

A Distributed Object Platform Infrastructure for Multimedia Applications

Geoff Coulson

Distributed Multimedia Research Group,
Department of Computing,
Lancaster University,
Lancaster LA1 4YR,
UK.

telephone: +44 (0)1524 65201
e-mail: geoff@comp.lancs.ac.uk

ABSTRACT

Although distributed object computing has developed rapidly over the past decade, and is now becoming commercially important, there remain key application areas inadequately supported by current standards and implementations. This paper describes research aimed at support for one of these areas: distributed soft real-time/ multimedia applications. The approach is to provide a low level platform which offers generic middleware services useful for the implementation of a range of multimedia capable distributed object systems. The design of the platform is influenced on the one hand by the real-time/ multimedia-oriented computational model of the RM-ODP and on the other hand by recent research results in the efficient engineering of communications systems and operating systems. The platform provides support for quality of service (QoS) and application specific protocols as required by multimedia capable distributed object systems. A novel scheme for flexible QoS specification and management is described. A performance evaluation of the platform is given and a sample application program is presented to illustrate the platform's API.

Keywords: distributed object computing, distributed multimedia, CORBA, quality of service.

1. Introduction

Distributed object computing has developed rapidly over the past decade and has now moved out of the research lab into the various fields of industry and commerce. The benefits of using currently available distributed object computing products are that application writers view a uniform distributed computational model and are isolated from the heterogeneity of underlying systems (i.e. different networks, end-systems, communications protocols, operating systems and language environments). The resulting ease of application development has boosted the widespread deployment of distributed applications, and companies providing commercial distributed object platforms are expanding rapidly and becoming highly successful.

A key factor in this success has been the development of international standards for distributed object computing. These standards include the Object Management Group's Common Object Broker Architecture (CORBA) [OMG,96], Microsoft's DCOM, the Java RMI, the Open Group's Distributed Computing Environment (DCE), and the ISO's and ITU-T's Reference Model for Open Distributed Processing (RM-ODP) [ITU-T,95a]. CORBA has

been particularly influential, and most of today's distributed object technologies are based on the CORBA standard. The RM-ODP is less concrete in nature than CORBA but is broader in scope. In addition, the CORBA specifications themselves have been considerably influenced by RM-ODP.

Despite these developments, there are key application areas in which distributed object platforms have lagged behind ad-hoc approaches to building distributed applications (e.g. using a UNIX based sockets/TLI interface to TCP/IP). In particular, support for *distributed multimedia applications* is weak or non-existent in today's distributed object products. Furthermore, multimedia support is not yet mature in the CORBA specifications although it has been considered in general terms in RM-ODP. In this paper, we discuss a middleware design called GOPI (viz. General Object Platform Infrastructure), that addresses the requirements of this application area by providing core services for the support of multimedia-capable distributed object platforms. It is intended that the semantics of *particular* distributed object platforms be implemented on top of GOPI as toolkits (e.g. a student is currently building a CORBA based toolkit layer featuring RM-ODP influenced multimedia extensions). This separation between mechanism and API (cf. personalities in microkernel based operating systems) allows us to concentrate our research on generic performance and QoS management issues which are crucial for distributed multimedia application support.

GOPI is implemented as the following set of modules: *base* (a collection of generally useful foundation programming classes), *threads* (a user level 'real-time' thread package), *msg* ('real-time' inter-thread message passing and buffer management), *comm* (an architecture for accommodating application specific protocols) and *bind* (a module supporting communication endpoints called *irefs*, and a binding protocol and QoS negotiation framework). Each module is realised as an independent library with minimal dependencies. For example, the threads module is standalone and it is possible to replace the bind module without impacting the comm module.

The remainder of this paper is structured as follows. Section two briefly discusses the key influences on our design. These are i) the RM-ODP's computational model for multimedia, and ii) a number of engineering principles abstracted from recent developments in multimedia communications and operating systems research. Section three then gives a detailed description of the GOPI system in terms of its modular structure and section four describes example application specific protocols. Section five complements the structural description with a dynamic view of the system that focuses on the non-multiplexed operation of the control and data paths (this is GOPI's prime means of supporting predictable QoS). Section six then presents a simple program to illustrate the use of the GOPI API, and section seven offers a comparative performance evaluation of the system. Finally, section eight considers related work and section nine presents our conclusions and plans for future development.

2. Influences on GOPI

2.1 The RM-ODP's Multimedia Computational Model

The RM-ODP's multimedia computational model implies specific requirements on a multimedia capable distributed engineering infrastructure and is thus of key relevance to the design of GOPI. The RM-ODP's starting point is the assumption that the model of object interaction conventionally adopted in distributed object platforms - i.e. operation invocation - is inappropriate for *continuous media* such as digital audio or video. For these media types, a *streaming* mode of communication is required rather than a request/ reply based operation invocation model.

The RM-ODP approach builds streaming interaction on top of the primitive notion of *signal* which is defined as the emission/ reception of a data item from/ to an interface¹. A stream is modelled as a sequence of signal emissions from a producer interface together with an associated sequence of signal receptions at a consumer interface. In RM-ODP the emission or reception point of such a sequence of signals is known as a *flow* (as opposed to an *operation* in the traditional request/ reply style of interaction), and an interface containing flows rather than operations is known as a *stream interface*. Stream interfaces may contain many flows with varying *types* and *directionalities*. The type of a flow refers to the format of the data items to be carried across the interface and is defined, like an operation, in terms of a type signature (i.e. a flow name together with named and typed input and output parameters). The directionality of a flow is either *in* or *out*.

The model assumes an *explicit*² interface at each end of an interaction -- a client and a server interface in the case of the operational interaction and a producer and consumer in the case of the stream interaction. RM-ODP also makes explicit the association or *binding* between pairs (or, more generally, sets) of interfaces. A binding is an object with its own (operational) interface through which the binding can be monitored and controlled. Monitoring operations allow the user of the binding to obtain information about the real-time performance (i.e. QoS) of the underlying data communication, and control operations allow the user to adapt the operation of the binding (e.g. to increase the bandwidth or add additional jitter-smoothing buffers). In general, binding objects abstract over the potentially complex details of multimedia communications and selectively hide these details from the programmer. Bindings do not have to be created directly by objects involved in the binding but may instead be created by *third party* objects which obtain references to interfaces owned by those objects. This facility eases the configuration and structuring of potentially complex multimedia applications containing many per-media objects.

The final key element in the RM-ODP computational model is *QoS specification*. QoS refers primarily to the required temporal constraints that must be satisfied by communicating objects and is specified in terms of temporal predicates over signals. For example, a QoS specification may specify that *n* signal instances must arrive at a particular flow within a certain time interval (i.e. a statement of throughput requirement). Similar statements can be used to describe constraints on jitter and delay and on timing relationships across multiple interfaces (as required by multimedia synchronisation).

2.2 Engineering Principles for Supporting Multimedia

Recent years have seen a flurry of research in supporting multimedia (or, more particularly, continuous media) in networks and operating systems. Some general principles which have emerged from this research, and which have influenced our design goals, are as follows.

Application Specific Protocols It has become clear that a ‘one size fits all’ approach to protocol provision is not appropriate; there are potentially large numbers of media types used by multimedia applications each with their own separate and individual QoS requirements [Campbell,93] and a single protocol capable of supporting all such requirements would be extremely large and unwieldy. Beyond this, protocol functionality required by future

¹ Signals underpin both streaming and operation invocation interaction styles; in fact they are intended as a generic abstraction on which to layer all conceivable interaction styles. An operational invocation is built as the composition of four signals (i.e. an emission from the client interface, a reception and emission at a server interface and a reception back at the client).

² Many object platforms do not use explicit references at the client end of an interaction, or provide client interface references that have less generality than server interfaces.

applications cannot possibly be known in advance due to the ever increasing diversity in media encodings, network types, application topologies etc. What is required is an architecture in which it is straightforward to add new protocols, preferably at the user level [Thekkath,93], so that unforeseen application specific and media specific communication requirements can be easily accommodated. *Application layer framing* (ALF) [Clarke,90] is an important principle guiding the implementation of application specific protocols. ALF helps to minimise protocol processing overhead by assuming that the application layer chooses packet characteristics appropriate to the underlying network services.

System structuring This covers three main areas: *multiplexing*, *upcall structuring* and *integrated layer processing* (ILP). Multiplexing multiple protocol entities at level n over a single protocol entity at level $n-1$ is to be avoided in multimedia systems because it results in so-called ‘QoS crosstalk’ [Tennenhouse,90]. This means that it is difficult for a single layer $n-1$ protocol underlying multiple layer n protocol instances to simultaneously meet the differing QoS needs of the various layer n instances. A related problem is that of resource accounting; it is difficult in a multiplexed system to cleanly delineate the resources used by different activities and therefore to accurately maintain per-activity resource budgets. In addition, Schmidt [Schmidt,97] has shown that demultiplexing is a significant source of inefficiency in CORBA implementations. In an *upcall structured* system [Clarke,85] receivers do not poll for incoming messages as is common in traditional communications systems. Rather, the control flow is initiated from below; each layer is informed of the arrival of data via an *upcall* and it, in turn, upcalls the layer above. This scheme ensures timely delivery of data and moreover minimises buffer occupancy time. ILP [Clarke,90] proposes that systems should be structured into as few layers as possible, and that in implementation, layers and processing loops should be collapsed as far as possible.

Soft real-time support Initial attempts at supporting multimedia borrowed heavily from the real-time community and adopted ideas such as hard real-time scheduling (e.g. earliest deadline first or rate monotonic scheduling [Liu,73]) and resource admission testing. Although these principles are still required for high quality multimedia in applications (such as remote surgery [Williams,92]), more recent work on multimedia over the Internet has demonstrated that low quality multimedia (e.g. as required for computer based conferencing or application sharing) can be adequately supported in an environment of standard workstations and communications protocols such as TCP/UDP/IP. The key operating principles in such an environment are *fair share* and *adaptivity*; fair share ensures that resources are applied equitably across multiple sessions at a relatively coarse level of granularity, and adaptivity builds intelligence into applications so that they can make best use of the resources they are given. A prime example of adaptivity is a jitter control mechanism which dynamically adjusts the size of an elastic buffer so that it is large when network jitter is high and small when network jitter is low (to minimise end to end latency) [Jacobson,92].

General efficiency considerations Although it is neither sufficient or necessary for a multimedia system to make efficiency its highest priority goal, it is clearly desirable to optimise those aspects of the system that are particularly stressed by typical multimedia applications. In practice this amounts to minimising and optimising *copy operations* (multimedia applications handle large data units and therefore copy operations are time consuming in terms of both CPU cycles and cache misses) and *context switches* (multimedia applications are typically structured as many co-operating processes or threads and the resulting context switch overhead significantly impacts performance through domain crossing latency and cache misses). Recently developed *zero copy architectures* [Druschel,96] perform all protocol processing in both kernel and user space on a single buffer and thereby avoid all copy operations between the user application and the network. Minimisation of context

switches can be achieved through the use of *user level threads* which, ideally, are integrated with the kernel scheduler to avoid priority inversions [Govindan,91].

3. GOPI Modules

3.1 The Base Module

The base module contains generally useful low level programming objects such as lists, stacks and hash tables. These objects can be directly used by stubs and language bindings as well as by the core GOPI modules. The base module also contains abstractions of all the operating system calls used by the whole GOPI system. This eases porting, as all operating system dependencies are in one place, and moreover enhances the readability of the code in the other modules as operating system dependent idiosyncrasies are hidden from the higher layer code.

3.2 The Threads Module

The threads module provides lightweight user level threads together with primitives for thread synchronisation and timing. The main motivation behind providing a new threads module rather than using currently available services such as Posix threads was to facilitate experimentation with a range of scheduling policies for multimedia. An additional benefit is efficiency; GOPI threads are significantly faster than Solaris 'processor scoped' pthreads, for example¹.

GOPI threads are preemptible by external events such as timer expiry and IO availability but are not timesliced by default (although a timeslicing option is available as a compile time option). In the absence of timeslicing, a thread can be made to yield and enter the thread kernel by an explicit `thread_yield()` call. Threads are created using a call which allows the creator to assign optional scheduling parameters (i.e. priority, deadline and period). There are also calls to modify the scheduling parameters of a thread as it runs. The scheduling policy used is priority based with earliest deadline first scheduling within priority bands.

For synchronisation, the threads module provides semaphores. These have timeouts on the `P()` operation (similar to the Posix `cond_timedwait()` service) to assist in controlling temporal behaviour:

```
bool thread_P_timeout(SemId sem, long us);
```

`Thread_P_timeout()` returns FALSE if it returns due to a timeout and TRUE if it returns due to a `thread_V()` operation invoked by another thread.

The threads module also supports per-thread timers. The call to set a timer is as follows:

```
thread_timerset(int timerid, Function timerfunc, long us);
```

`Timerfunc` is a pointer to a function which is upcalled after `us` microseconds have elapsed on timer `timerid`. The number of timers available per thread (e.g. 20) is fixed at compile time. The implementation of timers relies on an operating system service such as Unix's SIGALRM; the per thread timers are multiplexed over a single instance of such a service using a delta queue.

¹ A program was written that creates two threads which both execute a tight loop. Thread 1 executes: `while(1) {P(sem1); V(sem2);}`, and thread 2 executes: `while(1) {V(sem1); P(sem2);}` (the semaphores are both initialised to 0). A version using GOPI threads executes 2.2 times faster on a Sun SPARCserver-1000 than a version using SunOS 5.5 detached 'processor scoped' threads with the SCHED_OTHER scheduling policy and the `sem_wait()` and `sem_post()` operations for `P()` and `V()` respectively.

The threads module employs a generic and extensible ‘deferred interrupt’ framework for operating systems level event notification (e.g. of timer expiry, or of IO availability as used in the comm module). For each type of event to be handled, the scheme employs i) an operating system level *event handler* (e.g. a UNIX signal handler routine) ii) an event specific *event flag* and iii) an event specific *callback routine*. The event handler simply sets the event flag, deferring to the callback routine to actually handle the event. The callback routine is invoked each time execution enters or leaves the thread kernel and acts upon the event in case the event flag is set. As well as being generic and extensible, this scheme is also highly *responsive* as it is arranged that the callback routine is called as soon as possible after the event flag is set. This works as follows. If the event handler sees that the thread kernel is not locked, it directly enters the thread kernel itself which, as above, directly causes the event specific callback to be invoked. If, however, the thread kernel *is* locked, the event handler immediately returns as execution is known to be already in the thread kernel and the event flag will be soon be acted upon anyway. Because OS level handlers are non preemptible, only a simple boolean variable is required to implement the kernel lock: if the handler sees that the variable is unset it enters the kernel, else it simply sets the event specific flag and returns.

The threads module employs the abstraction of *virtual processors* (VPs) to enable it to exploit parallelism in shared memory multiprocessors. VPs, which are implemented as operating system provided kernel-threads, spend their lives executing GOPI user level threads. The intention is that one VP per available physical processor is created so that VP level concurrency is (potentially) truly parallel. Unfortunately, VPs incur inevitable overheads. In particular, a configuration with more than one VP must use operating system level mutex locks to implement the thread kernel lock rather than the simple boolean variable referred to above.

3.3 The Msg Module

The msg module provides message-based inter-thread, intra address space, communication services which build on timed semaphores. Messages are sent and received on intermediary *channels*. The msg/ buffer module also manages *buffers* which are the unit of data transfer in inter-thread communication calls (in the same address space). The buffer manager allocates and de-allocates variable sized buffers from a pre-allocated pool of memory owned by the msg/ buffer library. A ‘buddy system’ allocation scheme [Knuth,73] is used which efficiently allocates and de-allocates buffers whose size is a power of two¹.

The message passing primitives are as follows:

```
bool msg_asend(ChanId c, Buffer *buf, bool blk);
bool msg_arecv(ChanId c, Buffer **buf, long us);
```

Both routines return FALSE if a buffer was not sent (resp. received). A buffer will not be sent and `msg_asend()` will return immediately if its `blk` argument is FALSE and the channel `c` is full (otherwise, if `blk` is TRUE, the call will block until space is available and the buffer will then be sent). Similarly, `msg_arecv()` will return having failed to receive a buffer if the timeout specified by `us` expires before a buffer has been placed in `c` by a corresponding `msg_asend()`.

`Msg_asend()` and `msg_arecv()` implement a pass-by-reference style of communication in which the caller of `msg_arecv()` provides the address of a buffer pointer; thus no copy

¹ In practice, it is not necessary to request buffers that are an exact power of two; the buffer manager rounds up the requested size to the nearest power of two and sets a ‘user data size’ field in the buffer descriptor to the user’s requested size.

operation is incurred. This enables very efficient inter-thread communication¹ although it does require the programmer to be careful - the sender must not attempt to write to a buffer having sent it. The convention is for a sender to allocate a buffer and send it and for the receiver to de-allocate the buffer after it is finished with. An optimised mode of inter-thread communication is available for short messages. Here, a zero length buffer is allocated and sent. Allocating a zero length buffer returns a buffer descriptor that does not point to any allocated pool memory. However, the buffer descriptor itself contains a 20 byte ‘annexe’ field which can be used to pass a short message. Buffer descriptors are maintained in a cache so there is no overhead in allocating memory for the descriptor.

3.4 The Comm Module

3.4.1 Services

The comm module adopts a multi-level protocol architecture consisting of *transport protocols* (provided by the operating system) and *application specific protocols* (ASPs). ASPs sit on top of the available transport protocols and provide application specific services (examples are given later in this section).

The comm module provides the following services:

- an architecture to support the implementation and integration (either at compile time or at run time through dynamic linking) of ASPs,
- a *transport abstraction layer* which gives ASPs uniform access to the available range of underlying operating system level transport protocols.

The comm module also exports generic primitives to send and receive data on specified connection identifiers using the appropriate ASP and transport protocol.

3.4.2 Transports

The transport protocols currently configured are TCP, UDP and UNIX named pipes (FIFOs). As a general rule, TCP is used by ASPs which support reliable bindings, e.g. operational bindings, and UDP is used in stream bindings. It is simple to add support for other transports where available; e.g. AAL5 for ATM networks.

The *transport abstraction layer* eases both the integration of new transports and the implementation of ASPs. It provides uniform connection oriented connection management access to all installed transports (whether or not they themselves are connection oriented). Similar to the ASP layer (see below), the transport abstraction layer’s connection management service uses Berkeley socket style `listen()`, `connect()` and `accept()` entry points.

3.4.3 ASP Framework

To allow new ASPs to be easily added to the system, GOPI provides a standard typedef through which a set of per-ASP entry points can be specified:

```
typedef struct asp {
    Function listen;
    Function connect;
    Function accept;
    Function hdrsize;
    Function flipqos;
}
```

¹ In addition, no context switch occurs on `msg_asend()` if `blk` is `FALSE` or on `msg_arecv()` if `uS` is 0.

```

    Function send;
    Function receive;
    Function call;
} Asp;

```

The `listen()`, `connect()` and `accept()` entry points are used for connection management (including QoS specification) and the `send()`, `receive()` and `call()` entry points are used for data transfer. The `hdrsize()` entry point is used by higher level code that allocates buffers and needs to leave buffer space for ASP specific headers. The `flipqos()` entry point is used by external un-marshalling routines to request the byte-swapping of a per-ASP QoS specification (see below) which has been sent from a machine with an opposite endian architecture to that of the receiver.

The `listen()`, `connect()` and `accept()` operations build on the services provided by the transport abstraction layer to provide QoS mapping and resource management functions. In GOPI, individual ASPs are responsible for defining a schema for QoS specification and for mapping QoS specifications expressed using this schema into low level GOPI resources (primarily threads, buffers and transport service access points or TSAPs). The reason for delegating QoS management to individual ASPs rather than attempting to handle it generically is that it is extremely difficult to provide a generic set of QoS parameters and resource management functions sufficient for any arbitrary application. If, however, these functions are provided on a per-application/ per-ASP basis, the mapping from QoS specifications to resource requirements becomes trivial (it is simply hard wired into the ASP's code). More details of the QoS mapping and resource management roles of ASPs are given in sections 3.4 and 3.5.

GOPI guarantees that ASP implementations can assume that their data transfer entry points (i.e. `send()`, `receive()` and `call()`) are only called when the underlying operating system is ready to accept/ deliver data; e.g. when the ASP's `receive()` entry point is called, the ASP knows there is data available to be read from the transport abstraction layer. This assumption considerably eases the implementation of ASPs.

To detect the availability of incoming messages (and/or operating system buffer space for outgoing messages) on behalf of ASPs, GOPI uses IO polling based on the UNIX `poll()` call (or its equivalent in other operating systems) as will be described in section 5. Alternatively, in order to respond in a more timely (although possibly less efficient) fashion to IO events, GOPI can be configured via a compile time switch to use *asynchronous IO notification* facilities such as those provided by SIGPOLL signals in the UNIX environment (all UNIX specific details are hidden, however, by the operating system abstraction code in the base module).

To install a new ASP, it is only required to provide i) a source file that implements the calls contained in the `Asp` struct, and ii) an initialisation routine (to be called from the main GOPI initialisation routine) which calls `comm_aspregister()`; the latter takes as its argument a pointer to an `Asp` structure filled in with the ASPs entry points. Having installed an ASP in this way, communication endpoints can be created that refer to the new ASP (see section 3.5). Subsequently, when `send`, `receive` and `call` invocations are made on these endpoints, GOPI arranges for the routines of the appropriate ASP to be called via a *protocol switch* data structure which is a table of sets of ASP entry points indexed by an ASP integer identifier.

Given the above framework, it becomes relatively easy to implement and install an ASP. As illustrated in figure 1, ASPs have a fixed upper interface and also have a fixed lower interface to the available transports via the transport abstraction layer. Furthermore they can

exploit all the core GOPI services such as threads, timers, buffers and inter-thread communication to realise their required protocol services. Example ASP implementations are described in section 6.

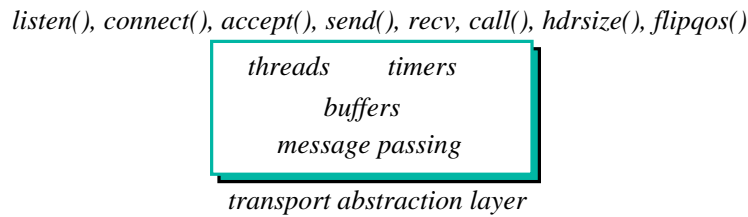


Figure 1: ASP Implementation

3.4.4 Currently Implemented ASPs

The current version of GOPI provides seven ASPs:

- LOCAL efficiently binds irefs in the same address space by means of GOPI channels; it therefore incurs no data copy and only a user level context switch per communication,
- TCPASP provides a minimally value added service over TCP,
- FIFOASP provides similar facilities to TCPASP over UNIX named pipes (FIFOs),
- SHMEM provides shared memory based communications; it uses UNIX FIFOs to synchronise access to per-binding shared memory segments,
- UDPFRAG is a UDP based service which provides simple fragmentation/ re-assembly and error recovery services over UDP,
- AUDIO is a dedicated ASP for audio transmission over UDP,
- IIOP is a subset of the OMG's Internet Interoperability Protocol and is used for request/ reply communications only.

The UDPFRAG and AUDIO ASPs are described in detail as case studies in section 6; their description is deferred until the relationship between ASPs and the binding protocol has been clarified in the following section.

3.5 The Bind Module

3.5.1 Services

The bind module provides the following services:

- the abstraction of *irefs* (cf. RM-ODP interface references) which serve as endpoints of communication,
- (with the assistance of ASPs) a framework for the specification, negotiation and mapping of QoS, and
- an 'out-of-band' connection management protocol to support the *binding* of irefs, via an ASP, with a given QoS.

The bind module also exports ASP independent primitives to send and receive data between bound irefs using the appropriate ASP.

3.5.2 Irefs

Irefs are the endpoints of communication in GOPI. Each iref has a *flowtype* of either *outin*, *inout*, *in* or *out*, and a *role* of either *provider*, *customer* or *provider/customer*. The flowtype designators identify the types of signal supported by the iref; *outin* and *inout* support paired signals for traditional client/ server style operational interactions, and *in* and *out* support stream interactions where signals are emitted at one interface and received at the other. The provider/ customer terminology is adopted because the more usual *client/ server* distinction becomes meaningless in the case of interactions between *in* and *out* irefs¹. A *provider/customer* iref is able to simultaneously act as a provider and a customer (see section 3.5.4.3).

Irefs also contain a list of *profiles* which contain *i*) the name (integer identifier) of an ASP, *ii*) an associated QoS specification and *iii*) an associated transport endpoint address. As explained above, the representation of QoS is transparent to GOPI and is viewed and mapped only within individual ASPs. The QoS specifications attached to an iref contain ‘default’ values that can be overridden by the QoS parameters passed to the `bind_bindreq()` service (see below).

Finally, it is possible to register a *handler* with an iref. A handler is a C function that is upcalled by GOPI when data is to be delivered (assuming an *in* or *inout* iref), or obtained from the sender (assuming an *out* iref). Handlers interface GOPI to a stubs/ language binding layer where they are used to, for example, call a method on an invoked C++ object. For traditional server interface references there will likely be a one-to-one correspondence between irefs and computational model level interface references; calls to all the methods in the interface will be multiplexed over the same iref. For stream interfaces, however, a stubs/ language binding layer may use a separate iref for each flow in the interface if each flow requires its own individual ASP and QoS.

3.5.3 Binding Protocol

3.5.3.1 Binding Establishment

For communication to take place in GOPI it is necessary that irefs are first explicitly *bound* using some ASP. The primary interface to the binding protocol is the following:

```
bind_bindreq(Iref *customer, Iref *provider, Aspname asp, CharSeq *qos);
```

This call binds the two irefs `customer` and `provider` using the ASP specified in `asp` and the QoS specified in `qos`. QoS mapping and resource allocation (e.g. of threads, buffers and TSAPs) are performed by the `listen()`, `connect()` and `accept()` operations of individual ASPs rather than by the binding protocol itself². All the binding protocol provides is a means to exchange QoS/ resource information between peer ASP instances, the interpretation of which is left entirely to the ASPs. QoS information is generically passed to ASPs by

¹ For example, the ‘service’ in such a case may be either *in* (e.g. a continuous media file server onto which the output of a camera is being stored) or *out* (e.g. a video on demand service).

² Note that the GOPI model of binding differs from the Berkeley sockets scheme in that GOPI passes binding protocol messages *above* the level of the `listen()`, `connect()` and `accept()` API (see the following description). In the Berkeley sockets API, the inter-peer communication is *below* the level of the API and thus hidden from the API user. In addition, in GOPI it is the customer (cf. client) that calls `listen()` and `accept()` while the provider (cf. server) calls `connect()`. In the sockets API, the server calls `listen()` and `accept()` while the client calls `connect()`.

superimposing ASP-specific QoS representations on a generic `CharSeq` struct¹ (cf. the `sockaddr` mechanism in the Berkeley sockets interface). The form of the binding protocol is illustrated in figure 2.

As detailed description of the operation of the binding protocol is as follows (we assume in the following that `customer` is co-located with the caller of `bind_bindreq()`²).

1. The customer side first performs a sanity check on the provider `iref` and then calls the local ASP entity's `listen()` entry point which determines resource requirements, and attempts to allocate local resources. Resource requirements are derived from QoS information³ passed to the `listen()` call. The `listen()` call returns a possibly modified `CharSeq` QoS struct (e.g. it may have been downgraded depending on local resource availability).
2. The customer side of the protocol then upcalls the customer `iref`'s handler (if a handler is attached) to inform the application code associated with the `iref` that an attempt is being made to bind to it and to obtain its permission to be bound. This upcall uses the `event` parameter of the standard handler prototype (an example is given in section 6).



Figure 2: Binding Protocol

3. The customer side then builds a message containing the two `irefs` and the required QoS specification and sends it to the provider end using the IIOP protocol. The address used is a standard *binding manager* address which is specified in the provider `iref`. There is only one management address per address space and all bind, unbind and control (see below) requests for all `irefs` supported in that address space are directed to this address.
4. At the provider side, the binding protocol first confirms that the `irefs` are compatible (i.e. that their flowtypes match and that they share a common ASP). It then checks and allocates resources and possibly modifies the QoS struct again via the provider side ASP entity's `connect()` entry point which operates in an analogous manner to `listen()`.
5. The provider side then upcalls the provider `iref`'s handler (if present) to obtain the permission of the application code to create the binding in an analogous way to step 2.

¹ This is defined as: `typedef struct charseq {int length; char *data} CharSeq;` ASPs may choose to provide a utility function to allow higher level code to build a `CharSeq` data structure given a set of QoS parameters; see the code in section 3.9 for an example.

² The customer and provider `irefs` specified to `bind_bindreq()` do not have to have been created in the same address space (process) as the caller of `bind_bindreq()`; it is possible to pass `irefs` around a distributed application and bind them from any 'third party' location. In the case where the caller of `bind_bindreq()` is not co-located with the customer `iref` (i.e. a third party bind), the management protocol first (transparently) contacts the customer which then initiates the above procedure.

³ The QoS information used is either the `QoS` argument passed to `bind_bindreq()` or, if a non specific "ANY_QOS" `qos` argument was passed to `bind_bindreq()`, the default QoS in the provider `iref`.

6. In case the provider *iref* has an associated handler, the binding protocol creates a dedicated *per-binding thread* for the provider end of the binding. If the *iref* has an *out* flowtype, the thread is periodic and runs at a rate determined by the ASP. All ASPs must provide this information to the binding protocol, together with information on the default fragment size so that the sender thread can allocate suitably sized buffers. The information is provided in the form of *period* and *fragsize* parameters returned from the `connect()` call. It is the responsibility of every ASP to provide these values by performing a suitable mapping from its ASP specific QoS parameters. Sender threads run a standard routine that repeatedly obtains data from the handler and then calls the ASP's `send()` entry point. Similarly, receiver per-binding threads run a routine which calls the ASP's `receive()` entry point and then delivers the data to the handler. These routines also perform simple generic QoS monitoring of the rate of stream connections.
7. If all is well, the provider end replies to the customer's request. The customer end calls the ASP's `accept()` entry point with the `CharSeq` QoS struct returned from `connect()`, and the customer ASP entity is given the final say (via its `accept()` entry point) on whether the binding will be accepted (with the QoS proposed by the `connect()` call); if it is decided to refuse the binding request, or if step 8 fails, the `bind_unbindreq()` service is invoked which carries out a further request/ reply transaction to free any resources held by the provider.
8. In case the customer *iref* has an associated handler, the binding protocol creates a per-binding thread for the customer end of the binding in an analogous manner to step 6 using information returned from the `accept()` call.

The ASP's connection management calls are able to make decisions as to resource availability by mapping the given ASP specific QoS parameters to primitive resource interfaces (i.e. threads, buffers and TSAPs) provided by the lower level modules of GOPI. As argued in section 3.4.3, it is easy for ASPs to perform this mapping as they are entities with a very specific purpose. GOPI's thread, buffer and TSAP creation routines perform simple admission tests to determine whether the creation of a new resource instance may compromise the QoS of existing bindings. For example, the thread creation routine performs a standard EDF admission test [Liu,73] and the buffer allocation routine checks a 'low water mark' indicating current buffer usage levels. Currently, these admission tests are rather rudimentary as GOPI runs in a standard operating system environment with minimal QoS based resource management. The resource management scheme is already useful in helping to provide QoS predictability in a non real-time environment; it would be of even greater utility in a QoS aware operating system environment [Coulson,95].

Finally, in addition to mapping QoS and allocating thread resources, an ASP's `listen()`, `connect()` and `accept()` routines will presumably conspire to create a connection at the transport layer via the transport abstraction layer described in section 3.4.2. This is entirely transparent to the binding protocol.

3.5.3.3 Binding Operation

Having created a binding, it is possible to pass data between *irefs*. In the case of an *in/ out* stream binding, data begins to flow as soon as the binding is created (i.e. the per-binding thread at the *out* end starts to execute immediately). For stream bindings it is possible for any involved party to stop and restart the flow of data on the binding by means of a `bind_ctl()` call (which, like `bind_bindreq()`, also works in a third party style). For traditional operational bindings there is no per-binding thread at the *outin* end of the binding and it

necessary for the stubs/ marshalling layer to explicitly call the following routine:

```
bind_call(Iref *customer, Buffer *outbuf, Buffer **inbuf);
```

`Bind_call()` simply invokes the `call()` routine of the binding's associated ASP. `Customer` is the `iref` at the customer ('client') side of the binding and `outbuf` is the buffer to send to the provider ('server'). On return, `inbuf` will point to a buffer that contains the reply from the provider.

Note that it is also possible for ASP peers to send and receive private protocol data unit messages between themselves transparently to the user of the binding.

3.5.4 Extended Bindings

So far, bindings have been considered to comprise an association between two `irefs` involving a two level, ASP and transport protocol, stack. While such bindings are adequate for many applications, GOPI attempts to provide maximal flexibility by providing two varieties of *extended binding*. These are called *ASP stacking* and *compound bindings* and are described in the following two sub-sections. Figure 3 illustrates the two forms of extended binding (note that the two forms can be used together).

3.5.4.1 ASP Stacking

ASP stacking allows new ASPs to be defined in terms of existing ASPs in a manner reminiscent of subclassing in object oriented frameworks. The purpose of ASP stacking is to support the incremental development of complex protocols. As an example, consider an ASP that transparently optimises intra-machine and intra-process communications. This could be implemented simply by selecting (in the connection management routines) from among from the already existing TCPASP, FIFOASP and LOCAL protocols depending on the location of the `irefs` being bound. Only a few tens of lines of code would be required for such an ASP. As another example, the AUDIO ASP described in section 4.2 is layered on top of the UDPFRAG ASP.

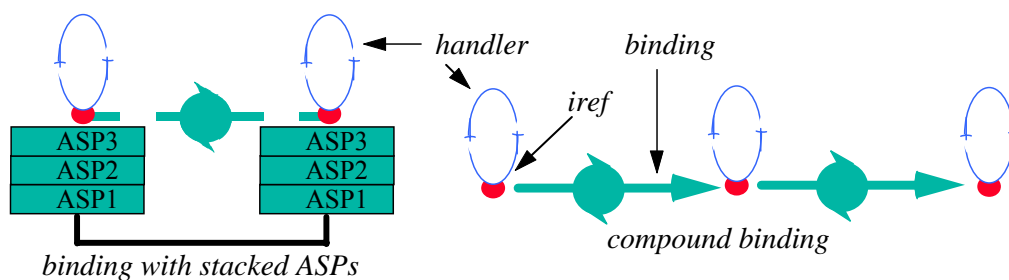


Figure 3: ASP Stacking and Compound Bindings

Unlike the protocol composition facilities provided by x-kernel [Hutchinson,91], ASP stacking is *static*: higher level ASPs explicitly name the ASPs on which they build. This is less flexible than the dynamic x-kernel scheme in which protocol modules can be composed on a per connection basis in ways unforeseen by the protocol implementers. A static scheme was chosen in GOPI because of the requirement for QoS mapping; adopting a dynamic scheme would imply the use of a fixed, generic set of QoS parameters because otherwise it would be impossible for a higher level ASP to know how to map QoS onto an unspecified lower level protocol. It is, in any case, possible to employ compound bindings (see below) if more dynamic protocol composition is required.

3.5.4.3 Compound Bindings

Compound bindings comprise more than two irefs chained together so that data flows along the chain from iref to iref (both stream and operational bindings can be compounded). As data flows along the binding, it can be manipulated and monitored at handlers attached to the constituent irefs in the normal way. As a simple example, a two stage compound binding could be used to implement a bridge or proxy across administrative domains or to insert a media filtering stage into a stream binding. In addition, dynamically composed protocol stacks can be implemented as compound bindings as mentioned above. It would also be possible to use compound bindings as the basis of a *component framework* implementation [Waddington,97] realised as a toolkit on top of GOPI.

Compound bindings build on facilities already described. In particular, *customer/provider irefs* fulfil the role of intermediate pipeline stages, *third party binding* is used to build compound bindings in an incremental manner and *local bindings* (implemented through the local binding ASP mentioned in section 3.4.4) enable the construction of multistage filtering pipelines with low overhead¹.

Compound bindings become really flexible and powerful when augmented with a *factory service*. Factory services allow a third party binder program to request the instantiation of an iref, together with an associated handler containing application specific code (e.g. media compression/ decompression code), on a remote site. Factory services can easily be layered on top of the services currently provided by GOPI. Given a factory service, a binder program can create a set of appropriate irefs/ handlers and combine them into an arbitrarily complex compound binding. Because it is easy to wrap up a binder program as an object at a higher level of abstraction, compound bindings with factories provide direct support for ODP binding objects (see section 2.1). An example of a simple binder program is shown in section 6.

4. Example ASPs

4.1 UDPFRAG

UDPFRAG is a simple ASP (\approx 350 lines of C code) that carries arbitrarily sized messages over UDP. It may lose messages but will not deliver corrupted or out of order messages. The QoS structure used by UDPFRAG is as follows:

```
typedef struct uqos {
    int utu;
    int reliability;
    int rate;
} udpfrag_qos;
```

The *utu* specification is inspired by the ALF concept. In this case, the *utu* specification is a hint to UDPFRAG that it should break large messages into *utu* sized fragments (e.g. if the transport was UDP over Ethernet, the *utu* would ideally be set to 1500 bytes). The *rate* specification is used for *in/ out* stream bindings to determine the period of the sending thread at the *out* end of the binding. The *reliability* specification (see also below) is used to determine the percentage of frames UDPFRAG may silently discard (there is no error recovery mechanism; if a binding exceeds its specified reliability threshold it simply upcalls the application and informs it of the fact).

¹ Local bindings are particularly useful for exploiting parallelism on multiprocessor machines as each intermediate iref in the pipeline has its own thread instantiated by the binding protocol.

When a binding is being established, the binding protocol passes `udpfrag_qos` structs (encapsulated as `CharSeqs`) to UDPFRAG's `listen()`, `connect()` and `accept()` entry points. UDPFRAG determines the resources required for the new binding (i.e. a TSAP and some buffers are required at each end) and attempts to allocate them. If the allocation is successful, the binding protocol will eventually report success.

In operation, the protocol takes a GOPI buffer at the send side, logically fragments it (if necessary) into `utu` sized packets and sends them via UDP to the receiver where they are built back into a GOPI buffer and delivered to the GOPI user. Fixed sized fragments are used which means that the last fragment of a multi-fragment message may contain some garbage. This disadvantage is traded off against the corresponding advantage that the receiver knows in advance the size of packet to receive and therefore only one `recv()` system call is required in the (common) case of single fragment messages (with variable sized fragments, one `recv()` call would be necessary to read a fragment size in the header and another to read the appropriate number of data bytes). Initially, the receiver allocates a buffer of size `utu` for the receipt of a new message. If the `utu` sized message turns out to be the first fragment in a multi-fragment message, the receiver allocates a new buffer (the size required is given in the header of the fragment) and copies the contents of the initial buffer into this. This, of course, incurs a penalty on multi-fragment messages, but it is assumed that the user has chosen the `utu` appropriately so that multi-fragment messages will be rare¹. With a wise choice of `utu` (it should be chosen to be large enough that fragmentation will not be required in the common case but not so large that there will be too much wastage per message) the common case efficiency should be high.

For reliability, only a minimal amount of work is done by UDPFRAG to avoid confusing the fragmentation code and thus passing out of order or corrupt messages to the layer above. If an out-of-order fragment (as signified by a sequence number in the packet header) is detected, the protocol discards the whole of the current message and waits for the first fragment of the following message. The `reliability` QoS parameter is used by UDPFRAG to determine how many dropped messages can be tolerated before the user is informed (see the example in section 6). This strategy is simple and efficient and arguably sufficient for applications such as video-over-UDP where reliability is not as important as speed. However, improvements could clearly be made by, for example, stashing out of order fragments that arrive early.

4.2 AUDIO

AUDIO [Parlavantzas,97] is a more complex and less conventional ASP than UDPFRAG. Its purpose is to carry voice audio packets over UDP (in fact it is layered on top of UDPFRAG) while attempting to minimise end-to-end delay, jitter and packet loss. The audio data is encapsulated in RTP [Schulzrinne,96] packets and RTP encoding identifiers are used. The protocol is based on the *elastic buffer* principle established in the `vat` Internet audio tool [Jacobson,92] and takes advantage of silences between talkspurts to calculate the optimal *playout offset* of samples in the forthcoming talkspurt. It also exchanges private intra-ASP protocol data units between the sender and receiver peers to implement flow control.

The QoS structure used by AUDIO is as follows (note that all sizes are in units of samples rather than bytes):

```
typedef struct audioqos {
```

¹ Of course other policies are possible in this area -- as they are in other areas. This is precisely why it is useful to allow applications to specify their own ASPs which are appropriate to their particular needs.

```

int samples_per_packet;      /* fragment size in samples */
int packetisation_interval; /* inter packet gap measured in ms */
int encoding_type;          /* RTP encoding type */
int playout_buffer_size;    /* size in samples */
int initial_delay;          /* no of samples to buffer initially */
int playout_delay_algorithm; /* select alternative algorithms */
bool transparent;
} audio_qos;

```

The first five parameters are used to configure the internal behaviour of the elastic buffer algorithm and will not be discussed further. The `playout_delay_algorithm` parameter is used to select from a number of alternative algorithms for determining the playout offset of audio samples. The final parameter, `transparent`, controls the way that audio data is acquired and delivered. If `transparent` is `FALSE`, audio data is acquired and delivered via handlers in the normal way; i.e. it is up to the user of the binding to produce and consume the audio packets. If, on the other hand, `transparent` is `TRUE`, the AUDIO ASP itself produces and consumes the audio packets by opening the local audio device inside its `connect()` and `accept()` routines and itself reading and writing to the local audio device inside its `send()` and `receive()` routines. In this latter case handlers are used only for synchronisation purposes to notify the user when samples are being sent/ delivered (the flow of data will not resume until the user lets the handler return so active synchronisation with other activities can be achieved). The `transparent` option is useful in simple bindings whereas the non transparent option can be used in compound bindings where pipeline stages perform pre/post processing of the audio stream. The `playout_delay_algorithm` and `transparent` parameters illustrate that the range of functions controllable by QoS parameters is very wide: any behaviour that requires agreement between peers can be controlled and configured as a QoS parameter.

Note that implementing an adaptive audio protocol as an ASP gives much greater flexibility to application writers than by using a standalone audio tool such as `vat`. For example, programmers can create multiple concurrent streams of audio and other media and perform synchronisation between them (through the use of handlers). They can also potentially accommodate audio in CORBA distributed programs in a highly integrated manner.

5. Control and Data Path Multiplexing

In keeping with the GOPI philosophy, each binding is non-multiplexed; i.e. it has its own operating system level TSAP and its own per-binding thread with which to produce/ consume data and execute its ASP code. This leads to the generic advantages of non multiplexing identified in section 2.2.

In the following, we describe the control/ data path on the receive side of an *out/ in* stream binding. Initially, GOPI runs a routine called `io_handler()` which uses `os_poll()`¹ to determine which TSAPs have data ready on them. The call to `io_handler()` can be initiated either *implicitly* if GOPI is configured for asynchronous IO notification (see section 3.4.3), or *explicitly* if GOPI is configured for polled IO. In the case of asynchronous IO notification, the deferred interrupt scheme described in section 3.2 is used, `io_handler()` being the callback routine. In the case of polled IO, the call to `io_handler()` is made from a low priority *sentinel thread* which runs only when no other threads in the system are runnable.

Having executed the `os_poll()` call, `io_handler()` performs a `msg_asend()` call to the per-binding thread associated with each ready TSAP. The appropriate thread (or more accurately the IPC channel associated with the thread) is determined by a simple mapping indexed through the TSAP identifier. As explained in section 3.3, the `msg_asend()` call does

¹ `os_poll()` maps, in UNIX, to the UNIX `poll()` system call.

not cause a thread context switch (this is crucial to avoid a fatal recursion when `io_handler()` is called from within the thread module; it also means that `io_handler()` executes very quickly). Eventually, at a time determined by the thread scheduler, each of these per-binding threads will return from the corresponding `msg_arecv()` call, thus receiving notification that data is available on their TSAP. The thread will then (indirectly) call the bind module's `receive()` call which in turn calls the appropriate ASP's `receive()` call through the protocol switch before upcalling the user's handler to deliver the data to the layer above GOPI.

There are three important properties of this scheme:

- as the operating system level `recv()` system call is initiated from a high level in the system it can be ensured that the buffer used is the final buffer to be passed to the application; thus no copy operation is required;
- the order in which the threads doing the `recv()` system call get to run is determined by the deadline of the thread as determined by the QoS of its associated binding; thus GOPI is able to avoid processing low urgency messages when high urgency messages are available;
- a corollary of the above is that we avoid the situation where one-shot control messages (e.g. video stop/ start) are indefinitely delayed by the steady flow of packets on one or more stream bindings; this is a common pathology of many current systems when used to support multimedia applications.

Note that the scheme uses a combination of upcall and downcall structuring. Downcall structuring is used to receive data from the protocol stack and upcall structuring to deliver data to the user's handler. The advantage of this arrangement is that the per-binding thread in the middle is the arbiter of all activity and can thus control and monitor the rate of data flow (as specified by the ASP's *period* parameter; see section 3.5). In addition, the arrangement simplifies the implementation structure of ASPs which can simply assume a downcall structure.

The send side of a binding operates in an analogous fashion to the receive side. When the IO event code determines that it is possible to send data on the binding's TSAP and when the scheduler determines that it is time to send that data (as indicated by the binding's *period* parameter), the per-binding thread upcalls the user's handler to obtain a buffer full of data. It then calls the ASP's `send()` routine which, in turn, executes the operating system level call to actually send the data.

The above describes the multiplexing structure of GOPI's stream bindings. Ideally, bindings would be entirely non multiplexed; however, as we have seen, it is necessary in practice to employ a single level of multiplexing on the control path (i.e. the IO availability notification path; not the data path) because operating system APIs typically employ an IO multiplexing call such as `poll()` which, for efficiency reasons, must be serviced by a single thread at the lowest GOPI level. We also adopt a partially multiplexed scheme for operations bindings (i.e. *outin/ inout* bindings). Here, there is a dedicated per-iref rather than per-binding thread at the *inout* end (there is no dedicated thread at the *outin* end of these bindings). The implementation is sufficiently flexible that it is possible to experiment with the multiplexing strategy quite easily. Other possible policies to investigate would be, at one extreme per-binding threads or, at the other extreme, a single *inout* thread (or pool of threads) per process.

6. Example Program

6.1 Program Structure

To illustrate the facilities available in GOPI and to give a flavour of the programming interface, we now describe a simple program¹ that uses the GOPI API to set up a stream binding between a customer program, *cust*, and a provider program, *prov*. The binding is established by a third party binder program called *binder*. The provider program reads buffers of size `SIZE` from an audio device and sends them to the customer program which, in turn, writes them to its audio device.

There is a fourth program used in this application that is not shown below; this is a name server program called *trader* to/ from which irefs and associated names can be exported and imported. Calls to interact with the trader are provided by the `bind` library although the trader is implemented as a separate program.

6.2 Provider Program

The provider program initialises GOPI (`gopi_init()`) and then binds to the trader. This involves creating a customer iref with flowtype *outin* and a null handler (`((Function)0)`). A library routine `bind_traderiref()` returns the (provider) iref of the trader. This contains a ‘hard wired’ address that acts as a bootstrapping mechanism. An ASP argument of `ANY_ASP` and a QoS argument `ANY_QOS` are used to specify that a default ASP/QoS can be used for the binding to the trader.

Having bound to the trader, the program creates an *out* provider stream iref which uses the `UDPFrag` ASP and a `CharSeq` QoS data structure that specifies a `period` of 10ms, a `utu` of 1500 and a `reliability` of 10; the `CharSeq` data structure is built by a utility function `udpfrag_buildqos()` which is provided by the `UDPFrag` ASP. The program also attaches a handler, `shandler`, to the iref. This will be periodically upcalled when GOPI requires a message to send on the binding when the iref is bound. The `event` argument to `shandler` lets the handler know why it was called (e.g. to notify that a binding has been requested, to obtain or deliver data, to report a QoS violation, etc.). In this case, the handler takes no specific action unless the event is of type `DATA`. The second handler argument specifies a pointer to a buffer into which the handler should place data to be sent on the binding. The two levels of indirection used allows the user to replace the given buffer with another in case it has previously filled another buffer. `shandler` simply reads a message from the audio device identified by the `fd` (file descriptor) variable into the GOPI buffer. The `msg_BUFDATA()` call is a macro that returns the address of the first byte of the data area of the buffer.

Having created its stream iref, the program exports this iref to the trader under the name “*prov*” before breaking its binding to the trader and calling `thread_exit()`. When `thread_exit()` is called the program as a whole does not terminate; rather it waits dormant for bind requests to arrive.

```
int shandler(int event, Buffer **b)
{
    if (event==DATA) read(fd, msg_BUFDATA(*b), SIZE);
    return OK;
}

main()
```

¹ The code in this section is slightly simplified for purposes of presentation.

```

{
    Iref *cust_trader, *prov_stream;
    CharSeq qos;

    gopi_init();
    bind_bindreq((cust_trader = bind_irefcustomer(OUTIN, (Function)0)),
                bind_traderiref(), ANY_ASP, ANY_QOS);
    udpfrag_buildqos(&qos, 10, 1500, 10);
    prov_stream = bind_irefprovider(OUT, shandler, UDPFRAG, &qos);
    bind_traderexport(cust_trader, "prov", prov_stream);
    bind_unbindreq(cust_trader);
    thread_exit();
}

```

6.3 Customer Program

The customer program is very similar to the provider. It binds to the trader and exports its customer iref. If the `event` parameter in the shandler routine has been given the value `QOS_NOTIFICATION` by the ASP, the handler simply prints an error message, otherwise it accepts a buffer of audio data and writes it to the audio device.

Note that there is no ASP or QoS associated with a customer iref.

```

int shandler(int event, Buffer **b)
{
    if (event == QOS_NOTIFICATION) {
        printf("shandler: lost some messages\n");
    } else if (event == DATA) {
        write(fd, msg_BUFDATA(*b), SIZE);
    }
    return OK;
}

main();
{
    Iref *cust_trader, *cust_stream;

    gopi_init();
    bind_bindreq((cust_trader = bind_irefcustomer(OUTIN, (Function)0)),
                bind_traderiref(), ANY_ASP, ANY_QOS);
    cust_stream = bind_irefcustomer(IN, shandler);
    bind_traderexport(cust_trader, "cust", cust_stream);
    bind_unbindreq(cust_trader);
    thread_exit();
}

```

6.4 Binder Program

The binder program first binds to the trader and then imports two iref names (as specified on the command line) and binds them together. In our case, of course, the specified names would be those of the customer and provider irefs previously exported by the provider and customer programs. The binder passes the two imported irefs to `bind_bindreq()`, together with `ANY_ASP/ ANY_QOS` arguments that results in the default ASP/ QoS contained in the provider iref being selected. `bind_bindreq()` arranges for a third party binding to be established between the irefs, following which data begins to flow on the stream. Having initiated the flow of data, the binder sets a timer that stops the flow and destroys the binding after 60 seconds have elapsed.

```

Iref cust_service;

```

```

int monitor()
{
    bind_ctl(cust_service, STOP);
    bind_unbindreq(cust_service);
}

main(argc, argv)
int argc;
char *argv[];
{
    Iref prov_service, *cust_trader;

    gopi_init();
    bind_bindreq((cust_trader = bind_irefcustomer(OUTIN, (Function)0)),
                bind_traderiref(), ANY_ASP, ANY_QOS);
    bind_traderimport(cust_trader, argv[1], &cust_service);
    bind_traderimport(cust_trader, argv[2], &prov_service);
    bind_unbindreq(cust_trader);
    bind_bindreq(&cust_service, &prov_service, UDPPFRAG, ANY_QOS);
    thread_timerset(1, monitor, 60000000);
    thread_exit();
}

```

7. Performance Evaluation

To empirically evaluate the performance potential of GOPI, we have carried out two experiments; one to measure performance relative to other comparable systems and the other to evaluate the scalability of the non-multiplexed communications architecture.

Packet size (bytes)	1	1024	8192
Socket (no poll())	1015	861	484
Socket (with poll())	999	839	442
GOPI (polled IO)	743	734	402
GOPI (IO notification)	664	522	350
GOPI vs Sockets overhead	25%	12%	9%
COOL-ORB	464	420	260

In the first experiment, we compared GOPI to a minimal ‘base line’ Berkeley sockets program and to a commercial ORB: COOL-ORB 4.1 from Chorus Systems¹. In fact, two Berkeley sockets programs were used; one with a null `poll()` system call inserted before each `read()` call and another without the `poll()`. In addition, two GOPI programs were used, one configured to use polled IO and the other using asynchronous IO notification (see section 3.4.3). The COOL-ORB program was modified so that no marshalling overhead was incurred so as to obtain a fair comparison with GOPI. All five test programs were configured as client/server pairs in which the client made repeated request/reply calls on the server. TCP was the underlying protocol for all the test programs, none of which touched the data sent/received in any way. Both the clients and servers were run on the same machine to minimise the effects of load fluctuation. The above table shows the number of round trip server invocations measured per second for each program (averaged over about 1,000,000 calls). Figures are given for three packet sizes, ‘small’, ‘medium’ and ‘large’. The ‘GOPI vs Sockets overhead’ results give the percentage overhead incurred by GOPI (with the polled IO configuration) as

¹ COOL-ORB was selected for comparison as it has been shown to be faster than the market leader in CORBA platforms, Iona’s Orbix [Blair,97].

against the minimal socket program (with `poll()`).

Clearly, the performance of GOPI compares favourably to that of COOL-ORB. It also, as would be expected, performs worse than the ‘base line’ sockets program although the overhead is probably not unacceptable when larger sized packets are used (particularly as large packets are the norm for multimedia data). In comparing GOPI with the sockets program we observe that GOPI carries the following overheads: thread context switching and message passing, buffer (de)allocation, protocol layer execution and header processing, and multiplexing/ demultiplexing to/from irefs. It should also be noted that no serious attempt has been made to optimise GOPI.

Polled IO adds overhead to the basic cost of `send()/recv()` calls and it is reasonable to use the socket program with `poll()` rather than the simplest and fastest socket program as a baseline reference for performance comparisons. This is because all user level multithreaded systems are inevitably required to use IO polling of some kind. The key implementation issue in minimising `poll()` overhead in GOPI is the selection of an appropriate trade off between the inefficiency resulting from frequent `poll()` calls and the higher IO latency resulting from less frequent polls. The solution adopted in GOPI is to check for IO availability only when no application thread is runnable (see section 5). This implicitly embodies the policy that no new messages should be taken from the operating system until current messages have been dealt with. In many cases this is an appropriate policy but it can lead to degraded IO responsiveness in applications with many CPU bound threads. One way to avoid the problems associated with polled IO is to use kernel threads rather than user level threads which can simply block in the operating system kernel (e.g. we could use multiple VPs in the threads module). This, however, introduces further significant overheads because of the need for operating system level context switches and synchronisation.

Another alternative to polled IO is asynchronous IO notification (see section 3.4.3). When GOPI is configured for asynchronous IO notification it is necessary to perform a `poll()` system call after each SIGPOLL to find out which of the currently open sockets are ready. This incurs, like the polled IO approach, two system calls per IO operation although the *responsiveness* will be better than polled IO in many cases. On top of this, however, some operating systems (e.g. SunOS) require *yet another* `poll()` call following each `recv()`. This is because SIGPOLL only informs of a change in state from *no data available* to *data available*, and another packet may have arrived while the previous one was being read. The need for two system calls per IO could be eliminated if the SIGPOLL signal itself carried the information returned by `poll()`. Furthermore, the need for the second `poll()` could be eliminated if a SIGPOLL was triggered on each and every packet arrival (cf. the semantics of the BSD UNIX SIGIO)¹. This semantic, however, is problematic where operating system level fragmentation is involved; e.g. TCP segments do not necessarily map directly onto GOPI messages. Although the test results show that the version of GOPI configured with IO notification performed relatively poorly, IO notification may still be useful in situations where busy processes need to respond instantly to occasional incoming control messages. It would be possible to adapt GOPI to use IO notification on a selectable, per TSAP, basis (presumably selected by a QoS parameter defined by ASPs for which rapid response to events was an issue) to take advantage of the benefits of IO notification without paying the overall performance penalty.

¹ GOPI uses a workaround in which there is no additional `poll()` following a `recv()`. Instead, it falls back on the polled IO strategy described above to detect the presence of any additional data not gathered by the `recv()` following a SIGPOLL/`poll()` combination. The drawback of this approach is, again, the potentially decreased IO responsiveness incurred.

The second experiment was intended to evaluate the scalability of GOPI's non-multiplexed structure. A GOPI customer/ provider program was written which established varying numbers of bindings to transmit a given total number of packets. The customer and provider parts of the program were again run on the same machine. Packet sizes of 1024 bytes were used over a stream binding using the SHMEM ASP (SHMEM was used to avoid either packet loss, possible with UDPFRAG, or slow start/ buffering effects, possible with TCPASP and FIFOASP). The binding was run as fast as the system would allow. The time to receive 160,000 packets was measured at the receiver (multiple averaged runs were performed to obtain the given figures).

Number of bindings	1	2	4	8
SHMEM	15.99 secs	8.55 secs	7.97 secs	7.52 secs

The results show that for more than one binding an *increase* in performance was observed over the single binding case. This is probably because fewer than one `poll()` call per receive is incurred when there are multiple receiving bindings in place (i.e. `poll()` reports more than one TSAP with data waiting). There was then little appreciable difference in overall throughput for $2 \leq n \leq 8$ bindings (i.e. configurations with n threads and n FIFOs, used for event notification in SHMEM, at each end). This result demonstrates that GOPI's improved QoS predictability (achieved through non multiplexing) is not bought at the expense of an unacceptable degradation in performance.

8. Related Work

Streams were first adopted in standards by the Interactive Multimedia Association in its Multimedia Systems Service (MSS) specification [IMA,94]. The MSS approach is based on a model in which the objects producing the streams are 'special' and inaccessible to applications; application can *control* but not directly *access* the real-time media. This is fundamentally different from the more integrated RM-ODP/ GOPI view of multimedia streams in which applications treat streams in as similar a manner as possible to conventional operational interactions. The IMA MSS is currently being adopted by ISO in its PREMO standard [ISO,96].

Recent work in the OMG's CORBA forum has addressed the need for streams and real-time services in distributed object platforms. In particular, a Request for Proposals (RFP) for the Control and Management of Audio/ Visual Streams by the OMG Telecommunication Special Interest Group [OMG,98a] has recently closed, resulting in proposals from a number of interested parties. However, this RFP is concerned only with the control and management of streams and not with the implementation of the streams themselves. Separately from the above mentioned RFP, an RFP from the OMG's Real-time Special Interest Group is also underway [OMG,98b]. Unfortunately, however, it is difficult to make a detailed comparison between these proposals and GOPI as there is as yet little detail in the OMG contributions.

The European Commission funded ReTINA project is designing a distributed object platform based on CORBA but with streams and QoS extensions [Dang-Tran,96]. The architecture is based on a clean separation between *i)* ORB support mechanisms such as interface reference management, threads, buffers etc., and *ii)* *binding classes* which provide communications services tailored to particular applications. Binding objects (i.e. instances of binding classes) are configured into applications though a *generic binding protocol*. The generic binding protocol is the centrepiece of the architecture and all bindable objects must support a minimal set of upcall operations to enable them to be bound by binding objects

using the binding protocol. The binding architecture is recursive in that new ‘value added’ binding objects can be built out of other binding objects. Compared to ReTINA, the GOPI work focuses more on the ORB support mechanism area than the binding architecture. The GOPI binding protocol is compatible in concept with the ReTINA protocol but support for binding classes would be provided at a higher level than the currently implemented GOPI services.

The DIMMA project [Li,97] at APM Ltd., Cambridge, UK is also addressing issues of real-time/ multimedia on distributed object platforms. DIMMA is based on the ANSAware distributed systems platform which has been enhanced with a modular protocol stack and a flexible multiplexing structure; like GOPI, the degree of per-binding internal multiplexing can be configured according to the needs of different protocols. DIMMA emphasises multimedia and QoS to a lesser extent than GOPI. There is a timed RPC protocol but no stream communications. In addition, there is no end-to-end binding protocol over which QoS information can be exchanged.

The TAO object request broker [Schmidt,97b] is a CORBA 2.0 compliant implementation that runs on real-time OS platforms. The system is primarily designed for hard real-time applications (one target domain is avionics for example) although multimedia support is considered. TAO features a real-time message scheduling service that can provide deterministic temporal guarantees (given a real-time OS infrastructure) by avoiding priority inversion and non-determinism. A real-time version of the CORBA event service has also been built. In contrast to TAO, GOPI is designed to provide soft real-time/ multimedia support in a conventional OS/ network environment and emphasises flexible user services support rather than hard deterministic performance.

Finally, the *Regis* distributed systems platform [Crane,96], developed at Imperial College, London, incorporates the notions of *in* and *out* interaction points and third party bindings. It also has a flexible protocol architecture in which it is possible to add/ remove application specific protocol modules at run time. This scheme, which is based on an x-kernel like graph of protocol entities, is also adopted in DIMMA and in the Horus platform developed at Cornell University [vanRenesse,96]. Although *Regis* has a more flexible application specific protocol architecture than GOPI, it has no QoS support (e.g. QoS oriented binding protocol) and, unlike GOPI, is not structured in a non-multiplexed manner which, as explained in section 2.2, is highly appropriate for QoS support.

9. Conclusions and Future Work

We have described a platform intended to support the implementation of multimedia/ QoS capable distributed object platforms. GOPI is specifically designed to support the model of distributed multimedia object systems described in the RM-ODP. It is designed in a modular fashion and permits the exploration of a number of interesting ideas current in the communications and operating systems research communities. The small size of the system (about 12,000 lines of reasonably well commented C) is evidence that multimedia capable distributed object platforms need not be large and unwieldy. Furthermore, the figures in section 7 demonstrate that such platforms can adequately meet the performance requirements of multimedia applications¹.

It can be seen that much of the design of GOPI is directly influenced by the work described in section 2. First, GOPI provides support for an RM-ODP like computational

¹ Audio and video stream bindings will not usually require marshalling so no performance penalty over and above that detailed in section 7 should be incurred.

model in that it supports stream bindings and compound bindings (cf. binding objects) between explicit interfaces. Furthermore, ODP signals are closely related to the invocation of handlers: a signal abstraction with associated QoS specification could readily be implemented on top of GOPI as could an operations based computational model such as CORBA.

In terms of engineering principles, the work on application specific protocols and application layer framing is reflected in the GOPI ASP architecture which provides a 'programmer friendly' environment for the implementation of new application specific protocols. GOPI is also influenced by the integrated layer processing approach in that a single thread is used to sequentially execute both the protocol stack and the application code that runs in the context of a handler. In addition, GOPI employs a hybrid upcall/ downcall control structure for protocol processing which simplifies ASP implementation while retaining the advantages of upcall structuring.

The principle of soft real-time support is not directly supported in GOPI except by virtue of it being implemented in a general purpose operating system environment. To better support this principle, more sophisticated admission testing would need to be added to the thread and buffer management modules. There is also an important role for ASPs in this area; see section 3.5.3. The timed semaphores, timed IPC primitives and EDF based scheduling are of some assistance in supporting soft real-time applications. Finally, the need for general efficiency is addressed through many GOPI mechanisms; for example, the zero copy architecture, the use of cheap user level threads and the efficient buffer allocation/ deallocation scheme.

GOPI offers a unique approach to QoS specification, QoS mapping and resource management. Rather than attempting to second guess the QoS requirements of all possible applications, GOPI's ASP framework lets applications define their own specific framework for QoS management. In this way applications/ ASPs can create arbitrary QoS specification schemas together with application specific mappings of these specifications onto the low level resources understood by GOPI (i.e. threads, buffers and TSAPs). At the same time, as much as possible of the generic functionality of QoS provision is provided by GOPI -- for example, the binding protocol and the provision of low level, policy free, resource managers for threads, buffers and TSAPs.

Regarding future work on GOPI, there are many more possibilities than we have the time and resources to investigate. Extensions we have planned for the near future include *i*) adding a CORBA object layer with C++ and a version of IDL extended to support *in* and *out* signals, and *ii*) enhancing CORBA interoperability through bridging and the use of CORBA Interoperable Object References (IORs), *iii*) investigating the integration of GOPI with operating system supported multimedia support mechanisms (e.g. full zero copy architecture, integrated operating system/ user level scheduling, better signal notification (see section 7)), and *iv*) providing multicast support underpinned by multicast IP.

Finally, we have longer term plans to restructure the internals of GOPI to further increase its modularity and flexibility. The approach we have in mind is to apply the spirit of the ASP architecture to other system modules such as binding protocols, transports and scheduling policies so that it is easy for programmers to add new instances of such modules to the system which are better tailored for their particular application. For example, using the proposed modular structure it would be possible to replace the current binding protocol by an alternative network QoS aware binding protocol using RSVP [Zhang,93], or a protocol for multi-party binding, or a ReTINA like binding protocol. We also intend to exploit the concept of *reflection* [Maes,87] in this context to provide a framework for controlling and managing the flexibility gained by adding such dynamic modularisation.

Acknowledgement

The work described in the paper was carried out in the context of the BT Labs funded Management of Multiservice Networks project at Lancaster University, UK. We would like to thank BT Labs for their generous support.

References

- [**Blair,97**] Blair, G.S. and J.B. Stefani, "Open Distributed Processing and Multimedia", Addison Wesley, ISBN 0201177943, pp 387-388, 1997.
- [**Campbell,93**] Campbell, A., Coulson, G., García, F., Hutchison, D., and H. Leopold, "Integrated Quality of Service for Multimedia Communications", *Proc. IEEE Infocom'93*, Hotel Nikko, San Francisco, CA, March 1993.
- [**Clarke,85**] Clark, D.D., "The Structuring of Systems Using Upcalls", *Proc. Tenth ACM Symposium on Operating Systems Principles*, pp 171-180, December 1985.
- [**Clarke,90**] Clarke, D.D. and Tennenhouse, D.L., Architectural Considerations for a New Generation of Protocols, *Proc. ACM SIGCOMM 1990*, Philadelphia, USA, also in *Computer Communications Review*, Vol 24, No 4, September 1990.
- [**Coulson,95**] Coulson, G., Campbell, A and P. Robin, "Design of a QoS Controlled ATM Based Communication System in Chorus", *IEEE Journal of Selected Areas in Communications (JSAC)*, Special Issue on ATM LANs: Implementation and Experiences with Emerging Technology, 1995.
- [**Crane,96**] Crane, S., Dulay, N., "A Configurable Protocol Architecture for CORBA Environments", Internal Report, Dept. of Computing, Imperial College of Science, Technology and Medicine, 180 Queen's Gate, London SW7 2BZ, UK, 1996.
- [**Dang-Tran,96**] Dang Tran, F., Perebaskine, V., Stefani, J.B., Crawford, B., Kramer, A and Otway, D., "Binding and Streams: the ReTINA Approach", *Proc. TINA '96*, ?? 1996.
- [**Druschel,96**] Druschel, P., "Operating Systems Support for High Speed Communication", *CACM*, Vol 39, No 9, September 1996.
- [**Govindan,91**] Govindan, R., and D.P. Anderson, "Scheduling and IPC Mechanisms for Continuous Media", *Proc 13th ACM Symposium on Operating Systems Principles*, Asilomar Conference Center, Pacific Grove, California, USA, SIGOPS, Vol 25, pp 68-80, 1991.
- [**Hutchinson,91**] Hutchinson, N. and L. Peterson, "The x-kernel: An architecture for Implementing Network Protocols", *IEEE Transactions on Software Engineering*, Vol 17 No 1, pp 64-76, January 1991.
- [**IMA,94**] Interactive Multimedia Association: Multimedia Systems Services, Parts I and II, IMA Recommended Practice, Second Draft, September 1994.
- [**ITU-T,95a**] UIT-T, ISO/IEC Recommendation X.902, International Standard 10746-2, "ODP Reference Model: Descriptive Model", January 1995.
- [**ISO,96**] International Standards Organisation, Information Processing Systems - Computer Graphics and Image Processing - Presentation Environments for Multimedia Objects (PREMO), Part 3: Multimedia System Services, ISO/IEC Standards Committee ISO/IEC JTC1/SC24 draft 1996-05-15 May 1996.
- [**Jacobson,92**] Jacobson, V., McCanne, S., The LBL audio tool vat, manual page, available from <ftp://ftp.ee.lbl.gov/conferencing/vat>, July 1992.

[**Knuth,73**] Knuth, D.E., "The Art of Computer Programming, Volume 1: Fundamental Algorithms", Second Edition, Reading, Massachusetts, USA, Addison Wesley, 1973.

[**Li,97**] Li, G., "DIMMA Nucleus Design", Technical Report, APM Ltd., Poseidon House, Castle Park, Cambridge CB3 0RD, UK, January 1997.

[**Liu,73**] Liu, C.L. and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment", *Journal of the Association for Computing Machinery*, Vol 20, No 1, pp 46-61, February 1973.

[**Maes,87**] Maes, P., "Concepts and Experiments in Computational Reflection", In Proceedings of OOPSLA'87, Vol 22 of ACM SIGPLAN Notices, pp 147-155, ACM Press, 1987.

[**OMG,96**] Corba 2.0 Specification, Object Management Group Technical Document PTC/96-03-04, available at <http://www.omg.org/>

[**OMG,98a**] Telecom A/V RFP
http://www.omg.org/library/schedule/AV_Streams_RTF.htm

[**OMG,98b**] Realtime RFP
http://www.omg.org/library/schedule/Realtime_CORBA_1.0_RFP.htm
(<ftp://ftp.omg.org/pub/docs/orbos/>)

[**Parlavantzas,97**] Palavantzas, N., "An Adaptive Audio Protocol over the Internet", MSc thesis, Lancaster University, Lancaster, UK, September 1997.

[**Schulzrinne,96**] Schulzrinne, H., Casner, S., Frederick, R and Jacobson, V., "RTP: A Transport Protocol for Real-Time Applications", IETF Audio/ Video Transport Working Group, RFC 1889, January 1996.

[**Schmidt,97**] Schmid, D.C., "Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA", Proc. GLOBECOM '97, Phoenix, AZ, November, 1997.

[**Schmidt,97b**] Schmid, D.C., "The Design of the TAO Real-Time Object Request Broker", <http://www.cs.wustl.edu/~schmidt/new.html#corba>.

[**Tennenhouse,90**] Tennenhouse D.L., "Layered Multiplexing Considered Harmful, Protocols for High-Speed Networks", Elsevier Science Publishers B.V. (North-Holland), 1990.

[**Thekkath,93**] Thekkath, C.A., T.D. Nguyen, E. Moy, and E. Lazowska, "Implementing Network Protocols at User Level." *IEEE Transactions on Networking*, Vol 5, No 1, pp 554-565, October, 1993. □ □

[**vanRenesse,96**] van Renesse, R., "Masking the Overhead of Protocol Layering", Proc. 1996 ACM SIGCOMM Conference, Stanford, September 1996.

[**Williams,92**] Williams, N., Blair, G.S., and N. Davies, "Multimedia Computing: An Assessment of the State of the Art", *Journal of Information Services and Use*, October 1992.

[**Zhang,93**] Zhang, L., Deering, S., Estrin, D., Shenker, S. and D. Zappala, "RSVP: A New Resource ReSerVation Protocol", *IEEE Network*, September 1993.