

A Configurable Multimedia Middleware Platform

Geoff Coulson

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

e-mail: geoff@comp.lancs.ac.uk

ABSTRACT

Experience has indicated that it is both beneficial and feasible to support soft real-time/ multimedia applications in distributed middleware architectures such as the Object Management Group's CORBA. However, the deployment of multimedia capable middleware platforms is not yet occurring on a large scale. This is due, in part, to a lack of experience in engineering such platforms in a performant, predictable and configurable manner as required by multimedia applications. This paper describes the design of such a platform and describes in detail the ways in which the platform attempts to maximise performance, predictability and configurability in a standard workstation operating system environment. Comparative performance measurements are given which demonstrate that, despite its configurability, the platform's performance is clearly superior to that of standard commercial ORBs.

1. INTRODUCTION

Experience has indicated that it is both beneficial and feasible to support soft real-time/ multimedia applications in distributed middleware architectures such as the Object Management Group's CORBA [OMG,98]. The potential *benefits* of such support to application developers are:

- i) the provision of high level programming abstractions for multimedia communications, including quality of service (QoS) specification and management, and
- ii) the seamless integration of multimedia with conventional distributed interactions.

In addition, the *feasibility* of such support has been demonstrated through the implementation of a number of CORBA based, multimedia capable, experimental middleware platforms by the research community (e.g. [Coulson,98], [Dang-Tran,96], [Donaldson,98], [Schmidt,97]).

Notwithstanding the above, however, the large scale deployment of such platforms seems unlikely to happen in the short term. One major obstacle to deployment is lack of standardisation of appropriate programming abstractions. Although guidelines for this are available in the ISO's Reference Model for Open Distributed Processing, the issue has not

yet been seriously addressed by the OMG¹. The other major obstacle lies in the *engineering* of soft real-time/ multimedia middleware platforms. It is still not well understood how best to structure middleware to achieve optimal QoS management, particularly in terms of *performance*, *predictability* (in terms of timeliness) and *configurability* as required by soft real-time/ multimedia applications [Blair,97].

It is useful to distinguish *static* and *dynamic* aspects of QoS management [Coulson,97a]. Static aspects involve QoS specification, mapping, negotiation and resource allocation at connection setup time. Dynamic aspects are then concerned with managing allocated resources at data transfer time to ensure that required levels of QoS are continuously maintained. In this paper, we focus primarily on dynamic QoS management issues². In particular, we describe low level concurrency and communications mechanisms designed to maximise performance and predictability in multimedia middleware. We describe in detail how these mechanisms are implemented on top of standard workstation operating system (OS) services and how they can be flexibly configured to best maintain QoS over a wide range of operating conditions. The mechanisms described are implemented in a multimedia middleware platform called GOPI (Generic Object Platform Infrastructure). GOPI attempts to deliver (as best it can) performance and predictability in a configurable manner in a standard OS environment. It is also designed to be backwardly compatible with CORBA.

The remainder of the paper is structured as follows. Sections 2 and 3 establish some essential background by respectively overviewing i) currently available OS support for soft real-time/ multimedia middleware and ii) the gross structure of the GOPI middleware platform. Following this, section 4 describes the GOPI approach to concurrency management and section 5 explores GOPI's communications architecture. Section 6 then evaluates the described mechanisms in terms of performance and section 7 suggests OS enhancements for improved support of soft real-time/ multimedia middleware. Finally, section 7 surveys related work and section 8 offers some concluding remarks.

2. SYSTEM SUPPORT FOR MULTIMEDIA MIDDLEWARE

Modern network-capable OSs provide a system call interface rich in functionality but difficult and error prone to use. The role of multimedia middleware is therefore to build higher level abstractions on top of the OS level API to ease the task of the application developer. The key challenge is to provide useful abstractions without compromising performance and predictability. This requires careful use of the OS level facilities (e.g. excessive use of system calls is expensive in terms of performance because system calls usually end in a context switch, and this in turn impacts predictability due to cache disruption). The ways in which GOPI addresses this challenge are discussed in sections 4 and 5. In this section, we prepare the ground for this discussion by briefly reviewing the OS level facilities typically found in modern OSs.

The most important system services from a middleware implementation perspective can be placed under the headings of *concurrency*, *communications* and *event* (or signal) *handling*.

¹ The OMG's response to demands for multimedia support in CORBA (mainly from the Telecommunications community) has been the Telecom SIG's "Control and Management of Audio/Video Streams" RFP [OMG,98a]. In our view, however, this represents a short term solution to the problem because it does not treat continuous media as first class data types [Blair97a]. In the long run, we believe, a more integrated approach to multimedia support will be required.

² Note that GOPI, the system described in this paper, comprehensively supports both static and dynamic QoS management. Static aspects are not, however, emphasised in this paper. See [Coulson,98] for more details of these aspects.

In addition, calls to manage memory resources and optimised local IPC services such as shared memory or FIFOs are heavily exploited by multimedia middleware.

The main concurrency related services are those associated with *kernel threads* (i.e. threads supported natively by the OS). These include calls to create kernel threads, synchronisation calls on mutexes and semaphores, and thread safe libraries (i.e. libraries whose non re-entrant routines are implicitly protected with mutexes). In addition, primitive facilities for the support of *user level threads* (i.e. threads supported in a user level library outside the OS) are often provided. In SunOS 5.5, for example, the `makecontext()` and `swapcontext()` calls allow a user level thread library to manage the CPU state of user level threads. Machine instructions such as test-and-set or compare-and-swap are also useful as a building block for synchronisation primitives in user level thread packages.

In communications, the *socket* abstraction is central. All modern OSs have calls to create network sockets, to bind them to addresses and to write and read data on them in either a blocking or non blocking fashion. In addition, a *scatter/ gather* IO facility is often provided through which structured data (e.g. packets with headers) residing in non contiguous memory locations can be written/ read to/ from a socket in a single call. Finally, an IO multiplexing call (e.g. `poll()` or `select()`) is usually provided to determine which of a number of sockets are ready for IO, or to block (with an optional timeout) until one or more sockets are ready.

Event handling services are used to inform user software of asynchronous events in the OS; e.g. alarm expiry, periodic timing signals or IO availability on sockets (i.e. SIGALRM, SIGVTIMER, SIGPOLL in the UNIX environment). Event masking calls are also provided so that user level critical sections can be protected from interruption by asynchronous events. Although the semantics of events are notoriously obtuse, they are still a potentially useful service for multimedia middleware implementations as they allow the middleware to learn of events in a (relatively) timely fashion without the overhead of polling with system calls.

3. THE STRUCTURE OF GOPI

GOPI is structured at the coarsest level of granularity into two sub-systems, both of which are realised as run-time libraries linked with applications. The two sub-systems are i) the application programmer's interface (API) personality, and ii) the GOPI core. The API personality presents GOPI core services to the application in terms of some standard interface such as CORBA (with appropriate extensions for the soft real-time/ multimedia functionality). As the API personality does not significantly impact the issues of performance and predictability addressed in this paper, we do not consider it further; see [Coulson,98] for more details. The core services themselves present the following low level API³:

- calls to manage location independent communication endpoints called *irefs*, and to bind these (using first or third party binding), according to a specified QoS, using a stack of user selected *application specific protocols* (ASPs),
- calls to send and receive data, in both request/reply and streaming mode, on bound irefs; the API optionally uses *upcalls* of application defined *handlers* to obtain and deliver data in addition to the traditional downcall based style of call,
- calls to create and manage threads in association with *application scheduler contexts* (ASCs); calls to create and manage semaphores and timers; calls to create and

³ Aspects of this API are discussed throughout the remainder of this paper. Definitive documentation of the API can be found at: <http://www.comp.lancs.ac.uk/computing/users/geoff/GOPI/index.html>.

manage buffers and communicate them between threads in the same address space via channels (*chans*).

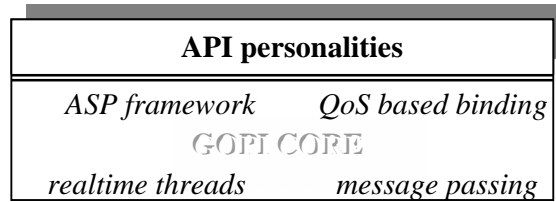


Figure 1: The structure of GOPI

In terms of modular structure (see figure 1), the GOPI core is implemented as a set of independent modules: *base*, a collection of generally useful foundation programming classes; *thread*, a ‘real-time’ concurrency package; *msg*, a ‘real-time’ inter-thread message passing and buffer package; *comm*, an architecture for accommodating ASPs; and *bind*, a module supporting irefs, plus a binding, or connection management, protocol with QoS negotiation capabilities. Each module is realised as a separate library with minimal dependencies. For example, the thread module can be used as a standalone package, and it is possible to replace the bind module without impacting the comm module.

GOPI is written mainly in C (the core consists of approximately 12,000 lines of code) and runs on a variety of UNIX platforms. A detailed description of the system including the API and the functionality of the various modules is available in [Coulson,97b] and [Coulson,98].

4. CONCURRENCY

4.1 API Overview

Threads are created using the following call provided by the core GOPI API:

```
int thread_create(Func f, int arg, ASC scheduler, CharSeq *params);
```

The *f* argument specifies a function to be executed by the newly created thread and the *arg* argument specifies a parameter to be passed to this function. The remaining arguments allow the caller to select an ASC for the thread and to assign per-ASC scheduling parameters (e.g. priority, deadline and period). The scheduling parameters are pre-specified by calling an ASC specific routine which marshalls ASC specific parameters, *p*₁, *p*₂, ..., *p*_{*n*}, into a CharSeq (character sequence) data structure:

```
void <asname>_buildparams(CharSeq *params, <p1>, <p2>, ..., <pn>);
```

There is also a call to modify the ASC and scheduling parameters of the current thread (see section 4.3):

```
bool thread_change(ASC newscheduler, CharSeq *newparams);
```

For synchronisation, the threads module provides semaphores and associated *thread_P()* and *thread_V()* operations. A timeout variant of the *thread_P()* operation (similar to the Posix *cond_timedwait()* service) is provided to assist in controlling temporal behaviour:

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

This returns FALSE if the return was due to a timeout and TRUE if the return was the consequence of a *thread_V()* operation invoked by another thread. There is also a call,

`thread_yield()`, which is used to explicitly yield the CPU and cause a context switch. The `thread_exit()` call is used to destroy the calling thread.

The thread module additionally supports an arbitrary number of per-thread timers. The call to set a timer is as follows:

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

This call results in the function `timerfunc` being invoked, in the context of the calling thread, after `us` microseconds have elapsed. The identifier of the timer is returned as a reference to be passed to related calls which stop the timer or read its elapsed time.

4.2 Threads and VPs

Internally, GOPI uses a two level concurrency architecture in which user level threads are multiplexed on top of kernel threads. Kernel threads are called *virtual processors* or VPs in the GOPI context. VPs spend their lives in an endless loop; they repeatedly take a user level thread context from an ASC and execute it until it either yields or is preempted (see below). The number of VPs per *capsule* (i.e. *process* in UNIX terms) potentially affects performance and predictability (again, see below) but does not impact the degree of user thread concurrency available to either GOPI or its applications. In particular, a capsule with a single VP can support an arbitrary number of user level threads.

The motivation for supporting user level threads is that i) context switches are cheap so high levels of concurrency can be used and ii) the choice of scheduling policy is not restricted to whatever is provided by the OS. The motivation for additionally using VPs is that i) intra process parallelism is possible on multiprocessors and ii) more flexible scheduling possibilities are opened up (see section 4.3).

The thread module (actually, each ASC; see section 4.3) can be configured according to one of the following options:

- *no preemption* in which the only way that a thread yields to another thread is either explicitly (using `thread_yield()`) or when the thread blocks (e.g. by calling `thread_P()` on a semaphore);
- *preemption* in which threads can be preempted by asynchronous *events* (*signals* in UNIX parlance) which are delivered to the thread module by the OS (examples of events are IO availability and alarm expiry; see sections 4.5 and 5.3 respectively);

The configuration can be altered dynamically at run-time. There is also an option to *timeslice* threads. This is simply built on top of the *preemption* option using a periodic alarm event which calls `thread_yield()` on each invocation.

4.3 Application Scheduler Contexts (ASCs)

A significant benefit of a user level thread implementation is that it allows considerably flexibility in terms of scheduling; a wide range of scheduling policies can easily be implemented (e.g. priority based, rate monotonic, earliest deadline first, least laxity first etc. [Stankovic,95]). The GOPI thread module provides a framework called *application scheduler contexts* (ASCs) for the purpose of implementing scheduling policies.

Each ASC supports the following interface:

```
int          asc_init(void);
bool        asc_admit(Thread t, CharSeq *params);
Thread      *asc_expel(int thread_id);
void        asc_deschedule(Thread t);
Thread      *asc_schedule(void);
```

Note that these ASC operations are called exclusively by code internal to the thread module; they are not directly accessible to the user. The user's role is limited to i) creating and installing new ASCs, and ii) creating threads that are managed by installed ASCs. ASCs can be installed dynamically at any time and multiple instances of the same type of ASC can be in operation simultaneously.

Each ASC instance encapsulates i) a scheduling policy, ii) a run queue and iii) one or more VPs. The run queue and VP resources are created by `asc_init()` which is called at ASC installation time. VPs are created with either the *preemption*, *no preemption* or *timeslicing* options as described above. The `asc_admit()` call is used to admission test a thread creation request and, if the admission test is successful, to register the given thread context with the ASC (which will place the thread on its run queue). The `params` argument passed to `asc_admit()` is the same `params` argument that was originally passed by the user to `thread_create()`. This is completely opaque to the thread module and is only interpretable by the ASC. The `asc_expel()` call is used to remove a thread from an ASC; it is used by the `thread_exit()` and `thread_change()` routines as described below.

`asc_schedule()` and `asc_deschedule()` are called by VPs executing the thread module's context switch code (inside `thread_yield()`). This code determines the 'home' ASC of the calling VP (which is found through a mapping from the VP's unique identifier) and calls `asc_deschedule()` on the appropriate ASC, passing the user thread context of the currently executing thread as an argument. Internally, `asc_deschedule()` updates any dynamic scheduling state associated with the descheduled thread (e.g. deadlines) and stores the descriptor on its private run queue. The `thread_yield()` code then calls `asc_schedule()` which returns the user thread context that the ASC's policy has determined should be the next thread to run. Finally, `thread_yield()` switches context to this newly selected thread.

The `thread_change()` routine (see section 4.1 above) attempts to migrate the current thread into a newly specified ASC by first removing it from its current ASC using `asc_expel()` and then calling `asc_admit()` on the new ASC using the scheduling parameters provided. `thread_change()` succeeds or fails depending on the outcome of this call. The caller of `thread_change()` may also specify the same ASC but with new scheduling parameters.

4.4 Locking Issues

In any concurrent system, the implementation of locking is crucial in maximising performance and predictability. Performance is maximised by minimising the inherent overhead of the locking operations and by appropriately trading of the number and length of critical sections. Similarly, both performance and predictability are maximised by minimising the number of context switches incurred. In GOPI, the degree and type of locking, and thus the overhead involved, are configurable depending on i) whether or not any currently installed ASC has the *preemption* option enabled and ii) whether just one or multiple VPs are currently in existence. GOPI implements three levels of locking as follows.

The first level of locking is to ensure that `libc` and other library calls are *atomic* in the face of preemption. Therefore this level of locking is not required unless some ASC has configured the *preemption* option (in the rest of this paragraph the *preemption* option is assumed). For a configuration with more than one VP, it is further necessary that appropriate multi-thread safe (i.e. 'MT-LEVEL Safe' in SunOS 5.5 parlance) versions of the standard libraries are used; these include their own internal, VP level, locking based on OS level mutexes. For a configuration with only a single ASC/VP, however, a more efficient solution suffices. In such a situation, concurrency problems can only arise if the thread module

preempts a thread while it is executing inside a standard library routine. All that is required to prevent such problems in the single VP case is a simple boolean flag; a thread making a library call simply sets this flag on entry and clears it on exit⁴. Asynchronous event handlers can then decide whether or not to evoke a preemption (by calling `thread_yield()`) on the basis of the flag's value (see also section 4.5). Note that in this solution it is not necessary to use OS level signal masking calls. This is an important benefit as these calls carry a significant overhead in frequently executed sections of code.

The second level of locking in the thread module is concerned with protecting the data structures of the thread module and ASCs (e.g. run queues and semaphore queues) in the face of preemption. With a configuration of more than one VP, an OS level mutex is required to protect these data structures from OS level preemptions. Again, however, the far more efficient flag based solution described above suffices in the special case of a single VP configuration.

The third level of locking is concerned with protecting higher level GOPI modules and applications. At this level, of course, the thread module's semaphore implementation is available to implement locking. Overhead here is minimised in that the user level semaphore implementation requires only very few instructions. In addition, each GOPI module attempts to maximise concurrency through the use of multiple, per resource, critical sections as opposed to single, per module, locks. Performance, as well as predictability, is further enhanced in that `thread_P()` does not cause a context switch unless it blocks, and `thread_V()` does not cause a context switch unless there are other threads blocked on the semaphore.

4.5 Event Handling Framework

The thread module employs a generic and extensible framework for OS level event handling which is closely integrated with the locking scheme. The objective of the framework is to handle events with minimum latency while maintaining low locking overhead. For each type of event to be handled, the user of the framework provides:

- ii) the address of a boolean *event flag* variable,
- i) the address of an OS level *event handler* (e.g. a UNIX signal handler routine) which, among other things (see below), sets the *event flag*, and
- iii) the address of a *callback routine* which is conditionally called from the thread module's context switch code depending on the state of the associated *event flag*.

In operation, when an event is triggered (i.e. the event handler is called by the OS) the first thing the event handler does is set the associated event flag. Its subsequent action then depends on whether or not the ASC associated with the interrupted VP is configured for *preemption* and, if it is, on the state of the level one and level two locks described above. If the *no preemption* option is selected *or* if one or both of these locks are set, the handler simply returns; the callback routine will be invoked later by the thread module's context switch code when the latter inspects the event flag and notices that it is set. If, however, the *preemption* option is selected *and* both locks are clear, the event handler itself calls `thread_yield()` before returning. This causes execution to immediately enter the context switch code (thus preempting the currently executing thread) and directly results in the

⁴ Note that the flag scheme is still required, in addition to multi-thread safe libraries, for the multiple VP case. This is because a signal handler could otherwise enter the thread module while a preempted thread was inside a standard library call. The thread module's scheduler could then run another thread which subsequently tried to make the same library call, resulting in deadlock on the library's mutex lock.

callback being invoked as above. In either case, the callback is invoked with the minimum possible latency in the circumstances (i.e. immediately in the case of *preemption* or as soon as the context switch code notices that the flag is set on the next context switch in the case of *non preemption*).

A further issue to be considered is the possibility of a race condition between the setting of an event flag in the event handler and the reading and clearing of the flag by the context switch code. Without any preventive measures, it would be possible for events to be missed if the flag was set by the event handler (i.e. a new event came in) between the context switch code detecting that the flag was set and it clearing the flag prior to invoking the callback routine. To prevent this potential pathology, a *test-and-set* instruction is used by the context switch code to read and clear the flag in a single atomic action.

One prominent user of the event handling framework is the timer implementation code. This multiplexes virtual timers, using a delta queue scheme, on top of (assuming a UNIX environment) the SIGALRM event. On each invocation of the timer callback routine, the callback fires ripe timers and also resets the OS alarm signal according to the firing time of the next alarm in its queue. These timers form the basis of the timed semaphores mentioned above and also form the basis of message passing routines in the *msg* module which feature timeout options. Other clients of the event handling framework are the periodic alarm handler which uses the framework to implement timeslicing (i.e. a preemption takes place on each periodic alarm) and the *comm* module which uses it to monitor input/ output availability as described in section 5. In the UNIX environment, the periodic alarm handler uses the SIGVTALRM signal and the comm module uses the SIGPOLL signal.

5. COMMUNICATIONS

5.1 API Overview

The primary abstraction employed by the communications system from the API user's point of view is the *iref*. Irefs, which are location independent communication endpoints, are created with the following call:

```
Iref *bind_irefcreate(FlowType cust_flowtype, FlowType prov_flowtype,  
                    Function h, Aspname asp, CharSeq *qos);
```

The two `FlowType` arguments refer to the *directionality* (e.g. stream or request/ reply) of the *iref* in each of its two *roles*, customer and provider (see below). The `h` argument specifies a *handler*: a C function pointer that will be upcalled when data arrives at or leaves an *iref*. Handlers are typically written to call stub/ skeleton code provided by the API personality. Data are passed to/ from handlers in *buffers* (see below). The `asp` and `qos` arguments respectively select and configure an *application specific protocol* (ASP) stack to be associated with this *iref* in subsequent bindings (see below).

Irefs are bound (connected) using the following call:

```
Iref *bind_bindreq(Iref *cust, Iref *prov, CharSeq *qos);
```

This call binds `irefcust` to `irefprov` (the call works regardless of the location of the *irefs* being bound) using the ASP that was associated with `prov` when it was created. `bind_bindreq()` works by invoking a generic *binding protocol* (see below) which negotiates a mutually acceptable QoS for the binding. If a non-null `qos` argument is given, then this is used as the target QoS; otherwise the default QoS that was associated with `prov` at creation time is used. `bind_bindreq()` returns a *iref* on which invocations can be made to control

(e.g. start, stop, renegotiate QoS, destroy) the newly created binding. These control invocations are made using the following routine:

```
int bind_invoke(Iref *iref, int operation, Buffer *args);
```

`bind_invoke()` is a general purpose service which uses the OMG's Internet Inter-ORB Protocol (IIOP) [OMG,98] to invoke an `iref` without requiring a prior explicit binding. To renegotiate the QoS of a binding, `bind_invoke()` is called with the following arguments: the control `iref` returned from `bind_bindreq()`, an `int` constant `RENEG` and a buffer into which a new target QoS has been marshalled from an ASP specific `CharSeq` specification.

It can be seen that ASPs play a role in the communications module that is very similar to that of ASCs in the thread module; both provide application specific behaviour and are configured and reconfigured using specifications defined according to their own schema. As with ASCs, per-ASP routines are provided which marshall QoS parameters into `CharSeqs` which can be passed to `bind_irefcreate()` and `bind_bindreq()`. As well as configuring ASP behaviour, QoS specifications are used by ASPs to determine the resource requirement (i.e. transport sockets, threads and buffers) of the requested binding. Furthermore, ASPs are often written in terms of other ASPs which provide a lower level service. In such cases, the QoS specification determines which lower level ASP(s) will be selected and what their QoS parameters will be. This process, recursively applied, leads to an arbitrarily deep protocol stack, the form of which is a function of the QoS specification passed to the top level ASP.

5.2 Application Specific Protocols (ASPs)

The *comm* module is a multi-level protocol framework which enables bindings to be built from stacks of ASPs which are in turn stacked on top of transport protocols (TCP/IP, UDP/IP and Unix FIFOs are available in the current implementation). To date, ASPs for IIOP, adaptive audio and inter-capsule shared memory communication have been implemented [Coulson,98]. Each ASP supports the following interface (slightly simplified):

```
int asp_listen(Profile *pf, FlowType flowtype);
int asp_connect(Profile *pf, FlowType flowtype);
int asp_accept(Profile *pf, int listening_sap);
int asp_relisten(int sap, Profile *pf);
int asp_reconnect(int sap, Profile *pf);
int asp_reaccept(int sap, Profile *pf);
int asp_close(int sap);
int asp_hdrsize(int sap);
int asp_flipqos(CharSeq *cs_qos);
int asp_send(int sap, Buffer *buf);
int asp_receive(int sap, Buffer **outbuf);
int asp_call(int sap, Buffer *inbuf, Buffer **outbuf);
```

`asp_listen()`, `asp_connect()` and `asp_accept()` are used by the binding protocol (see figure 2) to establish a mutually acceptable level of QoS between the customer and provider using the specified ASP⁵. In the first instance, the target QoS from `bind_bindreq()` is passed to `asp_listen()` at the customer side (this is passed in a `Profile` datatype which also contains addressing information). `asp_listen()` tentatively allocates resources based on the given `Profile` and returns a local identifier for the binding which will later be passed to `asp_accept()`. `asp_listen()` may also modify the QoS information in its `Profile` argument if it would like to propose a different QoS from the initial target. The binding protocol then sends this (possibly modified) `Profile` to its peer which passes it to the provider's `asp_connect()` routine. This works similarly to `asp_listen()`, again returning a local identifier and a possibly modified `Profile`. Finally, this `Profile` is passed to

⁵ As these calls are primarily related to static QoS management, they are not discussed in detail in this paper; see [Coulson,98].

`asp_accept()` back at the customer side which makes the final decision as to the viability of the binding.

`asp_relisten()`, `asp_reconnect()` and `asp_reaccept()` operate in a similar way to `asp_listen()`, `asp_connect()` and `asp_accept()`, but are used when the binding protocol is invoked on a current binding to *renegotiate* the QoS of the binding (see section 5.1). As with the initial binding procedure, the semantics of renegotiation are entirely determined by the ASP involved. For example, some ASPs may ignore renegotiation requests, others may only renegotiate if the binding is currently in a ‘stopped’ state etc.

The remaining ASP routines are conceptually straightforward. `asp_close()` destroys an existing binding, `asp_hdrsize()` returns the size of header used by this ASP (also see section 5.5) and `asp_flipqos()` is used to reverse the byte ordering of words in a QoS specification when a `CharSeq` QoS specification is received from an opposite endian machine. Finally, `asp_send()`, `asp_receive()` and `asp_call()` are the data transfer operations supported by the ASP⁶. ASP implementations can assume that their data transfer operations are only called when the underlying system is ready to accept/ deliver data. For example, when an ASP’s `receive()` entry point is called it knows there is data ready to be read from the ASP/ transport protocol below it without blocking. In general, ASPs are written to be useful for both request/reply or streaming mode interactions⁷; the choice of interaction style is made by the *bind* module above (it is determined by the `FlowType` argument passed to `asp_listen()` and `asp_connect()`). Full details of the ASP architecture and its relationship to the *bind* module are given in [Coulson,98].

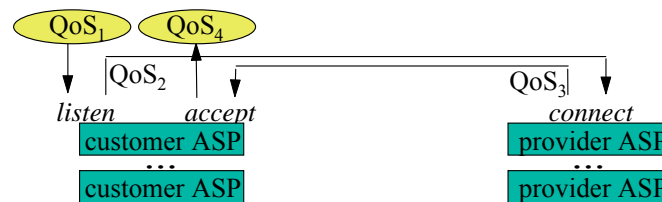


Figure 2: The binding protocol

To maximise performance in a middleware based communications system it is essential that as few context switches, locking operations and copy operations as possible are involved. To maximise predictability it is further essential that bindings are as isolated from each other as possible. To help achieve both these ends, the GOPI design employs *non multiplexed* bindings; each binding has its own OS level socket, its own thread and its own state in each ASP (all these resources are allocated during binding protocol execution). Non multiplexed bindings eliminate the ‘QoS crosstalk’ that is incurred when bindings with different QoS requirements are multiplexed on to some common layer⁸.

⁶ Note that these operations, in common with the rest of the ASP operations, are not called directly by applications. Applications send and receive data either via handlers (see section 5.1) or via send, receive and call operations defined by the *bind* module. These transparently map to the appropriate operations of the ASP associated with the *iref* identifying the binding.

⁷ Certain ASPs, however, may choose to specialise in either request/ reply or streaming mode interactions even to the extent of only providing null implementations of `asp_call()` or `asp_send()/asp_receive()` respectively.

⁸ In some situations it may not be desirable for *all* connections to be non multiplexed. For example, a server dealing with RPC requests from a large number of clients or from clients without any particular QoS requirements would not necessarily want to dedicate a thread to each client. Similarly, IIO connections cannot be given a dedicated server thread as GOPI’s negotiated binding protocol is not used for standard

Each binding in the *comm* module can be configured according to one of the following options:

- *IO polling* (the default option) in which IO availability is explicitly checked every so often (by the *sentinel thread*; see below) by means of an OS primitive such as UNIX's `poll()`, or
- *IO notification* in which the thread module's event handling framework (see section 4.5) is used to allow the OS to asynchronously inform the *comm* module of IO availability on the binding as soon as it happens.

The *IO notification* option provides maximally timely responsiveness to incoming packets from the network - particularly if the thread module's *preemption* option is selected. The *IO polling* option, on the other hand, avoids the overhead of event handling (both options require `poll()` as will become clear below) and thus may be more efficient in a capsule with few threads and little else to do other than receive packets from the network.

5.3 The IO Data Path

The bottom level of the communications module is a routine called `io_handler()`. Where bindings are configured for *IO notification*, `io_handler()` is used as the callback routine in the thread module's event handling framework (see section 4.5) and is thus called as soon as its associated event (e.g. `SIGPOLL`, assuming a UNIX environment) is notified by the OS. If, on the other hand, the *IO polling* option is selected, `io_handler()` must be called explicitly. This opens up a range of possibilities as to how often and under what circumstances `io_handler()` should actually be called. In the current implementation it is called from a distinguished thread called the *sentinel thread* which runs only when no other thread in the system is runnable.

When `io_handler()` executes, it first calls the `poll()` system call (or equivalent) to determine which currently open sockets are ready for IO. `io_handler()` then sends a notification message, using the services of the *msg* module, to the per-binding thread associated with each ready socket. The per-binding threads, on receiving this message call their associated ASP stacks to read/write data from/to the network as appropriate. A per-binding *masking scheme* is employed in the receive direction to prevent per-binding threads from being notified more than once for the same incoming packet. Messages are only sent to per-binding threads if the mask is clear. If the mask is clear, `io_handler()` sets it on sending a message and the per-binding thread clears it when the corresponding packet has been received.

Note that, as each binding has its own thread, it is the thread's associated ASC (the scheduling decisions of which are, in turn, controlled by the QoS specifications of individual bindings) which determines the time/ order in which bindings actually get to read/write from/to the network. This means that GOPI is able to avoid processing low urgency messages when high urgency messages are available. It also means that situations can be avoided in which 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 ORBs when used to support multimedia applications.

IIOP bindings. In such situations either a single thread can be used or a pool of threads can be reserved. GOPI permits either of these options to be configured.

5.4 IO Notification Issues

The IO data path described above is over simplified in that it omits discussion of certain issues arising from the use of *IO notification*. The semantics of event driven IO in operating systems are often rather complex and require careful design to exploit them effectively.

The first issue is that events such as UNIX's SIGPOLL do not carry any information about *which* sockets are ready for IO when there are multiple open sockets⁹. Therefore, as mentioned above, the middleware must explicitly issue a `poll()` call following receipt of the event to obtain this information (in GOPI, `io_handler()` is called which, in turn, calls `poll()`). This implies the overhead of an event, a `poll()` call and a read/write for each network packet received via the *IO notification* option.

The second issue is that in many systems (e.g. SunOS 5.5) the middleware should not assume that each packet arrival will generate exactly one event. On the one hand, when a number of packets addressed to different sockets all arrive from the network at around the same time, only a single event may be evoked. On the other hand, many OSs only generate an event for incoming data when the state of one or more sockets changes from “no data available on this socket” to “some data available on this socket”. This means that while the middleware is about to read a packet whose arrival has been notified by an event, another packet can arrive which does not evoke an event (because the condition “some data available on this socket” still holds). This means in turn that relying exclusively on events to signal the presence of arriving packets may result in packets received at the OS level being ‘unperceived’ and thus unreceived at the middleware level.

The default solution adopted by GOPI is simply to receive a single packet for each event and to additionally rely on a subsequent `io_handler()` call to detect any further packets that may have arrived without evoking an event. A subsequent `io_handler()` call is guaranteed to occur sooner or later *either* because a further event occurs when a packet arrives on some other socket *or* because the *sentinel thread* will eventually run and issue an `io_handler()` call as described in section 5.3.

Beyond this default solution, two additional possibilities are available in GOPI. In the first, the system is configured to always issue an explicit `io_handler()` call after each packet has been read. This buys possibly improved predictability in receiving unperceived packets at the expense of a not inconsiderable performance hit. In the second, ASPs are designed to attempt to read not just a single packet, but as many packets as there is buffer space available to the binding. A non blocking read is used so that the call returns immediately however many packets are actually available. This scheme is attractive in that only a single system call is needed to receive possibly multiple packets. However, it can only be used where the ASP knows in advance the size of expected packets (e.g. it uses fixed size fragments; see section 5.5). This is because, assuming non contiguous buffers, the byte offsets of the unperceived packets, as required by a scatter/ gather mode read, cannot otherwise be known at the time the read is issued.

5.5 Buffer Management

Buffers in GOPI are allocated from pre-allocated pools of memory owned by the *msg* module. A ‘buddy system’ allocation scheme [Knuth,73] is used which efficiently allocates and de-allocates power-of-two sized buffers. The benefit of a user level buffer manager is

⁹ UNIX implementations do typically have a facility, called *siginfo*, for associating additional information with signals. This facility, however, does not provide information on each individual socket for the SIGPOLL signal.

that its performance can be tuned according to expected usage patterns. In addition, it avoids the overhead of frequent `malloc()` calls which are particularly expensive when OS level locking must be used in a multiple VP environment. GOPI buffers are implemented as C structs which, among other things, contain a pointer to the data area of the buffer. Buffers are passed by reference between ASPs in a stacked binding; no copying is incurred.

GOPI requires that all ASPs use fixed sized packet headers. Before allocating a buffer, an `asp_hdrsize()` call is made on the top level ASP in a binding to determine the required header size for the whole stack (this call is propagated down to lower level ASPs to obtain the total header size requirement). Buffers have an associated 'current header pointer' which is 'pushed' and 'popped' by ASPs as the buffer passes up and down the stack.

The calls to allocate buffers, together with their headers, and to free buffers are as follows:

```
Buffer *msg_bufget(int hdrsize, int size);
void msg_bufput(Buffer *b);
Buffer *msg_bufmake(int hdrsize, char *header, int size, char *data);
void msg_bufgetfrags(Buffer *origbuffer, int fragsize, char *fraghdr,
                    int fraghdrsize; Buffer *frags[], int *numfrags);
```

`msg_bufget()` allocates a buffer of size `size` from the buffer pool. If a non zero `hdrsize` argument is given to `msg_bufget()`, a header area of size `hdrsize` is additionally allocated contiguously with the start of the buffer's data area. `msg_bufput()` decrements the given buffer's *reference count* and deallocates the buffer if the reference count has become zero. `msg_bufmake()` differs from `msg_bufget()` in that i) the buffer's data area is supplied by the *caller* (in the `data` argument) rather than being taken from the buffer pool, and ii) if `header` is null, then a header area of size `hdrsize` is taken from the buffer pool, otherwise the given memory is used as a header. `msg_bufmake()` is useful in situations where ASPs manage a device such as an audio card or an area of framebuffer which is mapped to an address in the user virtual address space. In such a case a buffer can be built using this address so that data can be received/ sent straight from/ to the device without incurring any copy overhead.

`msg_bufgetfrags()` is used to perform logical fragmentation on buffers. The call takes a buffer to be fragmented and returns an array of new buffers, each of which points into the data area of the original buffer at an appropriate offset. The fragment header size is specified in `fraghdrsize` and the number of fragments produced is returned in `numfrags`. If the `fraghdr` argument is non null, this same piece of memory is used as the header for all the fragments (this can be useful in situations where fragments are sent successively rather than in a single scatter/ gather IO send, and do not need to be kept for possible retransmission); otherwise a new area of pool memory is allocated for each fragment header. To ensure that the original buffer is not deallocated until all the fragments have been deallocated, the original buffer's reference count is incremented by `numfrags` and the fragment buffers keep a reference to it so that they can decrement its reference count when they are deallocated. `msg_bufgetfrags()` is implemented in terms of `msg_bufmake()`. The `data` argument to `msg_bufmake()` is some address within the data area of the original buffer and the header related arguments depend on the value of `fraghdr`.

Ideally, an ASP stack will be designed so that fragmentation is only performed *once*, by the bottom level ASP that interfaces directly with the transport layer. In such cases a technique called *header borrowing* may be used as an alternative to fragmenting with `msg_bufgetfrags()`. In header borrowing, fragment headers are written into header-sized pieces of memory within the packet buffer's data area so that the headers are *contiguous* with their payloads. For the first fragment of a packet, the header area specified in the original

`msg_bufget()` call is used. For the remaining fragments, the ASP saves into temporary storage the header sized piece of data directly in front of the fragment and uses this memory for the fragment's header. When each fragment has been transmitted the borrowed memory is restored (a similar scheme is used in the reverse direction for defragmentation). The relative costs of using header borrowing and `msg_bufgetfrags()` based fragmentation are as follows:

- i) header borrowing involves two copy operations of a header sized piece of memory per fragment (scatter/ gather IO is not required as the headers and payloads are contiguous);
- ii) `msg_bufgetfrags()` involves allocating memory for fragment buffer structs, headers and scatter/ gather data structures, and assigning the associated pointers.

The tradeoff is likely to be in favour of header borrowing where headers are relatively small and fragments are relatively many.

Finally, when implementing ASPs for which timeliness is a prime requirement, it is advisable to use *fixed sized* packets so that the scheme described in section 5.4 for maximising the timeliness of *IO notification* can be employed¹⁰. If fragmentation is used, this implies that the final fragment of each packet should be padded with garbage data to make it the same size as all the others (to ensure that there is space for this garbage data, the buffer is over-allocated to be a multiple of the fragment size). The effectiveness of such a policy is highly dependent on the fragment size chosen. Large fragments are good in that many packets will not need to be fragmented at all but are bad in that the 'last' fragment may contain a lot of garbage data (the same is true *mutis mutandis* for small fragments). The approach employed in most of the ASPs developed so far is to allow the user to set the fragmentation size as a QoS parameter. This assumes that the user has prior knowledge of likely packet size distribution and can thus make an informed trade off.

6. Performance Evaluation

6.1 Scope of the Evaluation

This section describes experiments which evaluate the performance of the GOPI platform. The purpose of the experiments is to evaluate the extent to which GOPI is capable of performing well despite offering high level abstractions and a high degree of configurability. The experiments are specifically *not* designed to evaluate the configurability properties themselves. These would be more appropriately evaluated through API personalities and applications which exploited the platform's configurability in meaningful ways; this is an issue for future work. Furthermore, the experiments do not explicitly evaluate predictability. This is because predictability is primarily a function of specific policies (e.g. an earliest-deadline-first scheduling policy implemented as an ASC or a rate based transport protocol implemented as an ASP) and only secondarily a function of the middleware itself (the role of GOPI is to provide a configurable framework and environment for the *integration* of such policies).

6.2 Threads

A simple program was written to evaluate the performance of the user level thread package. The program creates two threads each of which execute a tight loop containing nothing but a 'yield' call. Each thread loops 500,000 times. Two versions of this program

¹⁰ Alternative, a scheme whereby the size of the next packet is included in the header of the preceding packet can be used in cases where the underlying transport protocol is reliable.

were run on a SPARCserver-1000 running SunOS 5.5¹¹. One program uses GOPI threads, `thread_yield()` and a simple, single VP, ASC which implements priority based scheduling with earliest deadline first scheduling within priority bands. The other program uses SunOS detached ‘processor scoped’ threads (i.e. user threads), the SunOS `thr_yield()` call and Sun’s SCHED_OTHER scheduling policy (this is a priority based policy).

GOPI threads	12.095 secs
SunOS threads	24.706 secs

Table 1: Relative performance of GOPI and SunOS threads.

The results, shown in Table 1, show a speedup for GOPI threads of $24.706/12.095 = 2.04$ over SunOS 5.5 user level threads.

6.3 Buffer Management

A simple program was written to evaluate the performance of the user level buffer management scheme. The program repeatedly allocates and deallocates buffers over 1,000,000 iterations. The size of each of the allocated buffers is taken from a pseudo-random sequence. One version of the program uses GOPI’s `msg_bufget()/msg_bufput()` and the other uses the UNIX `malloc()/free()` calls. The same pseudo-random sequence of buffer sizes was used by each program.

<code>msg_bufget(), msg_bufput()</code>	2.159 secs
<code>malloc(), free()</code>	5.352 secs

Table 2: Relative performance of `msg_bufget()` and `malloc()`

The results, shown in Table 2, show a speedup for the GOPI buffer management scheme of $5.352/2.159 = 2.48$ over `malloc()/free()`. This is despite the fact that the GOPI buffer management routines are allocating and initialising a `Buffer` struct (these are allocated from a separate dedicated pool) in addition to simply allocating memory.

6.4 End to End Communications

To evaluate the end-to-end performance of the GOPI communications architecture, two experiments were carried out; one to measure performance relative to comparable systems and the other to evaluate the scaleability 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 (IO polling)	743	734	402
GOPI (IO notification)	664	522	350
GOPI vs Sockets overhead	25%	12%	9%

¹¹ This machine and OS was used for all the performance measures reported in this section. In addition, a maximally efficient GOPI configuration was used in all cases (unless otherwise specified); i.e. 1 VP, no preemption and IO polling.

COOL-ORB	464	420	260
----------	-----	-----	-----

Table 3: End to end round trips per second

In the first experiment, GOPI was compared to a minimal ‘base line’ Berkeley sockets program and to a commercial ORB: COOL-ORB 4.1 from Chorus Systems¹². In fact, two Berkeley sockets programs were used; one with a null `poll()` system call inserted before each read and another without the `poll()`. In addition, two GOPI programs were used, one configured to use *IO polling* and the other using *IO notification* (see section 5.2). The COOL-ORB program was modified so that no marshalling overhead was incurred so as to obtain a fair comparison with the GOPI core. 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 transport 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. Table 3 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’ figures give the percentage overhead incurred by GOPI (with the *IO polling* configuration) as against the minimal socket program (with `poll()`)¹³.

Clearly, the performance of GOPI compares very favourably with 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, ASP 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. The performance of *IO notification* was, as expected, inferior to *IO polling* due the additional overhead of event handling. However, *IO notification* in GOPI still easily outperformed COOL and has the additional benefit of increased timeliness and predictability.

Number of bindings	1	2	4	8
Time for 160,000 packets	15.99 secs	8.55 secs	7.97 secs	7.52 secs

Table 4: Stream performance with varying numbers of bindings

The second experiment was intended to evaluate the scalability of GOPI’s non-multiplexed communications architecture. A GOPI program was written which established varying numbers of bindings to transmit a given total number of packets. The two parts of the program were again run on the same machine. Packet sizes of 1024 bytes were used over a stream binding using the shared memory ASP (this was used to avoid either packet loss, possible with a UDP/IP based ASP, or slow start/ buffering effects, possible with TCP/IP and FIFO based ASPs). The binding was run as fast as the system would allow. The time to

¹² 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].

¹³ It seems reasonable to use the socket program with `poll()` rather than the simplest and fastest socket program as a baseline reference for this performance comparison because all user level multithreaded systems are inevitably required to use IO polling of some kind.

receive 160,000 packets was measured at the receiver (multiple averaged runs were performed to obtain the given figures).

The results in Table 4 show that for more than one binding an *increase* in performance was observed over the single binding case. This counter intuitive result probably occurs 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 socket with data waiting). There was then little appreciable difference in overall throughput for $2 \leq n \leq 8$ bindings. This result provides evidence that GOPI's improved QoS predictability (achieved through non multiplexing) is not bought at the expense of an unacceptable degradation in performance.

6.5 ASP Stacking Overhead

A final experiment was performed to assess the overhead of the ASP stacking framework. The baseline for this experiment is the *IO polling* configuration of section 6.4. To assess the overhead of ASP stacking, a minimal ASP was written which could configure multiple instances of itself in a stack, the actual number of instances being determined by a QoS parameter. Varying numbers of instances of this ASP were configured above the baseline configuration. In terms of overhead, each instance contributed only a four byte header and the minimum possible amount of processing.

The results in Table 5 show an acceptable overhead for relatively small numbers of stacked ASPs. Optimisation would be required, however, if it were desired to implement stacks composed of large numbers of fine grained 'micro-protocols'.

Configuration	Calls per second	% overhead
Baseline	734	-
Baseline + 1 null layer	716	2.6%
Baseline + 8 null layers	607	17.4%
Baseline + 16 null layers	531	27.7%

Table 5: ASP Stacking Overhead

7. IMPLICATIONS FOR THE OS

Although the main goal of GOPI is to operate in a standard OS environment, our implementation experience has suggested that the performance and predictability of multimedia middleware could be considerably enhanced with a few localised and relatively straightforward modifications to the operating system. The first such modification, and the simplest to effect, would be to re-implement key informational system calls so that the information is available to the middleware without the overhead of a system call. Examples would be the UNIX calls `gettimeofday()`, used to read the system clock, and `pthread_self()`, used to obtain the id of a VP. Such calls could be re-implemented by mapping OS virtual memory areas, to be written by the OS and read by the application, into application address spaces.

The second suggested modification is to provide *per-socket* IO readiness information with the SIGPOLL signal, probably as an extension of the `siginfo` mechanism (see section 5.4). The availability of such information would eliminate the need for a `poll()` call after

each signal and would thus considerably reduce the overhead of *IO notification*. Note that we do *not* recommend that the semantics of SIGPOLL be changed so that a SIGPOLL is generated by each packet arrival. Although the fact that some packets may fail to evoke a SIGPOLL leads to complexity in the middleware (see section 5.4), there are good reasons to live with this semantic. Firstly, a per-packet SIGPOLL would only work for datagram sockets; with stream sockets using TCP, the OS has no way to identify a user level packet, so the middleware would still require the additional complexity as long as TCP was used. Secondly, per-packet SIGPOLLs would add overhead and it is not at all clear that this would be less than the overhead of the user level schemes described in section 5.4.

Our third suggestion is to consider replacing the traditional priority based scheduler with a fair share policy such as lottery *scheduling* [Waldspurger,94]. Clearly, this is a far more wide ranging and controversial suggestion than the above; nevertheless we believe it is worthy of investigation. The attraction of fair share scheduling is that it has the capability to offer a far superior basis on which to build predictable user level ASCs while simultaneously offering obligatory OS level properties such as fairness. Lottery scheduling works by assigning notional *lottery tickets* to processes (or kernel threads) and then holding a ‘lottery’ at each scheduling point; the winning process runs on the CPU until the next scheduling point¹⁴. Its key properties are i) starvation does not occur as every process will win some lotteries as long as it holds at least some tickets, and ii) if a process has $x\%$ of the available lottery tickets it will probabilistically take $x\%$ of the available CPU time. Thus, VPs implemented as lottery scheduled kernel threads are much closer to ‘real’ CPUs. In fact each VP approximates a CPU with a speed of S/x where S is the speed of the physical CPU and x is the percentage of lottery tickets it holds. Such support would allow user level ASCs to operate in a considerably more predictable manner and to perform meaningful admission tests at the user level. As a refinement, if the OS had to vary the percentage of lottery tickets assigned to a given VP, it could deliver an event to the thread module so that ASCs could appropriately adapt (e.g. by changing scheduling policy or admission test criteria).

8. RELATED WORK

Recent work in the OMG’s CORBA forum has addressed the need for streams and real-time services in distributed object platforms. In particular, the Telecom SIG’s specification for the ‘Control and Management of Audio/ Visual Streams’ [OMG,97] has addressed the need for streams in CORBA and the Realtime SIG’s (ongoing) ‘Realtime CORBA’ specification [OMG,98b] is addressing the need for more general real-time functionality. As stated in the introduction, GOPI’s approach to the support of streams differs fundamentally from that of the OMG. The GOPI approach is to treat stream data as far as possible in the same manner and in the same environment as conventional data whereas the OMG approach is to transport stream data ‘out of band’ and only use the ORB for control and management of streams. The Realtime CORBA activity defines a number of concepts also found in GOPI (e.g. flexible threads and scheduling, pluggable transports and higher level protocols). However, a detailed comparison is not yet possible because, despite the fact that aspects of the Realtime CORBA specification have been implemented by the various contributors, no integrated implementation of the whole design yet exists.

The European Commission funded ReTINA project is designing a CORBA platform featuring streams and QoS extensions [Dang-Tran,96]. The architecture is based on a clean

¹⁴ A common objection to the use of lottery scheduling is that the performance of implementations produced to date has been inferior to that of standard priority based scheduling schemes. However, this may be due more to the fact that standard schedulers have been intensively optimised over many years than to any *inherent* performance limitation of lottery scheduling.

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. While comparable in terms of their overall goals, ReTINA focuses more on static QoS management issues such as binding establishment than the dynamic QoS management issues emphasised in GOPI.

The DIMMA project [Donaldson,98] at APM Ltd., Cambridge, UK is also addressing issues of multimedia support in distributed object platforms. DIMMA is based on the ANSAware distributed systems platform which has been enhanced with abstractions for resource management and a flexible multiplexing structure; like GOPI, the degree of per-binding internal multiplexing can be configured according to the needs of different applications. Compared to GOPI, DIMMA focuses more on *global* QoS configurability and less on the *fine grained* QoS configurability of individual bindings. The finer grained configurability in GOPI is achieved largely through the ASC and ASP frameworks which are lacking in DIMMA.

TAO [Schmidt,97] is a CORBA 2.0 compliant platform that runs on real-time operating systems. Although multimedia support is considered, and the OMG's Control and Management of A/V Streams specification has recently been implemented in TAO [Munjee,98]), the system appears to be primarily designed for hard real-time applications such as avionics. 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. In contrast to TAO, GOPI is designed to provide integrated soft real-time/ multimedia support in a conventional OS/ network environment and emphasises flexible user services support rather than hard deterministic performance.

GOPI is also influenced by related work in more specific areas. In terms of *scheduling*, GOPI's two-level thread architecture is influenced by split level scheduling [Govindan,91] and by SunOS 5.5's two-level thread architecture [Sun,94]. The GOPI architecture differs from split level scheduling in that it does not depend on specialised OS support for user level threads. It differs from the SunOS design mainly in terms of flexibility. For example, GOPI allows new schedulers (both preemptive or non preemptive) to be added dynamically, and the event handling framework allows close integration between external events and scheduling. GOPI's ASC framework can also be compared to *scheduling classes* in UNIX SVR4. The significant difference is that ASCs are user level and scheduling classes are kernel level. As a result of this ASCs suffer less from mutual interference between classes (see [Nieh,94]). This is particularly true when the underlying VPs are lottery scheduled.

In terms of *communications*, GOPI's ASP framework is clearly influenced by flexible protocol frameworks such as the x-kernel [Hutchinson,91] and Ensemble [vanRenesse,96]. GOPI differs from these earlier systems primarily in its approach to QoS configurability. In most previous designs, the builder of a stack would attempt to realise a given QoS requirement by explicitly selecting layers and individually configuring them in an appropriate way. Although superficially attractive, this approach has the drawback that the stack builder code must 'know' how to map the given QoS specification to the configuration parameters of each selected layer. Furthermore, it must be aware of dependencies between layers with various configurations (e.g. the degree of compression supported by an MPEG layer may impact the choice of throughput and delay parameters given to a transport layer). Although these problems are surmountable where small numbers of layer types are involved, the approach does not scale well when many layer types are involved and new QoS parameters have to be introduced. To avoid these 'QoS mapping' problems and to better support extensibility, GOPI builds stacks in an implicit, encapsulated manner. The stack builder simply specifies a single 'top level' ASP and configures it using the ASP's own QoS schema.

The ASP instance itself then autonomously realises the required QoS by mapping its QoS parameters to those of further, encapsulated, ASP instances (and so on recursively).

A second way in which GOPI differs from most previous protocol frameworks is in its approach to scheduling. The x-kernel and Ensemble, in common with most other frameworks, execute stacks by dedicating a single thread to all stacks and using an ‘external’ scheduler. In other words, they divide the total workload into discrete steps (e.g. the execution of individual layers in individual stacks) and serially execute these steps according to some policy (e.g. FCFS or priority order) as they become ready. Although this approach allows some limited control over the QoS of each individual stack, typically by assigning different priorities to the steps of different stacks, the GOPI thread-per-stack approach is much more configurable and fine grained.

9. CONCLUSIONS

We have described a middleware platform intended to support soft real-time/ multimedia applications in a standard OS environment and have also suggested ways in which relatively minor alterations to the OS could enhance the level of support offered to the platform. GOPI is designed in a modular fashion and implements a number of configurable mechanisms to enhance performance and predictability. The performance figures in section 6 demonstrate that despite its high degree of configurability, GOPI compares favourably with related systems and can adequately meet the performance requirements of multimedia applications¹⁵.

The main dimensions of configurability in GOPI can be summarised as follows.

- *threads*
 - configure by choosing appropriate ASC and scheduling parameters
 - change ASC and scheduling parameters of threads at run time
 - add new ASCs dynamically
 - configure ASCs with *preemption*, *no preemption* or *timeslicing*
- *bindings/ communications*
 - configure by choosing appropriate ASPs and QoS
 - renegotiate ASPs and binding QoS at run time (not addressed in this paper)
 - add new ASPs dynamically
 - configure ASPs with *IO polling* or *IO notification*

The ASC and ASP frameworks are central to this configurability. These frameworks make it straightforward to customise middleware for specific applications by designing and installing specialised plug-in modules. Furthermore, these frameworks effectively sidestep the much discussed ‘QoS mapping problem’ in which QoS specifications in some general purpose high level notation must be somehow mapped to low level resources such as transport sockets, VPs and memory. This mapping is trivial in GOPI because ASCs and ASPs define their *own specialised QoS specification schema* and the mapping from this schema to generic resource managers (provided by GOPI) is simply hard wired into their code.

In future work, we intend to further develop support for configurability. One direction is to port GOPI to the Windows NT environment so that NT’s flexible support for dynamic loading can be exploited to load new ASCs and ASPs at run time. We are also interested in designing appropriate APIs to permit maximal configurability while not overburdening the

¹⁵ Audio and video stream bindings will typically require zero or only minimal marshalling so little if any performance penalty over and above that detailed in section 7 should be incurred in a GOPI based CORBA implementation.

programmer with confusing detail. To this end, we are investigating the concept of reflection [Maes,87] which separates out basic behaviour from ‘meta-behaviour’ (i.e. configurability). A report on our early work in this direction can be found in [Blair,98].

REFERENCES

[Blair,97] Blair, G.S., Stefani, J.B., “Open Distributed Processing and Multimedia, Addison-Wesley, In Press, 1997.

[Blair,98] Blair, G.S., Coulson, G., Robin, P. and M. Papatomas, An Architecture for Next Generation Middleware, Proc. Middleware ’98, The Lake District, England, November 1998.

[Coulson,97a] Coulson, G and de Meer, J., Guest Editor's Introduction, Special Issue on Quality of Service, Distributed Systems Engineering Journal, Vol 4, No 1, pp 1-3, March 1997.

[Coulson,97b] Coulson, G. and Clarke, M.W., "A Distributed Object Platform Infrastructure for Multimedia Applications", *Proc. 3rd Workshop on High Performance Protocol Architectures (HIPPARCH'97)*, Uppsala, Sweden, June 1997.

[Coulson,98] Coulson, G and Clarke, M.W., A Distributed Object Platform Infrastructure for Multimedia Applications, Computer Communications, Vol 21, No 9, pp 802-818, July 1998.

[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. Available at: <http://www.uk.infowin.org/ACTS/RUS/PROJECTS/ac048.htm>.

[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.

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

[Donaldson,98] Donaldson, D., Faupel, M., Hayton, R., Herbert, A., Howarth, N., Kramer, A., MacMillan, I., Otway D. and Waterhouse, S., “DIMMA - A Multi-media ORB”, Proc. Middleware ’98, The Low Wood Hotel, Ambleside, England, September 1998.

[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.

[Mungee,98] Mungee S., Surendran, N and Schmidt, D., “The Design and Implementation of a CORBA Audio/Video Streaming Service, Washington University Technical Report WUCS-98-15, Department of Computer Science, Washington University at St Louis, MO 63130, USA, May 1998.

[Nieh,94] Nieh, J. et al., “SVR4 UNIX Scheduler Unacceptable for Multimedia Applications”, Proc, 4th Workshop on Network and Operating System Support for Digital Audio and Video, Lancaster, England, April 1994.

[**OMG,98**] The Common Object Request Broker: Architecture and Specification 2.2, available at <http://www.omg.org/>

[**OMG,97**] Control and Management of Audio Visual Streams, OMG Document number telecom/97-05-07; available from http://www.omg.org/library/schedule/AV_Streams_RTF.htm

[**OMG,98a**] Realtime CORBA V1.1, Initial Submission to Realtime SIG's RFP on Realtime CORBA, OMG Document number orbos/98-01-08, available at <http://www.omg.org/>*

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

[**Stankovic,95**] Stankovic, J., "Implications of Classical Scheduling Results for Real-Time Systems", IEEE Computer, 1995.

[**Sun,94**] SunOS 5.5 Manual Pages, Sun Microsystems, 1994.

[**vanRenesse,96**] 19. van Renesse, R., "Masking the Overhead of Protocol Layering", Proc. 1996 ACM SIGCOMM Conference, Stanford, September 1996, ACM Press, pp96-105.

[**Waldspurger,94**] Waldspurger, C.A., Weihl, W.E., "Lottery Scheduling: Flexible Proportional-Share Resource Management", Proc. First USENIX Symposium on Operating Systems Design and Implementation, November 1994. Proceedings at: <http://www.usenix.org/publications/library/proceedings/osdi/index.html>.