *receive* work queues, their associated doorbell resources and the user's registered memory regions. To initiate a network data transfer, the user process builds a descriptor and inserts it into appropriate work queue by placing a token in the queue's associated doorbell. When a VI descriptor is completely sent or received, the NIC places an entry into the completion queue that indicates which VI completed the operation. The user process receives this notification either through polling or via interrupts.

3.2 Active Messages over VIA - AMVIA

AMVIA uses a three-way request-reply primitive for robustness and error recovery as shown in Figure 3.1. We call the node that sends the request as the *initiator* and the node that computes and sends the reply as the *target*. When the target receives a request, it extracts the handler id from the message and performs a lookup using this index. On successful discovery and invocation of the handler, an acknowledgement is sent back to the initiator. Subsequently, the target computes and sends the reply.

We thus see that our protocol differs from a general two-way request/reply protocol due to the addition of the acknowledgement. The major advantages of this addition are -

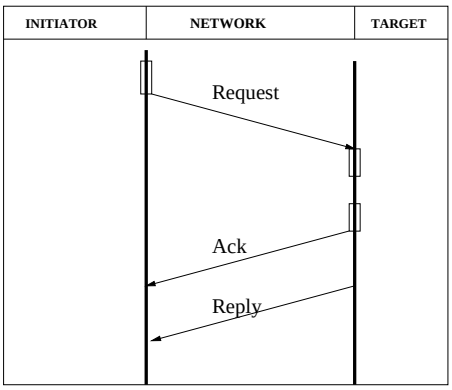

Figure 3.1: **AMVIA Three Way Request Reply Protocol**

1. It provides extra error recovery information -on timeout after successive retransmission attempts, the initiator can now localize the error to be the failure of remote node or the network. In case of a general two-way protocol, a request timeout could additionally be due to handler crash or loss of the reply.

2. The acknowledgement also allows us to set the request timeout value independent of the handler computation time, and some factor of the round trip time. For AMVIA, we set it to be about twice the round trip time. In case of a generalized two-way protocol, the request timeout will have to be pessimistically set to the maximum handler execution time, which is usually set at around one second, as the handler may perform some I/O operations.

3. It allows the layer to be designed for finite handler execution time. Since the acknowledgement signifies the start of the handler, the initiator can give up on the handler after a finite predetermined time and log an abandon state. This feature is different from most messaging layers as they are designed to allow the handler to run infinitely.

Another important design consideration was the logging of the intermediate states in a memory mapped file to allow applications to recover from process failures. For error recovery, we designed AMVIA carefully as presented in the state transition diagram for the initiator and the target nodes. Each request operation is bound by the instant the initiator issues a request and when it receives the reply. Each operation has two state

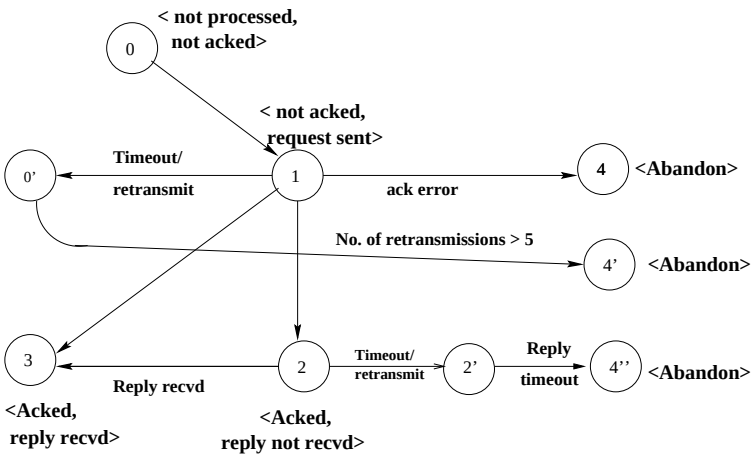

Figure 3.2: **State Diagram for Initiator Node in AMVIA**

variables, *Ack/Reply Status* and *Operation Status*. The application can always query the AM layer to find the state of the send operation encapsulated by the <Ack/Reply status, Operation Status> tuple.

3.2.1 State Diagram of the Initiator Node

Figure 3.2 shows the AMVIA state diagram at the initiator node. The system is said to have failed when it moves to any of the Abandon states.

- *State 0*: On an AM_Request(), the operation is initialized using the Tx_creat command to have default states of <NOT_ACKED, NOT_PROCESSED>.

- *State 1*: On successfully posting the message, the entry is updated to hold the data that was sent and the status of the operation is set to <NOT_ACKED, REQUEST_SENT>.

- *State 2*: On receiving the acknowledgement before the timeout, the state is updated to <ACKED, REQUEST_PROCESSING>. The acknowledgment also indicates that the request handler was found. Else, the operation returns to State 0', and the retransmission mechanism is triggered.

- *State 3*: On receiving a reply for the request, the state is updated to <ACKED,PROCESSED> and the reply data is logged on to the transaction

record before calling the reply handler. This ensures persistence even if the process crashes at this time, or the reply handler is not found.

- *State 0'*: On an ack timeout, the message moves to the timeout state and the request is retransmitted for a maximum of k times.

- *State 4*: If the remote handler cannot be found, the target node sends a negative acknowledgement, which is logged as an error state <ACK_NOT_FOUND, REQUEST_SENT>. The message request is hence abandoned.

- *State 4'*: In case there is no acknowledgement/reply even after k retransmissions, the message is abandoned in state <NOT_ACKED, REQUEST_RTX_EXCEEDED>. This error state indicates a catastrophic error like the cable being disconnected, switch failure or remote node crash.

- *State 4"*: In case of a reply timeout, an identical retransmission mechanism is initiated. In case of no reply for N retransmissions, the message is logged with error state <REPLY_RTX_EXCEEDED,REQUEST_SENT>.

On reaching state 3, the record can now be freed and reset to default state for next request.

3.2.2 State Diagram at the Target Node

The receive operation is initialized with an id assigned to it on an AM_Request command. Each operation has two state variable, *AckStatus and OperationStatus.* These variables take on values and encapsulate the state of a receive at any instant.

The states are described as a *<AckStatus, OperationStatus>* tuple.

- *State 0*: On an AM_Reply(), the state is initialized using the Tx_creat command to have default states of <NOT_ACKED, NOT_PROCESSED>.

- *State 1*: On receiving a request, the handler id is extracted from the message and a lookup is performed in the handler table. If found, the status is set to <NOT_ACKED, REQUEST_RECVD>. If the handler is not found in the

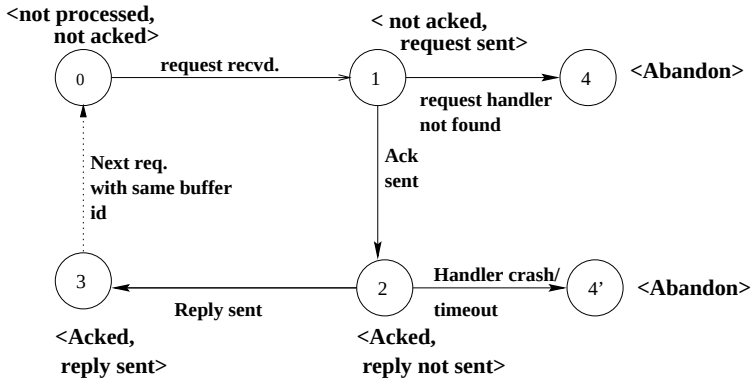

Figure 3.3: **State Diagram for Target Node in AMVIA**

endpoint, an error acknowledgement is sent and the operation state proceeds to Abandon <ERR_ACK, NOT_PROCESSED>.

- *State 2*: On sending the acknowledgement, the state is updated to <ACKED,REPLY_NOT_SENT>. The acknowledgment also indicates that the request handler was found.

- *State 3*: On sending a reply, the operation state is updated to <ACKED,REPLY_SENT> and the reply data is logged on to the record before calling the reply handler.

- *State 4*: If the handler cannot be found, the target node sends a negative acknowledgement, which is logged as an error state <ACK_NOT_FOUND, REQUEST_SENT>. The message transaction is hence abandoned.

- *State 4'*: On an reply timeout, the message moves to the timeout state and is abandoned.

The AMVIA protocol has the possibility of the message moving to Abandon state, State 4‘ in Figure 3.3 where the state of the request is indeterminate. At this point, the handler may have commenced execution, after which it is no longer possible to abort the operation. This state weakens the consistency of our model as compared to atomic commit protocols like two-phase and three-phase protocols where a transaction can be either committed or aborted.

We see that the Abandon State 6 in Figure 3.2 and State 5 in Figure 3.3 allow the application to recover from requests that lead to infinite execution or when the handler does not give the control back to the library in a reasonable pre-estimated time.

3.2.3 Message Logging

As mentioned earlier, the AMVIA messaging layer provides feedback to user level applications about the state of messages sent/received to/from remote connections. Thus, the actual decision regarding the action to be taken on intermittent or permanent link/network failures is left to the applications themselves. For example, a sending process could use the AMVIA primitives to constantly check if a reply has arrived or not. If it did not arrive till a timeout period, it could retransmit the request message to another symmetric node. Similarly, a target node could look at an incoming message and determine if the message had arrived earlier and was already processed (from our earlier discussion of the AMVIA states, this could happen if the initiator node crashed after sending out a message request, recovered and resent the message since it was not acknowledged with a corresponding reply) and if so, send out the earlier response thus providing primitives for delivering *at-most once* transaction semantics to higher level applications.
A transaction data structure maintained by the AMVIA library is composed of the following data:

- A unique transaction id.
- The packet itself to be sent out.
- Status of the message. This has two components: an *acknowledgement* that indicates whether the message actually made it across the network to the remote endpoint and a *status*, that indicates whether the message was actually processed at the remote endpoint.

Each process has a *send pool* and a *receive pool* of operation states. Every time messages are sent out, they are logged into the send pool and when corresponding replies arrive,

they are logged into the receive pool. The application process sending the request message is given a handle to the send pool entry corresponding to the operation just requested. When the acknowledgement for the request arrives, the state of the request is updated to reflect this. Later on, the process can use the handle to check if the message was actually processed or not. The action to be taken in case of a failure in any of the intermediate states is left to the application process. Once the message is written into the VI descriptor's data segment, the doorbell is rung and the descriptor is shipped out. On the receiving end, when the message arrives, an acknowledgement is first sent out to the endpoint that sent the request message. The message is then extracted from the descriptor, its id is obtained, and a corresponding log is created before the message is actually processed. After the message is processed, the result is logged again using the id, and a reply message is sent back to the requesting endpoint.

3.3 AMVIA - Implementation Overview

The fundamental abstractions of Active Messages have been implemented in accordance with the AM specifications. The complete specifications of AM can be found in [*Add reference here to the AM specification*]. The main features of the AMVIA implementation include:

1. Endpoints and Bundles: The AM *endpoint* is an abstraction for a process' connection to the network. Endpoints implement a two-phase request/reply scheme in which a request message is paired with a subsequent reply message. The AM *bundle* abstraction permits user-level polling of an arbitrary collection of endpoints.

2. Different Message Sizes: AMVIA supports three message sizes for flexibility for different applications: short (< 32 bytes), medium (< 128 bytes) and bulk transfers ($<$ network MTU size).

3. Flow Control Mechanism: Endpoints use a credit-based flow control scheme prevent network congestion and buffer overflow

4. Naming: In order to hide the network addressing details, endpoints are given unique names and registered with an internal AMVIA Name Server. Endpoints that need to make connections refer to remote endpoint by these names, thus providing an useful abstraction to user applications.

5. Peer-to-peer Connection Model: This provides both blocking and non-blocking semantics and ensures symmetrical code on both the initiator and the target nodes (or connecting processes), thus obviating any need for prior knowledge of the hierarchy of the communication framework. This is implemented by using the primitives provided by the underlying VIA Giganet API [add reference to the Giganet API specs].

6. Logging Mechanisms: Outgoing and incoming messages are logged into user-specified memory mapped files. The motivation behind this is to derive extra state from network traffic without loss of performance (by eliminating the extra copies for maintains dedicated message logs) to perform recovery in the event of intermittent and permanent network and link failures.

3.3.1 AMVIA Operation - Sequence Diagrams

AMVIA Initialization Sequence

During initialization, each participating node creates their respective endpoints using the AM API call `AM_AllocateEndpoint()`. The respective endpoints are then registered with the AMVIA Name Server by using the AMVIA API function `AMVIA_RegisterEndpoint()`. On successful registration with the Name Server, the participating endpoints (and respective processes) are ready to perform subsequent send and receive operations. The sequence of operations that occur during this process is shown in Figure 3.4.

AMVIA Send and Receive Operations

The sequence of events that occur during send and receive operations in AMVIA are depicted in Figure 3.5. Sending operation in the AMVIA are straightforward and vary

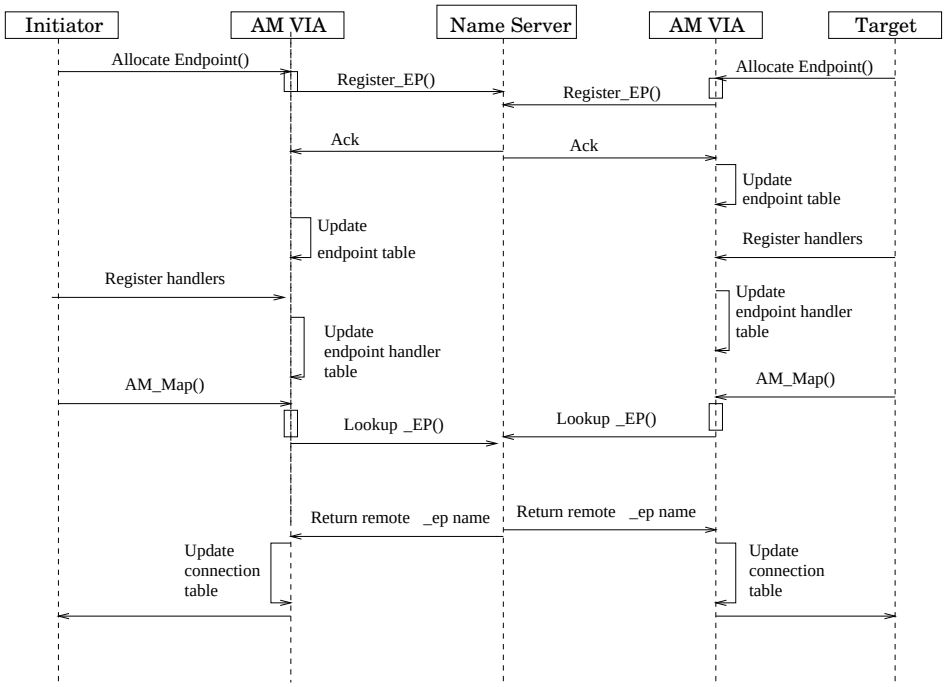

Figure 3.4: **Sequence of Events that Occur During AMVIA Initialization**

slightly for the short and medium message. For an `AM_Request()`, the function attempts to obtain a free descriptor and receive credit. If both are not available, the function polls until it can proceed. The data payload is then copied into the descriptor memory and posted onto the send queue. The operation then returns without blocking for the descriptor to complete. The `AM_Reply()` functions in an identical manner except that it does not wait for a request credit.

The sequence of operations that take place during an AMVIA receive operation vary depending on the message size. Short messages are processed by directly invoking the designated handler with the data arguments(the copy here is implicit). For medium messages however, the copy is avoided by passing the pointer to the data region directly to the handler. Thus, incoming medium messages are able to exploit the zero-copy semantics intended for VI architecture. Once the handler returns, the associated receive descriptor is cleared and re-posted to the VI's receive queue. The fact that the receive descriptor is not recycled until *after* the handler completes requires the receive queue to contain one extra element. This is to ensure that a reply sent by a

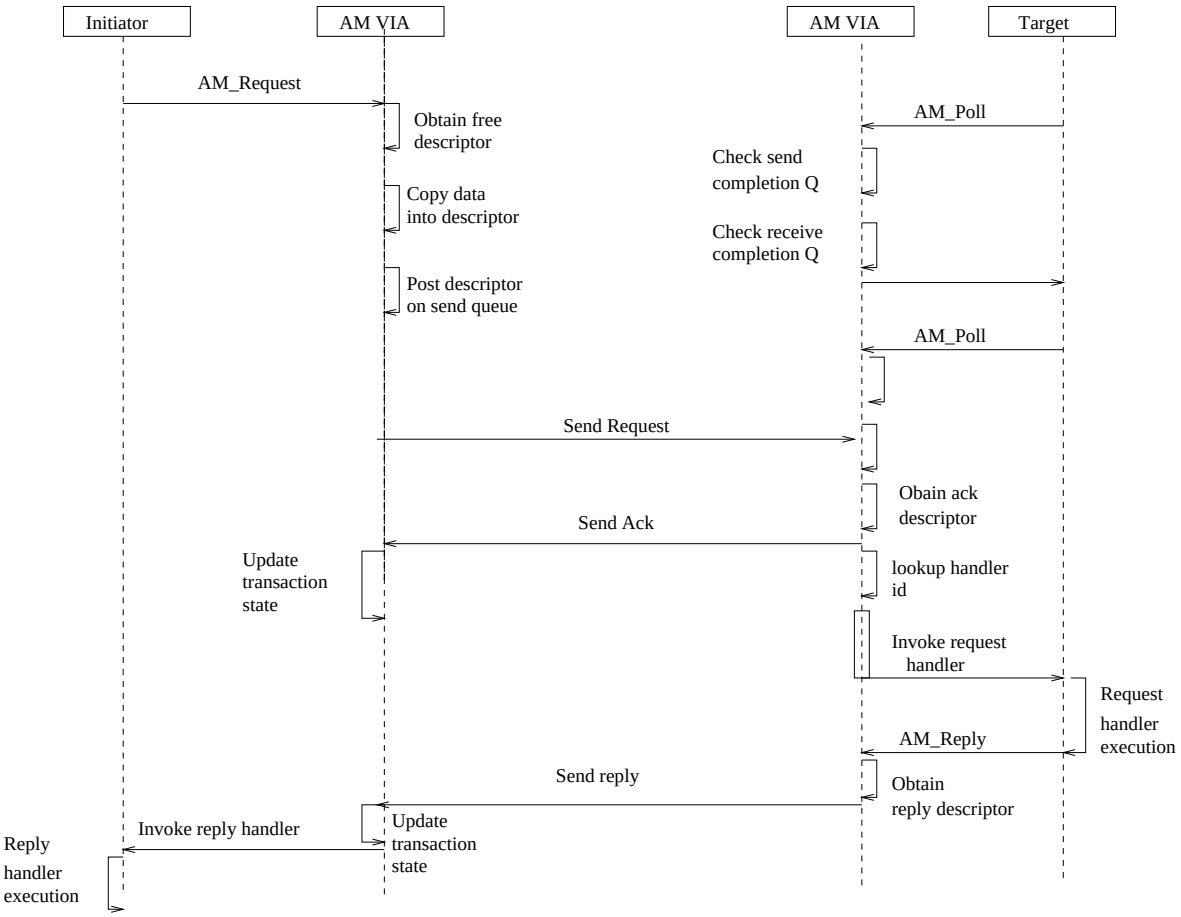

Figure 3.5: **Sequence of Events that Occur During AMVIA Send and Receive Operations**

request handler does not create a new request handler from which there is no available buffer. Recycling the receive descriptor before invoking the handler would require extra data copies that would degrade performance. It also allows us to maintain the copy of the reply till the end of the transaction to maintain *exactly once* handler execution in case of possible duplicate requests or retransmissions. The receive operations are channelled to the respective queues in the `AM_Poll()` operation. Both the short and the medium message queues are associated with the bundle send and receive completion queues. The `AM_Poll()` routine checks the receive completion queue in the bundle for incoming messages. For each received message, the routine performs a lookup of the associated AMVIQ, determines the request type, request or reply, and routes the

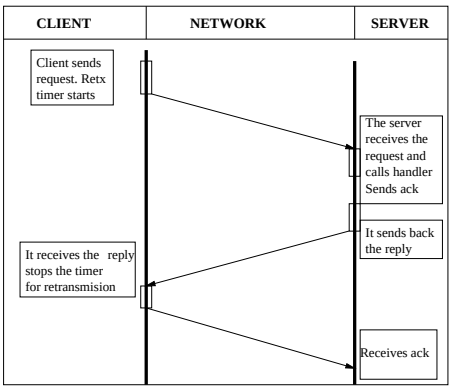

Figure 3.6: **AMUDP Three Way Request Reply Protocol.**

message to the respective handler. The handlers, on completion, return the control back to the `AM_Poll()` which sends the reply descriptor, if required. The reason for postponing the reply till after the handler execution is that, in this case, the sending of the reply indicates that the handler had successfully completed execution. This also clears up the reply descriptor for reuse. The other purpose of the polling routine is to recycle the send descriptors. The head of the send completion queue is checked once per call to `AM_Poll()` and the completed send descriptor marked available for reuse.

3.4 Performance

3.4.1 Resource Allocation

We now start to evaluate the performance of AMVIA in the absence of faults. To give context in which to interpret these results, we contrast our protocol against an existing AMUDP messaging architecture also built on the Active Messages paradigm. This protocol also implements the three-way request reply model as shown in Figure 3.6. AMUDP is essentially a two way request/reply paradigm with an additional acknowledgement at the end for release of reply buffer. The main characteristics that differ between these protocols are:

1. **Acknowledgement:** In AMVIA, we observe that the request timeout is the maximum time required for the acknowledgement to return and is independent of the request type or handler execution times. However, in AMUDP, the request

timeout is maximum handler execution time and needs to be optimally set at a much higher value.

2. **Resource Buffer Allocation:** AMVIA three-way and AMUDP three-way have a slightly different approach to resource release. To ensure *at most once* semantics, AMVIA holds its reply descriptor till the next request comes in. If each connection uses two buffers of size k, one for send and one for receive for each connection, and a network of size N can have potentially N×N connections. The total buffer size required at each node is thus equal to $N \times N \times 2k$. Conversely, AMUDP uses a common send pool for all send requests and replies, and the extra acknowledgement to release the buffer. This ensures better scalability for bigger networks, as the buffer size now is k, where k is independent of N.

3. **Buffer lifetime:** We see that the lifetime of the reply buffer is bound to the acknowledgement in AMUDP, whereas in AMVIA the buffer lifetime is unbound.

The detailed state diagrams for AMUDP are given in Appendix B.

3.4.2 Throughput

The first step in quantitative evaluation of any messaging layer is its performance. We use the LogGP [7, 1] model (Figure 3.8) to analyze the cost of sending and receiving messages because it allows characterizing the performance of the key resources but not their structure. Thus, this cost model is independent of particular hardware/software implementations.

LogGP

L: the *latency*, or delay, incurred in communicating a message containing a small number of words from its source processor/memory module to its target. The latency includes the time spent in the network interfaces and fabric, but not in the processor.

o: the *overhead*, defined as the length of time that a processor is engaged in the

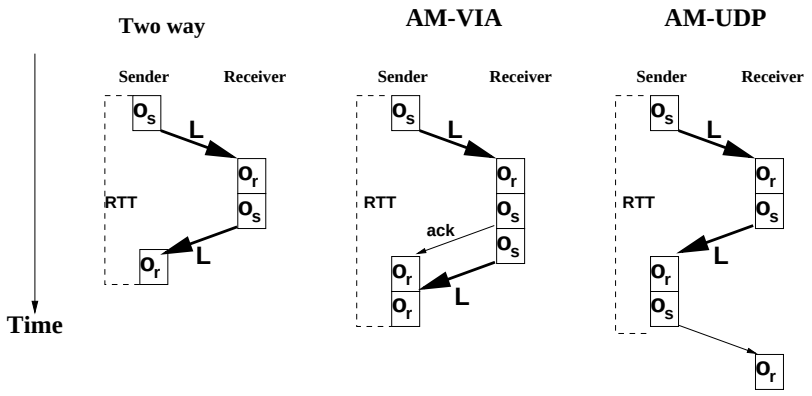

Figure 3.7: **Protocols of 3 Active Message Layers.** *The Figure shows the internal protocols used by the "generic" layers in previous work, and our two AM layers*

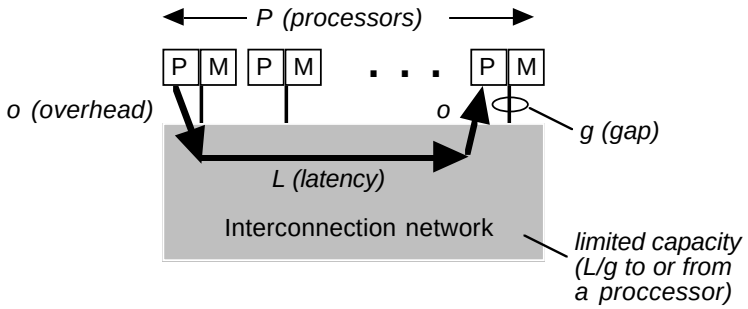

Figure 3.8: **LogP**

transmission or reception of each message; during this time, the processor cannot perform other operations.

g: the *gap*, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a module; this is the time it takes for a message to cross through the bandwidth bottleneck in the system.

G: the *Gap*, or time-per-byte for long messages. The inverse of G is the peak bandwidth. This parameter was added [1] because many platforms have special acceleration for long messages (e.g. DMA).

P: the number of processor modules.

Figure 3.7, shows the LogGP parameters of the two systems.

Measuring LogP

We use the LogP benchmark [6] to measure the LogP values for AMVIA and AMUDP. The benchmark works by sending incrementally increasing bursts of messages to generate a signature curve for average Message Cost as a function of number of messages sent, M as shown in Figure 3.9. For small M, the sender will issue all the requests without receiving any replies as indicated in the 'send-only' portion of the graph. For larger M, a fraction of the replies will arrive during the issue phase and the message cost will increase till it becomes equal to the gap g. The average message cost in the send-only regime reveals o_s. We can then measure gap by direct measurement. With these two parameters and by inserting extra delay between every message for δ time, we can observe receive overhead, o_r. Finally, we can derive the latency, L, by measuring the round-trip-time (RTT) and subtracting out the overheads.

send overhead

o_s is measured by sending a small burst of messages and computing the average cost per message.

gap

For a burst with only a small number of sends, the average messaging cost defines the send overhead, o_s. In bursts with large numbers of messages, the cost of each send approaches the steady-state initiation interval g. For both our communication systems, the bottleneck is the send and receive overheads. Figure 3.7 shows that in both our layers, an "extra" message is required in the three-way protocol. Taking into account this extra message, g is always equal to steady-state cost of the slower side, which, assuming that $o_r > o_s$, is $o_s + 2o_r$.

receive overhead

We can compute o_r in units of time by using measured values of o_s and g since $g = o_s + 2o_r$.

Latency

Measuring L with a 3-way protocol again requires care. We can see from Figure 3.7 that there is a *critical path* for a round-trip message. Assuming that other work is perfectly

overlapped, we know the critical path of an RTT is composed of only o_s, o_r, and L. Having measured the RTT, o_s, and derived o_r, we can compute L for both protocols as:

$RTT = 3o_s + 2o_r + 2L.$

Gap

To measure G, we send bursts of large messages, each with a fixed size. We then derive the bandwidth from the steady-state initiation interval and message size.

3.5 Results

3.5.1 LogP

Our experimental test-bed is two 400MHz Celeron nodes which have 66 MHz system buses and 33 MHz PCI buses. For the AMVIA measurements, we use a pair of Giganet cLan cards connected back-to-back, running the clan-1.0.1 driver. For the AMUDP measurements, we use a pair of Kingston 100 Mb/s ethernet cards (DECDC21140 chips), running the tulip driver connected by an ethernet switch. All experiments were run on Linux 2.2.12. We present the parameters are presented as obtained from the LogP performance benchmark [8, 18].

We observe in Figure 3.10(a) that the gap cannot be measured for AMVIA 3 way protocol. This is generally true of layers where the latency of the wire is lesser than the sum of send and receive overhead. This indicates that a new request does not have to wait for the previous reply to be drained. The analogous LoGP performance signature for AMUDP is shown in Figure 3.10(b). The values of the LoGP parameters derived from these figures are summarized in Table 3.1. We see that we have successfully built AMVIA as a high performance messaging layer with an additional round trip overhead of 16 μ secs over VIA RTT of 8μsecs. The corresponding overhead for AMUDP is close to 60 μsecs over raw UDP RTT. In addition, the send and receive overheads show a improvement by a factor of 10 for AMVIA as compared to the AMUDP implementation.