

A RETROSPECTIVE ON THE DESIGN OF THE GOPI MIDDLEWARE PLATFORM¹

Geoff Coulson, Shakun Baichoo and Oveeyen Moonian

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

contact: geoff@comp.lancs.ac.uk

ABSTRACT

This paper offers a high-level retrospective overview of the GOPI middleware platform which is the outcome of a three year project aimed at the development of generic, configurable and extensible middleware. GOPI has a clearly defined modular structure, is widely extensible with plug-ins at all levels of the architecture, and natively supports stream interactions as well as standard operation invocation. It offers a generic framework for quality of service (QoS) specification and management, and supports a high level multimedia oriented programming environment that is backwardly compatible with the OMG's CORBA. At its lower levels it supports QoS-driven resource management and features an optimised IOP stack. Despite its enhanced functionality, GOPI's IOP performance equals or exceeds that of state-of-the-art CORBA platforms.

1. Introduction

Middleware based on the OMG's CORBA [OMG,01], Microsoft's DCOM [Microsoft,01] and Java's Remote Method Invocation (RMI) [Sun,01] has emerged in recent years as a highly successful technology. Commonly referred to as *object-based middleware*, the technology is now widely deployed in industry and commerce and seems set to play an important role in the foreseeable future.

As a consequence of its success, there is now increasing desire to apply object-based middleware in areas of distributed computing for which it was not originally intended. Prime examples are mobile, real-time and multimedia computing; fault tolerant systems; systems that must be continuously available (often referred to as "7x24" systems); and systems involving groups and multipeer communication. In addition, there is growing demand to deploy middleware in a burgeoning diversity of hardware and operating system (OS) environments. Examples range from traditional client/ server environments, through scientific supercomputers interconnected by high speed networks, through small PDAs employing low-speed and wireless connectivity, through embedded devices with very primitive OS support.

¹ Coulson, G., Baichoo, S., Moonian, O., "A Retrospective on the Design of the GOPI Middleware Platform", ACM Multimedia Journal (to appear), 2002.

In light of these considerations, the research project that is the subject of this paper has developed, over the past three years, a new middleware platform called *GOPI* (Generic Object Platform Infrastructure). *GOPI* [Coulson,98] addresses the above challenges by attempting to render middleware functionality, including low-level ‘core’ areas, as generic, configurable and extensible as possible while retaining backward compatibility with CORBA. It also emphasises the need to integrate multimedia, QoS and multipeer-oriented functionality as ‘first class’ facilities in the middleware domain.

In contrast to previously published papers that have addressed specific aspects of *GOPI*’s design in depth [Coulson,98,99a,99b,99c,00,01a,01b], this paper offers a high-level retrospective overview of the platform and focuses on experiences and overall contributions. The remainder of the paper is structured as follows. First, section 2 enumerates and discusses the design principles underlying *GOPI*. Then section 3 briefly surveys the platform design as a necessary grounding for the rest of the paper. This is followed, in section 4, by a discussion of key aspects of the design in more depth in terms of our experience over the course of the project. Finally, section 5 relates our efforts to other relevant research and section 6 concludes and outlines our current work and future directions.

2. Design Principles

As mentioned, the design of *GOPI* has been guided by general principles of *genericity*, *configurability*, and *extensibility*. Ideally, we wanted our platform to be capable of meeting the needs of any conceivable class of distributed application with any conceivable requirement along any conceivable dimension, e.g. in terms of types of interaction, topology, scalability, performance, predictability, memory footprint, reliability, availability or adaptivity. In cases where the required functionality could not be attained by configuring the existing platform, it should be possible to extend the platform in the required direction in a natural and explicitly supported manner.

A supporting goal was to construct the platform in terms of a well-defined *modular structure*. The issue of internal platform structure has been overshadowed by the middleware standardisation process which, rightly, focuses on ‘black box’ behaviour rather than internal structure. Nevertheless, without a well-defined internal structure, middleware platforms have severely restricted scope for configurability and extensibility. On the other hand, given a suitably modular structure, it is much easier to particularise platforms to niche application domains or deployment environments. For example, a Real-Time CORBA-like particularisation [OMG,99a] might select a profile that included thread pools together with a communications infrastructure that supports explicit binding, client-propagated priorities, etc.

At the programming model level, our most fundamental design decision was to base our design on the computational model defined by the ISO’s Reference model for Open Distributed Processing (RM-ODP) [ITU-T,95], [Blair,97]. In particular, we wanted to support the following aspects of RM-ODP:

- *Integrated Interaction Types*

CORBA supports traditional request/ reply operation invocations, media streams [OMG,00a] and events [OMG,00b], but only the former interaction type is directly supported by the core computational model in a ‘first class’ manner (i.e. is apparent in interface definitions). In contrast, RM-ODP advocates an integrated

approach in which streams and events are first class entities in the computational model. Advantages of this approach are that synchronisation and coordination between interactions of different types (e.g. using operations to control media streams) is easy to program and need not rely on specialised extensions like CORBA's A/V Streams service [OMG,00a], which may be heavyweight, complex and not exactly what the programmer needs. Furthermore, we wanted all interaction types to be supported by a common integrated platform infrastructure. Such an infrastructure is well placed to make informed resource management trade-offs which take account of the diverse *quality of service* (QoS) requirements of different interactions and interaction types. Such an infrastructure can also support synchronisation and coordination between interactions in a direct and efficient manner.

- *Quality of Service Support*

RM-ODP allows interaction points in interfaces to be annotated with *QoS attributes* that tailor interaction point behaviour in terms of any conceivable dimension; examples are latency, periodicity, reliability, and levels of 'guarantee' of other attributes. This implies the need for infrastructure support in terms of: *i*) suitable ontologies, or *schema*, for QoS specification, *ii*) mapping of QoS specifications between layers or modules that may employ different schema, *iii*) mapping of QoS specifications to appropriate platform-level resources like threads, buffer pools or transport connections, *iv*) admission controlling of resource allocation requests, and *v*) negotiation and perhaps dynamic re-negotiation of resource allocation among principals such as the various end-systems involved and the network. In addition, it implies that the infrastructure must support resource schedulers that can appropriately multiplex dynamic resources like CPU and network capacity in accordance with given allocations.

- *Explicit Bindings*

In CORBA, the process of 'binding' to a remote object, i.e. establishing a connection and per-connection state, is *implicit* in the narrowing of a local proxy object². In contrast, the process of binding in RM-ODP is *explicit*: it requires the binding creator to submit a list of endpoints that will participate in the binding (including the 'client') together with a QoS specification. This submission then results in the creation of a *binding object*, visible to the user, that encapsulates the (QoS determined) infrastructure supporting the binding. One key benefit of this approach is that it provides structure for the isolation of per-binding resources, which is crucial in an environment that must support bindings with particular QoS needs. Also, it naturally supports run-time management of the binding; for example, the binding object can generate events when its QoS degrades, provide operations that enable the binding user to alter its QoS, or provide operations to add and remove participants (e.g. in a group binding). Finally, explicit binding lends itself to a 'plumbing' approach in which bindings can be composed by third

² This is not true in Real-Time CORBA which does employ a species of explicit binding [OMG,01]. However, our point is that, rather than being available in only one specialised environment, explicit binding should be an integral part of a generally applied computational model and therefore should be available in any environment (e.g. mobile computing or group communications).

parties into multi-stage topologies such as pipelines or trees. This is particularly useful in media stream processing and group applications.

Building on explicit bindings, we also wanted the platform to support the straightforward definition of *new types* of binding with arbitrary application specific semantics. For example, in a multimedia application environment it may be useful to define a *multiplexor* binding type that merges media packets from multiple sources to a single sink according to a specified policy (e.g. round robin or first come first served); the associated binding object would offer operations to dynamically add and remove sources and/ or sinks. We also wanted to be able to define new binding types in terms of value-added compositions of existing binding types. The motivation here was to reduce development effort and encourage the establishment of extensible libraries of binding types.

Also in the spirit of extensibility, we rejected from the outset the notion of a single ‘standard’ schema for QoS specification, and opted for the co-existence of multiple application-domain-tailored, per binding type, schemata. For example, the above mentioned multiplexor binding type might employ a simple enumeration-based schema that enables choice from a selection of named policies, whereas a more sophisticated schema used by an audio transport binding might support the specification of rules (expressed, e.g., in XML) for switching to alternative encodings as a function of bandwidth variation [Coulson,99c].

As a final goal, we decided to focus on the support of *distributed multimedia* applications. We saw multimedia as a particularly challenging application domain that demands a superset of the functionality required by many other domains. For example, it has strong requirements for interaction type integration, particularly of media streams and operations, and for QoS specification and resource management. It also has obvious extensibility requirements in terms of media specific protocols and filters. In addition, multipeer operation could be exercised in the form of media dissemination scenarios, and, because multimedia is demanding in terms of performance, we would be forced to keep our design and implementation tight and efficient.

3. An Overview of GOPI

3.1 Architectural Overview

As shown in Figure 1, GOPI’s architecture at the coarsest granularity comprises two levels:

- the *GOPI-core* level, and above that,
- the *API personality* level.

This architecture is clearly related to that of micro-kernel operating systems such as Spring [Sun,93], and is inspired by similar reasoning. The aim is to provide common, generic functionality in the core, and application-domain-specific, specialised programming models (i.e. high level abstractions and semantics) at the personality level. At a finer granularity, each level of the architecture is internally structured in a modular manner. For example, GOPI-core includes independent concurrency, communication and binding modules (see below).

Multiple personalities, each with their own independent programming model, can co-exist in the same address space and share the generic low-level GOPI-core API. To date, two personality layers have been built: a multimedia oriented personality (an early version was described in [Coulson,99b]), and a standard CORBA personality [Coulson,01a]. We briefly describe these in section 3.3, having first discussed noteworthy aspects of the design of GOPI-core in section 3.2.

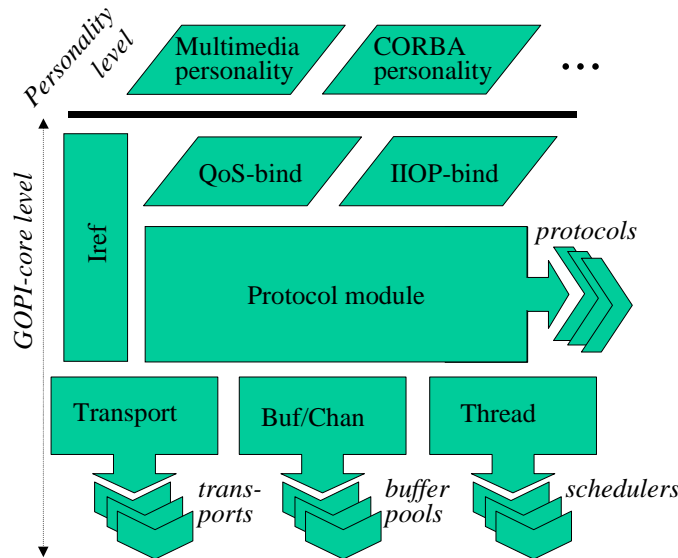


Figure 1: Overall GOPI Architecture

3.2 GOPI-core

GOPI-core consists of approximately 20,000 lines of C and runs on a variety of platforms including SunOS, Linux and Win32. It comprises the following modules:

- *Thread* is a sophisticated concurrency package that simultaneously supports multiple co-existing plug-in modules, called *schedulers*, that encapsulate their own scheduling policy for user-level threads. Each scheduler can be configured in terms of its own QoS schema and autonomously manages its own encapsulated resources (kernel threads).
- *Buf* manages buffer pools which are implemented as plug-ins; the associated *Chan* module provides an efficient inter-thread buffer passing service.
- *Transport* provides a common abstraction layer over a set of OS-supported plug-in transport protocols (such as TCP, UDP, pipes, shared memory etc.).
- *Protocol* provides a framework above the transport layer for the deployment of stacks of user-defined plug-in modules, called *protocols*, each of which can be configured in terms of its own QoS schema. Protocols also monitor their own QoS and can generate events when QoS degrades.
- *Iref* supports location transparent communication endpoints called *irefs* that act as participants in bindings at the GOPI-core level; for CORBA compatibility, routines are provided to translate irefs to and from CORBA IORs.
- *QoS-bind* supports the abstraction of *QoS bindings* which conform to the model of explicit binding outlined in section 2.

- *IOP-bind* provides a set of support services for *IOP bindings*. These are standard CORBA bindings and are layered over a plug-in protocol implementation of CORBA's GIOP protocol [OMG,01] and used by the CORBA personality.

As can be seen, a notable feature of GOPI-core is its aggressive use of *plug-ins*. For example, the Thread module supports plug-in schedulers, and the Transport and Protocol modules support plug-in transports and higher-level protocols respectively. In addition, the IOP-bind module supports plug-in request demultiplexors and thread-pools. In the GOPI context, plug-ins are defined as software objects that, while conforming to a well-defined interface, implement that interface in terms of varying behaviours. Crucially, plug-ins can be added to the system not only statically, at compile time, but dynamically, at run-time. They are uniformly and efficiently implemented as arrays of pointers to functions.

Whereas plug-ins represent GOPI-core's unit of *fine grained* extensibility, GOPI-core can also be extended at a *coarser granularity* by adding new top-level modules. To facilitate this mode of extension, which is only available statically, a layering principle is adopted whereby earlier listed modules (as ordered in the above list) are not permitted to employ the services of later listed modules. This makes it easier to modify, add or replace higher-level (later listed) modules without widespread impact. The linkages between modules are straightforwardly implemented as procedure calls.

3.3 The Personality Level

As mentioned, we have implemented two separate personalities on top of GOPI-core; the standard CORBA personality is layered atop IOP-bind, while the specialised multimedia programming environment builds on QoS-bind. In this section we focus on the multimedia personality as this is essentially a superset of the CORBA personality. Both personalities are implemented in C++. Detailed descriptions of an early version of the multimedia personality are available in [Coulson,99b] and [Coulson,00].

To support interaction type integration at the personality level, the multimedia personality extends the CORBA Interface Definition Language (IDL) with *signals*, *streams* and *QoS-groups*. Signals are used for one-off messages whereas streams are used for continuous message flows (e.g., video or audio streams). Both these interaction types are unidirectional and capable only of either emitting or receiving messages [ITU-T,95]. So-called *QoS-groups* are used to partition the interaction points in an interface into disjoint sets that will share the same QoS at run-time. More specifically, each QoS-group will, at run-time, be underpinned by a separate GOPI-core binding, the protocol stack and QoS of which will be determined at either service creation time or bind time (or both).

The following example illustrates the IDL extensions:

```
// GOPI extended CORBA IDL
interface MediaServer {
    typedef char Title [512]
    sequence<Title> Titles;
    sequence<char> VideoFrame;
    sequence<char> AudioPkt;

    videoqos streamout video(out VideoFrame f);
    audioqos streamout audio(out AudioPkt a);
    void getTitles(out Titles ts);
}
```

```

void selectTitle(in Title t);
command signin start();
command signin stop();
titleevent sigout newTitleAdded(out Title t);
};

```

This ‘media server’ emits video and audio at two producer (‘*streamout*’) stream interaction points, respectively *video()* and *audio()*, each of which is placed in its own dedicated QoS-group (respectively, *videoqos* and *audioqos*). In addition, the service has standard operational interaction points to browse the available video titles (*getTitles()*) and to select a particular title (*selectTitle()*). These operations have no associated QoS-group and therefore will use a default protocol and QoS at bind time. There are also two consumer (‘*signin*’) signal interaction points, *start()* and *stop()*, in a common QoS-group (*command*) which are used to turn the flow of media on or off, and there is a producer (‘*sigout*’) signal, *newTitleAdded()*, in its own dedicated QoS-group (*titleevent*), which asynchronously notifies the availability of a new title (titles are assumed to be added via a separate management interface).

As well as the IDL extensions, the multimedia personality employs a number of run-time abstractions as follows³:

- *InterfaceRefs* are location independent endpoints that encapsulate a set of GOPI-core irefs, one for each QoS-group in an associated IDL specification. InterfaceRefs also embed typing information and are characterised with a ‘role’ that can be either *provider* or *requirer* or both. The *provider* role represents a unit of service provision (an object implementing an IDL interface), while the *requirer* role represents a unit of service requirement (a client proxy). Note that role is orthogonal to the directionality (in or out) of stream and signal interaction points. Provider InterfaceRefs also contain a default protocol type and QoS specification (expressed in the protocol’s native schema) for each QoS-group.
- *GenericBinding* is a C++ class that can be used to bind any pair of type and role compatible InterfaceRefs (one must be a requirer and the other a provider), regardless of the location of their referents (i.e. *third party* binding is supported). The class supports operations to get and set the binding’s QoS; these allow the programmer to override the default QoS associated with the provider iref both at bind time and while the binding is active. These operations are implemented in terms of underlying QoS negotiation and renegotiation services offered by GOPI-core’s QoS-bind module as discussed in section 4.3.
- *QoS specification* classes are used to specify per-protocol QoS for both InterfaceRefs and GenericBindings. Each QoS class, conventionally named *<protocol-name>_QoS* after one of the protocols supported by the GOPI-core Protocol module, derives from a base class called *QoS*. Each derived class typically adds constructors that take parameters relating to the associated protocol’s QoS schema, and get and set methods relating to individual elements of the QoS schema. Base QoS class functionality offers generic operations to marshal and unmarshal the encapsulated QoS data so it can be passed around the distributed system.

³ More detail on the use of these abstractions can be found in Appendix A.

- Optional *QoS manager* classes are also associated with a particular protocol (although each protocol can have multiple QoS managers) and derive from a base class called *QoSManager*. QoS managers are attached to particular *GenericBindings* at run-time. Each QoS manager implements a virtual method *qos_event(QoS *current_qos)* which encapsulates a QoS-manager-specific policy for dealing with a protocol-specific event. For example, a *qos_event()* implementation in a video protocol stack's QoS manager may respond to a 'degraded frame rate' event by renegotiating the QoS of the *GenericBinding*.

Another key dimension of flexibility offered by the multimedia personality is the availability of a variety of *binding styles*. These are supported partially by the underlying protocol and partially by having the IDL compiler generate multiple, per binding style, stub and skeleton formats that can be selected at run-time. The *active binding* style is useful for media stream bindings and employs upcall oriented stubs so that the upcall style of user/ binding interaction traditionally employed at the provider (or server) side of a binding can also be employed at the requirer (or client) side. This allows QoS management to be fully delegated to an specialised underlying protocol stack, which creates a thread that repeatedly upcalls the stub with a periodicity that is a function of the QoS specification that was passed to the binding on its creation. Also for media streams, we support *direct bindings* in which the underlying protocol bypasses the application and directly sources or sinks messages itself. This binding style is typically used for reasons of efficiency. For example, we have a video protocol that directly gets or puts packets from or to a video card without incurring the overhead of passing them through the application. In the direct binding style, stubs and skeletons are still upcalled on message dispatch or arrival in the usual way, but only vestigial 'meta-data' (i.e. an integer representing a counter) are passed so that higher level code can track and synchronise with the flow of data.

Furthermore, to support the aggregation of multiple *GenericBindings* we provide *pipelined* and *multipeer* binding styles. In the former, multiple *GenericBinding* instances (which must employ the same IDL type) are linearly concatenated. At each 'link' in the pipeline a combined stub/ skeleton is employed for which a single upcall suffices to both deliver an incoming message from the upstream binding and, on return of the upcall, to pass the message (or a transformation of it) to the downstream binding. This implementation yields a very simple programming model for pipelines and is also highly efficient as no context switch is involved between stages. In *multipeer* bindings, a single distinguished provider *InterfaceRef* is used to represent a 'group', and group members are represented by requirer *InterfaceRefs*. Requirers become group members by being bound individually to the distinguished provider *InterfaceRef* using the standard point-to-point *GenericBinding* mechanism discussed above. However, at the protocol level, transparently to the user, these multiple bindings are mapped to a single multicast address by the underlying (multipeer binding style capable) protocol⁴.

Finally, layered on the above abstractions, the multimedia personality supports the definition of *application specific bindings* (ASBs). ASBs are encapsulations of

⁴ To avoid a single point of failure in multipeer bindings, the provider *InterfaceRef* can be passively replicated in an arbitrary number of address spaces using a standard GOPI-core service [Saikoski,00]. However, machinery to replicate state held in the 'group' protocol instance is not provided by default and must be specially implemented if required. Group management functionality such as authorisation and authentication is not provided at this level either.

multiple GenericBindings (especially pipelined and multipeer bindings) and/ or other (recursively nested) ASBs. Crucially, like GOPI-core protocols, ASBs offer specialised QoS schema and run-time QoS management classes that either build directly on GenericBinding-level QoS or QoS management classes, or on those of their nested ASBs.

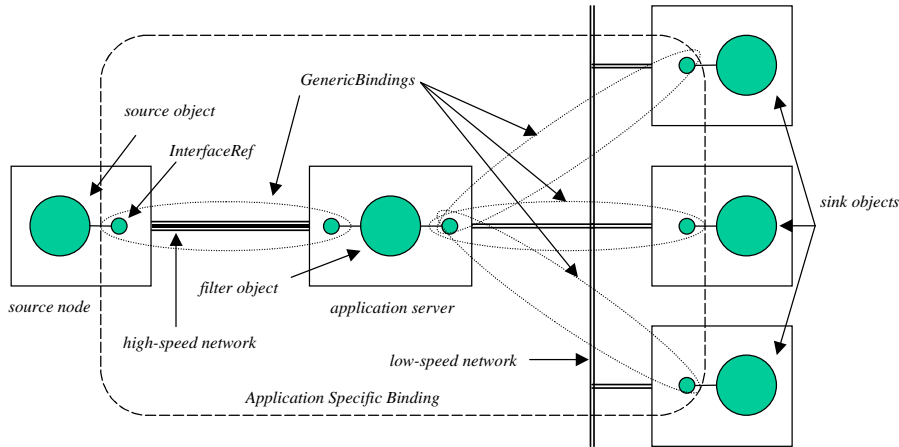


Figure 2: Application Specific Binding Example

As an example of an ASB, consider the media dissemination scenario of Figure 2 in which a media stream is piped through compression software on an application server machine before being disseminated over a low speed segment of the network. This scenario can be realised as an ASB that *i*) remotely instantiates the compression filter object (this is done using a generic GOPI service), *ii*) creates, using an active GenericBinding (or, indeed, a nested ASB), a third party binding between the media source object and the filter, *iii*) similarly creates a multipeer binding over the low speed network by remotely establishing multiple GenericBindings between the filter and the sink objects. In addition, a pipelined binding style could be employed to ensure an efficient implementation of the compressor object.

4. Experiences

4.1 Overview

In this section we discuss our experiences with the design and implementation of GOPI. More specifically, the following sub-sections discuss our experiences in the following areas: GOPI's modular structure, its approach to QoS management, its approach to resource management, programming with the multimedia personality's API, and performance.

4.2 Modular Structure

We have found a liberal application of the 'plug-in' concept to be extremely useful in rendering key areas of the architecture extensible in terms of both mechanism and policy (specific examples were given in section 3.2). For example, we have implemented a wide range of protocols including: *i*) a plug-in that implements the CORBA GIOP protocol, *ii*) a simple stream-oriented protocol for generic continuous media types, *iii*) more sophisticated 'adaptive' protocols for audio and video, and *iv*) unreliable and reliable message-oriented multicast protocols. In addition we have implemented a number of scheduler plug-ins including dynamic priority, earliest deadline first (EDF) and rate monotonic.

On the down side, however, when we asked our students to write plug-ins they found it initially difficult to deal with the significant amount of context involved. Protocol implementers in particular have to deal with many issues over and above basic protocol functionality; for example, QoS mapping, event reporting and binding style semantics. There were also problems with the coarser grain modular structure. In particular, our simple procedure call implementation of inter-module linkages was found to be a limitation as it supports only compile time extensibility in a natural way. Furthermore, we found it hard to maintain the discipline of restricting dependencies between modules without explicit programming model support. Our currently favoured solution to all these problems is to implement both modules and plug-ins as *components* (see section 5). We also exploit the related notion of *component frameworks* [Szyperski,98] to give additional structure and guidance to plug-in developers.

4.3 Approach to QoS Management

GOPI's approach to QoS specification and management is in many ways its most unique and innovative characteristic. We have found that the use of per-plug-in QoS schemas (both in the Protocol and Thread frameworks and in ASBs) yields maximal simplicity, flexibility and extensibility in comparison to related systems which use a fixed QoS ontology and QoS mapping rules (e.g. MULTE-ORB [Kristensen,01]). For example, new protocols do not have to be written in terms of a fixed, pre-existing set of QoS parameters such as delay, jitter and throughput. The apparent downside to this is that each plug-in must be responsible for interpreting and mapping its own QoS specifications without generic system support. In practice we have not found this to be a burden; it is usually straightforward for plug-ins to map to either generic GOPI-core resources such as threads, buffers or transports, or to the QoS schema of other plug-ins on which they may be layered.

Protocol layering and QoS mapping function as follows: newly instantiated protocol instances can choose, on the basis of their given QoS specifications, to instantiate below themselves a further protocol instance (or instances) to which they will pass a QoS specification expressed in the schema of that protocol type. When this process is applied recursively it naturally leads, with minimal framework support, to the instantiation of *stacks* of QoS configured protocols. In this scheme, unlike that employed by traditional protocol stack frameworks such as Ensemble [van Renesse,98], the mode of protocol stacking is implicit and hidden from the user who merely selects a top level protocol and provides an associated QoS specification. The shape of the subsequently established stack is determined solely by the private choices (presumably made on the basis of its given QoS specification) of each recursively instantiated protocol instance.

We have found that, despite supporting arbitrary QoS schema, a simple generic QoS negotiation protocol has sufficed for all the protocols we have implemented to date or can envisage implementing. The form of this negotiation protocol, which is implemented in the QoS-bind module, is shown in Figure 3. As mentioned, the protocol can also be re-invoked while a binding is in existence to renegotiate the binding's QoS.

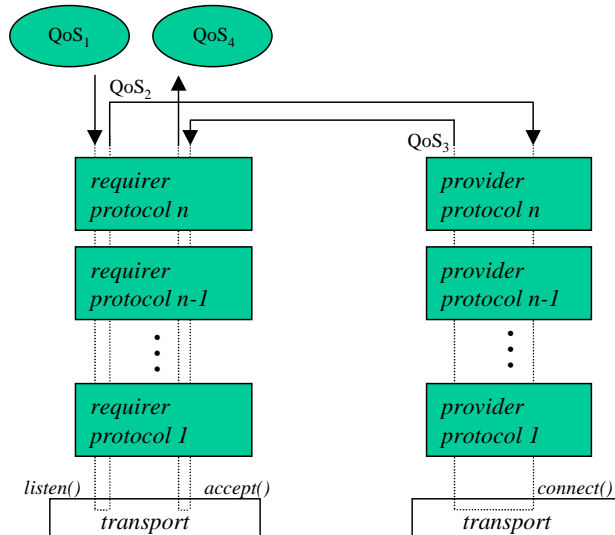


Figure 3: GOPI's Generic QoS Negotiation Protocol

Referring to Figure 3, each protocol instance on the requirer side, starting with the initially instantiated 'requirer protocol n ', first maps its given QoS specification (initially QoS_1), either directly to a transport or to the QoS schema of a further protocol instance that it may choose to instantiate below itself at layer $n-1$. Then, having tentatively reserved resources⁵, it returns a possibly revised QoS, QoS_2 , which will differ from QoS_1 if the latter cannot be achieved. QoS_2 is then carried by the binding protocol (using a standard CORBA binding) across to the provider side where a similar process unfolds. Eventually, the top level revised QoS at the provider side, QoS_3 , is passed back to the requirer side when it is again passed down through the stack for ultimate confirmation, and finally returned as QoS_4 .

Although the recursive protocol stack instantiation scheme has worked well, we have identified certain drawbacks. One limitation is that, in terms of topology, only *trees* of protocols (rooted at the top) are supported. This, of course, allows linear stacks but precludes more general *graphs* of protocols. A second limitation is that 'receive' calls can only be issued from the top protocol in a stack and this precludes the use of upcall-driven control structures which are beneficial where it is useful to have incoming packet headers determine a path up through the stack. Finally, a third possible limitation is that, because protocol stacks instantiate themselves transparently, it is not possible to change the QoS of an existing binding by dynamically inserting new protocols into a stack from the 'outside'; this can only be achieved via top-down QoS renegotiation⁶.

Subsequent work that attempts to address the above limitations (at the cost of an increase in complexity) is reported in [Kramp,00] and the relation of our framework to other protocol framework is further discussed in section 5.

⁵ In addition to a layer $n-1$ protocol or a transport, protocols may of course map to and allocate any other GOPI resource type—e.g. buffers and threads. They may also instantiate multiple layer $n-1$ protocols to form a tree topology.

⁶ On the other hand, inserting protocols from the outside is potentially dangerous. It is arguably far safer to declaratively specify a changed requirement and then allow the stack itself to perform any required reconfiguration.

4.4 Approach to Resource Management

The QoS specification, negotiation and management framework discussed above relies on the availability of low level resources with which to underpin its QoS provision. In this regard, we have found that, even given a lack of OS support for guaranteed resource allocation, there still is a lot that can usefully be done at the user level. First, we have implemented a buddy based [Knuth,73] buffer management plug-in that supports multiple buffer pools (as multiple plug-in instances) so that individual bindings (or, more precisely, protocol plug-ins participating in a binding) can allocate from a ring-fenced pool of buffers. The buddy based buffer manager is able to allocate and deallocate buffers at over twice the speed of a simpler and smaller plug-in based directly on *malloc()* [Coulson,99a].

The most important aspect of resource management, however, has proved to be the GOPI-core Thread module's scheduler framework. This framework enables the co-existence of independent scheduling environments each of which offers user-level threads with a particular scheduling policy, concurrency style (i.e. non-preemptive, preemptive or timesliced) and underlying set of resources in terms of kernel threads. In accordance with the principles laid out in section 1, the form of each scheduler's QoS schema is entirely the concern of the scheduler itself. As a simple example, the schema associated with our EDF scheduler consists simply of two integer parameters: 'deadline' and 'period'.

We have found the scheduler framework to be a crucial underpinning for predictability, relative prioritisation and general QoS support. By employing distinct schedulers for, say, incoming invocation handling and media stream handling, we can coherently schedule instances of each of these interaction types (e.g. using priority scheduling for the former and EDF for the latter), while simultaneously dedicating processing resource (i.e. underlying kernel threads) to each type so that neither can starve the other. Furthermore, we can dynamically alter the QoS and scheduling policy of existing threads by migrating them to a different scheduler, and can load and unload schedulers as application activity evolves. We also believe that the plug-in scheduler concept has significant potential in implementing *real-time* middleware environments such as Real-Time CORBA.

An apparent downside to the use of the scheduler framework is that, at least to some extent, it compromises GOPI's portability because user-level thread implementations are inherently less portable than kernel threads (this is because they require small sections of CPU specific code to initialise a run-time stack and to save and restore CPU state). Despite this, our use of user level concurrency has not proved problematic in practice. For example, we have quickly and easily ported GOPI from its original SunOS reference platform to a number of other UNIX platforms and to Win32. The main reason for this is that machine dependent code can be eliminated on almost all modern platforms through the use of widely deployed standard facilities such as POSIX's *ucontext(3)* (see [Engelschall,99]). This issue, together with a detailed treatment of the whole scheduler framework, is further discussed in [Coulson,01b].

4.5 Programming with the Multimedia Personality's API

Our experience has been that the multimedia personality makes it very easy to create distributed applications that employ and control media streams, QoS and multipeer topologies. Apart from the generally high degree of abstraction and

integration, the fact that standard IDL parameter structuring is generally available is particularly useful for applications that employ highly structured streams and signals (e.g., distributed virtual reality or remote data collection). In addition, the availability of third party binding has proved invaluable, particularly as an underpinning for ASBs. To more concretely illustrate the use of the API, Appendix A outlines the code required to set up a simple video connection based on the *MediaServer* example of section 3.3.

At a more detailed level, we have found that allowing a mixture of interaction types in the same interface has proved a fortunate design decision. This is in contrast to RM-ODP and other stream capable platforms (e.g. MULTE-ORB [Kristensen,01]) that require all interactions points in an interface to be of the same type. We have found that our approach permits a more flexible separation of concerns; in particular, as illustrated in the *MediaServer* example in section 3.3, it is often more useful to separate concerns in terms of base functionality versus management functionality rather than simply in terms of interaction type (in the *MediaServer* example, management functionality consists of adding titles, which involves only standard operational interactions, whereas the service's base functionality additionally relies on streams and signals). Furthermore, the notion of QoS-groups has proved very successful. The essence of this mechanism is to provide compile-time structure for QoS specification but to defer the actual specification to run-time. It thus represents a satisfying compromise between a static compile time QoS specification style and a style that relies exclusively on run-time QoS specification.

We have also found the multipeer binding style to be a great success in terms of ease of programming. Multipeer bindings render multicast based group communications transparent so all the user has to do is build up a desired topology in terms of point-to-point connections. As an example of the naturalness of integration of multipeer support, it is trivial (and probably sensible) to choose at service creation time to underpin the 'titleevent' QoS-group in the example of section 3.3 with a multipeer binding style compliant reliable multicast messaging protocol rather than a conventional peer-to-peer protocol. The downside of our multipeer approach is that group management functionality (e.g. admission testing of new members) is left to individual protocols in GOPI-core's Protocol module. Ideally, there should be a better way of *i)* providing generic and reusable implementations of such services and *ii)* integrating them with the personality level API. As with the general modularity issues outlined in section 4.2 our currently favoured approach is to employ *reflection* (see section 6) as a solution to this problem so that group management is available at the 'meta-level' through a standard interface accessible to both GOPI-core level protocols and personality level programmers.

Finally, in terms of extensibility the ASB concept has proved a great success in offering extensibility at a higher level of abstraction than protocols. ASBs are particularly useful for wrapping pipeline or tree topologies. Furthermore, their relatively high level of abstraction does not significantly impact performance because at run-time they are ultimately just concatenations of GenericBindings. On the other hand, extensibility at the protocol level has proved less successful. Beyond the general issues mentioned in section 4.2, adding a new protocol involves not only integrating a new protocol into the GOPI-core Protocol module but also providing personality level QoS and QoS_M classes that are programmed in a different language

and environment. We are currently investigating methods of streamlining and automating the procedure for adding new protocols and their associated machinery.

4.6 Performance

We have investigated the performance of stream bindings in the multimedia personality using a variety of media specific protocols. In particular, we have compared the performance of simple stream binding with that of a minimal C-language socket-based client/ server program that has none of GOPI's general middleware related support. In such a comparison we have found that GOPI can stream packets at around 57% of the rate of the socket-based program [Coulson,01a]. Furthermore, the attained rate is around 72% of that of a minimal socket-based program written in Java. We believe that these results provide strong evidence that the sophisticated programming environment made possible by a stream capable middleware platform need not be bought at the expense of unacceptable performance.

However, our most detailed performance analysis has been directed at the standard CORBA personality. This has revealed that GOPI's modular structure and built-in extensibility has not been detrimental to performance: as shown in detail in [Coulson,01a], GOPI's performance equals or exceeds that of well-known high-performance ORBs like OmniORB [Lo,98] and TAO [Kuhns,99]. A digest of these results is given in Figure 4⁷.

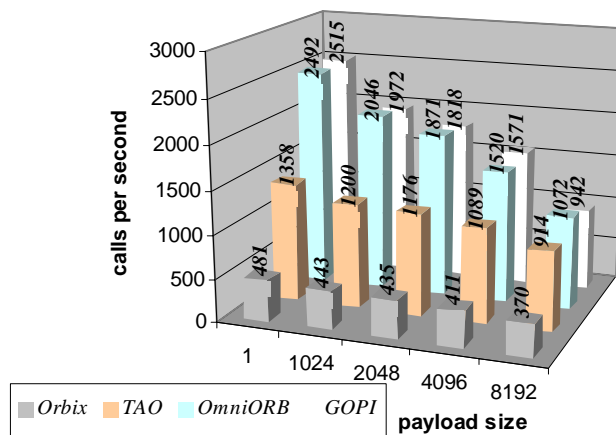


Figure 4: Comparative Performance of the CORBA Personality

Overall, we believe that the following optimisations have most directly contributed to GOPI's high GIOP performance:

- an optimisation that caches GIOP headers and predicts the contents of commonly used fields based on typical usage;
- the use of only a subset of the GIOP protocol while remaining fully GIOP compatible; in particular, GIOP's *fragmentation* and *request cancellation*

⁷ Figure 4 illustrates numbers of invocations achieved per second as a function of different payload sizes. In all cases the tests involved a minimal IDL interface with a single operation that accepts and returns an array of octets (the 'payload'). No processing or copying of the payload was performed at either the client or the server, both of which ran on the same machine (an otherwise unloaded 360MHz Sun SPARC Ultra 5 with 64MB of main memory and running SunOS 5.7) and communicated over TCP/IP loopback. The compared ORBs were Washington University's TAO 1.1/ ACE 5.1, AT&T's OmniORB 2.8.0, and Iona's Orbix 3.0. See [Coulson,01a] for more detail.

messages are not used, and the *close connection* message is ignored wherever possible.

- a ‘read-ahead’ optimisation in which the number of OS level *recv()* calls per GIOP message converges to one; this is in contrast to the two *recv()* calls employed by most ORBs (one for the header and another for the payload);

In addition, our GIOP implementation supports the option of *non-multiplexed, per-binding, connections* which boost performance by avoiding the overhead of reply demultiplexing at the client side. Finally, the implementation is supported by a number of general GOPI-core level optimisations relating to notification of incoming messages, context switching, inter-thread communication, connection cache management, request demultiplexing and buffer management. Full details of all these optimisations are given in [Coulson,01a].

5. Related Work

The OMG’s CORBA forum has produced a comprehensive body of work relating to the interests of this paper. For example, in recognition of the need to support interaction types other than operations, CORBA has produced specifications for the control and management of audio and video streams [OMG,00a], for event handling and notification [OMG,00b] and for messaging [OMG,98]. Similarly, in recognition of the need to particularise the core middleware architecture for various purposes, it has produced or is producing specifications such as real-time CORBA [OMG,99a], embedded CORBA [OMG,00c], high-performance CORBA [OMG,99b], pluggable protocols [OMG,99c] and portable interceptors [OMG,99d]. In addition, CORBA supports ‘policy objects’ which are used to configure aspects of the ORB’s service (e.g. in areas such as the portable object adapter, asynchronous messaging, security, transactions and real-time).

While these extensions successfully address genuine needs, their architects are often forced to compromise to meet the overarching imperative of minimal change to existing specifications. In this way, for example, the A/V streams specification settles for merely ‘supervising’ streams which must themselves be implemented outside the CORBA environment. Similarly, portable interceptors can only be attached at limited, pre-specified, points in the middleware architecture. Furthermore, the extensions frequently have the appearance of having been ‘grafted on’ to the existing standard in a way that meets the immediate need but may lead to complications in the long run. For example, real-time CORBA is obliged to define an independent architecture distinct from that of the original, non-real-time, design. These limitations are probably inevitable given the environment of standards conformance and (hence) incremental development in which they were developed. Our long term, unconstrained and ‘idealistic’ perspective is that such extensions should be capable of being accommodated without compromise and without necessitating fundamental changes to the computational model or core middleware architecture. We believe that GOPI, despite being conceived before these CORBA extensions were defined, is nevertheless capable of accommodating many of them in a far more integrated manner without change to its fundamental computational model or modular structure.

In the academic research field, workers at Washington University, St. Louis publish widely on their *TAO* middleware platform. This is at the cutting edge of CORBA research; the TAO team have contributed to many, and implemented all, of the above mentioned CORBA extensions. Of particular interest, [Kuhns,99] reports

on the integration of ‘pluggable protocols’ into TAO and [Wang,00] discusses the use of the CORBA component model to support QoS. However, because of its close adherence to CORBA, TAO does not support equivalents of GOPI’s integrated interaction types, or the latter’s open approach to QoS specification, mapping and negotiation, or GOPI’s various binding styles (active, direct, pipelined and multipeer), or run-time QoS management and renegotiation capabilities, or ASBs. Furthermore, although it features a high degree of compile-time and deploy-time configurability, TAO does not support an equivalent of the run-time configurability offered by GOPI’s plug-ins.

At the University of Illinois, workers have developed a platform called *dynamicTAO* [Kon,00] which enhances TAO with ‘configurator’ meta-objects that facilitate dependency tracking and support run-time attachment and detachment of pre-loaded components. DynamicTAO currently supports configurable components in the areas of threading, request multiplexing, request scheduling, and connection management. They have also developed a fully componentised ORB called *LegORB* [Roman,00] which applies similar ideas and additionally incorporates dynamically loadable components. Generally speaking, this works offers run-time configurability but does not explicitly address QoS. For example, it does not offer QoS specification, negotiation, and run-time management as does GOPI. Other research at Illinois (see, e.g., [Wichadakul,01]) does address QoS issues but is primarily concerned with higher level services like QoS compilation, profiling and mapping, distributed resource management and visual programming. It is thus complementary to our work which focuses more on enabling mechanisms in the ORB core.

Researchers at BBN [Zinky,01] have developed a reflective ORB called *QuO* that employs specialised declarative languages to specify QoS, together with courses of action to be taken when QoS degrades. QuO’s approach to QoS specification is arguably more sophisticated than GOPI’s, but, again, GOPI takes a lower level, more procedural, approach to the realisation of QoS. In addition, QuO is not specifically targeted at multimedia; e.g. it does not support streams. GOPI’s approach can therefore be seen as largely complementary to QuO’s (for example, a QuO-like personality could be built on top of GOPI-core).

Citrix’s *DIMMA* project [Donaldson,98] has addressed issues of multimedia support in distributed object platforms. DIMMA enhances an earlier platform (ANSAware) with a flexible multiplexing structure and abstractions for resource management. However, it focuses on global QoS configurability rather than fine-grained, per-binding, configurability. The finer grained configurability in GOPI is achieved largely through explicit bindings and the protocol and scheduler frameworks. These are lacking in DIMMA, as is a run-time QoS management facility. *FlexiNet* [Hayton,98] and *FlexiBind* [Hanssen,99] are more recent offerings from Citrix. These are Java ORBs that focus on the provision of a highly flexible binding framework through dynamic tailoring of protocol stacks. They do not address the wider issues of configurability explored by GOPI.

The *ReTINA* project [ReTINA,99] has designed a CORBA platform featuring streams and QoS extensions. Similarly to GOPI, the architecture offers a clean separation between generic ORB support mechanisms such as interface reference management, threads, buffers etc., and extensible ‘binding classes’ that provide tailored communications services. 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. A related project at France Telecom is continuing the main themes of the ReTINA research in the context of a Java ORB called *Jonathan* [Dumant,98].

At the University of Oslo, researchers are developing a multimedia capable ORB called MULTE-ORB [Kristensen,01]. This work has so far integrated a protocol framework called Da Capo into an existing, traditionally architected, CORBA v2 ORB called COOL. The resultant platform, which incorporates an extensible set of binding protocols—cf. GOPI's QoS negotiation protocol, supports flexible QoS enabled bindings but does not provide full support for multimedia streams as it relies on a traditional CORBA API. In their K-ORB project [K-ORB,01] researchers at Trinity College Dublin are applying their previous work on the Iguana reflective C++ extension to the area of modular, pluggable, ORB architectures. In terms of pluggability, this work is taking a basically similar approach to that of GOPI but the emphasis is not particularly on QoS or multimedia support—rather it is on the OMG's minimumCORBA specification. The work is also at a relatively early stage of development, with only the IIOP area of the ORB architecture having been fully developed to date.

Finally, GOPI's contributions in the specific area of protocol frameworks can be related to work such as Horus and Ensemble from Cornell University [van Renesse,98], Tau from the Georgia Institute of Technology [Clayton,98], Coyote from the University of Arizona [Bhatti,98] or Da Capo++ from the University of Zurich [Stiller,98]. Generally speaking, these frameworks are considerably more sophisticated than GOPI's Protocol module, but do not address the latter's central aims of simple and flexible QoS specification, mapping, negotiation, and run-time QoS management (i.e. event production and renegotiation). As mentioned, subsequent work in [Kramp,00] has attempted to extend GOPI's framework (e.g. in terms of fully general graph topologies as opposed to GOPI's tree topologies, and exception handling support) while retaining GOPI's core emphasis of minimally prescriptive QoS support.

6. Conclusions, Current Work and Future Directions

Looking back over the project, we believe we have successfully demonstrated that ORB architectures can incorporate fully integrated media streaming and QoS support in a natural, efficient and easy to program fashion while retaining backward compatibility with CORBA. In particular, we would characterise GOPI's major contributions as follows:

- the definition of a configurable and run-time-extensible modular ORB structure that has the potential to be applied in a wide variety of applications and systems environments;
- the realisation of an RM-ODP based multimedia programming model that gives first class status to media streams, events and QoS support;
- the definition of an approach to QoS management that accommodates QoS specification, mapping, monitoring and (re)negotiation, in a completely generic and non-prescriptive manner;
- the development of a sophisticated multimedia programming environment that features various binding styles (active, direct, pipelined and multipeer) and extensibility through the notion of application specific bindings (ASBs);

- a comprehensive approach to resource management that helps underpin the QoS requirements of individual bindings even in a commodity operating system environment;
- the development of a novel protocol stack framework that focuses on QoS specification and management;
- the development of a novel thread scheduling framework that focuses on QoS specification and management;
- a highly optimised IIOP stack that equals or exceeds the performance of other ORBs.

As a result of experiences from the GOPI project, we are currently developing a new middleware platform called *OpenORB* [Blair,01] that builds on GOPI's successes and addresses its weaknesses. In particular, while retaining GOPI's basic structure and functionality, and reusing a significant amount of its codebase, OpenORB makes radical enhancements to GOPI's models of extensibility and configurability.

First, OpenORB strengthens GOPI's modular structure by applying a well-founded *component model* for the definition of modules and plug-ins. This facilitates dynamic loading and unloading of components, explicitly records the dependencies of each component in terms of other components, enables components to be written in different languages, and permits components to have multiple interfaces. Second, OpenORB provides sophisticated reflective facilities that significantly enhance the management of configurability and reconfigurability in the middleware architecture.

We believe that reflection will play a key role in future middleware architectures as it promises a principled and consistent methodology for (re)configuration, management, extensibility and long term evolution. The essence of reflection is to provide *causally connected meta-models* of the structure and functionality of the architecture. This means that when the meta-model is changed, the underlying middleware changes accordingly, and vice versa. As an example, OpenORB's *architectural meta-model* comprises graph structures that represent current component configurations (e.g. of the middleware as a whole or of a single binding or ASB). By inspecting this graph one can discover the current configuration and then change it simply by manipulating the graph.

A final area in which we hope to further enhance GOPI's level of extensibility is in terms of interaction types. GOPI supports integrated operations, signals and streams but this list is currently non-extensible. Our aim is to build a framework that allows new interaction types to be defined in terms of existing types. For example, we may want to define an asynchronous interaction type for mobile computing [Davies,97] by building on the signal interaction type. Due to its fundamental nature, this mode of extensibility requires considerably more sophisticated support than GOPI's plug-ins. However, its availability should yield an entirely new dimension of flexibility. Our current explorations in this area are reported in [Blair,01].

References

[Bhatti,98] Bhatti, N.T., Hiltunen, M.A., Schlichting, R.D., Chiu, W., "Coyote: A System for Constructing Fine-Grain Configurable Communication Services", ACM Transactions on Computer Systems, Vol 16, No 4, pp 321-366, November 1998.

- [Blair,01]** Blair, G.S., Coulson, G., Anderson, A., Blair, L., Clarke, M., Costa, F., Duran-Limon, H., Fitzpatrick, T., Johnston, L., Moreira, R., Parlavantzas, N., Saikoski, K., “The Design and Implementation of Open ORB v2”, Special Issue of IEEE Distributed Systems Online on Reflective Middleware, <http://www.computer.org/dsonline/>, 2001.
- [Blair,97]** Blair, G.S., Stefani, J.B., “Open Distributed Processing and Multimedia”, ISBN 0201177943, Addison-Wesley, 1997.
- [Clayton,98]** Clayton, R., Calvert, K., “A Reactive Implementation of the Tau Protocol Composition Mechanism”, Proc. IEEE Conference on Open Architectures and Network Programming (OpenArch), San Francisco, California, 1998.
- [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.
- [Coulson,99a]** Coulson, G., “A Configurable Multimedia Middleware Platform”, IEEE Multimedia, Vol 6, No 1, pp 62-76, January - March 1999.
- [Coulson,99b]** Coulson, G., and Baichoo, S., “A Distributed Object Platform for Multimedia Applications”, Proc. IEEE Multimedia Systems, Florence, Italy, pp 122-126, June 1999.
- [Coulson,99c]** Coulson, G., Blair, G.S., Davies, N., Robin, P. and Fitzpatrick, T., “Supporting Mobile Multimedia Applications through Adaptive Middleware”, IEEE Journal on Selected Areas in Communications, Vol 17, No 9, pp 1651-1659, September 1999.
- [Coulson,00]** Coulson, G., and Baichoo, S., “Experiences in Implementing a Distributed Object Platform for Multimedia Applications”, Software Practice and Experience, Vol 30, pp 663-683, 2000.
- [Coulson,01a]** Coulson, G., and Baichoo, S., “Implementing the CORBA GIOP in a High-Performance Object Request Broker Environment”, ACM Distributed Computing, Vol 14, No 2, pp 113-126, April 2001.
- [Coulson,01b]** Coulson, G., and Moonian, O., “A Quality of Service Configurable Concurrency Framework for Object Based Middleware”, Concurrency Practice and Experience (to appear), 2001.
- [Davies,97]** Davies, N., Wade, S., Friday, A., Blair, G., “Limbo: A Tuple Space Based Platform for Adaptive Mobile Applications”, Proc. IFIP International Conference on Open Distributed Processing and Distributed Platforms (ICODP/ICDP), Toronto, Canada, 27-30 May 1997.
- [Donaldson,98]** Donaldson, D., Faupel, M., Hayton, R., Herbert, A., Howarth, N., Kramer, A., MacMillan, I., Otway D. and Waterhouse, S., “DIMMA - A Multimedia ORB”, Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware), The Lake District, England, November 1998.
- [Dumant,98]** Dumant, B., Horn, F., Dang-Tran, F. and Stefani, J.-B., “Jonathan: an Open Distributed Processing Environment in Java”, Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware), The Lake District, England, November 1998.
- [Engelschall,99]** Engelschall, R., “Portable Multithreading: the Signal Stack Trick for User-Space Thread Creation”, paper included with *GNU Portable Threads* distribution, <http://www.gnu.org/software/PTH/>, 1999.
- [Hanssen,99]** Hanssen, Ø., and Eliassen, F., “A Framework for Policy Bindings”, Proc. IEEE International Symposium on Distributed Objects and Applications

- (DOA), Edinburgh, September 1999.
- [Hayton,98]** Hayton, R., Herbert, A., Donaldson, D., “FlexiNet: A Flexible Component-oriented Middleware System”, Proc. ACM SIGOPS European Workshop on Support for Composing Distributed Applications, Sintra, Portugal, September 1998.
- [ITU-T,95]** ITU-T, ISO/IEC Recommendation X.902, International Standard 10746-2, “ODP Reference Model: Descriptive Model”, January 1995.
- [Knuth,73]** Knuth, D.E., “The Art of Computer Programming, Volume 1: Fundamental Algorithms”, Second Edition, Reading, Massachusetts, USA, ISBN 0201896834, Addison Wesley, 1973.
- [Kon,00]** Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhães, L.C., and Campbell, R.H., “Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB”. Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware), IBM Palisades Executive Conference Center, New York, April, 2000.
- [K-ORB,01]** Trinity College’s K-ORB project: <http://www.dsg.cs.tcd.ie/Research/minCORBA>, 2001.
- [Kramp,00]** Kramp, T and Coulson, G., “The Design of a Flexible Communications Framework for Next-Generation Middleware”, Proc. IEEE International Symposium on Distributed Objects and Applications (DOA), Antwerp, Belgium, September 2000.
- [Kristensen,01]** Kristensen, T., Berlin Kalleberg, I., Plagemann, T., “Implementing Configurable Signaling in the MULTE-ORB”, Proc. IEEE Conference on Open Architectures and Network Programming (OpenArch), Anchorage, Alaska, pp 137-146, April 2001.
- [Kuhns,99]** Kuhns, F., O’Ryan, C., Schmidt, D.C., Othman, O. and Parsons, J., “The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware”, Proc. IFIP International Workshop on Protocols for High-Speed Networks (PfHSN), Salem, MA, USA, August 1999. (See also: <http://www.cs.wustl.edu/~schmidt/PfHSN.ps.gz>)
- [Lo,98]** Lo, S.L. and Pope, S., “The Implementation of a High Performance ORB over Multiple Network Transports”, Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware), The Lake District, England, November 1998. (See also: <http://www.uk.research.att.com/omniORB/omniORBPerformance.html>)
- [Microsoft,01]** Microsoft’s DCOM web page: <http://windows.microsoft.com/com/tech/dcom.asp>.
- [OMG,98]** CORBA Messaging Submission, <http://cgi.omg.org/cgi-bin/doc?orbos/98-05-05>, 1998.
- [OMG,99a]** Real-Time CORBA Specification, <http://www.omg.org/cgi-bin/doc?orbos/99-02-12>, 1999.
- [OMG,99b]** CORBA Real-Time PSIG High Performance Working Group Homepage, http://www.omg.org/homepages/realtime/working_groups/high_performance_corba.html, 1999.
- [OMG,99c]** CORBA Telecom SIG’s RFP on Extensible Transport Framework, <http://www.omg.org/cgi-bin/doc?telecom/99-10-05>, 1999.
- [OMG,99d]** CORBA Portable Interceptors Working Draft, <http://cgi.omg.org/cgi-bin/doc?orbos/99-10-01>, 1999.
- [OMG,00a]** Object Management Group, Control and Management of Audio/Video

- Streams, v1.0, <http://www.omg.org/>, 2000.
- [**OMG,00b**] Object Management Group, Event Service v1.0, OMG Document formal/2000-06-15, 2000.
- [**OMG,00c**] CORBA Embedded Systems Working Group Homepage, http://www.omg.org/homepages/realtime/working_groups/embedded_systems.html, 2000.
- [**OMG,01**] The Common Object Request Broker: Architecture and Specification, <http://www.omg.org/>, 2001.
- [**ReTINA,99**] ReTINA, “Extended DPE Resource Control Framework Specifications”, ReTINA Deliverable AC048/D1.01xtn, European Union ACTS Project AC048, Brussels, January 1999.
- [**Roman,00**] Roman, M., Mickunas, D., Kon, F., and Campbell, R.H., LegORB and Ubiquitous CORBA”, IFIP/ACM Workshop on Reflective Middleware, IBM Palisades Executive Conference Center, New York, April 2000. (See also: <http://www.comp.lancs.ac.uk/computing/rm2000/program.html>)
- [**Saikoski,00**] Saikoski, K. B. and Coulson G., “Configurable and Reconfigurable Group Services in a Component Based Middleware Environment”, Proc. Workshop on Dependable and Group Communication (DSMGC) at IEEE International Symposium on Reliable Distributed Systems (SRDS), Nürnberg, Germany, October 2000.
- [**Stiller,99**] Stiller, B., Class, C., Waldvogel, M., Caronni, G., Bauer, D., Plattner, B., “A Flexible Middleware for Multimedia Communication: Design, Implementation, and Experience”, IEEE Journal on Selected Areas in Communications: Special Issue on Middleware, Vol 17, No 9, pp 1580-1598, September 1999.
- [**Sun,01**] Sun’s RMI-IIOP web page: <http://java.sun.com/products/rmi-iiop/index.html>, 2001.
- [**Sun,93**] Sun’s Spring Microkernel, <http://www.sun.com/research/technical-reports/1993/abstract-14.html>, 1993.
- [**Szyperski,98**] Szyperski, C., “Component Software: Beyond Object-Oriented Programming”, ISBN 0201178885, Addison-Wesley, 1998.
- [**van Renesse,98**] van Renesse, R., Birman, K.P., Hayden, M., Vaysburd, A., Karr, D., “Building Adaptive Systems Using Ensemble”, Software Practice and Experience, Vol 28, No 9, pp 963-979, August 1998.
- [**Wang,00**] Wang, N., Parameswaran, K., Kircher, M., Schmidt, D.C., “Applying Reflective Middleware Techniques to Optimize a QoS-enabled CORBA Component Model Implementation”, Proc. International Computer Software and Applications Conference (COMSPAC), Taipei, Taiwan, October 2000.
- [**Wichadakul,01**] Wichadakul, D., Nahrstedt, K., Gu, X., Xu, D., “2KQ+: An Integrated Approach of QoS Compilation and Reconfigurable, Component-Based Run-Time Middleware for the Unified QoS Management Framework”, Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware), Heidelberg, Germany, November 2001.
- [**Zinky,01**] Zinky, J., Schantz, R., Loyall, J., Anderson, K., Megquier, J., “The Quality Objects (QuO) Middleware Framework”, Special Issue of IEEE Distributed Systems Online on Reflective Middleware, <http://computer.org/dsonline/>, 2001.

Appendix A: Example Use of the Multimedia Personality API

In this appendix we present a slightly simplified and abbreviated example of application programming with GOPI's multimedia personality. The example builds on the *MediaServer* IDL interface that was presented in section 3.3 by illustrating the creation and management of a binding between a provider service supporting this interface and a remote requirer (client). The example is coded in C++.

A.1 Compiling the Interface

When the IDL interface is compiled, templates of requirer-side and provider-side *implementation classes*, called *r<interface_name>* and *p<interface_name>* respectively, are generated. In our specific case, these are named *rMediaServer* and *pMediaServer*. As is traditional, the provider-side implementation class template contains skeletons corresponding to each IDL operation (these are to be completed by the application programmer). But, in addition, to support signal and streams, the provider class also contains skeletons for all *signin* and *streamin/ streamout* interaction points. Furthermore, in addition to the traditional stubs, the requirer class contains upcall oriented skeletons corresponding to all *sigout* and *streamin/ streamout* interaction points⁸.

The implementation class templates also contain implementations of two standard management methods that will be referred to later:

```
InterfaceRef *getInterfaceRef(void);
void set_default_QoS(String qosgroupname, QoS *qos);
```

A.2 Creating the Service and Specifying Default QoS

The following code fragment associates a protocol called *STRM*, together with a corresponding default QoS specification, with the *videoqos* QoS-group of a newly created provider-side *MediaServer* implementation object. The *STRM_QoS* class through which the QoS is defined offers a constructor to which a desired frame rate can be passed. It also offers the following set and get methods that will be referred to below in section A.3: *int get_rate()* and *void set_rate(int rate)*.

```
// C++
pMediaServer *pMS = new pMediaServer();
STRM_QoS *my_video_qos = new STRM_QoS(24); // 24 is the frame rate
pMS->set_default_QoS("videoqos", my_video_qos);
```

The call of the provider-side implementation class' *set_default_QoS()* method creates a per-QoS-group GOPI-core-level iref, suitably configured with the given protocol and default QoS specification. Similar calls would be made to associate other default protocols and QoS specifications with the other *MediaServer* QoS-groups. These irefs will eventually be embedded in an *InterfaceRef* that can be obtained using the above mentioned *getInterfaceRef()* call.

A.3 Implementing a QoS Manager

The following is an example QoS manager that can be associated with the *STRM* protocol used above. Note that provision of QoS management is optional and, furthermore, QoS managers do not have to be provided for all QoS-groups in an IDL

⁸ These are needed at the source end of stream bindings in case the user chooses to employ *active binding* semantics; see sections 2.3 and A.4.

defined service.

In the following, we use the following methods defined in the `GenericBinding` class: `QoS get_QoS(String qosgroupname)`, `void set_QoS(String qosgroupname, QoS newqos, QoSM *qosmanager)` and `void bind()`. Additionally, `this_bind` is a public variable of type `GenericBinding` that is defined in the virtual class `QoSM` from which `STRM_QoSM` derives.

```
// C++
class STRM_QoSM : QoSM {
    STRM_QoSM(GenericBinding *b) {QoSM(b);};
    void qos_event(QoS *current_qos)
    {
        GenericBinding *b = this_bind;
        STRM_QoS *oldq = b->get_QoS("videoqos"); // last config'd QoS
        STRM_QoS *newq = new STRM_QoS();
        STRM_QoS *nowq = (STRM_QoS *)current_qos;
        int new_rate;

        if (nowq->get_rate() < oldq->get_rate() / 2) {
            // if rate has dropped significantly...

            /* now, do something to set the source codec to run at a
             * lower rate, new_rate - CODE NOT SHOWN HERE
             */

            // adjust binding rate to new codec rate
            newq->set_rate(new_rate);

            // modify QoS, renegotiate and rebind
            b->set_QoS("videoqos", newq, this);
            b->bind();
        }
    }
};
```

The `qos_event()` implementation (which overrides a virtual method defined in class `QoSM`) reacts in an appropriate way to QoS reports from the underlying protocol stack. Note that the above code just illustrates the principle; it is not necessarily intended to represent a realistic policy.

A.4 Establishing and Operating the Binding

Having deployed the provider-side service and implemented the QoS manager, we are ready to establish and manage a `GenericBinding`. Assuming that `InterfaceRef prov` has already been obtained from the previously instantiated `pMediaService` implementation object (using the `getInterfaceRef()` method referred to above), we now create a requirer implementation object, create a suitable binding, customise the latter with the appropriate QoS specification and QoS manager, and set the binding in motion. Note that we specify the use of the active binding style when creating the requirer implementation object. Note also that, thanks to third party binding, the following code can run anywhere in the distributed system; it does not have to run on either the client or the server machine.

```
/* create the requirer implementation object; select 'active binding'
 * semantics; obtain an InterfaceRef from the requirer
 */
rMediaServer *rMS = new rMediaServer(ACTIVE); // create requirer
InterfaceRef *reqr = rMS->getInterfaceRef(); // get its InterfaceRef

/* create the binding */
```

```
GenericBinding *b = new GenericBinding(reqr, prov); // create binding

/* set up QoS and QoS management for the videoqos QoS-group */
STRM_QoS *my_video_qos = new STRM_QoS(8192, 24);
STRM_QoS *my_video_qosm = new STRM_QoS(b);
b->set_QoS("videoqos", my_video_qos, my_video_qosm);

/* allocate resources and start the binding */
b->bind();
```

At this point, the binding is in existence, resources have been allocated, and data is flowing on the active QoS-bindings (in this case, those underlying the *video()* and *audio()* streams). The binding is also ready to have its *video()* stream managed by the *my_video_qosm* object when the underlying *STRM* protocol stack delivers an appropriate QoS report.