An Enhanced Communications Package for Amadeus

Karl Jeacle

B.A. (Mod) Computer Science
Final Year Project, May 1992

Supervisor: Vinny Cahill

Abstract

This report describes extensions to the Communications Subsystem of the Amadeus Kernel developed at Trinity College Dublin.

In the Amadeus system, the CS provides the communication services necessary to carry out all remote operations. Inter-kernel communication in Amadeus is accomplished through message passing between corresponding kernel components of distinct kernels. The Inter-Kernel Message protocol is a lightweight message transaction protocol, which provides the necessary support for reliable communication between kernels.

Currently, the IKM has been implemented above UDP/IP on Digital Equipment Corporation's Ultrix 4.2 and Sun Microsystem's SunOS, on MicroVAXes, DECstations, and SPARCstations.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Amadeus & the IKM

Amadeus[1] is a development project undertaken by the Distributed Systems Group of the Department of Computer Science, Trinity College Dublin (TCD). The Amadeus environment provides an integrated platform for the development of distributed applications. It extends C++ for distribution and persistence[2]; the extended language is called C**.

The Inter Kernel Message protocol (IKM) is the communications subsystem of Amadeus. All nodes running Amadeus communicate using the IKM protocol.

The aim of the IKM is to provide communicating Amadeus nodes with a reliable transport service. It was originally designed [3] for use with high-speed, low-error rate LANs. Currently, the IKM is built on top of UDP/IP — an unreliable datagram service with limited packet size. The IKM overcomes these limitations, and provides many additional features.

1.2 Project Aim

The aim of this project was to extend the IKM to provide support for the following:

- Variable Sized Messages — Previously, messages had a maximum size of 2 Kb, and had to reside in a contiguous block of memory. The size of the reply message had to be equal to the size of the request message.
- Message Fragmentation — Since arbitrary sized messages were to be supported, a fragmentation scheme was necessary within the IKM which split large messages into smaller chunks suitable for transmission over the network.

- Message Forwarding — Support for forwarding was added. Previously, this was not available. A node is able to forward a message to another node, without leaving any dependencies on itself. This action is transparent to the original sender.
- Multicast & Multisend — Provision for synchronous and asynchronous simultaneous message sends to specified 'groups' of nodes.

In order to add support for the above, much of how the existing IKM was designed and implemented had to be re-evaluated. On the outside, the user interface remained mostly the same, but internally, the IKM underwent a complete overhaul, both in terms of design and how it was implemented. In fact, apart from the interaction with UDP, the IKM code was completely rewritten.

A more modular approach was adopted throughout the implementation and the code was split up into layers to provide for future maintenance and enhancements.

With state machines (see Chapter 3) governing the operation of the IKM, the code implementation was able to strictly follow the design.

1.3 Report Layout

Chapter Two discusses *related projects* which influenced the design of the IKM.

Chapter Three describes the *IKM interface* and the *design* of Message Fragmentation, Message Forwarding and Multiple Send. State diagrams for both sending and receiving sides are included and explained.

Chapter Four discusses the *implementation* of the IKM from the ground up, including how the extensions mentioned above were incorporated.

Chapter Five contains *performance* tables and graphs. Justification of these results and comparison with other systems is included.

Chapter Six is a *summary* of the report. This includes a *critique* of the IKM as it stood at the end of the project and discusses shortcomings, and possible improvements.

Bibliography

Appendix A is a *Programmer's Guide to the IKM*, including example source code showing how to use the IKM.

Appendix B contains the complete project source code.

1.4 Acknowledgements

Obviously, I'd like to thank my supervisor, Vinny Cahill, for all his insights on how to approach various problems which arose during the year. I'd also like to thank John Moreau for his help, advice and reassurance throughout the year, especially in the early stages, when the project seemed like such a large undertaking.

Chapter 2

Background

2.1 Introduction

This chapter is intended to provide the reader with background information on related projects which influenced the design of the IKM.

2.2 Protocol Classification

A basic strategy in common use which provides *reliable* transmission between sender and receiver is Automatic Repeat Request (ARQ) [4]. There are two types of ARQ — Idle RQ and Continuous RQ, both of which are discuss below.

The essential characteristic of Idle RQ is that only one block can ever be outstanding during transmission. When the sender sends block n, he can not send block $n+1$ until until he knows that the receiver has successfully received block n. There are several ways to achieve this:

1. Full Acknowledgement — an ack/nack (or message of some sort) is sent by the receiver for every block received/not received. The BSC [5] protocol for example requires the receiver to transmits alternate 'ack 0' and 'ack 1' messages for each message received. The sender can identify from these which blocks (if any) went missing.
2. Positive Only Acknowledgement — the receiver only acks blocks successfully received. The sender will timeout and retransmit if no ack is received for a block.
3. No Acknowledgement — UDP is an example of this. Unreliable transmission, often known as a 'Datagram Service'. Error control must be handled at higher levels. There is no flow control.

4. Negative Only Acknowledge — Receiver only sends back a nack when an error is detected. This scheme is not used in data communications.

Continuous RQ systems differ from Idle RQ systems in that more than one frame can be outstanding. In cases where the 'bit-length' of the link is greater than the block length (e.g. satellite channels), Idle RQ can be very inefficient. This is analogous to the delayed response when talking to a person on the other side of the world via telephone. The sender may have (almost) timed out and sent a retransmission. In cases such as this, Continuous RQ is used. Three types of Continuous RQ are:

1. Selective Retransmission — sender transmits blocks without waiting for immediate acks. The receiver sends ack(n) when it receives block n. The sender retransmits blocks which were not acked. Blocks can arrive out of order at destination, leading to a lot of buffering if the blocks are related.

2. Go-Back-N — If block n is corrupt, nack(n) will cause all blocks from n onwards to be retransmitted. Frames arrive in order so no buffering is required, but this is more wasteful in terms of network throughput than Selective Retransmission, since many blocks which were transmitted error-free may be discarded simply because one previous block was lost or corrupt.

3. Sliding Window — both sender and receiver maintain state on the number of blocks sent and received so far. By sending an ack(r), the receiver is acknowledging all blocks up to $r-1$. There is a 'maximum window size' associated with the link. This is the maximum number of blocks the sender can transmit before receiving an ack. The receiver can choose how much to acknowledge — thus forcing the sender to match its capacity.

Both Idle and Continuous RQ protocols are pessimistic. Neither sends all the blocks at the same time; Idle RQ systems allow only one block to be outstanding, acking/nacking each block, while continuous RQ systems allow several blocks to be outstanding, but still ack/nack for chunks of blocks.

2.3 Amoeba's FLIP

The Amoeba Distributed Operating System[6] was designed and implemented by the Distributed Systems Group at the Vrije (Free) University in Amsterdam. The Amoeba Kernel was designed with many of the components of a conventional OS such as the file system outside the kernel. Interprocess Communications, however, is included *within* the kernel.

The IPC facilities of Amoeba are intended to support *remote operations*. All communication takes place between (threads in) processes taking on the roles of *clients* making requests to *servers* to carry out operations. The act of sending a request to a server and receiving a reply is called a *message transaction*.

A client invokes a message transaction by using

```
trans(reqhdr, reqbuf, reqlen, rephdr, repbuf, replen)
```

At the server side, the call to receive is

```
getrequest(server, reqhdr, repcnt)
```

A reply is generated with

```
putreply(rephdr, repbuf, repcnt)
```

Both `trans()` and `getrequest()` are blocking calls. It is interesting to note that message reply size is fixed at request time.

Amoeba provides two protocol layers. The bottom layer provides a rate-control, secure, but not totally reliable datagram service. Request size ranges from a single header to 1 Gigabyte of data. The protocol used at this layer is known as the *Fast Local Internet Protocol* (FLIP).

The top layer, called the *transaction* layer consists of the `trans()`, `getrequest()` and `putreply()` calls, and provides a reliable remote-operation service. The reason for this layering is efficiency. Large messages should be split up at as low a level as possible to streamline the sending of individual packets in the device handler, so that kernel scheduling does not have to take place for each packet.

FLIP is implemented using 'stop-and-wait' at the packet level i.e. Idle RQ. At the transaction layer, timeouts and retransmissions are used to provide reliable transactions.

2.4 V's VMTP

The Versatile Message Transaction Protocol was designed for the V Kernel [7] at Stanford University. VMTP is a transport protocol designed to "support the transaction model of communication, as exemplified by the Remote Procedure Call".

The basic protocol is a request/response protocol. A message is sent as one or more *packet groups*. A packet group consists of up to 16k of data, or 21 *segment blocks*. A network packet (e.g. Ethernet) can contain one or several segment blocks. A bitmap of 32 bits, called the *delivery mask* indicates which segment blocks of a packet group are in a network packet. These masks can be ORed together to identify which blocks failed in transmission.

VMTP caters for what it calls *overruns*, when the receiver cannot keep up with the sender. To remedy this, an *interpacket gap* is introduced to allow sufficient time for a slow receiver to get ready for the next packet in time.

Like Amoeba, VMTP uses a 'stop-and-wait' mechanism. Other features include servers labeling responses as being idempotent (so a client can request a transaction many different times without there being any side effects — e.g. requesting time of day), provision for 'best-effort' multicast is supported, where the request is retransmitted until at least one reply is received. VMTP also provides asynchronous message sends which don't require replies; it confusingly calls these *datagrams*.

Secure message transport is another option in VMTP. Each transaction can be made to be secure individually. The mechanism guards against outsider message tapping, replays and imposters. An authentication server and packet encrypter is used to achieve this.

2.5 Protocol Comparison

All of the protocols above are pessimistic in that they assume packets will be lost over the network during large sends. They continually ack/nack every block transmitted. Protocols with state on both send and receive sides will perform well[1] on error-prone connections. However, on networks with low packet loss such as Ethernet LANs, the process of acking/nacking every block will be very wasteful on large message sends, and will increase the message transmission time.

The IKM, on the other hand, is optimistic in its approach. It assumes that only a small percentage of packets will be lost. All blocks are transmitted without delay to the destination. On error-free connections, this results in transfer rates much higher than the 'pessimistic' protocols. On error-prone connections, the IKM will just timeout and enquire which blocks were lost, and then just resend the missing blocks.

According to the classifications outlined above, both FLIP and VMTP use Idle RQ while the IKM uses a form of Continuous RQ. FLIP and VMTP use the Positive Acknowledgement scheme whereas the IKM uses what could be described as a Negative Acknowlegement version of Selective Retransmission, where the receiver only informs the sender of blocks that went missing.

[1] in terms of speed

Chapter 3

Design

3.1 Introduction

This chapter deals with the relaxation of constraints on IKM messages, the introduction of message forwarding, and multiple send.

Messages previously had to reside in a contiguous chunk of memory and were limited to a maximum size of 2 Kb. Message forwarding and multiple send were not supported.

3.2 Layers

Originally, in IKM terms, 'messages' and 'blocks' meant the same thing. A request/reply had a maximum message size of 2 Kb and was sent to another node as a request/reply block.

In order to handle arbitrary sized messages, a fragmentation scheme was needed to split large messages into smaller chunks suitable for sending over the network. These chunks are now known as 'IKM blocks', while the request/reply is known as an 'IKM message'. This request/reply is also known as an 'IKM Transaction', and 'blocks' are often referred to as 'packets', 'fragments' or 'datagrams'.

Figure 3.1 shows how a fragmentation layer was added to the IKM. Clients queue their 'messages' for transmission using the IKM interface. The message layer registers the request within the IKM, and monitors its progress throughout the request/reply transaction. All timeouts and retransmissions shown in Figures 3.3 and 3.4 are dealt with at this level. The fragmentation layer is responsible for the fragmentation scheme outlined above. It deals solely in blocks — directly above the network.

A message can now be presented to the IKM in one of two formats — contiguous or fragmented.

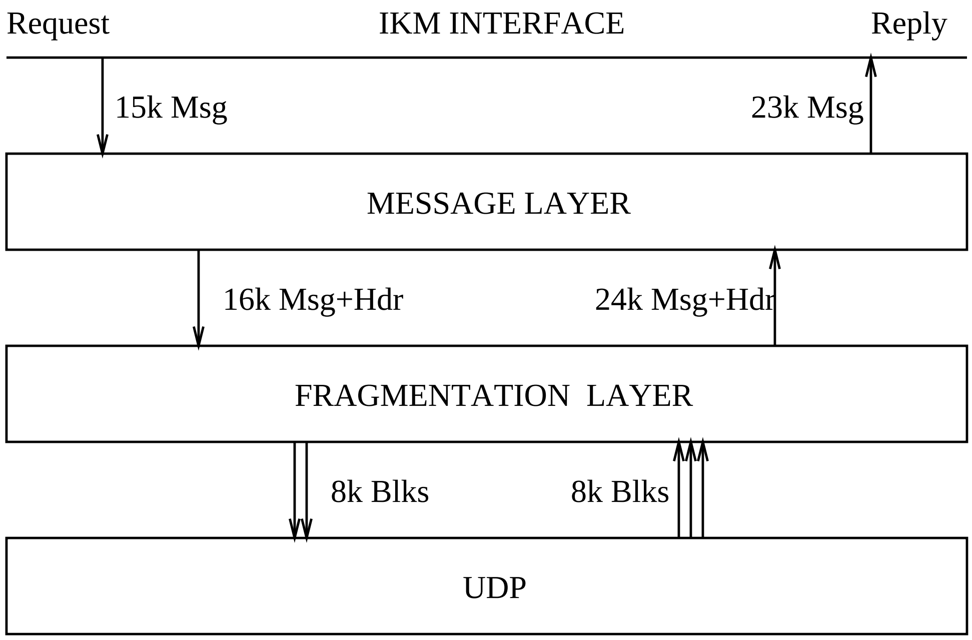

Figure 3.1: IKM Layers

Passing the message contiguously is the approach traditionally used; one large block with preallocated space for the header is presented.

However, now messages can be passed in a fragmented format, using an 'IkmBuffer Message'. IkmBuffers are essentially linked lists of blocks, as shown in figure 3.2. They can be created, read and written to using a set of custom routines (which are outlined in the next section).

Normally, when a message is presented, the IKM has to split it into fragments suitable for network transmission. Internally, the IKM always deals in these fragments. Therefore, the aim of the IkmBuffer message is to save the IKM from fragmenting messages, by letting the user present them in a form which is already optimally sized for transmission.

The advantages to the user are speed and convenience. Speed, since the IKM can process these messages more efficiently, and convenience since the user does not have to preallocate space for IKM headers, or know in advance how long the request message will be.

3.3 The IKM Interface

There are thirteen user-interface routines to the IKM.

`aon_ikm_init()` Initializes the IKM. Opens an Internet Domain socket[11] for network communications. Creates two lightweight processes, one to handle incoming messages, and one to handle timeouts.

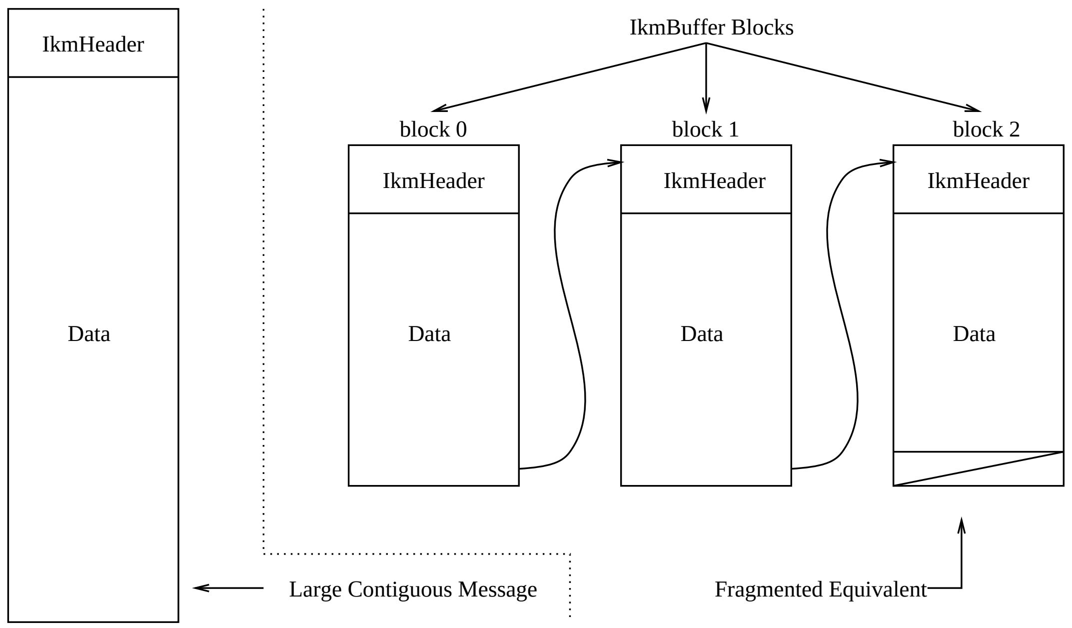

Figure 3.2: Contiguous & Fragmented Message Representation

`aon_ikm_exit()` Shuts down the IKM. Closes the network socket and cleans up in general.

`aon_ikm_send()` Sends an arbitrary sized contiguous message to another node. The message is delivered at the remote node in the same format as it was sent i.e. as a contiguous chunk. The send is synchronous, returning when a successful reply has been received or if the transaction fails. If the transaction was successful, the pointer to the request message passed in is changed to point to the reply.

`aon_ikm_send_frag()` Same as the above, but takes a pre-fragmented request message, and returns a fragmented reply i.e. IkmBuffer messages. The message is delivered at the remote node in the same format as it was sent i.e. as an IkmBuffer message.

`aon_ikm_asend()` The asynchronous version of the contiguous send routine above. No attempt is made to guarantee delivery of request message.

`aon_ikm_asend_frag()` Asynchronous version of the fragmented send routine above. No attempt is made to guarantee delivery of request message, again no MTT entry is made.

`aon_ikm_reply()` Sends a contiguous reply message to a request. The function returns as soon as the message has been transmitted, but the message is kept in memory for some time in case of transmission requests. The request message can be reused for the reply to avoid calling malloc(3).

`aon_ikm_reply_frag()` As above, but the reply message is an IkmBuffer list.

`aon_ikm_msend()` Multiple Send version of the above contiguous send routine. Pass in a list of hosts, and the message will be sent to each at the same time.

`aon_ikm_msend_frag()` Fragmented version of above.

`aon_ikm_amsend_frag()` Asynchronous version of above.

`aon_ikm_amsend()` Contiguous version of above.

`aon_ikm_forward()` Forwards a request message to a specified node. No attempt is made to guarantee delivery — this is the responsibility of the original sender.

3.4 IKM Message Types

There are four basic types of messages exchanged between remote kernels involved in IKM transactions. These were used by the original IKM to provide a reliable transport service. Their functions have been changed somewhat to cater for IKM extensions:

Request A request message consists of the original kernel request, fragmented into a number of smaller blocks. Each of these blocks are sent asynchronously. The calling kernel waits on a reply to the entire message, not to individual blocks.

Reply Like requests, these can be of arbitrary size, and are split into blocks. They are no longer restricted to the same size as that of the request message to which they are replying.

PING When the calling kernel has not received a reply to a reliably transferred request within a specified timeout, it begins to send PING messages to the remote kernel to ensure that it is still alive and processing the request.

ACK An ACK is sent in reply to a PING message, to tell the remote kernel that the complete request message has been received, and is being processed.

Previously, the IKM would timeout on a reply, and just retransmit the request message. However, since request messages are no longer fixed to one block, it would be inefficient to resend the whole message when only one or two blocks may be needed. An ENQuiry message and Negative ACKnowledge message have been added to the IKM to cater for this selective retransmission:

ENQ When the calling kernel has finished the unreliable transmission of a message's blocks, it waits for a reply from the remote kernel. If no reply is forthcoming, it sends an ENQuiry to ask what the status of the request is, and which, if any, of the blocks were not received. The ENQ message contains the first (and perhaps only) block of the message, and hence also serves as a retransmission.

NACK If a message has only partially been received and the kernel receives an ENQ, it counts which blocks have not been received, and sends a request for their retransmission in the form of a NACK message.

Finally, to support message forwarding:

FACK If a node receives an enquiry about a message which has been forwarded, it replies to the caller with a Forward ACKnowledge message. The FACK tells the caller that the message has been forwarded to node 'x' and that all future correspondence concerning this message should be conducted with node 'x'.

3.5 The IKM State Machine

Each IKM message transaction is a sequence of messages passed between the IKM at a source node and the IKM at a destination node. Because of the complexity involved in operating above an unreliable transport mechanism such as UDP, a state machine is needed to keep track of each transaction. An IKM transaction will always be in one of a number of well-defined states. These states are outlined below.

Figure 3.3 shows the state machine for message transaction states on the sending side. Each state is explained in Table 3.1.

Figure 3.4 shows the state machine for message transaction states on the receiving side. Each state is explained in Table 3.2.

3.6 Typical Transactions

Four typical transactions are shown in Figures 3.5 and 3.6. In each case, both request and reply messages are three blocks long.

Figure 3.5 shows the sequence of messages sent in an error free transaction, with normal and long processing times. A normal transaction consists of a request sent, serviced and reply received. A long service time on the remote node causes the sender to timeout and send an ENQ. The remote node will ACK this to tell the caller it is servicing the request. After a short delay, since the reply has not been received, the caller sends a PING to check the remote node is alive. The remote node responds with another ACK, and shortly afterwards, finishes processing the request, and sends the reply.

Figure 3.6 shows the sequence of messages sent when the third request and third reply blocks are lost. When the request block is lost, the caller times out and sends an ENQ. The remote node responds with a NACK detailing which blocks were not received. The caller now knows to resend block three. The remote node, on receipt, sends the caller an ACK to tell it that the complete request has finally been delivered and is being serviced. When a reply block is lost, the caller timeouts on a complete reply, and sends a NACK saying which blocks were not received. The remote node receives the NACK, and retransmits the correct reply block.

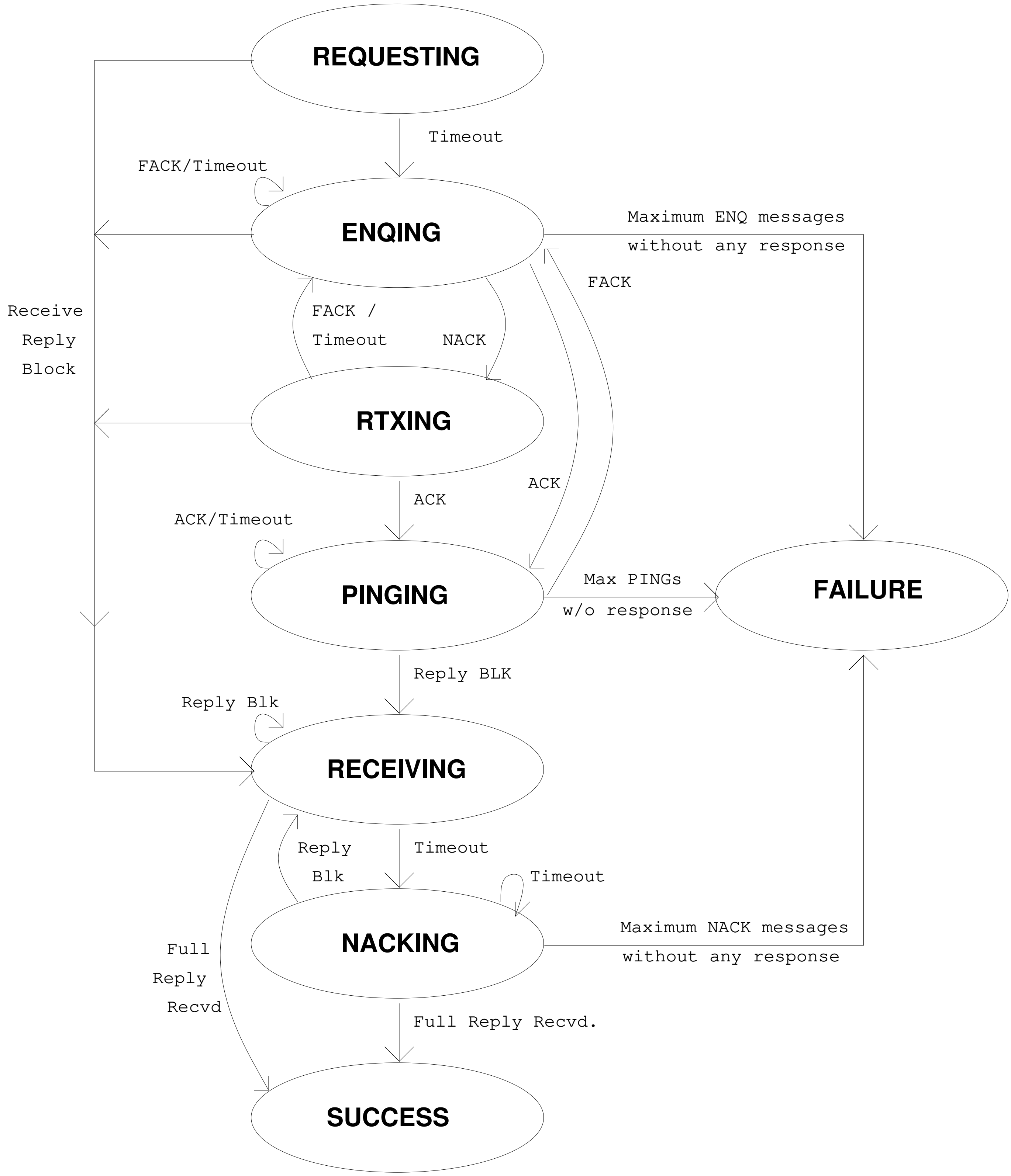

Figure 3.3: Message Send States

State	Description
REQUESTING	The fragmented message blocks are transmitted by lower level protocols. The kernel awaits a reply.
ENQING	An ENQuiry message is transmitted periodically until either an ACK, NACK or full reply is received. A NACK tells the IKM which blocks were not received.
RTXING	A NACK message was received by the IKM. The remote node did not receive the blocks specified in the NACK message. The missing blocks are retransmitted.
PINGING	An ACK has been received, so the IKM periodically sends a PING message to the remote node to check that it is still alive. If an ACK is not received after several PINGS, the remote node is assumed to have died.
RECEIVING	A reply block has been received. The IKM stays in this state until all the reply blocks have arrived. If not all arrive within a given time, the message moves into the NACKING state.
NACKING	Not all the reply blocks were received, the IKM continually transmits NACK messages demanding the missing blocks. If a reply block arrived, the message goes back to the RECEIVING state.
SUCCESS	When the full reply message is received, the message goes into the SUCCESS state. It is passed up to the client above the IKM.
FAILURE	If, after a fixed number of ENQs, PINGs or NACKs, there is no response from the remote kernel, it is assumed to have died. The message moves into the FAILURE state, and the IKM client is notified of the failed request.

Table 3.1: Message Send States

State	Description
REQRECEIVE	Fragmented blocks of the message are assembled on arrival.
REQSERVICE	The message has now fully arrived, and the request is being processed by the local kernel.
REQREPLY	The request result is sent back to the calling kernel.
REQFORWARD	The request has been forwarded to another node.
CLEANUP	The calling kernel has not sent any further blocks concerning this transaction for a fixed period of time. The message is discarded by the local kernel.

Table 3.2: Message Receive States

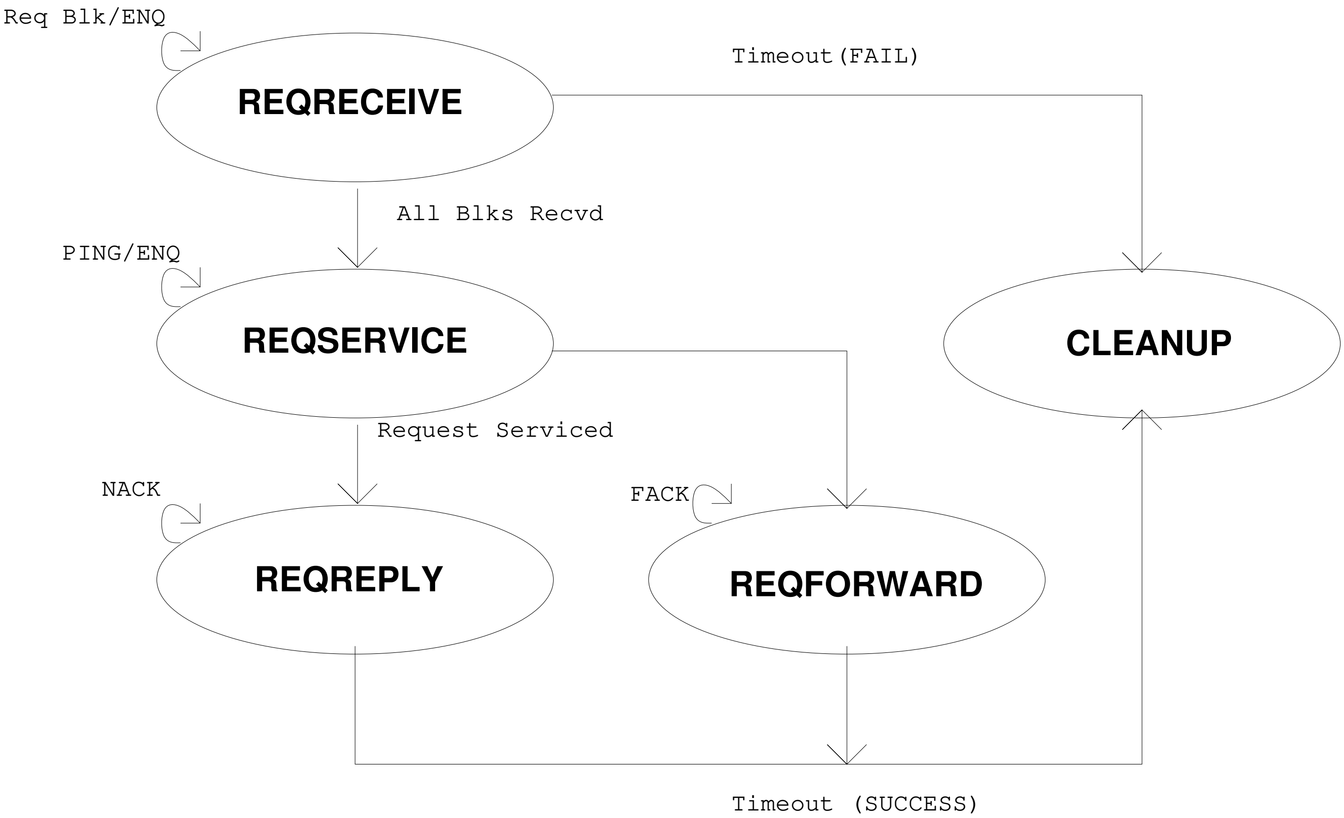

Figure 3.4: Message Receive States

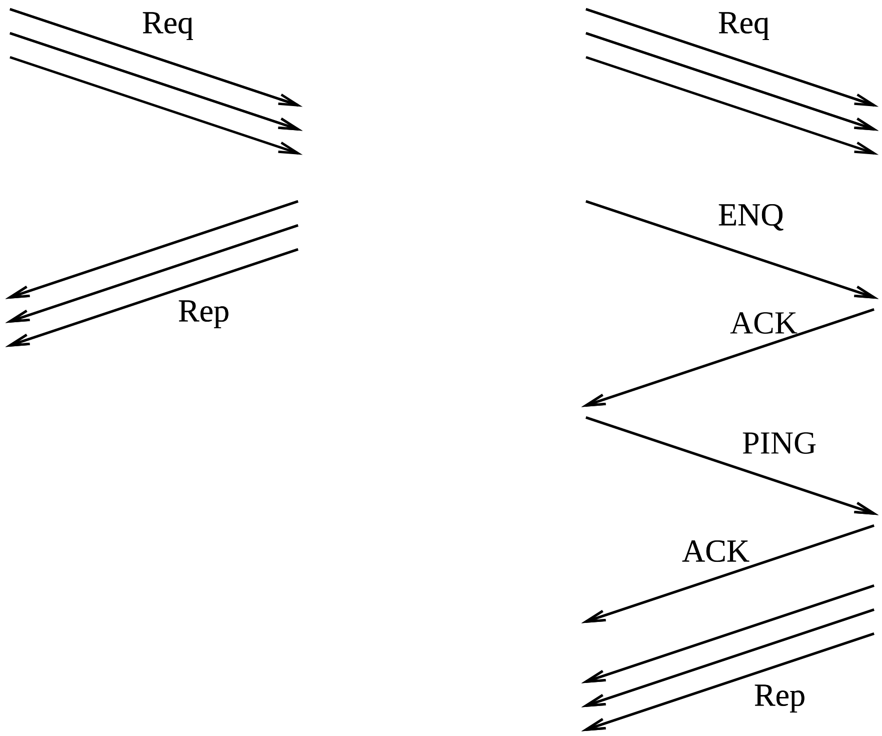

Figure 3.5: Normal & Long Processing Delay

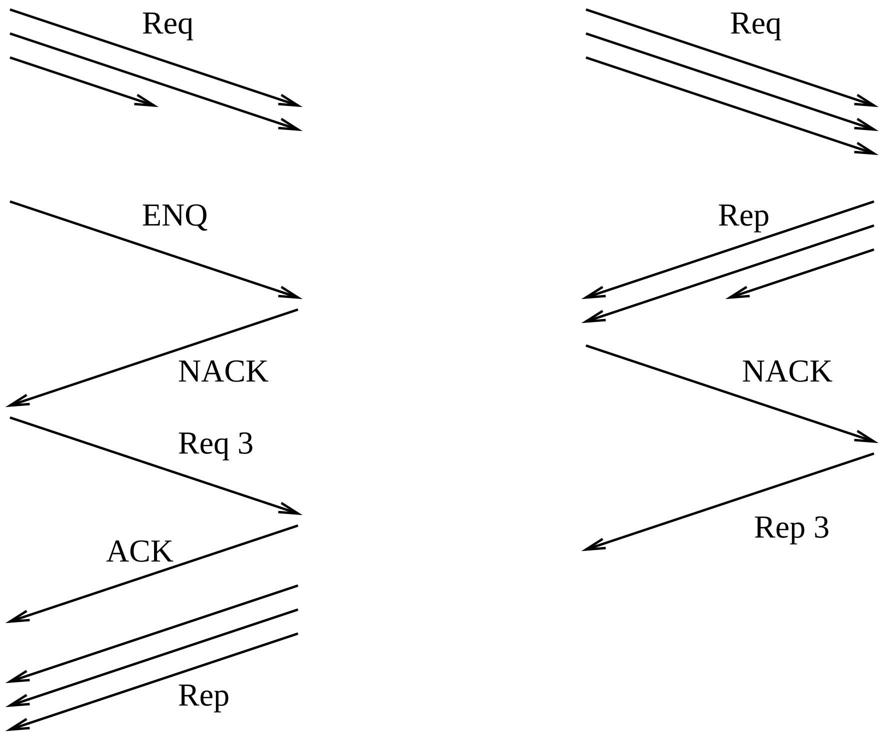

Figure 3.6: Lost Request & Lost Reply

3.7 State/Transaction Correspondence

A typical sequence of states on both send and receive sides, along with messages sent (Figure 3.7) might be as follows:

Sending Initially, the message goes into the REQUESTING state. It is fragmented into the appropriate number of blocks, and transmitted unreliably using UDP sockets. In this case, the third request block is lost.

When the all the fragments have been transmitted, the IKM will wait until a reply is received from the remote kernel. Since a number of blocks may not have arrived successfully at the destination node, the remote kernel will not be able to reply. The IKM times out, moves into the ENQING state and transmits an ENQ message to see what has happened.

The remote side replies with a NACK as it missed the third block. The NACK message tells the local IKM that the last block was lost. The caller now moves into the RTXING state and retransmits block three.

When an ACK is received, the sending side now knows that the request has been successfully delivered and moves into the PINGING state waiting for the reply message.

A short time later, and no reply block has arrived, the sender transmits a PING message to make sure the remote node is still alive. An ACK is received confirming that the node is alive and still processing.

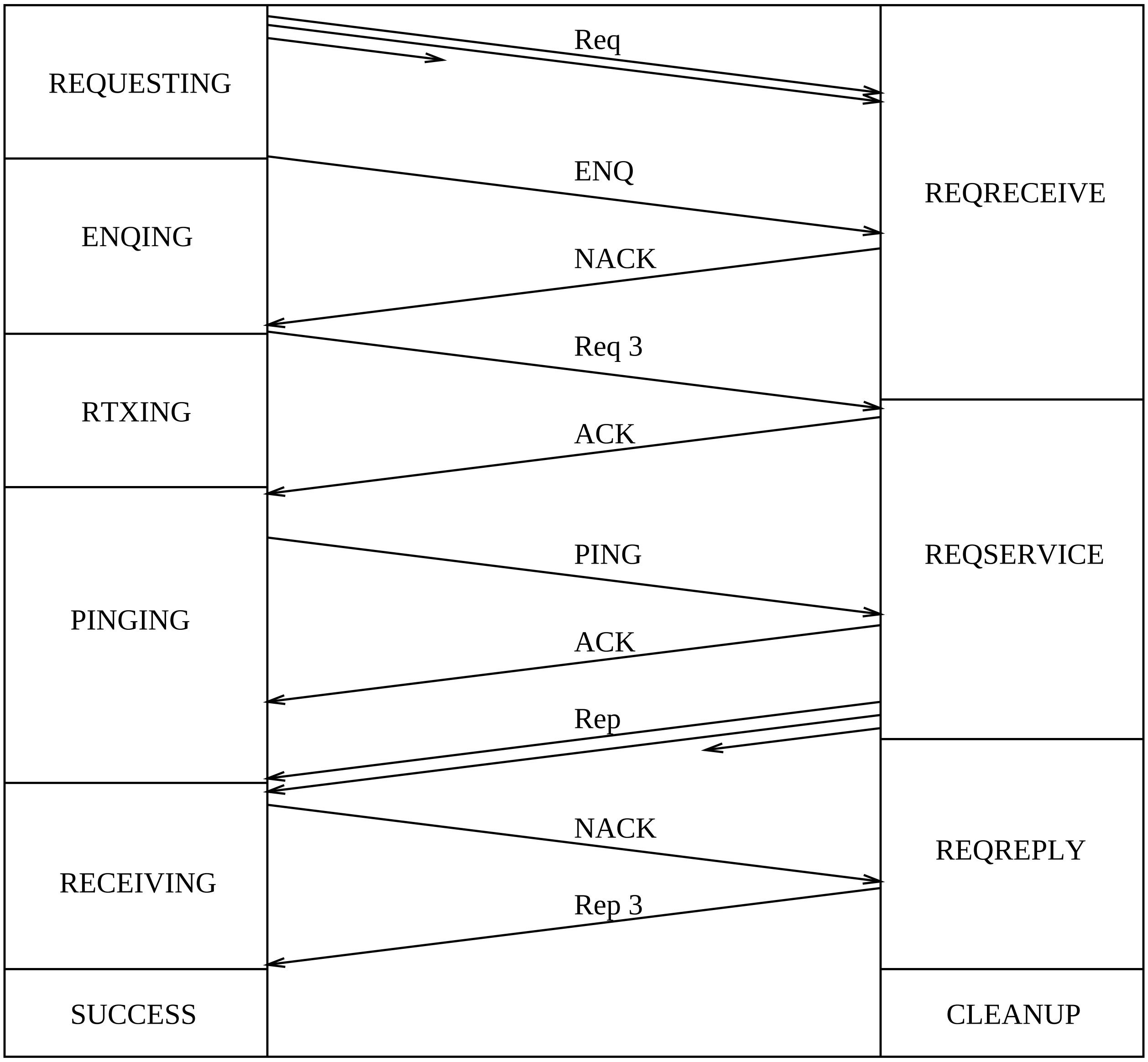

Figure 3.7: State/Message Correspondence

Soon after, the first two reply blocks are received. The message is moved into the RECEIVING state. Since the third reply block does not arrive, the local IKM transmits a NACK message to the remote side demanding a retransmission.

The remote side resends the missing block. It is received, and the message moves into the SUCCESS state — the full request/reply transaction has completed successfully. The IKM client is notified of the reply.

Receiving As fragmented message blocks arrive into the kernel, each message is sorted into the correct order using their block ids. The message is in the REQRECEIVE state.

The message will stay in the REQRECEIVE state until all request blocks have arrived. Since some blocks were unsuccessful in transmission, the calling kernel sends an ENQ message to ask which blocks did not arrive. The local kernel replies with a NACK message, demanding retransmissions. The message stays in REQRECEIVE.

The third block arrives and the request message is complete. The message moves into the REQSERVICE state, as the local kernel processes the request. The request takes a long time to process, and a PING message arrives; the local kernel replies with an ACK to assure caller that all is well.

When the request has been serviced, the complete reply is transmitted to the caller. The message moves into the REQREPLY state. Here, the third reply block gets lost. A NACK is received from the calling kernel to inform the local kernel of this, and the lost block is resent.

The message stays in the REQREPLY state for a sufficient period of time to allow the calling kernel to send NACKs and request these retransmissions.

3.8 Message Forwarding

When a complete request message is received, the receiving kernel component may elect not to service the request, but forward it to another node. This process is transparent to the IKM client, who submits a request and receives a reply.

Figure 3.8 shows the normal sequence of events when a message is forwarded. Node A initially sends a request to node B. Node B elects to forward the request to Node C. It moves the message into the REQFORWARD state, forges some fields in the message header and then transmits it to Node C. Node C processes the request and sends the reply to Node A.

Figure 3.9 shows what happens when a forwarded request block is lost. Node A times out on a reply, and sends an ENQ to Node B. Node B is in the REQFORWARD state when this ENQ arrives. It replies with a FACK telling node A to resend the ENQ to node C. Node A receives the FACK, changes all message references of Node B to Node C, reissues the ENQ, and continues the transaction as normal.

Figure 3.10 shows the case where a reply block of a forwarded request is lost. Node A sends

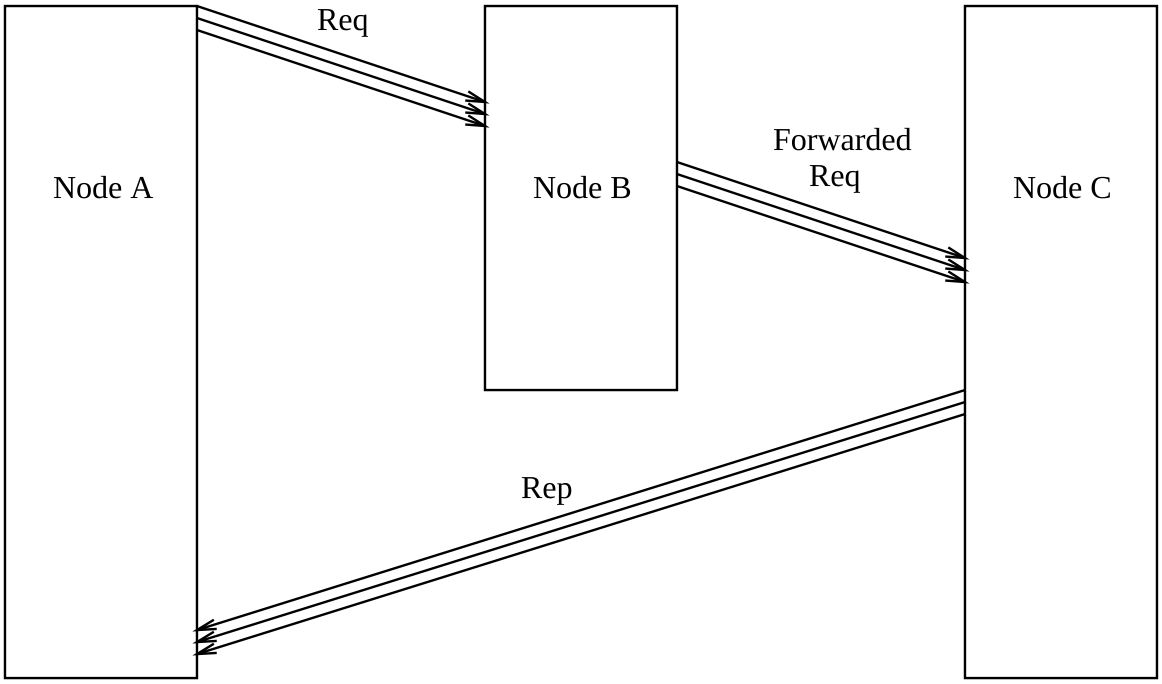

Figure 3.8: Normal Message Forward

the NACK for this reply block to Node B, as this was the original request destination. Node B replies with a FACK, as in the lost request case, and Node A then reissues the NACK to node C.

There were several other options available for the forwarding protocol but the above was chosen as the chief design aim was to minimize Node B's involvement, and to exclude it from the transaction as quickly as possible.

3.9 Multiple Send

First, some definitions:

A *unicast* message is sent by a host to one specific host on the network. A *broadcast* message is one that is transmitted by a host to every host on the network. A *multicast* message is one that is transmitted by a host to a specified group of hosts on the network; a multicast message implies that some subset of host systems belong to a multicast group.

Figure 3.11 shows the advantage in using a Multicast. If each individual message has to be sent synchronously, the time taken for a request/reply to, say, three nodes, will be $t_1 + t_2 + t_3$. Using multicast however, the IKM will transmit all three message requests one after the other without delay, and simultaneously wait for each reply to arrive. The time taken for the transaction will be $max(t_1, t_2, t_3)$ where t_n is the time taken for the n^{th} request/reply transaction to complete.

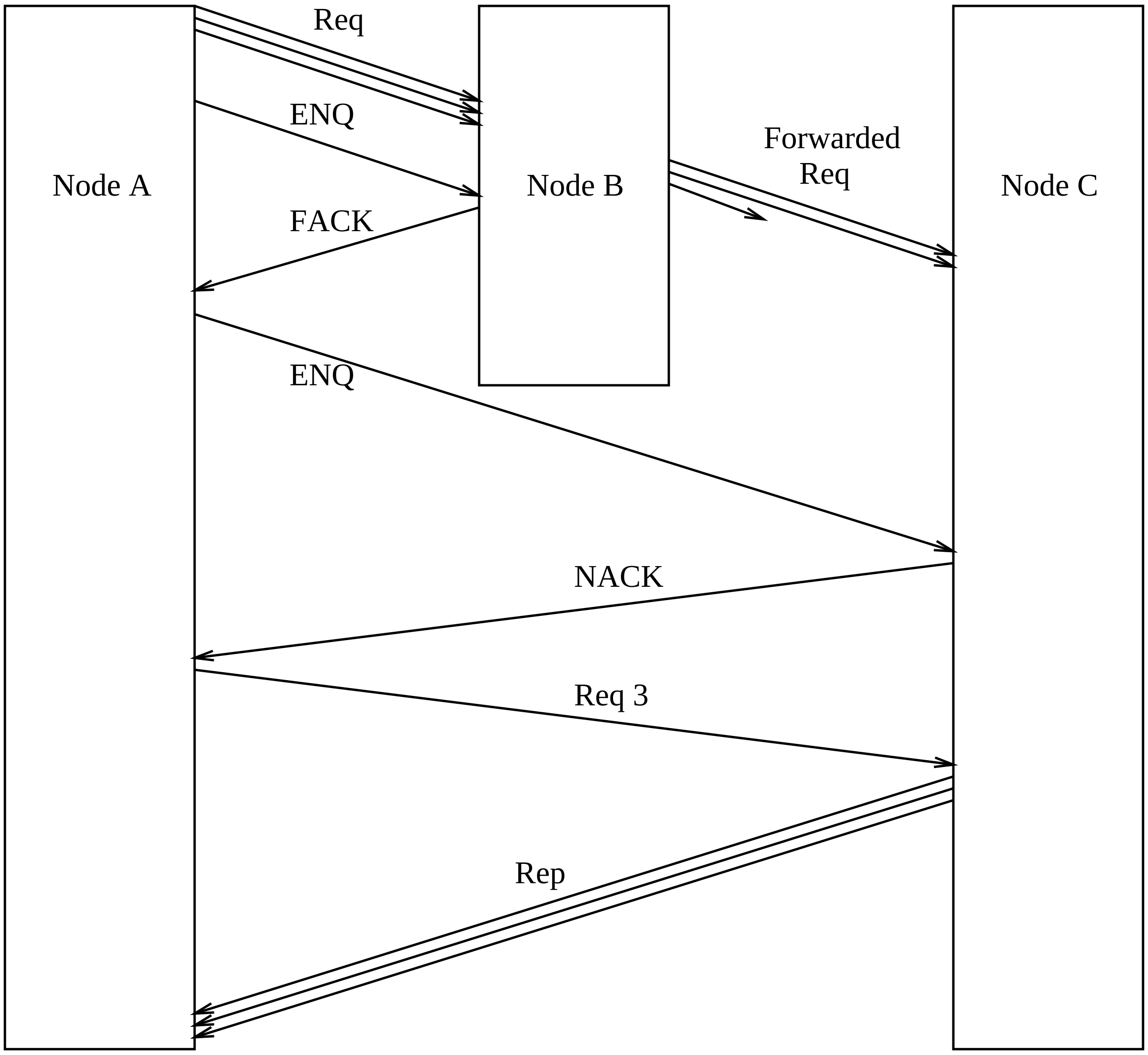

Figure 3.9: Lost Request Block in Forward

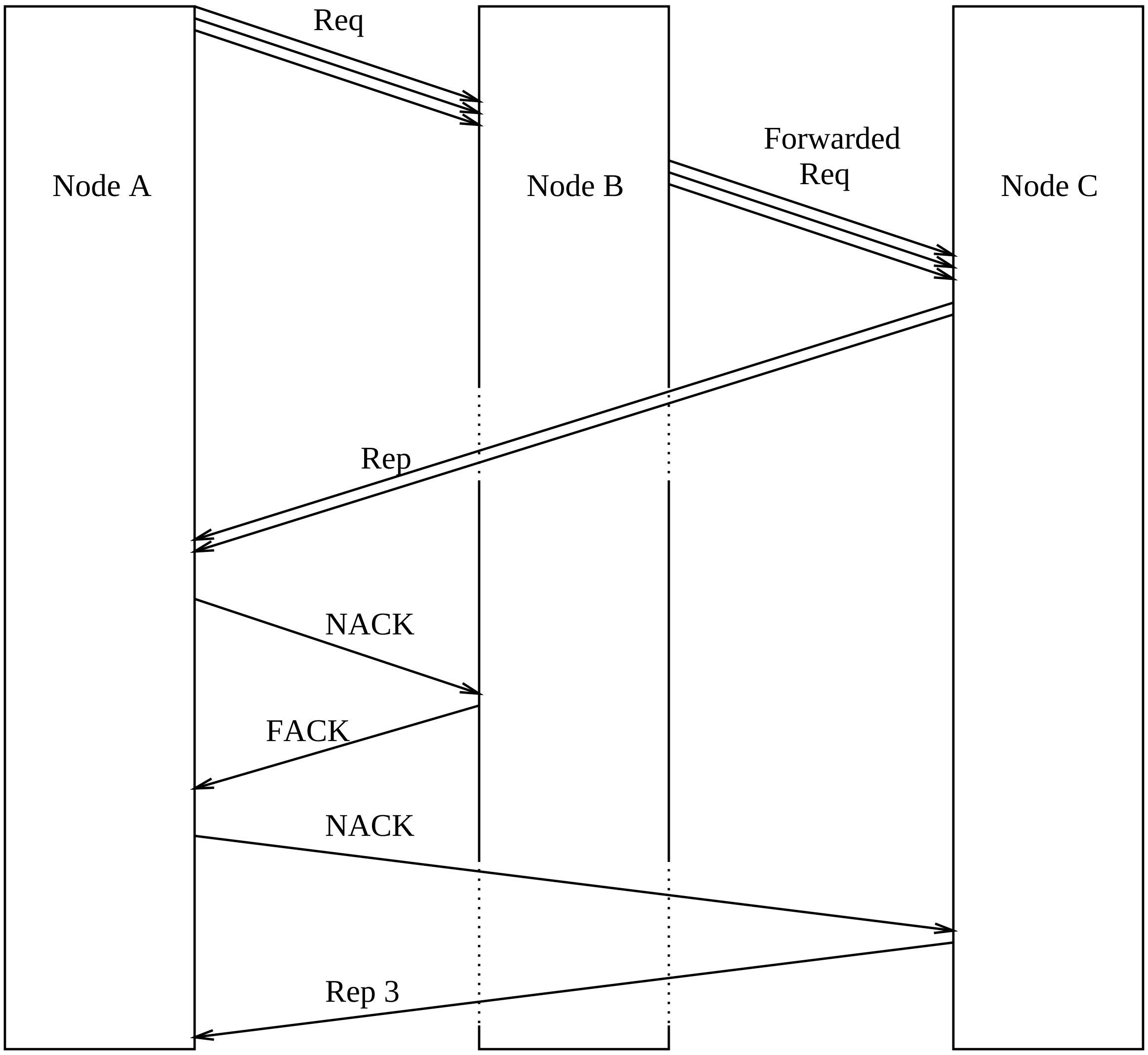

Figure 3.10: Lost Reply Block in Forward

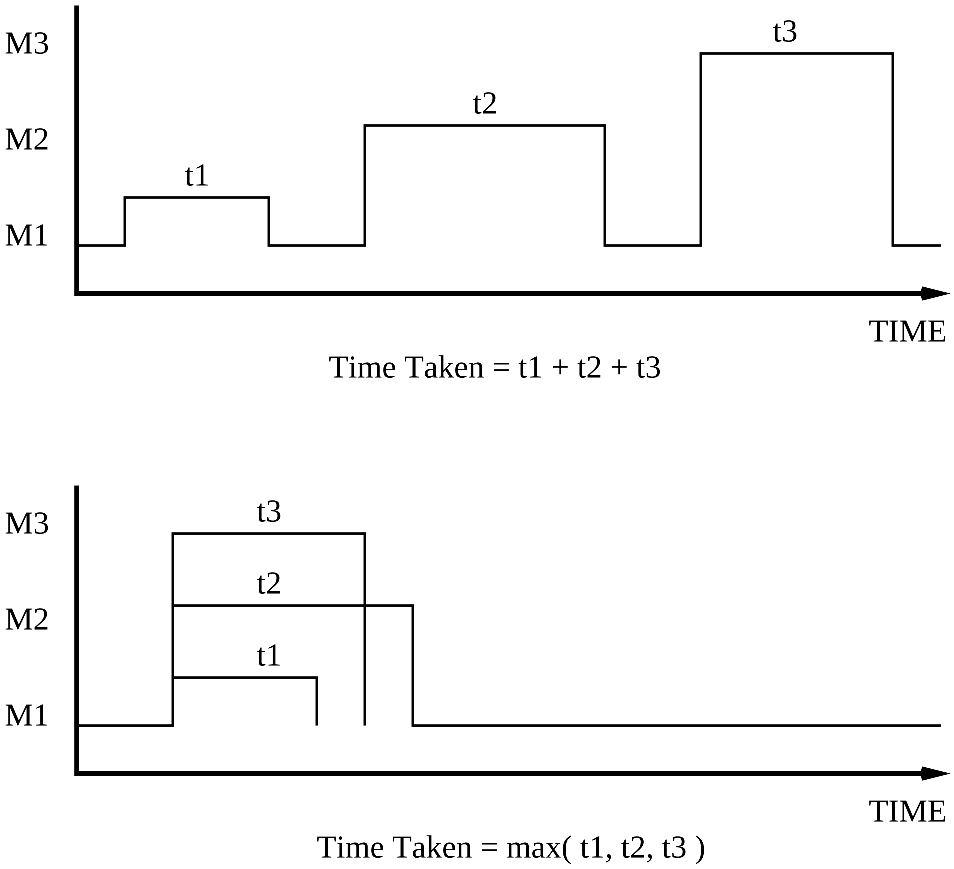

Figure 3.11: Multiple Send Timings

Design aims for IKM MSend included:

- To keep the amount of changes to the IKM code at a minimum. It was important that the inclusion of this new feature would not incur any performance loss on the rest of the IKM. For example, creating a thread to deal with each destination node reply to MSends would have led to huge IKM overheads in context switching when sending to many nodes.
- Existing clients should not have to be recompiled for use with the new system. Following on from above, MSend use should almost be on a 'need to know' basis. For example, when a node MSends a request to several destinations, each of these receivers treat this message as a normal request, they are unaware that this is part of an MSend. They do not need to know this information. They process the requests and reply as normal.
- To make the MSend as simple to use as possible. The interface routine should be similar to a single send, allowing the programmer a painless transition.
- Support for flexibility in what governs when a client call should return. To provide a mechanism whereby clients could have a call terminate when, for example, half of the replies to a request had been received.

See [8] for discussion on CMU's MultiRPC.

3.10 Summary

This chapter has described the design of the IKM and the functionality that it provides. The IKM state machine underwent a complete redesign while support for message forwarding and multiple send was introduced for the first time.

Chapter 4

Implementation

4.1 Introduction

This chapter describes the implementation of the 'new-look' IKM, and serves as a guide to the source code listing in Appendix A.

4.2 Internal Representation

There are three main data structures used within the IKM — the Message Transaction Table (MTT), the IKM Header, and the IKM Buffer.

Figure 4.1 shows the relationship between each of these structures, and their role within the IKM.

4.2.1 The IKM Header

The following fields are contained within the IKM header:

len Length of the appended data block

src Logical Node Address of Source

dest Logical Node Address of Destination

msgid Unique Mesage ID

curblk Current Block Number (numbered 0...(n-**1**))

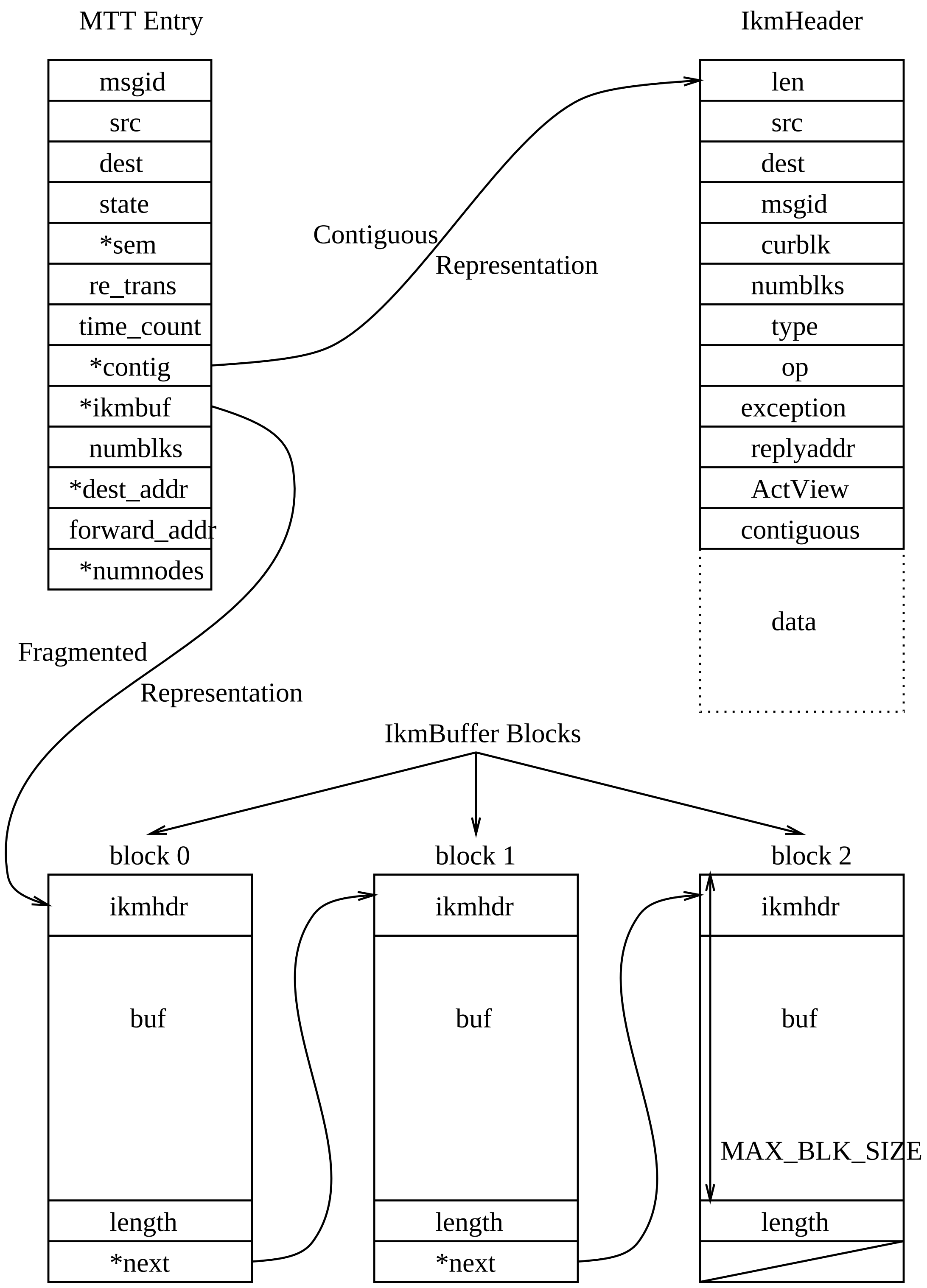

Figure 4.1: The main IKM data structures

numblks Number of blocks in message

type Type of Message (request/reply/etc)

op Operation Required

exception Informs caller of remote failure on return

replyaddr Address of node to reply to

ActView Identify Amadeus Activity

contiguous True if Message Contiguous, False if Fragmented

4.2.2 The IkmBuffer Message

The aim of the IkmBuffer message and its routines is to provide the programmer with a simple interface with which s/he can manipulate arbitrary sized messages suitable for transmission using the IKM.

Originally the malloc(3) and free(3) system calls were used to build messages. However this was inconvenient as the programmer does not always know in advance how large a message will be.

The IkmBuffer routines are built around a linked list which can dynamically grow in size — each element in the list is optimally sized for transmission. To the programmer, an IkmBuffer looks like one open-ended contiguous chunk of memory.

Normal usage would consist of opening an IkmBuffer, writing data to it, and then presenting it to the IKM for transmission.

The routines for all of this are as follows:

`aon_ikm_openbuffer()` Creates and returns a pointer to an IkmBuffer message. The first block in the linked list is created. Returns NULL on failure.

`aon_ikm_closebuffer()` Each element in the IkmBuffer message list is freed from memory. Void function.

`aon_ikm_appendbuffer()` This routine writes an arbitrary number of bytes from a specified address into a specified IkmBuffer list. New blocks are created at the end of the linked list as more space is needed. Returns True if successful.

`aon_ikm_readbuffer()` Reads a specified number of bytes from a specified offset from an IkmBuffer message into a contiguous block of memory. Returns the number of bytes read.

aon_ikm_lengthbuffer() Simply returns the number of bytes of data contained within an IkmBuffer message. Is it important to remember that the value returned excludes IKM header sizes.

The following buffer routines are used internally by the IKM:

aon_ikm_insertbuffer() This inserts a single IkmBuffer block into an IkmBuffer list. It is used internally on receipt of individual message request/reply blocks. The `curblock` field in the IKM Header indicates where the block should be inserted. Returns True if a new block was inserted, False if the block was already in the list.

aon_ikm_resetbuffer() Resets the `length` fields of each block in the list. Allows a buffer to be reused — avoiding the expense of a malloc(3).

aon_ikm_cleanbuffer() Usually called after a buffer has been reset with the above routine. Frees memory associated with each unused element in the IkmBuffer list.

4.2.3 The MTT and Message IDs

The Message Transaction Table (MTT) is used to keep track of all transactions currently being processed by the IKM. Each transaction has an MTT entry; the information held is as follows:

state The current state of the message in the IKM state machine. Each state is described in tables 3.1 and 3.2.

sem The semaphore on which the IKM interface send routine is sleeping. Needed to wake up sender when the reply arrives.

src Message Source Node

dest Message Destination Node

time_count The number of IKM timer units which have passed since the last (re)transmission. Every 'x' units, another transmission is invoked.

re_trans Number of retransmissions so far without response. This can be in relation to RTXs, ENQs, PINGs or NACKs. After a specified number of retransmissions without response, the transaction will fail.

numblks Number of blocks received so far for this message.

ikmbuf A pointer to the IkmBuffer request/reply which is being transmitted/received over UDP. New blocks are inserted here as they come in.

`contig` A pointer to a contiguous version of the above. Required if the message is to be delivered in a contiguous format.

`dest_addr` A pointer to the destination address of the message.

`forward_addr` The address of the node to which this message has been forwarded.

The Message Transaction Table is currently implemented as an array of MTT elements. Message transactions expire fairly quickly, so searching is not too expensive. Nevertheless, this should be recoded as a hash table in the next version.

Four routines are used to access and maintain the MTT. They are:

`aon_ikm_mtt_init()` Simply initializes the MTT for use.

`aon_ikm_mtt_get()` Allocates an entry for a message. The memory address of the entry is returned and the calling routine can then fill out the appropriate fields.

`aon_ikm_mtt_free()` Frees a message's MTT entry for use by another transaction.

`aon_ikm_mtt_search()` Searchs for an entry given source node and message ID.

A single routine is used to generate a unique message ID for each IKM transaction:

`aon_ikm_get_mid()` This works by getting the number of milliseconds passed so far in the current day. The threads package ensures that no two routines can call this function in the same millisecond. Multisend will, however, call this routine repeatedly, but provision is made for this.

4.3 Message Fragmentation

4.3.1 Sending side

A new function called `aon_ikm_send_msg()` has been introduced — its purpose being to send 'messages' as opposed to 'blocks'. Previously, messages and blocks were the same thing, and were handled by `aon_send_it()`. The latter function now deals exclusively with blocks.

The contiguous IKM interface Send and Reply functions use the IkmBuffer routines to fragment the message into blocks, before passing to `aon_send_msg()` which copies in the header and fills out two new fragmentation-related header fields. Each block is sent using `aon_send_it()`. The two new fields are `curblk` and `numblks`, which are the current block number and the total number of blocks in the message.

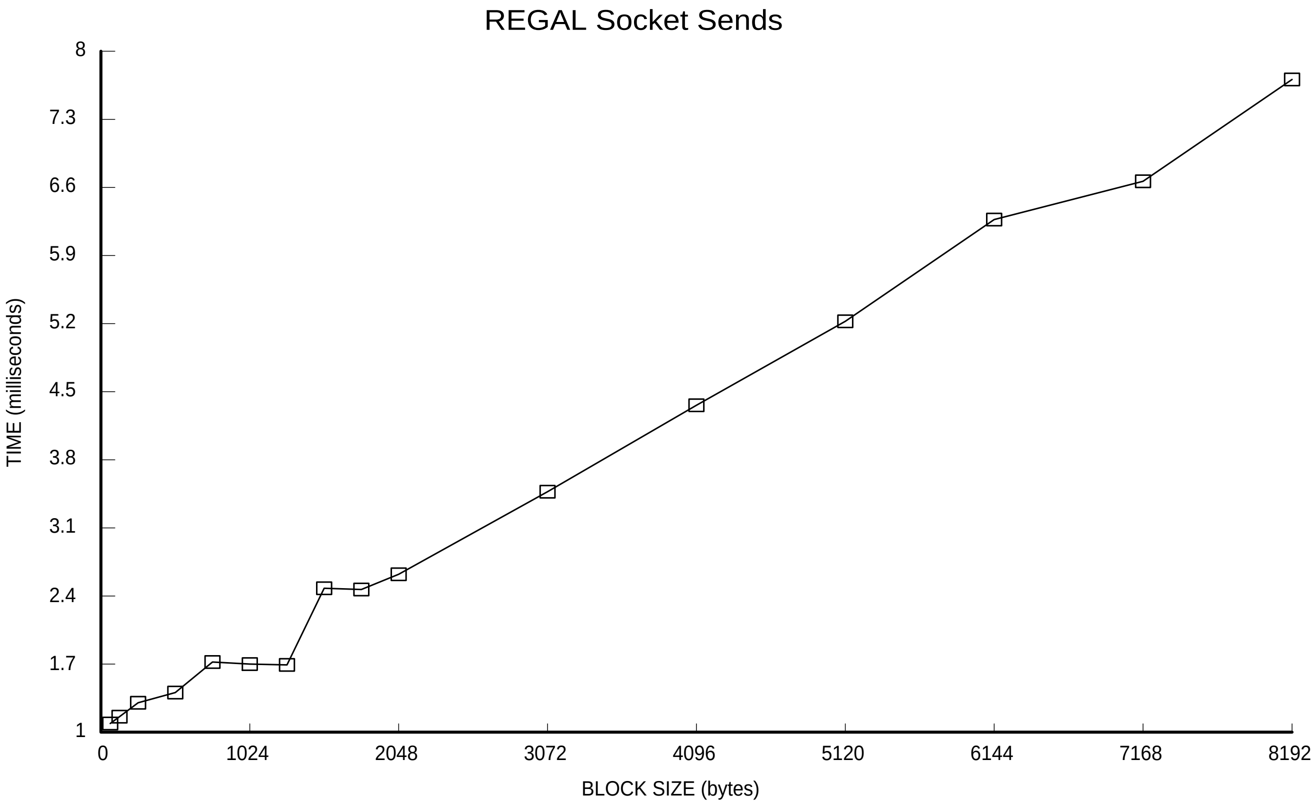

Figure 4.2: Internet Domain Datagram Socket Timings

When the fragmented IKM interface routines are used, `aon_send_msg()` can be called directly.

The write(2) system call is used to write a contigous block of memory to a socket. The writev(2) system call can be used to write a linked list to a socket. The writev(2) call is not used here as the maximum length of data which can be sent with a single writev(2) call is equivalent to that of a single write(2) call (8k in the case of an Internet Domain UDP socket). The only difference between the two calls is in how the data is presented.

As can be seen from figure 4.2, there is no optimum packet size for internet domain socket sends. Therefore, if a 10 Kb message is to be sent, we simply split it into 8 Kb and 2 Kb blocks. An alternative (and more complex) method might have been to split the message into several *optimally* sized blocks, and then transmit. However, it is worth noting that the increase in performance gained through this latter approach could potentially be lost due to the increased overhead in fragmentation/assembly of the smaller blocks.

4.3.2 Receiving Side

When message blocks arrive in, they will normally be dealt with by either `aon_ikm_process_req()` or `aon_ikm_process_reply()`.

The first of a message's request blocks to arrive will cause an MTT entry for that message to be made. Three new fields have been added to the MTT structure to allow each of the arriving blocks to be stored in a linked list hanging off the message's MTT entry. These are: `numblks` for the total number of received so far, `ikmbuf`, a pointer to the linked list of incoming message blocks, and `contig`, a pointer to an assembled version of the message. See figure 4.1.

When the final request block is received, the message moves into the REQSERVICE state, and is then delivered to the layer above the IKM by the function `aon_ikm_deliver()`. The message is assembled into a contiguous chunk if necessary before delivery. This involves copying the message fragments from the IkmBuffer message into a contiguous block of memory.

In the case of incoming reply blocks, an MTT entry already exists from the message request being sent. The pointer to the request message is discarded, and incoming blocks are assembled there instead.

When the final reply block arrives, the original request is discarded. The reply is made contiguous (if the original request was contiguous). The IKM client's request message pointer is pointed to the reply message, and the client is awakened.

4.4 Input-handler & Timer-handler

When the IKM is initialized with `aon_ikm_init()`, two threads are created to handle incoming messages and timeouts. These two routines are fundamental to the operation of the IKM.

4.4.1 The IKM Input-handler

The input-handler's general algorithm for operation is as follows:

```
while (True) {

        sleep until data arrives from network
        check message data
        call appropriate handling routine

}
```

The thread code which calls the above is contained in `aon_ikm_inputloop()`. This function puts the thread to sleep awaiting network activity. When a message arrives, `aon_ikm_input_handler()` is called. This routine reads the network data into an IkmBuffer block, validates it, and dispatches the appropriate handler function depending on the message type. These handlers are dealt with in the next section.

4.4.2 The IKM Timer-handler

The timer-handler's general algorithm for operation is as follows:

```
while (True) {

        sleep for 'x' IKM time units (fixed period)
        check status of each MTT entry,
        if necessary, send ENQ/PING or discard message

}
```

The function `aon_ikm_timerloop()` contains the thread code for the above. The function sleeps for a fixed period of time — the exact duration being derived from the Amadeus `aon_mspertick` configuration variable.

Each time it awakens, `aon_ikm_timer_handler()` is called. The aim of this function is to handle the timeouts associated with the IKM state machine (see figures 3.3 and 3.4).

When the function is called, each MTT entry's `time_count` variable is incremented, depending on the current state of the transaction. When this value reachs a certain threshold, a retransmission is invoked and the `re_trans` variable is incremented. If `re_trans` reaches another set threshold, the transaction is deemed to have failed. The IKM cleans up, and the client is informed.

This process is often referred to as a 'sweep-timer'.

4.5 Other Message Types

Not all blocks are data blocks, ENQs, PINGs, ACKs, and NACKs must also be handled. We'll first look at each of the routines which generate these messages, and then at each of the handler routines which process them.

`aon_ikm_ack_ping()` Sends an ACK or a PING message depending on the parameters passed.

`aon_ikm_send_enq()` Retransmits the block 1 of request message, with the type set to IKM_ENQ and not IKM_REQUEST.

`aon_ikm_send_nack()` Opens up an IkmBuffer list to build the NACK message. By using the `numblks` field in the request/reply block headers, and by scanning the list of blocks already received, the sequence numbers of the missing blocks can be written into the NACK message with `aon_ikm_appendbuffer()`.

And the handlers:

`aon_ikm_enq_handler()` Since an ENQ message serves as a retransmission, and contains piggy-backed data [1], `aon_ikm_process_req()` is called to insert a new request block which may not have originally been received. If the ENQ block did not provide the final message block, the MTT timeout on the message is reset, and a NACK is generated though `aon_ikm_send_nack()`. See below.

`aon_ikm_ping_handler()` Generally when a PING message arrives, a request message is being processed, and an ACK is sent back in reply through `aon_ikm_ack_ping()`. However, if a PING is received when the reply has been sent, then the first (and perhaps only) reply block of the message is retransmitted.

`aon_ikm_ack_handler()` When an ACK is received by a requesting node, it knows that the complete message request has been delivered successfully. It moves into the PINGING state, and awaits the reply.

`aon_ikm_nack_handler()` A NACK message's data consists of a sequence of block numbers. These block numbers refer to the message blocks which were not received by the NACK sender. To remedy this, the NACK handler runs through the request IkmBuffer list, and retransmits each block which the NACK claimed went missing.

4.6 Message Forwarding

Three new routines and a new message type were needed to implement message forwarding. They are as follows:

`aon_ikm_forward()` When a complete message request is delivered, the upper layers may decide to forward the message to another node for processing. This routine will then be called in place of `aon_ikm_reply()` or `aon_ikm_reply_frag()`. Since a fragmented copy of the request message is kept by the message's MTT entry, this can be used for retransmission to the new node. Before being forwarded, the message is put into the REQFORWARD state and the `dest` field of the request is forged to fool the new node

[1] In fact, the ENQ message may be the only data block in the request message.

into thinking the request came from the original sender and was always intended for this destination. We store the network address of the new node in the MTT's `forward_addr` field. The first block of the request is kept in case of further correspondence from the original sender. All other blocks are discarded.

`aon_ikm_send_fack()` If some of the blocks fail to reach the new destination during forwarding or some of its reply blocks fail, the original sender will timeout, and send an ENQ/PING. In this case, this routine will send an IKM_FACK (Forward ACKnowledge) message back to the original sender. The message consists of an Ikm Header, *with the reply address field set to be the address of new destination node.*

`aon_ikm_fack_handler()` When the original sender receives a FACK, it resets the message's timeout, puts the message back into either the ENQING or NACKING state (depending on whether request or reply blocks were lost), and changes all references to the destination node to the address in the FACK's `reply_addr` field.

4.7 Multiple Send

Some LANs (such as Ethernet or Token Ring) provide support for broadcast messages at the data-link layer. Network support for multicast, however, is less common, and even when it em is available, it can only be used at the hardware level (and not at a UNIX level).

Figure 4.3 illustrates how support for multiple send was added to the IKM. The interface routines create separate MTT entries for each destination node. The request is transmitted to each node sequentially, each request having a unique message ID.

Each MTT entry used to contain a semaphore upon which it would await a reply. Now, each entry contains a pointer to a semaphore. This allows each MTT request in a multisend message to share a semaphore upon which the client can be woken.

A new field `numnodes` is introduced to the MTT entry. This is used as a count variable which is decremented each time a complete reply is received from a destination node (or a reply is timed out). The IKM routine `aon_ikm_end_transaction()` performs this decrement and at present awakes the client if *all* replies have been received. This routines can be modified in future to change the constraints on replies from multiple sends. For example, a 'best effort' system similar to VMTP could be introduced where the client awakes when at least one reply is received.

When the client interface routine awakes, the messages are reassembled, if necessary, and the pointers to each request message are changed to point to each of the new reply messages.

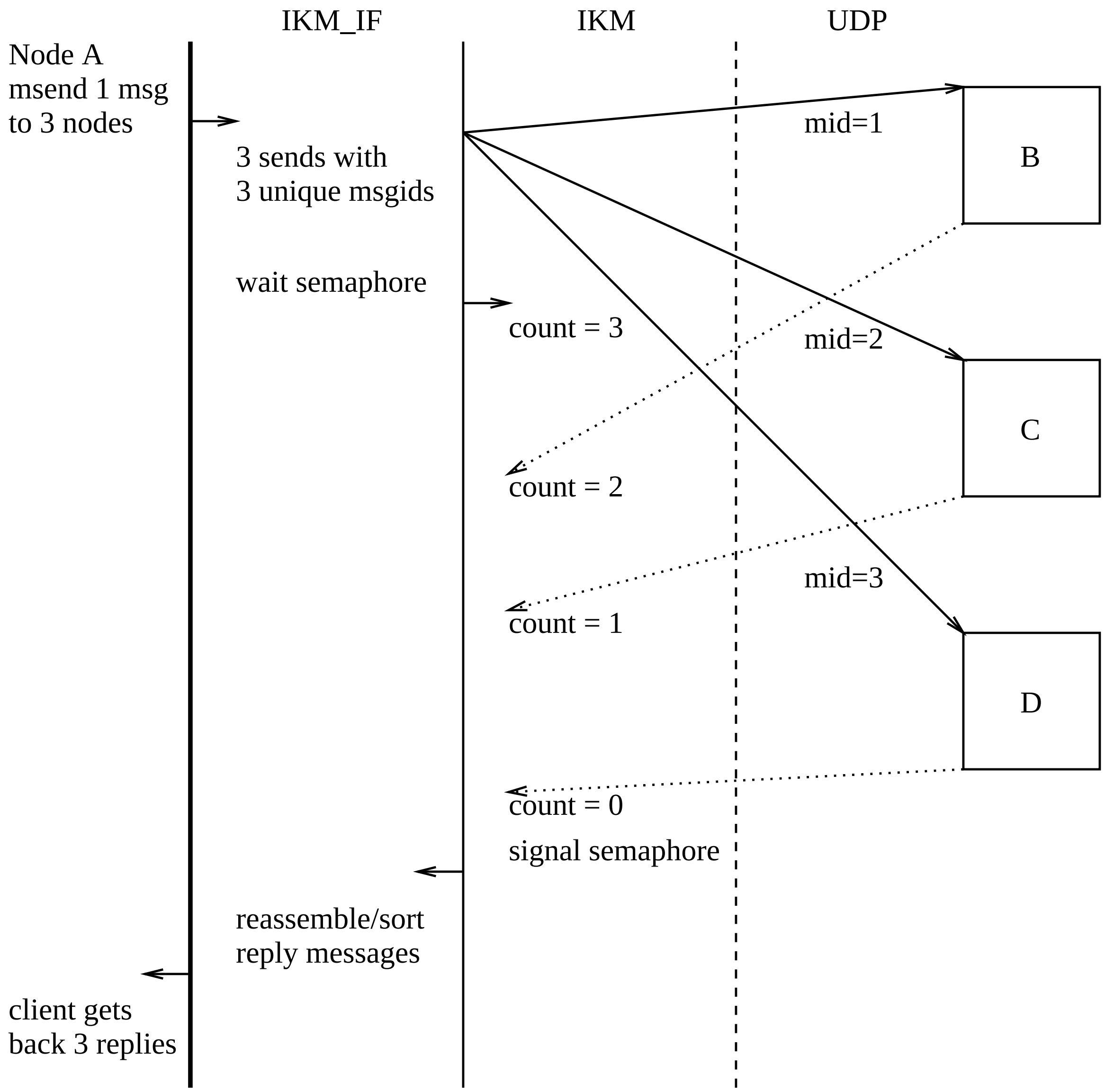

Figure 4.3: Multiple Send Implementation

4.8 Summary

This chapter has described how message fragmentation, forwarding and multiple send were added to the IKM.

Chapter 5

Performance

5.1 Introduction

This chapter explores various performance issues which are relevant to the IKM implementation. These include the choice of a suitable underlying transport mechanism, packet encapsulation, memory copying, and performance figures for the IKM itself.

5.2 Unix Domain vs Internet

Traditionally, both unix and internet domain sockets were supported by the IKM. When a message was being sent locally, unix domain sockets would be used. The benefits of this approach included higher transmission speeds and higher numbers of successfully transmitted messages.

Figure 5.1 shows timings for block sends using both unix and internet domain sockets. As can be seen, unix domain sockets are consistently faster than their internet domain counterparts, justifying the method used.

There were a number of drawbacks, however. Supporting two types of socket communication in the IKM code resulted in constant checks on which type of socket was being dealt with, leading to code which was often clumsy and difficult to follow.

In addition, unix domain sockets are limited to a maximum message size of 2 Kb, whereas internet sockets support sizes of up to 8 Kb. In order to provide a scheme whereby large messages could be fragmented into a series of small blocks suitable for transmission, a 2 Kb limit on block size was considered too restrictive. Using 2 Kb blocks in place of 8 Kb could increase the fragmentation/reassembly overhead by up to a factor of four.

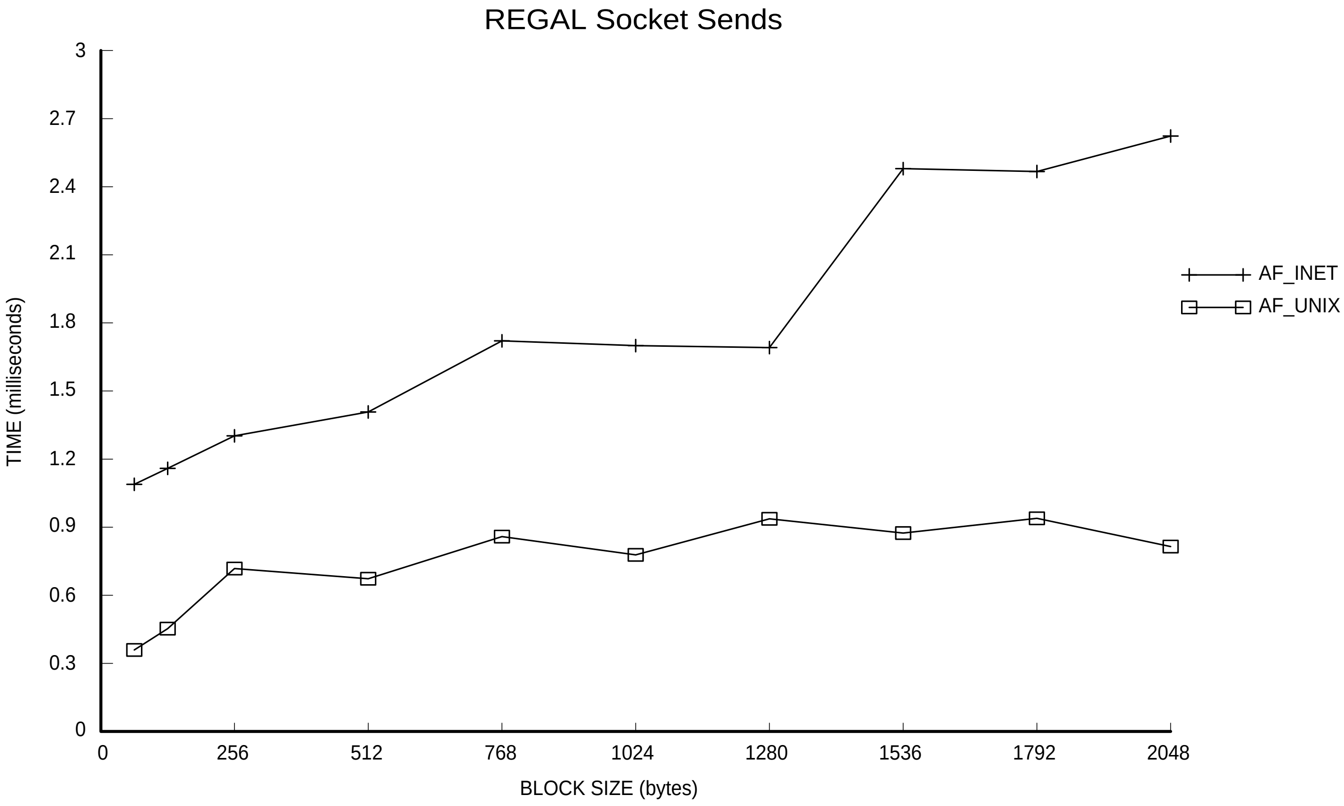

Figure 5.1: Unix/Internet Domain Datagram Socket Timings (Send Only)

The timings in figure 5.1 were taken on a machine with a relatively light load, and are therefore the ideal case. A more loaded machine will not show local sockets to have such a clear performance gain over internet domain. (See [9] for detailed discussion and further references to the effect of load on UDP/IP performance.)

Taking all of the above into consideration, it was decided that all IKM socket communications should be carried out using internet domain sockets.

5.3 Memory Copy & Scatter/Gather Write

5.3.1 Contiguous Messages

In the case of IKM Send and Reply routines, when contiguous messages are presented, the caller is forced to allocate space for the IKM header at the beginning of the message data. A pointer to this buffer is passed, along with the length of the message *including* the header.

If the message was not presented in this fashion, the header pointer and data pointer would have to be passed in separately, and a malloc(3) and two bcopy(3)s would be necessary to build the block into a format suitable for transmission.

As can be seen from figures 5.1 and 5.2, the time taken by a bcopy(3) can be a considerable percentage of the time taken for a datagram block transmission — the higher the rate of memory copies done during a request/reply transaction, the lower the performance.

The only alternative would be to pass pointers to the header and data to the writev(2) function. This call is sometimes referred to as 'Scatter/Gather Write' [10], as it takes pointers to data residing in different parts of memory and gathers them together to write as a single contiguous block.

It is possible, however, that in providing this extra functionality, the writev(2) call is less efficient than simply preallocating memory and executing the two bcopy(3)s.

5.3.2 Fragmented Messages

The alternative to all of the above for the IKM user is to simply use IkmBuffer messages when presenting data to be transmitted. The buffer routines allow messages to be built, sent and received with the minimum amount of copying.

The user does not have to worry about IKM headers and where they will fit in the message. He/She just has to write data into the IkmBuffer.

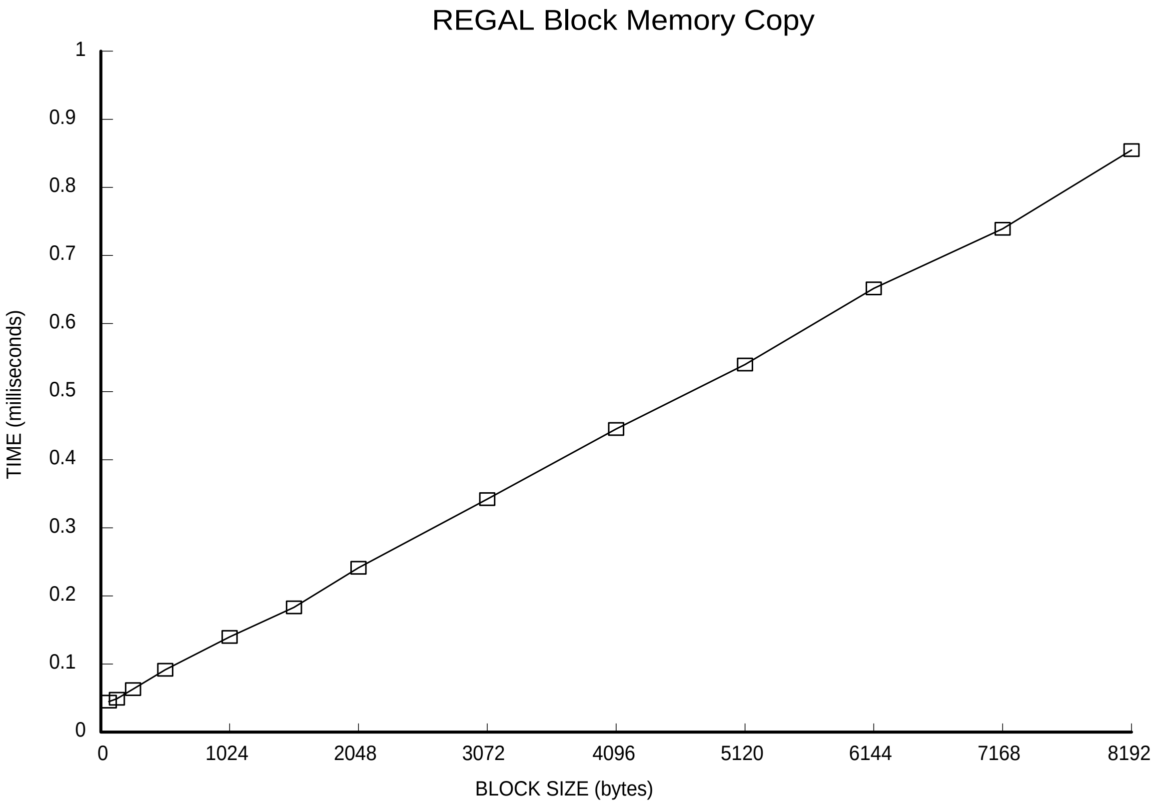

Figure 5.2: Block Memory Copy Timings

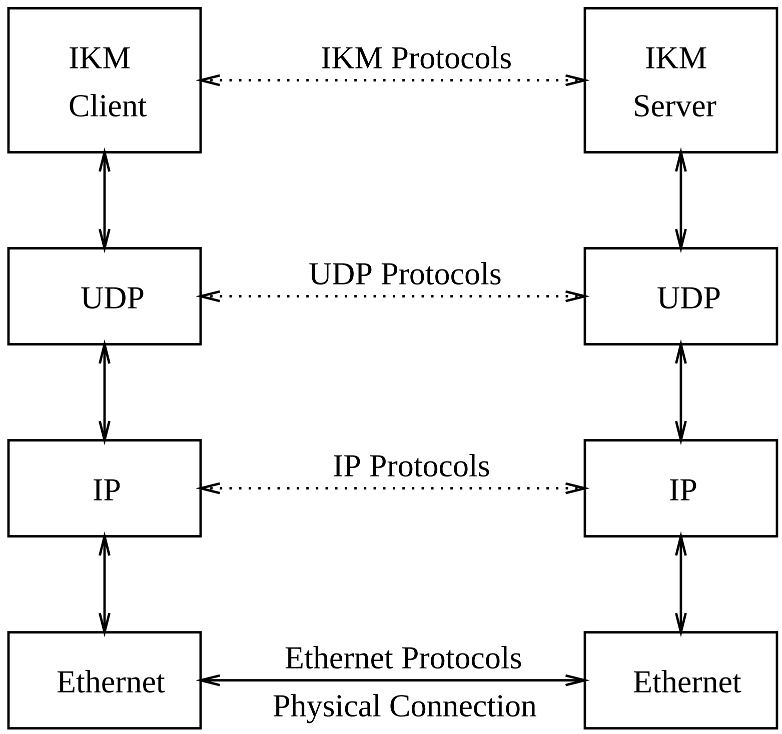

Figure 5.3: 4-layer Model for IKM over UDP/IP on an Ethernet

5.4 Encapsulation

Figure 5.3 shows the various layers between the user IKM process and the network hardware. Figure 5.4 shows how network data is 'encapsulated' at each of these layers.

While we say that all data has a 64-byte IKM header prepended to it before transmission, we must remember that when using UDP/IP as a transport mechanism, addition of new control information is added at each of the layers shown. What was initially a 64-byte header, 64-byte data IKM message, can turn into a considerably larger message by the time it hits the ethernet.

Much of the IKM header information is duplicated at the lower layers, but since each layer ignores data added by other layers, this cannot be avoided. The only solution would be to throw aside portability and implement the IKM directly above the network hardware.

5.5 IKM/UDP Features

Instead of being built on directly on top of hardware like FLIP and VMTP, the IKM sacrifices speed for portability. As can be seen from figure 5.3, the IKM is built above UDP above IP above hardware.

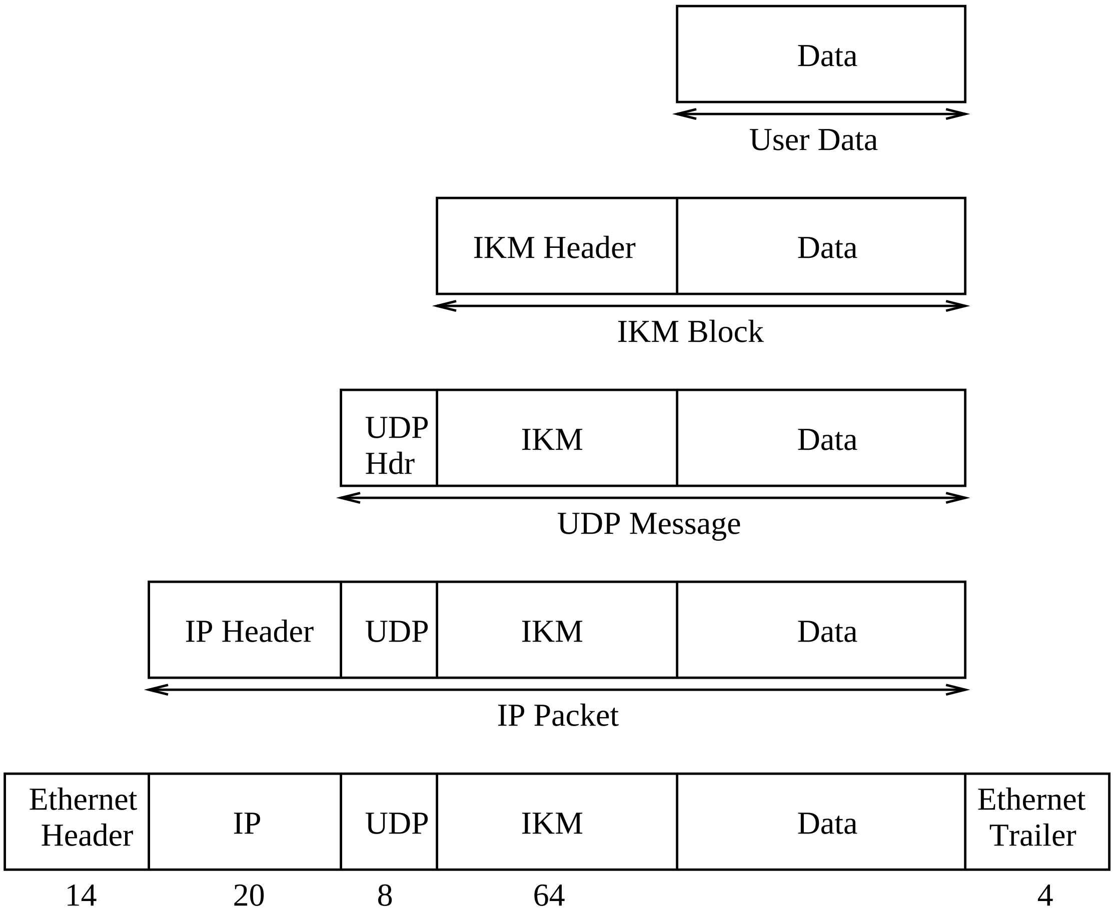

Figure 5.4: Data Encapsulation

IP stands for 'Internet Protocol' and offers two main services: 'Transmission Control Protocol' (TCP) and 'User Datagram Protocol' (UDP). TCP is a connection oriented service while UDP is a connectionless service.

Table 5.1 shows the different features both UDP and TCP provide over IP. It also shows how the IKM improves on bare UDP to provide a fast, reliable protocol (like TCP) but *without* the overhead.[1]

Feature	IP	UDP	TCP	IKM
Connection Oriented	no	no	yes	no
Message Boundaries	yes	yes	no	yes
Data Integrity	no	yes	yes	yes
Positive Acknowledge	no	no	yes	yes
Timeout & Retransmit	no	no	yes	yes
Sequencing	no	no	yes	yes
Flow Control	no	no	yes	no

Table 5.1: IP/UDP/TCP/IKM Feature Comparison

The IKM does not provide any flow control. All request data packets are sent on the network at the same time. On a typical 10 Mbps LAN, with current hardware, this is not a problem, however, if networks were to increase in speed, is is possible that blocks may be sent too fast for the receiver to receive. At present, if the IKM loses blocks in this manner, a NACK for the missing blocks will be sent, and they are retransmitted.

VMTP solves this problem by using an 'Interpacket Gap' between block sends. This gap is long enough for the receiver to be ready for the new block's arrival. See [6].

5.6 IKM Performance Figures

This section presents some performance figures for the IKM. All timings were taken on 'Regal', a DECstation 3100 running Ultrix 4.2.

Throughout the timings, asynchronous message sends are faster than synchronous message sends. This is simply because, asynchronous sends return to the caller immediately without waiting for remote servicing or replies.

Fragmented messages are, in general, faster than contiguous messages, since, internally, the IKM likes to deal with optimally sized network packets i.e. fragmented IkmBuffer messages.

[1]The overhead in setting up a reliable connection between two machines using a virtual circuit protocol such as TCP is a contributing factor in why connectionless protocols are often used to implement RPC.

Memory copying is kept to a minimum, where possible, within the IKM when contiguous messages are handled.

In all cases below, the remote service time is nil.

Table 5.2 shows observed best timings for a request/reply transaction consisting of 64 byte header and 64 byte data message being transmitted in both directions.

Request	Contiguous	Fragmented
Synchronous	5	4
Asynchronous	3	3

Table 5.2: Observed Best IKM Timings (millisecs)

Table 5.3 shows average timings for a request/reply transaction consisting of 64 byte header and data messages ranging from 64 bytes to almost 8 Kilobytes, moving in both directions.

Data Size	Synchronous		Asynchronous	
	Contiguous	Fragmented	Contiguous	Fragmented
64	10	7	4	4
128	10	7	4	4
256	10	7	4	4
512	10	7	4	4
1024	11	8	5	5
2048	13	11	8	6
3072	14	13	9	7
4096	16	15	10	9
5120	17	16	11	10
7168	20	20	14	11
8128	21	2l	15	13

Table 5.3: Average IKM Timings (bytes,millisecs)

Finally, figures 5.5 and 5.6 show table 5.3 graphically. Also included in the figures are timings for sending the equivalent amount of data over UDP sockets without any overhead protocol like the IKM.

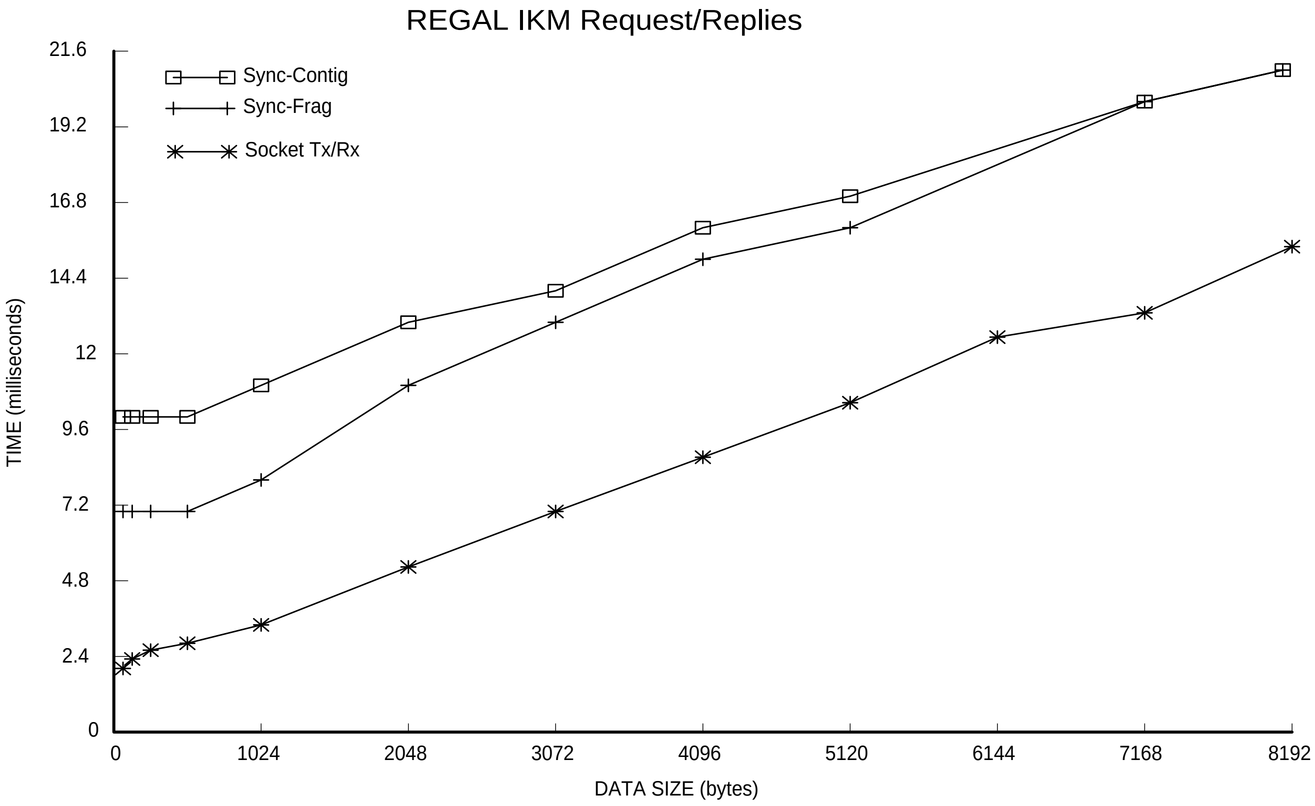

Figure 5.5: Average Synchronous IKM Timings

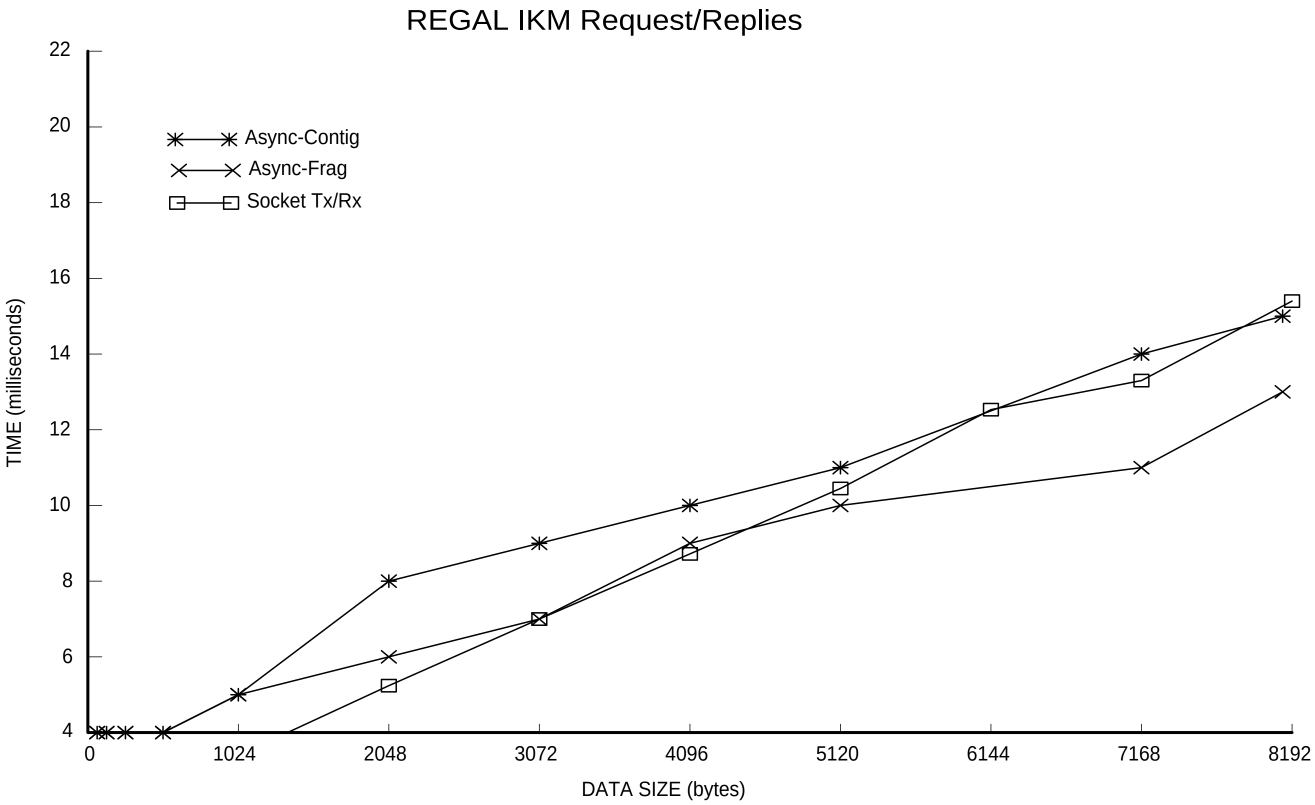

Figure 5.6: Average Asynchronous IKM Timings

Chapter 6

Summary

This final chapter concludes the report with a short critique of the current IKM, and provides the author with a soapbox...

6.1 Critique

No real optimization was done on the IKM. At present, it performs reasonably well, but there is always room for improvement. Any future work should consider the following points:

- At present, using the asynchronous IKM routines means potentially unreliable message transfers to the destination. This is acceptable in many cases, but it would be possible to create a thread to deal with asynchronous transfers, allowing the client to continue executing immediately but assured that best attempt is being made in completing the transaction.
- There should be more state on the receiving side. All request blocks are sent at the same time and so should arrive reasonably close together. At present, the sender times out on a complete request/reply; instead, the receiver should send back a NACK by itself for missing request blocks.
- ENQ messages currently always contains block 0 (to serve as a retransmission). This could make it smarter by sending the last block NACK'd (if appropriate) instead of block 0 all the time, which could be transmitted many times. Alternatively, we could just send the ENQ without the data after, say, the second retransmission.
- Perhaps writev(2) could be used to transmit the header and data message from the caller in a contiguous block. Currently, the caller has to preallocate space for the header in their data message. While the current method *might* be marginally faster, using writev(2) would provide the user with a cleaner and more natural interface. writev(2) performance needs to be studied.

- It is possible that using a readv(2) call on the receive side, may avoid a bcopy(3) if the message is to be delivered in contiguous form. Avoiding bcopy(3)'s on the receive side is always more difficult than on the send side since we don't know in advance what type of message is arriving from the network and hence do not know how to deal with it correctly until we have read it.

6.2 Conclusion & Soapbox

On the whole, the project went quite well. It was slow getting started, as I found the learning curve reasonably steep. Finding myself with little documentation and 2000 lines of code (which had been hacked on by a number of people over the years) was not a pleasant experience.

It took quite some time to get the IKM code running by itself outside of Amadeus — even though this work had supposedly been done already. Getting to grips with the threads package caused the most trouble — `aon_thread_exit()` did *not* seem to be the most obvious way of starting the threads running!

I liked the fact that both design *and* implementation were involved. Design without implementation can often be futile. If we lose sight of a final implementation, much of what is designed may be impractical to implement.

I have learnt various things from the project, some good and some bad. My knowledge of communications protocols and their implementations has obviously risen dramatically. I feel I know the C programming language inside out at this stage, but I'm not too sure whether that's good or bad.

I think the most important thing I learnt was how to tackle a large project — no matter what the subject material. This has certainly been the largest computer related work I have undertaken so far, both inside and out of college.

At the end of the day, the aims originally laid out were achieved. The IKM has undergone a major overhaul, and hopefully the DSG will find my work useful.

Bibliography

[1] Distributed Systems Group, Trinity College (1991). Overview of the Amadeus Project.

[2] HORN, C. and CAHILL, V. (1991). Supporting distributed applications in the Amadeus environment. *Computer Communications, Vol. 14, 6, 358.*

[3] SLATTERY, J. (1990). The Design and Implementation of a Communication Subsystem for a Distributed Operating System. *M.Sc. Thesis, Trinity College, Dublin.*

[4] HALLSALL, F. (1988). Data Communications, Computer Networks and OSI. *Addison-Wesley*

[5] MARTIN, J. and LEBEN, J. (1988). Data Communication Technology. *Prentice-Hall*

[6] MULLENDER, S. (1989) Distributed Systems. *ACM-Press, Addison-Wesley*

[7] CHERITON, D. R. (1984). The V Kernel: A Software Base for Distributed Systems. *IEEE Software, April 1984, 19.*

[8] SATYANARAYANAN, M. and SIEGEL, E.H. (1990). Parallel Communication in a Large Distributed Environment. *IEEE Transactions on Computers, Vol. 39, 3, 328.*

[9] CABRERA, L.F., HUNTER, E, KARELS, M.J. and MOSHER, D.A. (1988). User-Process Communication Performance in Networks of Computers. *IEEE Transactions on Software Engineering, Vol. 14, 1, 38.*

[10] STEVENS, W.R. (1990). UNIX Network Programming. *Prentice-Hall*

[11] LEFFER, S.J., FABRY, R.S., JOY, W.N., LAPSLEY, P., MILLER, S. and TOREK, C. (1986). An Advanced 4.3BSD Interprocess Communications Tutorial. UNIX Programming Supplementary Documents, Vol. 1 (PS1), 4.3BSD, CSRG, Berkeley, California.

[12] COULOURIS, G. F. and DOLLIMORE, J. (1988). Distributed Systems. *Addison-Wesley*

[13] ROCHKIND, M.J. (1985). Advanced UNIX Programming. *Prentice-Hall*

Appendix A

Programmer's Guide to the IKM

A.1 Introduction

This Appendix describes how to write programs using the IKM. Source code to a sample IKM client and sample IKM server is given. This shows how to use nearly all the IKM interface functions. A GNU Makefile is also included.

A.2 General Guidelines

The file `ikm_if.h` lists each of the IKM interface routines. Routines are provided for basic send, reply and forward. For some of these routines there are several versions: contiguous, fragmented, and asynchronous. The default is synchronous and contiguous. Fragmented versions have a `_frag` suffix. Asynchronous versions have an embedded `a` in their names.

Fragmented messages are created by using calls to the IkmBuffer routines (described in `ikm_buf.h`, see also the sample programs below). The advantage in using fragmented IKM routines over normal contiguous blocks of memory is both for convenience and efficiency.

IkmBuffer routines are convenient because you don't have to know in advance how large your message is going to be. You just open a buffer and append data to it. You then just present this to the IKM. They are efficient since the IkmBuffer structures come complete with preallocated space for IkmHeader's so the user doesn't have to leave space at the beginning of a message, avoiding malloc(3)'s and bcopy(3)'s on both send and receive sides. IkmBuffer structures are of optimum size for network transmission.

It is recommended that you use fragmented IKM interface routines wherever possible.[1]

[1] When you're using the IKM that is; you don't have to use them in the rest of your code:-)

For all synchronous routines, the address of a pointer to the request should be passed in. On successful transactions, this allows the interface routine to discard the request message, and change the pointer to point to the new reply message.

For asynchronous routines, just a pointer to the request is passed, since it is the responsibility of the caller to free the request after asynchronous transmission.

For multisend messages, arrays of pointers to destination node id's, addresses, and request messages need to be passed in. The arrays should be created by the caller and in each case, the address of the base element should be passed in.

Forward and reply routines are called on the receive side by layers above the IKM. The interface to upper layers on the receive side is through the routine `aon_ikm_deliver()` in `ikm.c`. This function gets called when a message has been successfully been received. You are free to change this function to suit your application.

A.3 IKM directory structure

The directory tree for these files should look something like:

```
regal> ls -R
example/     ikm/           support/

example:
Makefile*    demo_client*  demo_client.o demo_server.c
demo.h       demo_client.c demo_server*  demo_server.o

ikm:
ikm.c        ikm_buf.c     ikm_if.c      ikm_subr.c  ikm_types.h
ikm.h        ikm_buf.h     ikm_if.h      ikm_subr.h
ikm.o        ikm_buf.o     ikm_if.o      ikm_subr.o

support:
am.h         am_threads.h  diag.h        locore.o    unix_lib.c
am_sem.c     am_threads.o  eltypes.h     locore.s    unix_lib.h
am_sem.h     am_types.h    invoc_types.h pm_types.h  unix_lib.o
am_sem.o     config.h      locore.all    rt_obj.h
am_threads.c consts.h      locore.h      sets.h
```

Sample IKM Server

```
#define FileName        "demo_server"
#include "demo.h"

PortType        aon_serverport = IKM_DEMO_SERVER_PORT;
ProcDescType    aon_kernel_proc;
int             aon_currentnode;

/*
 * demo server mainline
 *
 * ask user for configuration variables, and then just
 * start ikm, launching threads which just wait to
 * handle incoming messages.
*/
main(argc,argv)
int argc;
char *argv[];
{

        /* Ask user for config/demo variables */
        aon_currentnode = askfor("Node ID?");
        FORWARD_NODE    = askfor("Forward?");
        JUNKING         = askfor("Drop Blks?");
        aon_csdebuglevel= askfor("Debug?");
        REPLY_SIZE      = askfor("Reply Size?");
        aon_serverport += aon_currentnode;

        /* initialise the threads package */
        aon_thread_init();

        /* initialise the IKM */
        if(! aon_ikm_init(aon_serverport) ) {
                Diag(("Can't init IKM"),1);
                exit(1);
        }

        /* start the threads running */
        aon_thread_exit();
}


/*
```

```
 * just convert yes/no answer from user to 1/0
*/
int askfor(str)
char *str;
{
        char ch[10];
        int ret;

        printf("%s ",str);
        scanf("%s",ch);

        switch(ch[0]) {
                case 'y':
                        ret=1;
                        break;
                case 'n':
                        ret=0;
                        break;
                default:
                        ret=atoi(ch);
                        break;
        }

        return(ret);
}
```

Sample IKM Client

```
#define FileName        "demo_client"
#include "demo.h"

ProcDescType aon_kernel_proc;
PortType aon_serverport =  IKM_DEMO_CLIENT_PORT;
int aon_currentnode;

Boolean         demo_send();
char            *filename;
NodeNameType    dest;
aon_thread_t    mythread;

#undef  FuncName
#define FuncName        "main"
```

```
/*
 * demo client mainline
 *
 * demo client sends a specified file to the destination node
 * check for it from command line, ask user for config variables,
 * initialise threads package, initialise IKM, and then create a
 * thread to send messages to destination.
*/
main(argc,argv)
int argc;
char **argv;
{
        if(argc!=2) {
                fprintf(stderr,"Usage: demo_client <filename>\n");
                exit(1);
        }

        /* Ask user for config/demo variables */
        aon_currentnode = askfor("Src Node ID?");
        dest            = askfor("Dest Node ID?");
        FRAGMENTED      = askfor("Fragmented?");
        ASYNC           = askfor("Asynchronous?");
        JUNKING         = askfor("Drop Blks?");
        aon_csdebuglevel= askfor("Debug?");
        MSEND           = askfor("Multisend?");
        filename        = argv[1];
        aon_serverport += aon_currentnode;

        /* initialise the threads package */
        aon_thread_init();

        /* initialise the IKM */
        if(! aon_ikm_init(aon_serverport) ) {
                Diag(("Can't init IKM"),1);
                exit(1);
        }

        /* create a thread for the demo_client */
        aon_thread_create(&mythread,0,(void(*))demo_send,0);

        /* start the threads running */
        aon_thread_exit();
}

#undef  FuncName
```

```
#define FuncName        "demo_send"

Boolean demo_send()
{
        CUR_PROC;

        NodeAddressType addr;
        IkmHeaderType   *request;
        int             len1,fd,strlen();
        OpIdType        opid;
        char            tempbuf[20];
        struct stat     statbuf;
        NodeNameType    mdest[3];
        NodeAddressType maddr[3];
        IkmBuffer       *mybuf[3];
        IkmHeaderType   *myreq[3];

        /* fix up threads - just don't ask why! */
        cur_proc->tdesc = aon_cur_thread;

        /* we assume server is on the same machine, and has its    */
        /* port number is the sum of IKM_DEMO_SERVER_PORT (demo.h) */
        /* and its NodeNameType value                              */

        /* build destination address for single sends */
        aon_getnw_addr(&addr,CurrNode,IKM_DEMO_SERVER_PORT+dest);

        /* build msend destination NodeNameType's */
        mdest[0]=1;
        mdest[1]=2;
        mdest[2]=3;

        /* build msend destination NodeAddrType's */
        aon_getnw_addr(&maddr[0],CurrNode,
                       IKM_DEMO_SERVER_PORT+mdest[0]);
        aon_getnw_addr(&maddr[1],CurrNode,
                       IKM_DEMO_SERVER_PORT+mdest[1]);
        aon_getnw_addr(&maddr[2],CurrNode,
                       IKM_DEMO_SERVER_PORT+mdest[2]);

        /* open the file we're going to transmit */
        if(stat(filename,&statbuf)) {
                printf("Couldn't open file\n");
                aon_ikm_exit();
                exit(1);
```

```
}

/* perform a checksum on it before sending. We do a checksum */
/* at the other end to prove message transfer was successful */
sprintf(tempbuf,"sum %s",filename);
system(tempbuf);

/* build contig version of message */
len1  = statbuf.st_size + IKM_HDR_SIZE;
request = (IkmHeaderType *)malloc(len1);
bzero(request,len1);

/* read file into contig buffer */
fd=open(filename,O_RDONLY,0);
read(fd,(Address)request+IKM_HDR_SIZE,statbuf.st_size);
close(fd);

/* for contig sends, send same request to each dest node */
myreq[0]=request;
myreq[1]=request;
myreq[2]=request;

/* build fragmented messages */
mybuf[0]=aon_ikm_openbuffer();
mybuf[1]=aon_ikm_openbuffer();
mybuf[2]=aon_ikm_openbuffer();
aon_ikm_appendbuffer(mybuf[0],(Address)(request)+IKM_HDR_SIZE,
                     statbuf.st_size);
aon_ikm_appendbuffer(mybuf[1],(Address)(request)+IKM_HDR_SIZE,
                     statbuf.st_size);
aon_ikm_appendbuffer(mybuf[2],(Address)(request)+IKM_HDR_SIZE,
                     statbuf.st_size);


/* start whichever message send was asked for */

if(FRAGMENTED) {

        if(ASYNC) {
                if(!aon_ikm_asend_frag(dest,&addr,mybuf[0],FRAGED))
                        Diag(("Couldn't send message"),1);

        } else if(!MSEND){
                if(! aon_ikm_send_frag(dest,&addr,mybuf,FRAGED))
                        Diag(("Couldn't send message"),1);
```

```
		} else {

		/* msend to 3 nodes */
		if(aon_ikm_msend_frag(3,mdest,maddr,mybuf,FRAGED)==False)
			Diag(("Message not entirely successful"),1);
		}


	} else {
		if(ASYNC) {
			if(!aon_ikm_asend(dest,&addr,request,len1,CONTIG))
				Diag(("Couldn't send message"),1);

		} else if(!MSEND) {
			if(!aon_ikm_send(dest,&addr,&request,len1,CONTIG))
				Diag(("Couldn't send message"),1);
		} else {

		/* msend to 3 nodes */
		if(aon_ikm_msend(3,mdest,maddr,myreq,len1,CONTIG)==False)
			Diag(("Message not entirely successful"),1);
		}

	}


	/* free up request (now reply) contig and fraged msg buffers */
	free(request);
	aon_ikm_closebuffer(mybuf[0]);
	aon_ikm_closebuffer(mybuf[1]);
	aon_ikm_closebuffer(mybuf[2]);

	/* shutdown the IKM and exit */
	aon_ikm_exit();
	exit(0);
}


/*
 * just convert yes/no answer from user to 1/0
*/
int askfor(str)
char *str;
{
```

```
        char ch[10];
        int ret;

        printf("%s ",str);
        scanf("%s",ch);

        switch(ch[0]) {
                case 'y':
                        ret=1;
                        break;
                case 'n':
                        ret=0;
                        break;
                default:
                        ret=atoi(ch);
                        break;
        }

        return(ret);
}
```

Demo Header File

```
#include "../support/eltypes.h"
#include "../support/am_types.h"
#include "../support/diag.h"
#include "../ikm/ikm.h"
#include "../ikm/ikm_if.h"
#include "../ikm/ikm_buf.h"

#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/timeb.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>

#define IKM_DEMO_CLIENT_PORT    3000
#define IKM_DEMO_SERVER_PORT    3100
```

```
/*
 * Some configuration constants
*/
int aon_mspertick    = 800;
int aon_csdebuglevel = 0;
int aon_amdebuglevel = 0;
int aon_debuglevel   = 0;

int MSEND,ASYNC,FRAGMENTED,JUNKING,FORWARD_NODE,
REQUEST_SIZE,REPLY_SIZE;
```

Demo GNU Makefile

```
.SILENT:

#
# Define compilers and assemblers
#
CC      = cc
LD      = /bin/ld
AS      = /bin/as
CPP     = /usr/lib/cpp

IKMFILES     = ../ikm
SUPPORTFILES = ../support

#
# Search path for makefile
#
VPATH   = .


#
# Set up flags for the C compiler.
#
CFLAGS  =  -I. -I/usr/include -I$(IKMFILES) -I$(SUPPORTFILES)
           -DIKM -DUSE_SIGIO -DMIPS


NAME="$(notdir $<)  ($(suffix $(basename $(subst /,.,$(dir $<)))))"
#
# Set up pattern rules
#
```

```
%.o     : %.c
        @echo " Compiling: " $(NAME)
        @$(CC) $(CFLAGS) -c $(CPPFLAGS) -o $@ $<

%.o     : %.s
        @echo "Assembling: " $(NAME)
        @$(AS) -o $@  $<

#
# Object files upon which amadeus server depends
#
OBJS =  $(IKMFILES)/ikm_buf.o \
        $(IKMFILES)/ikm.o \
        $(IKMFILES)/ikm_if.o\
        $(IKMFILES)/ikm_subr.o\
        $(SUPPORTFILES)/unix_lib.o\
        $(SUPPORTFILES)/am_threads.o\
        $(SUPPORTFILES)/am_sem.o\
        $(SUPPORTFILES)/locore.o

all     : demo_client demo_server

demo_client     : $(OBJS) demo_client.o
        @echo "Linking demo_client"
        @rm -f demo_client
        @$(CC) $(CFLAGS) -o demo_client $(OBJS) demo_client.o
        @chmod 755 demo_client

demo_server     : $(OBJS) demo_server.o
        @echo "Linking demo_server"
        @rm -f demo_server
        @$(CC) $(CFLAGS) -o demo_server $(OBJS) demo_server.o
        @chmod 755 demo_server

        @echo ^G
```

Appendix B

IKM Source Code

This appendix contains the complete project source code, consisting of the following IKM files:

`ikm_types.h` contains macros and declarations for all the main IKM data structures including the header, message types, MTT and its entries, and the IkmBuffer.

`ikm_subr.[hc]` contains routines to insert, delete and search for entries in the Message Transaction Table. Unique Message IDs are also generate here.

`ikm_buf.[hc]` contains all IkmBuffer related routines i.e. buffer creation, reading & writing, insertion, etc.

`ikm_if.[hc]` contains all the interface routines - both synchronous, asynchronous, fragmented and contiguous versions.

`ikm.[hc]` contains all the message handling handling routines to deal with ACK/PING/ACK/NACK/FACK etc, as well as the routines which generate them. The input handler and timer handler are also implemented here.

The IKM is dependent on the following files for compilation and correct operation. Listings of these files are *not* included here:

`unix_lib.[hc]` The interface to UDP sockets.

`am_threads.[hc]` Threads package.

`am_sem.[hc]` Semaphore package.

The following timing routines generated the UDP internet and unix domain socket comparison graphs, as well as bcopy(3) timings [1]. Source listings for these are not included either, but are available from the author.

[1]Plotted with Multiplot XLNe v.1.1 on a Commodore Amiga 2000.

`memcpy.c` The bcopy(3) Timings.

`un{client,server}.c` Unix Domain Socket Sends.

`in{client,server}.c` Internet Domain Socket Sends.