Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

DIMMA Nucleus Design

Guangxing Li

Abstract

Real world applications require many ORBs to cover a wide spectrum of functional and non-functional requirements, and to cope with new technology (such as high performance networks, real-time and multimedia applications). To help manage the complexity and to save engineering effort of different ORBs, a framework ORB can be used as a kernel, which provides a common base for the construction of fully functional, application and technology domain specific ORBs.

The DIMMA nucleus is such a framework ORB, on which real-time and multimedia ORBs can be readily constructed. The nucleus has a generic communication scheme, which offers a high degree of protocol configurability and extensibility.

The DIMMA nucleus supports the ODP API developed concurrently within the ANSA work programme, and can be extended to support other object APIs, such as CORBA API and TINA API.

APM.1553.01 **Approved** 20th January 1997
Technical Report

Distribution:
Supersedes:
Superseded by:

DIMMA Nucleus Design

DIMMA Nucleus Design

Guangxing Li

APM.1553.01

20th January 1997

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by APM Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

APM Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(01223) 515010
INTERNATIONAL	+44 1223 515010
FAX	+44 1223 359779
E-MAIL	apm@ansa.co.uk

Contents

1 Introduction

1.1 Context

This document explains the current status of the Distributed Interactive Multimedia Architecture (DIMMA) nucleus design and implementation.

The motivation and goals of the DIMMA project can be found in [APM.1295], and for completeness a summary is given as follows:

- meeting the requirements of a distributed processing environment as defined by the TINA architecture [TINA94]. This aspect is being investigated through the ReTINA project
- investigating the impact of lightweight approaches to ATM (desktop area networks, home/office multimedia systems) on operating system structures and interfaces. This aspect is being investigated through the DCAN project.

DIMMA addresses service management, service binding and service quality of service (QoS) management at a level of abstraction consistent with the application programming interfaces found in current distributed object computing systems such as ANSAware, ANSAware/RT, Microsoft's OLE 2, OMG's CORBA standard and Bellcore's INA.

The DIMMA project includes

- the extensions that are needed to current distributed computing object models (e.g. stream interfaces), and their engineering implications
- the additions and extensions that are needed to manage distributed services (e.g. explicit binding operations)
- the extensions needed to current distributed object computing infrastructures to enable interworking between them (e.g. support for multi-protocol ORBs)
- the extensions needed to current distributed object computing infrastructures to enable local fine grained control and monitoring of resources to give integrity to quality of service guarantees
- the extensions needed to support distributed object services to enable fine grained control and monitoring of resources across a distributed system.

DIMMA takes the ODP computational and engineering models, and the work to date in ANSA on real-time and multimedia computing [APM.1270] [APM.1393] [APM.1460] [APM.1222] [APM.1314], quality of service management and performance management as a baseline.

Having developed the overall architecture, current work is focused on detailed design and prototyping to validate, demonstrate and calibrate the architecture.

1.2 Audience

The audience is assumed to have general knowledge about distributed processing, network communication, ODP, CORBA and DIMMA architecture. In particular, the audience is expected to be aware of the DIMMA/ODP API [APM.1555] and CORBA Universal Networked Objects (UNO) [OMG95].

1.3 A framework ORB[1]

It is well understood that real world applications require many ORBs to cover diverse functional and non-functional requirements, and to cope with technology improvements (such as high performance communication, real-time and multimedia applications).

The approach taken in the DIMMA project is to structure an ORB as a modular set of components that plug into a minimal **framework** ORB, rather than the monolithic structures found in current ORBs.

The benefits of a framework approach are that:

- it is simpler to meet stringent performance and size constraints by omitting components from an ORB framework than attempting to strip down a monolithic ORB
- the cost of developing specialized ORBs for (often low volume) niche or emerging markets is reduced by re-using the framework and design patterns of replaceable components
- interworking between specialized ORBs is simplified because of their *family resemblance*, reducing system integration cost
- standard ORB interfaces and protocols (such as CORBA API, CORBA IIOP) can be supported as personality modules, benefiting standard-based portability, interoperability, common services etc.

A modular approaches has already been used successfully in operating systems, where traditional system functions are structured as separate services running over a *microkernel*, simplifying operating system design, implementation, configuration and extensibility.

The DIMMA nucleus design is based around an analogous framework or “microkernel” ORB. It provides generic services and components that are universal to a wide range of system requirements and platforms, including these of real-time and multimedia. The generic services allow flexible control of communications, processing and memory resources. They form a standard base which supports other system specific ORB functions.

1.4 Challenges of a framework ORB

An ORB [OMG90] provides the mechanisms for object creation, invocation and deletion. An ORB manages typically three types of resources:

- communication resources, such as network sockets and their binding to objects/interfaces
- processing resources, such as threads for concurrent activities

1. CORBA term Object Request Broker (ORB) is loosely used as equivalent to nucleus, as they mean the same thing in the context of this document.

- memory resources, such as network buffers.

The main design challenges of a framework ORB is to provide general abstractions for representing many different resource management policies and processing schemes, including these which enable application level control. These abstractions should be modular enough so that their interactions are minimal, to allow designers free choice of components to compose into a specific ORB.

In terms of functionality, a framework ORB needs to provide the following abstractions:

- a generic communication scheme to allow the exploitation of different communication resource multiplexing policies and the construction of arbitrary protocols
- a generic threading scheme to allow the adaptation of different concurrency handling techniques
- a generic buffer management scheme to allow different data presentation, fragmentation and protocol processing techniques
- a generic event processing scheme to allow arbitrary composition of resource management activities.

1.5 Document organization

Chapter 2 discusses the design of the nucleus generic communication scheme.

Chapter 3 presents the structure and major implementation notes of the nucleus.

Chapter 4 discusses a RPC system implementation within the DIMMA nucleus framework.

Chapter 5 briefs the implementation of the CORBA IIOP protocol within the DIMMA nucleus framework.

1.6 Acknowledgements

I would like to acknowledge the valuable comments of Andrew Herbert, Nigel Edwards, Andre Kramer and Rob van der Linden.

Dave Otway has always been patient and ready to discuss and help, and my special thanks to him.

2 A generic communication scheme

This chapter presents the communication scheme of the nucleus.

It is accepted wisdom that to help manage the complexity of communication systems, network software has better being organized through some generic abstractions, as illustrated by the *socket* abstraction for Berkeley UNIX, *stream* abstraction for System V UNIX and the *x-kernel* system [HP91] for general protocol writing. However they are all more oriented towards operating system constructions rather than for ORB implementations, and often lack QoS processing, binding processing and control of multiplexing. This motivates us to design a dedicated abstract communication framework for the nucleus.

2.1 Requirements

Network components are at the heart of any distributed system. To meet the DIMMA goals, its communication system must meet the following requirements:

- *coexistence of many communication protocols*. For example, RPC protocols for normal object interactions, stream protocols for video/audio transportation and real-time transport protocols for control (signalling) messages
- *protocol function composition*. It is generally accepted that protocols are layered components, and there should be some abstraction to group functional modules into a complete unit
- *protocol configuration*. Protocols are often platform specific, and should only be configured when necessary support exists. Protocol configuration can be done either statically (at compiler time) or dynamically (at run time)
- *independence of concurrency and multiplexing*, which are the major sources of protocol overheads and inflexibility of protocol architecture
- *QoS, explicit binding and stream processing.*

2.2 Abstractions

The nucleus generic communication scheme has eight major abstractions, namely **module**, **protocol**, **channel**, **binding**, **binder**, **address**, **message** and **QoS** object.

2.2.1 Module

A module is a layer of protocol function, such as TCP, IP, ANSAware REX and MPS.

A module provides two functions (called openMaster and openSlave) to create channels for communication and a demux function for message demultiplexing:

```
channel = openMaster (dispatch, qos)
channel = openSlave (address, qos, dispatch)
status = demux (message, timeout, qos)
length = header()
```

The openMaster and openSlave reflects the asymmetric nature of a pair of communication endpoints when connection oriented protocols are used or when a client/server interaction paradigm is used. The dispatch parameter is a function for a channel to upcall when a message arrives. The channel created by openMaster is called a *master* channel, and the one created by openSlave is called a *slave* channel. The *address* parameter in *openSlave* is the master channel address corresponding a slave channel.

The demux function demultiplexes messages arriving at a module to its channels. It does this typically by inspecting the header information of a message, and then either queues the message for a target channel or upcalls the channel's dispatch or *read* functions.

Other generic module functions include header (returns the length of this module's protocol header length), module_id etc.

2.2.2 Protocol

A protocol is a stack of modules. A protocol is constructed by pushing modules together. Generic protocol functions are:

```
push (module)
module = pop ()
status = start (qos)
status = terminate ()
channel = openMaster (dispatch, qos)
channel = openSlave (address, qos, dispatch)
length = header ()
status = active ()
message = buffer_make()
```

A protocol uses the *open* functions of its modules to create channels. The header function returns the length (in bytes) of the collective headers of all its modules. The active function indicates whether or not there are still channels in operation.

The module at the bottom of a protocol is called an *anchor* module (this term is borrowed from the *x-kernel*). The module at the top of a protocol is called a *dispatch* module.

An instance of a module object can appear in only one protocol, since it embeds state information always specific to a protocol.

A protocol processes a specific type of messages (buffers), which provide specific header, body and trailer for the protocol. Therefore, each protocol has its own buffer manager to create protocol specific buffers.

2.2.3 Channel

A channel provides functions for communication:

```
status = call (message, reply_message, qos)
status = cast (message, qos)
```

```
status = reply (message, qos)
status = write (message, qos)
status = read (message, qos)
status = close ()
status = control (opcode, arg)
```

The `call` and `cast` functions effect client object invocations. The *read*, *write* functions effect normal message processing and stream invocation; together with `reply` they effect object invocation processing at server side.

Channels created by an anchor module are called *anchor* channels, and those created by dispatch module are called *dispatch* channels.

An anchor channel typically uses a network interface (e.g. a socket or device driver) directly for message passing. A dispatch channel typically calls an application level function (such as a server stub) for message processing. Details of message processing are given in section 2.4.

A channel, by default, knows its lower layer channel in the protocol stack and upper layer module, but may not necessary knows its upper layer channel, this allows flexible channel multiplexing and demultiplexing.

2.2.4 Binder

A binder is used to setup necessary infrastructure components (i.e. bindings) to allow an activity in one object to invoke an operation of an interface in another object.

Bindings and binder provides the bridge between a computational API and the framework ORB.

In the communication scheme, a binder has a set of protocols, and creates *bindings* by using of these protocols. A RPC system binder typically has the following operations:

```
server_binding = svr_bind (dispatch, qos)
client_binding = clt_bind (interface_reference, qos, dispatch)
status = active ()
```

Bindings can be generated either implicitly by some object adapter API (see section 3.3, which does not require any programmer intervention) or explicitly by some explicit binding API (which requires a programmer to call the API).

Each binder understands a specific interface reference format.If the binder understands multiple protocols, the interface reference format must allow multiple protocol address representation.

The `active` function indicates whether or not the binder has any active bindings.

2.2.5 Binding

A binding is a set of channels through which object interactions take place.

An interface reference can be generated from a (server) binding by wrapping its channel address information. An interface reference can then be used to generate a peer (client) binding by some address selection procedure and protocol by using the wrapped address. This reveals the important nature of channel asymmetry i.e. the need of a master channel (have an address to be used by an interface reference) and a slave channel (only to match the address).

In a typical multiple protocols environment, a binder creates one channel per available protocol when creating a server binding. This allows any client which shares common protocol with the server to complete the binding.

A server binding provides a function to generate an interface reference: `interface_reference = if_ref ()`

A client binding normally contains only the necessary channel(s) (typically one for RPC system) for object interactions.

For stream bindings (where the distinguish of client and server is not obvious), multiple channels may required at both ends, for example, one for video processing and another for audio processing.

2.2.6 Message

Messages are linked buffers. Each buffer contains a block of data abstracted within a presentation policy. This policy can be overridden via object inheritance (polymorphism) to reflect a particular protocol specific policy requirement. The policy is represented as marshalling routines, which are defined along with the specific computational object API types system.

Buffers may have a manager. Buffer managers are protocol specific (see section 2.2.2). They are also buffer type specific so that different types of buffers are not mixed.

A buffer manager implements a specific management policy. For example, standard sized buffers may be cached for reuse, while other sized buffers are released immediately when free.

Buffer and protocol are related but orthogonal, e.g. one type of buffer may be used by several protocols.

Protocol and buffer together determines a data buffering policy, e.g. whether or not linked buffers are used, whether or not buffer size is automatically expanded etc.

Each message has an *address record*, which is used for identifying a message destination by the sending site, and which provides the complemental address information of a message header (which is used for identifying a message target by the receiving site).

2.2.7 Address

An address has a sequence of address record. Each address record is a sequence of octet. This is a generic address form.

Each channel has an address record for its channel address. An address is normally the concatenation of a stack of channels (corresponding to the module stack of a protocol). The content of an address record, i.e. the octet sequence, is only meaningful to a particular module or channel. Each protocol needs to provide a special mapping of its address format to this generic address form.

2.2.8 QoS object

A QoS object is a record, which contains a set of properties to drive operations in protocols, modules, channels and binders.

QoS objects have operations to put and get property values.

Two types of QoS objects are introduced, one for interface (binding) creation, one for object invocation.

A simple and often used case of QoS object is to implement *port based* applications, where a well known service address (an interface reference) must be able to be hand crafted, such as the interface reference for a Trader or a Factory in ANSAware. In such case, a QoS object may be used to drive protocols to generate channels with specific addresses.

2.3 Examples

Figure 2.1 illustrates a case of protocol configuration and composition. In the figure, a box represents a module, and a stack of module represents a complete protocol.

Figure 2.1: Protocol Configuration and Composition

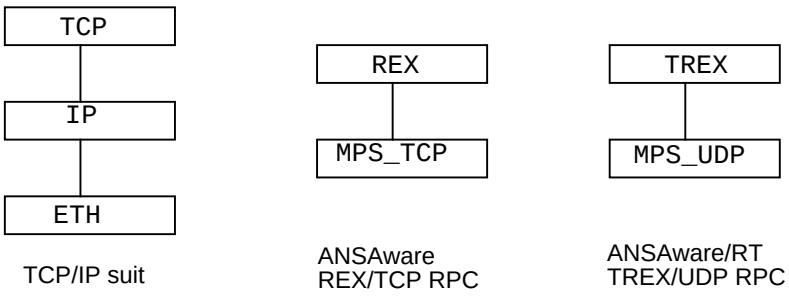


Figure 2.2: Server binding

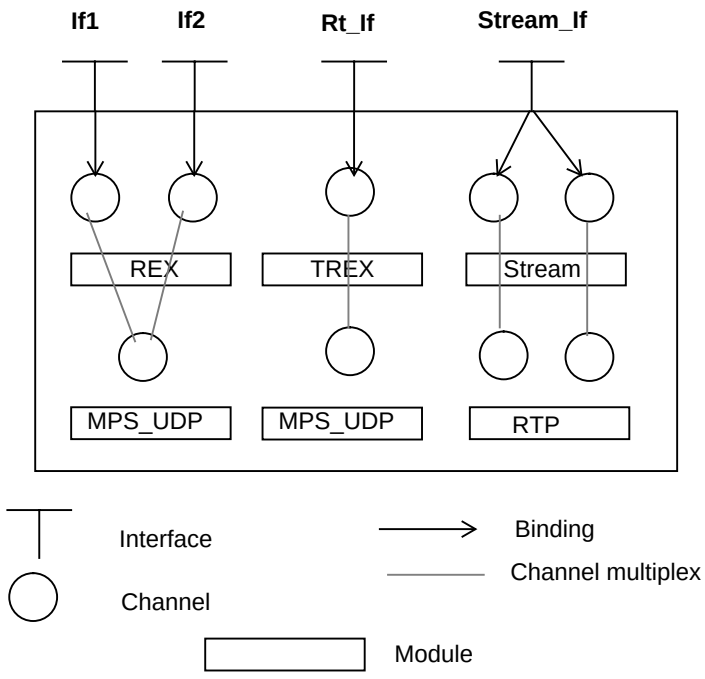
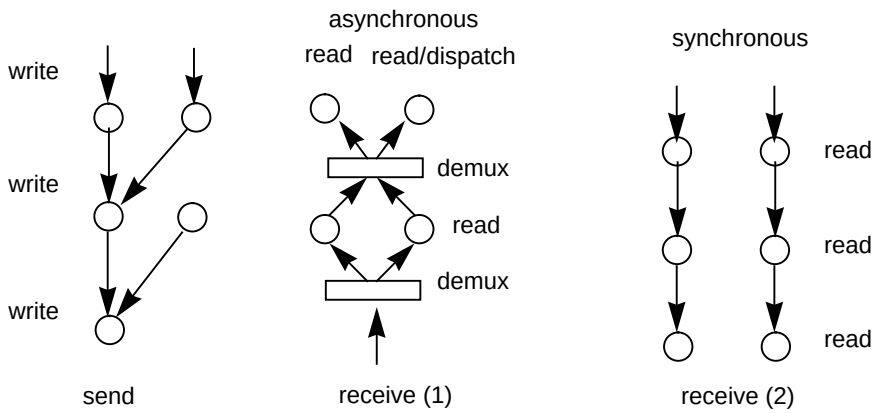


Figure 2.2 illustrates a case of server binding. There are three protocols in the example, namely REX/MPS_UDP for normal RPC, TREX/MPS_UDP for real-time RPC and Stream/RTP for stream transport. REX, MPS_UDP, TREX, Stream and RTP are modules. The three protocols may belong to three different binders (one for normal RPC, one for real-time signalling, one for stream processing). Two instances of MPS_UDP are used, one for REX/MPS_UDP and the other for TREX/MPS_UDP. Interfaces If1, If2 are each tied to a REX channel (which in turn shares a common MPS_UDP anchor channel). The real-time interface Rt_If has a TREX channel. The stream interface Stream_If has two Stream channels (each of which has a separate RTP anchor channel).

2.4 Message processing

Message processing is done by channel functions and module demux functions, as messages are passed by normal procedure calls between protocol layers. Message processing ends when processes arrive at a *dispatch* channel (for incoming messages) or at an *anchor* channel (for outgoing messages).

Figure 2.3: Message processing



Sending a message (including call, cast, reply and write) is processed down through a protocol stack. When a channel finishes its protocol processing for a message, it passes the message to the next lower layer channel. This procedure terminates when an anchor channel is encountered.

Receiving a message can be done asynchronously, synchronously or half synchronous and half asynchronously.

In the asynchronous receive case, shown in Figure 2.3 case (1), an incoming message is firstly processed by the demux function of an anchor module, which selects the target channel and passes the message to it. This channel either passes the message to its dispatch routine (if the channel is a dispatch channel), or passes the message to the demux function of the next upper layer module. The procedure ends when the message is dispatched.

In the synchronous case, shown in Figure 2.3 case (2), the read function of a channel calls the read function at a lower layer channel to get an incoming message. The read function of an anchor channel blocks waiting at a network interface. When a message arrives, it unblocks and passes the message to its upper layer.

Synchronous receive involves less protocol processing overhead but does not allow channel multiplexing. Synchronous and asynchronous message processing can also be combined, and it requires detailed knowledge about concurrency, channel multiplexing and event processing. This is further discussed in the next section.

2.5 Multiplexing and concurrency

Multiplexing is an essential part of any communication system to enable resource sharing. Arbitrary channel multiplexing is possible in the DIMMA communication scheme. However, the cost of multiplexing must be understood as it often prevents QoS and introduces extra protocol processing overhead. In applications where high performance and QoS are necessary, controlled multiplexing is an important technique.

Two simple channel multiplexing policies are

- maximum multiplexing - upper layer channels always multiplex at a lower layer channel, as used by IIOP and ANSAware REX
- no multiplexing - there is a one to one correspondence between a upper layer channel and a lower layer channel. This technique has been used by some multi-service RPC systems and most high performance RPC systems [McAuley89], e.g. to benefit the one multiplexing and demultiplexing ATM communication paradigm.

More complicated multiplexing policies require and are driven by QoS objects, one example is the Timed RPC (TREX) protocol developed in ANSAware/RT.

Controlled multiplexing and protocol concurrency processing are related topics, and they are better discussed by examples.

Figure 2.4: Client channel multiplexing and concurrency (1)

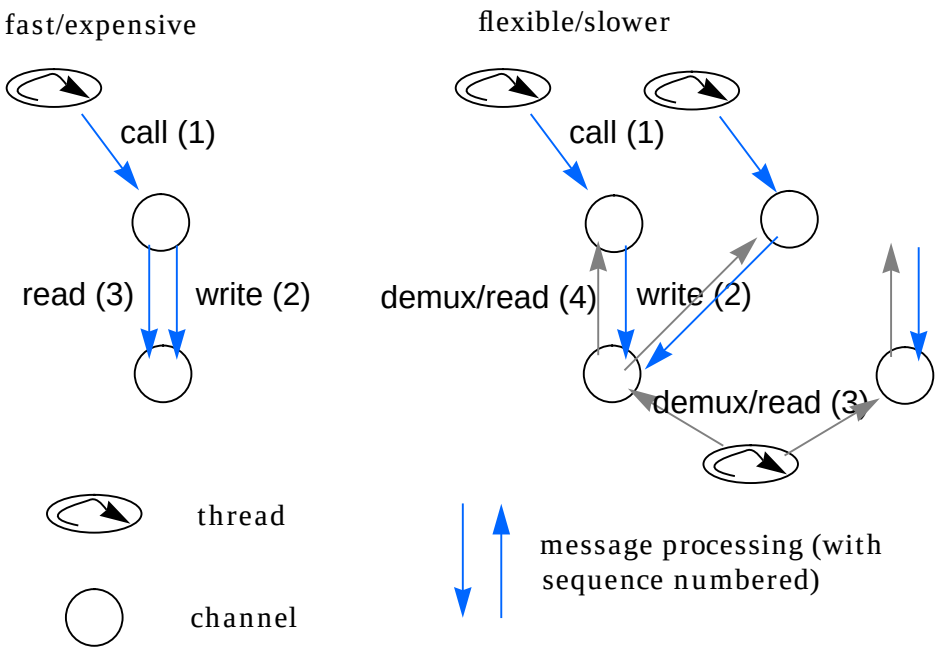
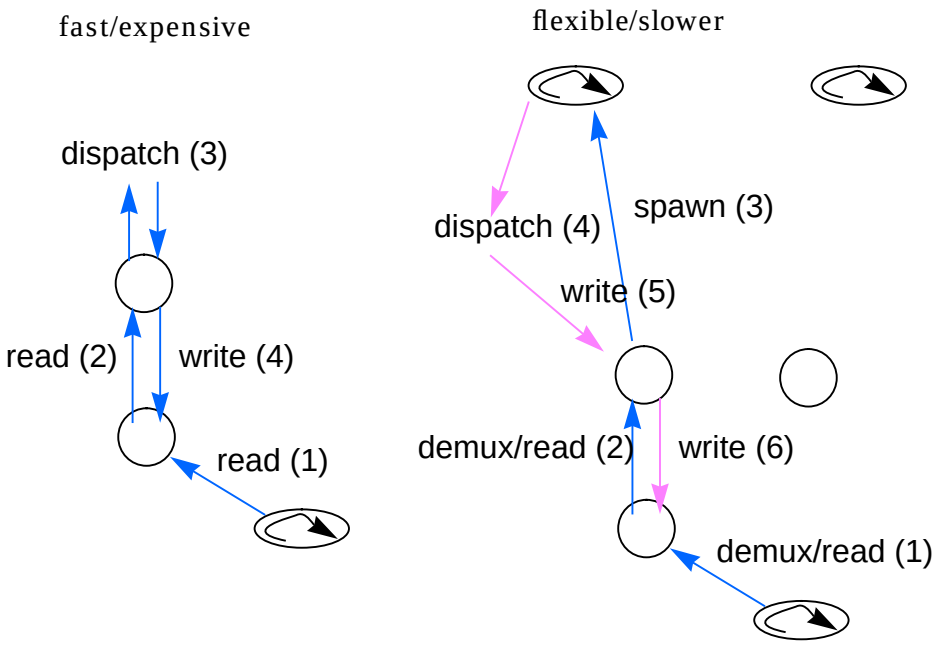


Figure 2.4 shows two cases of client message processing in a RPC system. The first case is the non-multiplexing case, an object invocation send the message along the channels and then waits at the anchor channel for a reply message. As no message demultiplexing is required, and the reply comes directly to the calling thread, protocol processing time is minimized. However, the price is high, for each RPC channel, a dedicated message processing anchor channel is required, and there is no concurrency access allowed for the RPC channel.

The second scenario uses maximum channel multiplexing. The calling thread waits at its own RPC channel for reply messages. All network incoming messages at anchor channels are processed by a dedicated network listening thread, which demultiplexes replies to destination RPC channels, and then wakes up the calling thread. This case allows RPC channels to share lower layer message processing (anchor) channels, and resource usages are minimized. The price paid is a demultiplexing processing and an extra thread context switch.

Figure 2.5: Server channel multiplexing and concurrency (1)



Similar cases for server channel message processing are given in Figure 2.5. In the second case a new thread is spawned for each invocations.

Figure 2.6 shows how to allow concurrent RPC calls by introducing a session layer at client side. A session caches the information about a channel and an invocation, and enables concurrent access to a channel.

Figure 2.7 shows how to have controlled server concurrency by introducing a message queue. It shows actually the ANSAware server side message processing scenario. The scheme is extended in ANSAware/RT by having multiple message queues (called entry), this allows separate (fine-grained) resource allocation and management for different (real-time and non-real-time) interfaces.

Figure 2.6: Client channel multiplexing and concurrency (2)

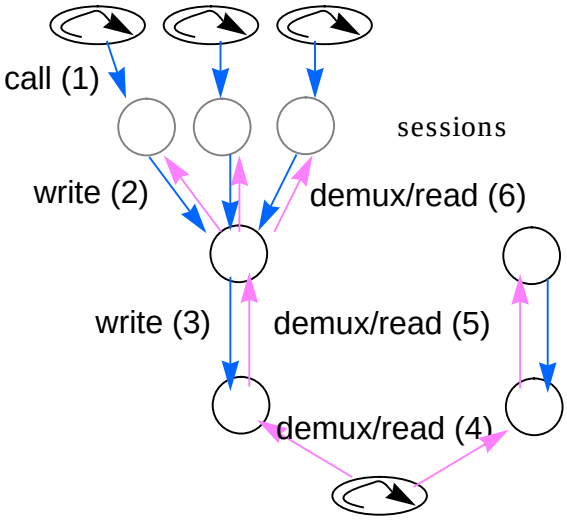
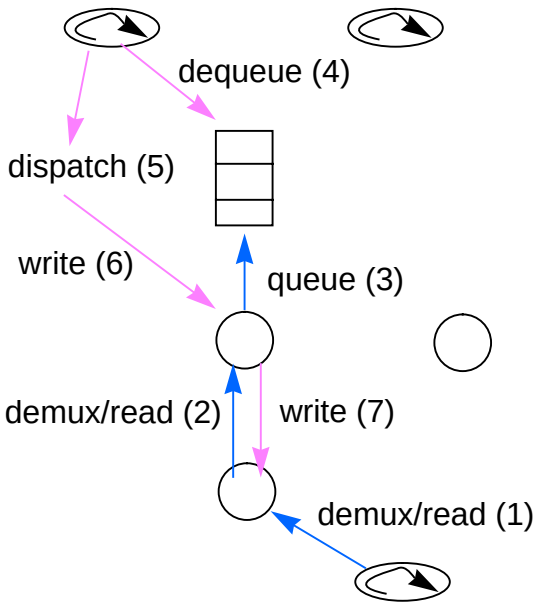


Figure 2.7: Server channel multiplexing and concurrency (2)



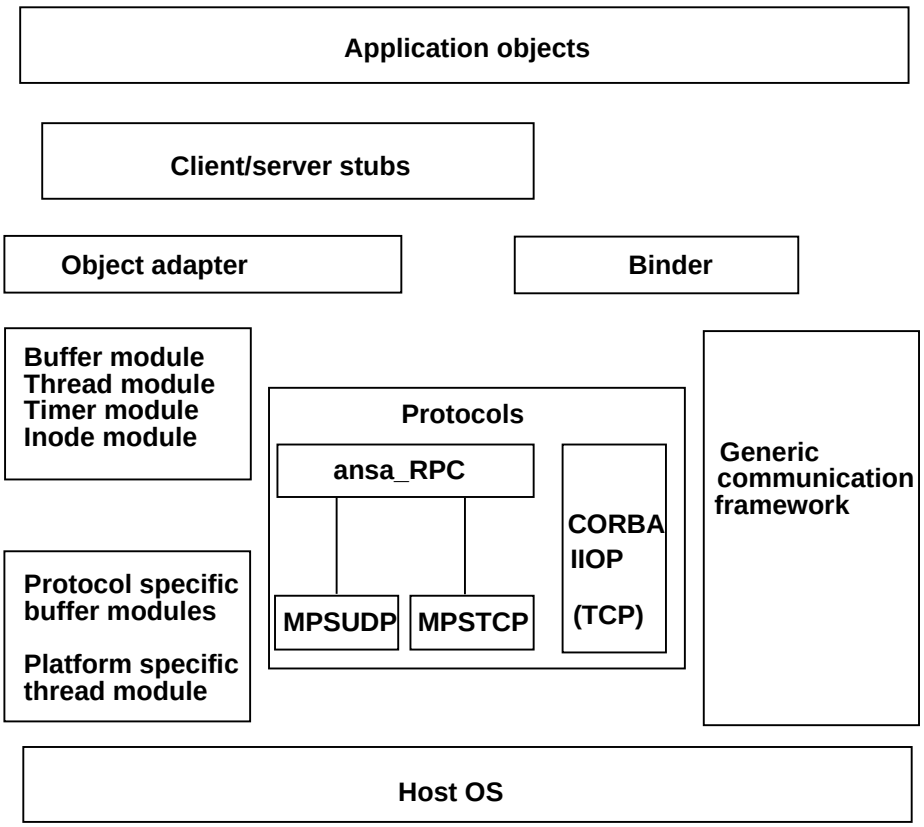
3 Implementation notes

This chapter gives a description of the various nucleus modules implementation in the C++ prototype.

3.1 Structure

The structure of the nucleus is given in Figure 3.1.

Figure 3.1: Nucleus structure



The nucleus consists of a number of generic software modules and some derived platform specific or protocol specific modules. They are explained in the following sections.

3.2 Protocols

The nucleus implements four protocol modules: `ansa_RPC`, `MPSUDP`, `MPSTCP` and `IIOP`. They can constitute three protocols: `ansa_RPC` with `MPSTCP`, `ansa_RPC` with `MPSUDP`, and `IIOP`.

Apart from providing standard abstractions defined by the framework, these protocols have an extra abstraction: **sessions**, which are used to associate invocations to channels.

The `ansa_RPC` and `IIOP` are further discussed in later chapters.

3.3 Binder and object adapter

The nucleus has a binder for setting up implicit bindings for the DIMMA API. The binder has three protocols as discussed in the last section.

The binder uses CORBA2 UNO Interoperable Object References (IOR) as its interface reference format. IOR supports multiple protocols and allows arbitrary protocol address formats as they can be encapsulated as IOR protocol profile (as a byte sequence).

A server binding is created when a server stub is generated. A server binding has three channels, each for a protocol of the binder. The interface reference (an IOR) for a server binding contains three tagged profiles, each for one of its channels.

A client binding is created the first time a remote invocation is made. The binder finds one of its common protocols with an remote interface (by searching the IOR profiles), and creates one channel for the binding.

A client/server binding is deleted when its related stub is deleted.

The object adapter implements routines for API to access the binder and its generated bindings. This module also implements the IOR to/from character string transformation routines defined by CORBA2 UNO.

3.4 Thread

An abstract thread package interface is defined and which must be implemented by the real thread package of a platform.

All threads are managed, i.e. have a manager, and are created by a manager. The nucleus has a *default* thread manager, and extra thread manager can be created by an application.

Thread manager is designed to apply some overall policies to its threads. At the moment, a thread manager only reports how many threads are active. This would be expected to be enhanced with other thread managing routines as defined in the ANSAware/RT entry abstraction.

Threads are created as normal objects. An optional property attribute can be used to generate real-time threads (if the real thread package supports real-time threads).

The following thread manipulation primitives are defined:

- join: wait till another thread finishes
- yield: let other thread execute

- cancel: send a signal to another thread
- detach: no thread will join this thread
- exit: force finishing.

The following synchronization mechanisms are defined:

- binary mutex
- conditional variable and timed conditional variable
- semaphore and timed semaphore.

The nucleus implements two real thread packages: one for POSIX thread environments and one for non-threaded environments (null thread).

The implementation of a semaphore package using POSIX synchronization primitives is not trivial when a spurious thread wakeup is allowed, and readers interested are suggested to study the source code.

The protocol implementation is designed in such a way that to take appropriate event processing actions depending on whether real concurrency is supported or not, and therefore able to work without forcing a specific concurrency scheme.

3.5 Buffer

Buffer is the basic abstraction for message processing as discussed in section 2.2.6.

A buffer defines virtual functions for implementing a data representation protocol, which are overridden by protocol specific procedures.

Buffers can be either managed (created by a buffer manager) or non-managed. A managed buffer is typically used for stubs and they are subject to some system defined management policy (such as free, creation and cache). Non-managed buffers are often for temporal use, and their management is done by protocol code. Non-managed buffers are used frequently in our IIOP implementation to marshal GIOP encapsulation.

Buffer managers are typed, and are dependent on the type of buffer they manage.

3.6 Index management

Table lookup to map some identifiers from network messages to some protocol objects (channels or sessions) is required by protocol processing.

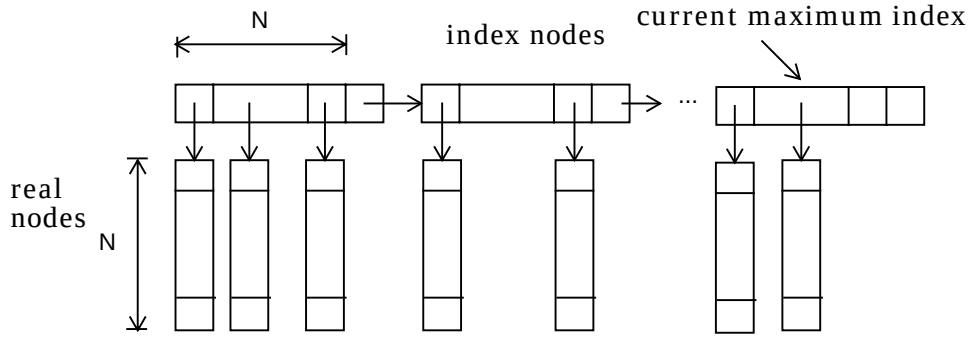
If a system is to scale (from few interfaces to millions of interfaces, and from few activities to millions of activities), this table lookup operation is one of the keys. The requirements are twofold

- be able to allocate real protocol objects (either channels or sessions) on demand
- be able to keep the lookup time constant (neither linear search nor hash tables do this).

A technique similar to UNIX **inode** management is used in the nucleus. But instead of using fixed length indirection (typically 3 levels in UNIX), the nucleus uses one level and automatically expanding index nodes. This is

illustrated in Figure 3.2. In this structure, both index nodes and real nodes (which can be channels or sessions) are allocated on demand (i.e. when the required real nodes exceed current allocated number). Real nodes are allocated in group of size `N`, to prevent excessive memory segmentation.

Figure 3.2: Index nodes



With such an arrangement, table lookup is an indexing operation from an index number to a real node. The object provides this service is called *inode* in this document.

3.7 Timer

The timer module provides a facility for the creation and management of both sporadic and periodic alarms, which can be used by both protocol code and application code.

A `Timer` abstraction is supported and the following overloaded operator can be applied:

- +, -, =
- <, >, <=, >=, ==

3.8 Tracing and debugging

The nucleus can be conditionally compiled to provide tracing messages.

Tracing is based on individual nucleus modules. To give tracing informations, an environment variables (ANSA_TRACE) can be set to the required modules value.

4 A RPC protocol

This chapter describes how an RPC protocol, called ansa_RPC, has been implemented in the DIMMA nucleus. It works on both TCP and UDP. The RPC protocol is lightweight and is designed for internal use, experimentation, and performance comparison with standard RPCs.

4.1 Structure

The RPC is a protocol layer in the generic communication framework. It implements the module and channel abstract classes in the framework.

The RPC module has two inodes: one for channel management, and one for session management.

The RPC channel implements both client channels (created by openSlave) and server channels (created by openMaster).

A session object is used for a client channel to maintain the state for each invocation.

In a multi-threaded environment, a dedicated message demultiplexing thread is created. It receives all network incoming messages and dispatches them to the right channels and sessions by the module demux function (single thread case is discussed in section 4.9).

The RPC module requires a stateless message passing service (MPS) protocol layer (like ANSAware). Two such layers are implemented: one for TCP, one for UDP. They are named as MPSTCP and MPSUDP. Each such MPS layers implements also the module and channel abstract classes in the framework.

Specifically, the RPC module provides the header, start, openMaster, openSlave and demux functions. The client channel provides call and cast functions. The server channel provides the dispatch function.

The MPS module provides also the header, start, openMaster, openSlave and demux functions. Its channel provides write and reply functions.

4.2 Client channel

Each client channel contains information about its peer (server channel) and maintains a session list for all ongoing concurrent invocations.

4.2.1 Session

A session is allocated for each invocation. Each session contains a request_id, a result status, a reply buffer and a semaphore. The request_id is allocated protocol module wise, and is used to guard against reused sessions (because of timeout, for example). The module manages its request_id in some *non-descent* order for this purpose. An invocation normally waits at its

session semaphore, until a reply buffer is collected by the module demux function. The invocation thread then is woke up (and picks up the reply) by the message demultiplexing thread. A session is immediately released after an invocation is finished.

A request message is sent out by a MPS channel write function.

4.3 Server channel

Each server channel has a nonce and a dispatch function. The nonce is a random number and is used to guard against reused channels (for reasons of reallocation, service destruction etc.).

An invocation request is dispatched by the demultiplexing thread, which generates a new thread for each request. The generated thread executes the channel dispatch function with the incoming message and thus upcalls to a server stub. A reply message is sent back by invoking a MPS channel reply function.

Client and server channels are not symmetric: for each server channel, there may be many client channels.

4.4 Messages

All RPC messages have a fixed header with the following fields:

- magic, 4 octets, for the indication of an ansa_RPC message
- version, 2 octets
- type, 2 octets, type of messages
- request_id, 4 octets
- channel, 4 octets, peer channel index
- nonce, 4 octets
- session, 4 octets, session id
- data_length, 4 octets, length of payload.

Four types of messages are defined: CALL, CAST, REPLY and NACK.

4.5 IOR profile

The interface reference generated by the nucleus binder is CORBA2 IOR. The IOR has a tagged profile for each protocol the binder understands. Each profile contains the required information for object (interface, to be more precise) location.

For the RPC protocol, its profile in the IOR has the following fields:

- channel id, 4 octets
- nonce, 4 octets
- host IP address, 4 octets,
- host port address, 2 octets

This profile is sufficient for the RPC to run on either TCP or UDP, because they use the same IP address family.

4.6 Presentation

User data are marshalled in network byte order and packed continuously as a byte stream (i.e. no alignment gaps are padded) in the RPC messages.

The RPC buffer marshalling functions pack the outgoing data into network byte order, and the unmarshalling functions transfer them back into host byte order.

4.7 MPSTCP

The `MPSTCP` protocol layer provides a message passing service based on host TCP protocol.

A single channel is allocated and used by all upper layer (RPC) channels. The channel encapsulates a passive TCP socket, which provides the address for IOR and enables the opening of connections for TCP transportation.

TCP sockets (connections) are opened on demand and cached, this allows the sharing use of socket resources and their concurrent access.

Only the `write` function in the `MPSTCP` channel enforces the opening of a new connection (if not found in the current cache). For the `reply` function, opening a new connection is not required, because a client must have abandoned a early connection, and this often means it is either down or not interested in a reply.

The `MPSTCP` layer adds another header to the RPC messages, the header has two fields:

- `magic`, 4 octets
- `data_length`, 4 octets

The `data_length` field helps the receive site to allocate a sufficient large buffer for an arriving message.

4.8 MPSUDP

The structure of the `UDP` protocol layer is similar to the `MPSTCP` layer. A single channel is allocated and used by all upper layer (RPC) channels. The channel encapsulate a UDP socket, by which all communications take place.

The `MPSUDP` layer does not add any header to the RPC message. Because UDP messages have a fixed largest length, it is always safe for the receiver to collect incoming messages with a buffer of the largest length.

It must be point out that because the `ansa_RPC` layer itself is not reliable, run it over `MPSUDP` is not reliable as well.

4.9 Null thread case

If the nucleus runs on a non-threaded platform (i.e. the nucleus is not provided with an implementation of its `Thread` class), the protocol code can automatically find this out and is designed to do synchronous data receive instead.

For an invocation, the calling thread, after sending out the request, just polls network incoming messages by calling the RPC module `demux` operation, until a reply message is come back or timeout.

For the server side, when the main thread calls `capsule_ready()`, it polls network messages, and dispatches requests to channels (and stubs) by using its own thread resource.

In a mixed server/client environment, the above scenarios can be combined, e.g. a client, while waiting a reply, may use its thread resource to serve invocation requests. But it is application writer's responsibility to avoid deadlock or liveliness problems because of lack of real concurrency.

5 An IIOP implementation

This chapter discusses the CORBA IIOP protocol implementation in the DIMMA nucleus. IIOP allows the interoperation between different CORBA compliant ORBs.

5.1 IIOP

CORBA2 defines the Universal Networked Objects (UNO) for interoperability. Its main interoperation protocol is called GIOP, and its mapping and implementation over TCP is called IIOP.

UNO defines a standard format for interface reference representation, called IOR, which is a type string plus a sequence of tagged protocol profiles. Each profile itself is an opaque octet sequence, and can be used to encapsulate any address information.

IIOP has a special mapping for its profile, called IIOP IOR profile, which has the following fields:

- `version`, 2 octets
- `host`, a character string
- `port`, 2 octets
- `object_key`, an opaque octet sequence.

GIOP defines the following components:

- Common Data Representation (CDR), which defines the wire format (as an octet stream) for all CORBA IDL types. Data in the octet stream are kept alignment and in local host byte order
- RPC Messages, GIOP defines seven types of messages for RPC processing. Each message has a common header plus a message type specific header
- Connection management, GIOP is designed to execute with a connection oriented protocol. Conventions are defined to open, close and multiplex network connections.

IIOP further details GIOP connection management in the case of TCP.

5.2 IIOP implementation

The nucleus IIOP implementation maintains the major features of the native `ansa_RPC` protocol given in the last chapter:

- it allows concurrent invocations by introducing sessions
- server stubs are automatically dispatched by a new thread
- object lifetime is guarded by nonce, this prevents invocations on dead objects (interfaces)

- network connections are cached for reuse and concurrent access.

Object keys are differentiated at client and server sites. A server site object key is a channel id plus its nonce (which can then be encapsulated to the GIOP object_key format for IIOP IOR profile), a client site object key remains the same as the object_key format in the IIOP IOR profile, which allows this IIOP implementation to talk to other vendors' IIOP.

Unfortunately, IIOP mandates a special network connection management scheme (IIOP failure semantics depends on TCP socket semantics), which makes it difficulty (if not impossible) for us to implement it as two protocol layers as is done in ansa_RPC. A single protocol module is used entirely for IIOP.

Particularly, a session list is not managed by a client channel, but is managed by a network connect, so that when the connection is closed or an error happens, related invocations can be informed.

GIOP request message has a request_id for the identification of invocation source (used by a reply message). Naturally, our implementation uses session id as this request_id. Because there is no ansa_RPC message request_id in GIOP, it would be dangerous to allow an invocation timeout (and a related session being reused), an obsolete reply may destroy another invocation. Therefore invocation timeout is not allowed in our IIOP implementation. However, this constraint only applies to clients using our API, it does not affect foreign clients.

GIOP CDR is implemented by a special buffer module for data alignment and byte order transformation.

5.3 Differences with ansa_RPC

Although the IIOP implementation inherits many of the ansa_RPC features, it has the following differences which are necessary by the standard:

- data presentation: ansa_RPC wire representation is network formatted and there is no data alignment padding, while IIOP's CDR is in local host byte order and data are kept alignment (by padding)
- message header: ansa_RPC messages have a fixed length protocol header, while IIOP messages have a fixed length protocol header plus a message type dependent header which is normally variable length
- IOR profile: ansa_RPC IOR profile uses IP address for its host address, while IIOP IOR profile uses a string for its host address. It means an IIOP implementation has to use some name server (normally DNS) to do the string to IP address transformation at some binding stage, this can be costly and also be a restriction for primitive (such as embedded) systems where no DNS is supported.

5.4 Test

A standard test (published by SUN Microsoft Inc.) has be carried between our implementation and SUN's implementation. Object invocations using basic types have been tested successfully. Other tests can only be done when the appropriate CORBA APIs are supported, and they are less relevant to the IIOP implementation.

5.5 A discussion of GIOP relocation service

GIOP object relocation facility is not implemented, because of lack of similar API. More importantly, it is still arguable that the GIOP approach to object relocation is useful as discussed as follows.

Most ORB implementations run as user level processes, and objects live within such processes. Therefore it is against intuition to have the GIOP object relocation scenario, i.e. an ORB is still there while some of its objects moved away. It is more nature to assume an ORB and its objects all moved, and the ANSAware approach to object relocation by using a relocation server is better, especially when the relocation service is used to cope with machine crash or failures. In such cases, it is simply impossible to run a relocation agent at the same machine and address as required by GIOP.

References

[APM.1295]

G Li, A Herbert and D Otway, An Overview of the Distributed Interactive Multimedia Architecture, Architecture Projects Management Ltd., Cambridge U.K., 1995.

[APM.1270]

G Li and D Otway, An Open Architecture for Real-Time Processing, Architecture Projects Management Ltd., Cambridge U.K., 1994.

[APM.1393]

D Otway, Streams and Signals, Architecture Projects Management Ltd., Cambridge U.K., May 1994.

[APM.1460]

G Li, ANSAware/RT Version 1.0: Programming and System Overview, Architecture Projects Management Ltd., Cambridge U.K., May 1994.

[APM.1222]

G Li, Some Engineering Aspects of Real-Time, Architecture Projects Management Ltd., Cambridge U.K., May 1994.

[APM.1314]

D Otway, Explicit Binding, Architecture Projects Management Ltd., Cambridge U.K., June 1994.

[APM.1555]

D Otway, ODP C++ API Design Overview, Architecture Projects Management Ltd., Cambridge U.K., 1995.

[OMG95]

OMG, CORBA 2.0/Interoperability, Universal Networked Objects, OMG TC Document, March, 1995.

[OMG90]

OMG, Object Management Architecture Guide, OMG TC Document, November, 1990.

[HP91]

N Hutchinson and L Peterson, The x-kernel: An Architecture for Implementing Network Protocols, IEEE Transactions on Software Engineering, 17(1):64-75, January 1991.

[McAuley89]

D R McAuley, Protocol Design for High Speed Networks, PhD Thesis, University of Cambridge Computer Laboratory, Technical Report 186, September 1989.

[TINA94]

Stefano Montesi, Requirements upon TINA-C Architecture, TINAC TB_MH.002_2.0_94, December 1994.