

Experiences in Implementing a Distributed Object Platform for Multimedia Applications

Geoff Coulson and Shakuntala Baichoo

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

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

ABSTRACT

Two current trends in distributed computing are the emergence of standardised distributed object platforms such as CORBA, and the increasing use of continuous media data types. This paper describes and evaluates a platform which supports both standard CORBA interactions and continuous media interactions in a fully integrated environment. The platform is implemented as a self-contained support infrastructure (GOPI core) and a separate ‘personality’ that sits on top of the infrastructure and provides a CORBA API. The platform user can create both request/ reply oriented and stream oriented bindings with quality of service specifications that are honoured (as far as possible) by the infrastructure. A framework for quality of service management which involves the monitoring and adaptation of quality levels is also supported. The level of performance attained by the platform provides evidence of the feasibility of natively supporting continuous media in a distributed object platform environment.

Keywords: Multimedia; CORBA; Quality of Service; Middleware; Distributed Systems.

1. Introduction

Two important trends are currently emerging in the field of distributed computing. The first of these is the growing prominence of *distributed object platforms* such as the OMG’s CORBA [1], the Java RMI or Microsoft’s DCOM. These platforms are popular because they offer high level, easy to use, standardised abstractions for request/ reply based distributed interaction. The second trend is the increasing use of continuous media data types such as digital audio and video in distributed applications. Continuous media applications require *stream based* as well as request/ reply based interaction and also require sophisticated support for *quality of service* (QoS) specification and management.

Although the importance of these two trends is universally acknowledged, it is interesting to observe that there is little consensus on how they should relate. Some contributors advocate a *non platform-integrated* approach in which the two trends remain separate and orthogonal. For example, the Internet community has defined the

Coulson, G., Baichoo, S., “Experiences in Implementing a Distributed Object Platform for Multimedia Applications”, *Software Practice and Experience*, Vol 30, pp 663-683, 2000.

RTP/RTCP/RTSP protocol suite [2] to support distributed continuous media applications. However, they do not prescribe any standardised mechanism for general purpose request/ reply control interactions when media streams are embedded in a larger object-based distributed application (although some in the Internet community see CORBA as a useful vehicle for this). Other contributors advocate a *partially platform-integrated* approach which accommodates the two trends in a common distributed object platform environment while stopping short of full integration. For example, the CORBA Telecom SIG defines an architecture [3] in which applications can *control* and *manage* continuous media streams from the standard CORBA environment but have no access to the stream data itself; the latter is carried out-of-band using QoS configurable communications services distinct from those used for CORBA communication. A similar architecture is adopted in Microsoft's Active X.

In this paper, we explore a *fully platform-integrated* approach¹ in which stream based interaction styles are accommodated in a standard distributed object platform environment *and* both stream and request/ reply interactions are supported by a common integrated infrastructure. Advantages of the platform-integrated approach are apparent at both the API and infrastructure levels. At the API level, there is less need for the application programmer to rely on specialised "systems programmers" to provide code that acts directly on continuous media, and there is no need to deal with two separate APIs. In addition, the synchronisation and coordination of continuous media and remote method invocation are easier to program through an integrated API. At the infrastructure level, a platform-integrated platform can make more informed resource management trade-offs which take account of the varied QoS requirements of all interaction types. Furthermore, the common infrastructure can support synchronisation and coordination in a more predictable and efficient manner.

The prototype platform described in this paper is called *GOPI* (for "Generic Object Platform Infrastructure"). GOPI is backwardly compatible with CORBA at the IDL and IIOP levels and, in addition, offers abstractions for continuous media programming and QoS specification and management. At the infrastructure level, the platform features QoS driven resource management and is implemented from the ground up to support both request/ reply and stream based interactions. The platform is structured at the coarsest level of granularity as two sub-systems, both of which are implemented as run-time libraries linked with applications. These are i) *GOPI core*, which provides a generic but low level programming interface, and ii) an *API personality* which provides an object based programming interface at a level of abstraction more suitable for application programmers. Architecturally, multiple distinct API personalities can simultaneously run on top of GOPI core, although only the CORBA based personality mentioned above has been implemented to date.

The remainder of this paper is structured as follows. Section 2 briefly describes the abstract computational model around which the platform has been designed. Section 3 then covers key aspects of the low level GOPI core interface on which the API personality, which is described in detail in section 4, is based. Section 5 then addresses performance issues and section 6 discusses related work. Finally, section 7 offers some concluding remarks.

¹ The integrated approach (although not based on a standardised distributed object platform) was first defined and advocated in [4].

2. Computational Model

The design of our abstractions for multimedia programming, at both the core and API personality levels, is influenced by the computational model of ISO's Reference Model for Open Distributed Processing (RM-ODP) [5]. In particular, we adopt the following characteristics from RM-ODP:

- Interfaces may include *signals* and *flows* in addition to conventional 'interaction points' (i.e. operations). Signals are primitive interaction points which are capable only of either emitting or receiving typed data items; *in* signals receive data items and *out* signals emit data items. Flows are like signals except that they emit/ receive continuous streams of data items according to a specified QoS defined over multiple emissions/ receptions.
- Bindings between interfaces are *first class objects* which themselves support an interface through which the binding can be monitored and controlled. Monitoring operations allow the user of the binding to obtain dynamic information about the binding's QoS, and control operations allow the user to adapt the operation of the binding (e.g. increase bandwidth).
- Bindings can be created and managed by *third party* objects which obtain and bind *interface references* for to-be-bound objects. This facility is particularly useful in the structuring of complex distributed multimedia applications containing many per-media objects.

Our model also differs from RM-ODP in a number of ways. In particular, we believe that the RM-ODP approach to QoS specification, which is to annotate interfaces with definitive QoS specifications, is overly static and prescriptive. Our approach is to defer the specification of QoS until bind time but to provide some structure for this specification by partitioning interaction points into disjoint sets, called *QoS groups*, which will share the same QoS. At bind time, each QoS group is given a dedicated QoS specification and is mapped to a dedicated GOPI core binding which is responsible for delivering this level of QoS.

A final feature of our computational model is that we abandon the traditional *client/ server* terminology for characterising the roles of interfaces in a binding. This is because the connotation of the traditional terminology implicitly links the notion of 'role' with data flow direction which, while unambiguous for operations, is confusing for signals and flows². In place of the traditional terminology we characterise interfaces as either *providers* or *customers* and emphasise that this distinction is orthogonal to the direction of data flow over the binding.

3. GOPI Core

3.1 Internal Structure and Major Abstractions

GOPI core is implemented as the following set of independent modules.

- the *bind* module supports the *iref* abstraction (see below), plus a generic *binding protocol* with QoS negotiation capabilities which is used to establish bindings between irefs using the *comm* framework;

² For example, the 'server' end of a flow binding could either be producing data (e.g. a video on demand service) or consuming data (e.g. a video file server recording from a camera).

- the *comm* module provides a framework for accommodating stacks of *application specific protocols* (see below);
- the *buf/chan* modules respectively manage buffers and provide a ‘real-time’ inter-thread buffer passing service;
- the *thread* module is a ‘real-time’ concurrency package which supports *application schedule contexts*; these enable the support of real-time threads with a range of application specific semantics;
- the *base* module is a collection of generally useful foundation programming classes.

GOPI core is written mainly in C, consists of approximately 12,000 lines of code, and runs on a variety of UNIX platforms. It is comprehensively described in [6] and [7].

Although only a minimally necessary amount of detail is given on GOPI core internals in this paper, *application specific protocols* (ASPs) as supported by the *comm* module require further elaboration. Each ASP embodies a communications protocol and associated QoS management machinery which is tailored to some specific media type. For example, the set of ASPs currently supported includes: a simple stream-oriented ASP for generic continuous media types, more sophisticated ‘adaptive’ ASPs for audio and video, a reliable message oriented multicast ASP and an ASP called *GIOP* that implements the CORBA GIOP/ IOP protocol³. Importantly, ASPs are QoS configurable according to an ASP specific ‘QoS schema’ (i.e. set of QoS parameter types) defined by the ASP developer. QoS schemas are only interpreted (e.g. mapped to resources such as threads/ application scheduler contexts and buffers) by ASPs themselves and are entirely transparent to the rest of GOPI core.

Typically, new ASPs are written by adding value to the services provided by pre-existing, lower level ASPs (and so on recursively). The fact that ASPs can be ‘stacked’ in this way is entirely transparent to GOPI modules other than *comm*. A single ‘top level’ ASP is explicitly chosen by the GOPI core user and this ASP then decides, as a function of its QoS parameters, whether to layer itself straight on to the transport layer or to instantiate and configure some other ASP or ASPs below itself (and so on recursively).

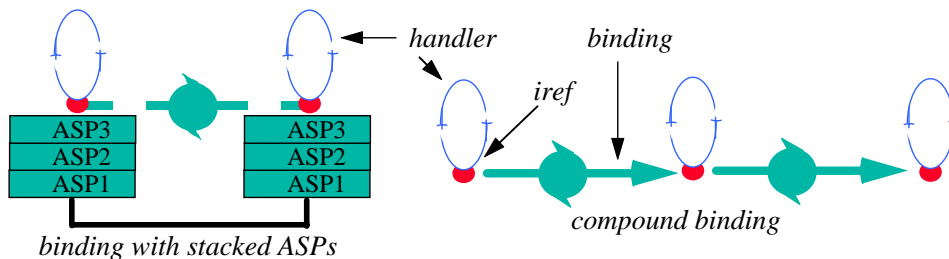


Figure 1: The main GOPI core abstractions

The following sections describe the various abstractions and associated calls exported by GOPI core. The main abstractions are *irefs* (communication endpoints),

³ The *GIOP* ASP only implements the packet header aspects of GIOP. The payload encoding aspect, which defines the on-the-wire presentation syntax used by CORBA stubs/ skeletons, is the concern of the API personality.

handlers (functions attached to irefs and upcalled to obtain or deliver data) and *bindings* (associations between irefs). Figure 1 visualises these abstractions and also illustrates the implementation of a binding as a stack of ASPs, and a compound binding (pipeline type) composed of two primitive bindings.

3.2 Irefs

3.2.1 Iref Creation

Irefs are location independent communication endpoints with two distinct and independent ‘roles’, called *customer* and *provider*, with which they can participate in bindings. Irefts are created with the following call:

```
Iref *bind_irefcreate(Intertype cust_inter, Intertype prov_inter,
                    Handler h, Aspname asp, CharSeq *qos);
```

The two *Intertype*⁴ arguments refer to the *interaction type* of the iref in each of its two roles. The interaction type is one of: *signal*, *flowin*, *flowout*. Bindings involving *flowin* and *flowout* roles are unidirectional whereas bindings involving *signal* roles are bi-directional⁵.

The *h* argument specifies a *handler*; i.e. a C function that is upcalled by GOPI core when data arrives at or leaves an iref. If a null handler is provided, it is still possible to send/ receive data to/ from an iref by means of an explicit interaction primitive. Details of handlers and interaction primitives are given in section 3.3.2 below.

The *asp* and *qos* arguments to *bind_irefcreate()* respectively select and configure an ASP stack to be associated with the newly created iref in subsequent bindings. This information is used only at bind time; no resources are allocated when irefts are created. The *qos* argument is of a generic type (i.e. a character sequence) because, as mentioned above, QoS specifications are interpreted only by ASPs, not by GOPI itself. It is expected that each ASP provides routines for the benefit of GOPI core users that map arguments specified according to their own particular QoS schema to/ from a marshalled character sequence form. For example, the minimal stream ASP *STRM* defines a QoS schema consisting of two integer parameters (packet size and rate) and provides the following routines:

```
strm_marshallqos(int pkt_size; int rate; CharSeq *csqos);
strm_unmarshallqos(CharSeq *csqos; int *pkt_size; int *rate);
```

3.2.2 Interaction with Irefts

Data can be sent/ received to/ from irefts either as a traditional downcall or in an upcall oriented manner. The following calls are available for interacting with bound irefts in the traditional downcall manner.

```
int bind_send(Iref *iref, Role r, Buffer *outgoing_data);
int bind_receive(Iref *iref, Role r, Buffer **incoming_data);
int bind_call(Iref *iref, Buffer *request, Buffer **reply);
int bind_mcall(Iref *iref, Buffer *request, Buffer **replies, int *n);
```

In these calls, the *iref* argument refers to an iref that was created in the local address space and has been bound to one or more other irefts (see section 3.3).

⁴ This is called *FlowType* in the released versions of the GOPI core software and in previous papers. We have renamed it here to better match the terminology adopted in the rest of the paper.

⁵ The property of unidirectionality of signals as defined by RM-ODP (see section 2) is layered above GOPI core in the API personality; see section 4.1. ‘Signals’ at the GOPI core level are bidirectional so as to be able to directly support request/ reply interactions as well as unidirectional RM-ODP signals.

bind_send() and *bind_receive()* both take a *Role* argument which is used to specify whether the data should be sent/ received on bindings in which the customer role or the provider role of *iref* is involved. *bind_call()* is used for request/ reply interactions (on signal irefs) and is implicitly applied to the specified iref's customer role. *bind_mcall()* is used where the iref's customer role is involved in multiple distinct bindings. In this case, a list of **n* reply buffers is returned, one from each of the multiple providers involved in the binding.

Upcall oriented interaction involves handlers as described above. Handlers are typically used to encapsulate stub/ skeleton code provided by the API personality. They have the following prototype:

```
typedef (*Handler)(Iref *iref, int op1, void *op2, Buffer **b);
```

The *iref* argument refers to the iref at which the incoming message was delivered, the *op1* and *op2* arguments are used to help demultiplex the incoming message to the higher layers, and *b* contains the message itself.

The default mode of operation for a handler attached to the *flowout* role of a bound iref is to insert data to be sent into the specified buffer *b*. Similarly, a handler attached to the *flowout* or *signal* role of a bound iref is expected to consume data from the specified buffer. Handlers, however, are also used where the ASPs associated with a binding choose to directly source/ sink data themselves in a so-called *direct connection* [8]. This mode of operation is typically used for reasons of efficiency. For example, an ASP may directly get/put video from/ to a video card (OS permitting) without incurring the overhead of passing the data through the application⁶. In direct connections, the buffer passed from the ASP to the handler typically contains only vestigial 'meta-data' parameters (e.g. an integer representing a framecounter) rather than the data itself. Even a null handler can be useful as it allows higher level code to closely track and synchronise with the flow of data.

Finally, an additional downcall, *bind_invoke()*, is provided:

```
int bind_invoke(Iref *iref, int op1, void *op2, Buffer *request,
**reply);
```

This differs from the downcalls discussed above in that no prior binding needs to be in place. The iref argument, which must be a signal iref with a handler attached, specifies a target to be invoked rather than the local end of a binding. *bind_invoke()* (implicitly) uses the *GIOP* ASP. The remaining arguments correspond directly to the arguments passed to the handler.

3.3 Explicit Bindings

3.3.1 Binding Creation

Bindings are created using the following call:

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

This binds the customer role of *cust* to the provider role of *prov* using the ASP that was associated with *prov* when it was created (note that the provider role of *cust* and the customer role of *prov* are not involved here). *bind_bindreq()* works regardless

⁶ Alternatively in such a situation, the 'direct connection' interaction style may be used only at the source; a handler encapsulating a software decompression algorithm could be used at the sink for increased flexibility.

of the location of the irefs being bound, i.e. *third party binding* is supported. The return value of *bind_bindreq()* is a signal iref on which *bind_invoke()* calls can be made to manage and control the binding; see section 3.3.2.

It is possible for irefs to be simultaneously involved in multiple bindings to form so called *compound bindings*. The simplest case is where an iref is acting as a ‘pipeline element’ and each of its roles are involved in distinct bindings with distinct peers. For example, the following calls establish a pipeline topology (see also section 3.3.3) with the customer role of iref *source* bound to the provider role of iref *pipe_element* and the customer role of iref *pipe_element* bound to the provider role of iref *sink*.

```
control_iref1 = bind_bindreq(source, pipe_element, qos1, h1);
control_iref2 = bind_bindreq(pipe_element, sink, qos1, h2);
```

It is also possible for *each role* of an iref to be bound more than once. For most ASPs the effect of this is to create two distinct bindings as might be expected. Group/multicast ASPs, however, typically implement a ‘join group’ semantic in which the provider role of a multiply-bound iref represents a ‘group manager’, and group members join/ leave the group by binding/ unbinding their customer roles to/ from this iref⁷. For example, given irefs associated with GOPI core’s reliable multicast ASP, the following calls create a group topology consisting of *member1*, *member2*, ..., *memberN*:

```
control_iref1 = bind_bindreq(member1, group_mgr_iref, qos1, h1);
control_iref2 = bind_bindreq(member2, group_mgr_iref, qos1, h2);
...
control_irefN = bind_bindreq(memberN, group_mgr_iref, qosN, hN);
```

Internally, *bind_bindreq()* invokes the bind module’s binding protocol which negotiates a mutually acceptable QoS for the binding (again, GOPI core provides the generic machinery for QoS negotiation while APSs take responsibility for QoS mapping and decision making). 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. If both irefs being bound have handlers attached and their *Intertype* is *flow*, data begins to flow on the binding immediately on creation. The rate of upcall of the *flowout* end’s handler (which in turn determines the flow rate of the binding) is calculated by the ASP as some ASP-specific function of its ASP-specific QoS parameters and provided to GOPI through an internal interface at binding creation time. A dedicated, per binding, thread is used to upcall both the *flowin* and *flowout* end handlers.

The final argument to *bind_bindreq()* specifies an optional *QoS handler*, the function of which is described in the following section.

3.3.2 QoS Management of Bindings

QoS management is supported by the QoS handler argument to *bind_bindreq()*. This handler is periodically upcalled throughout the lifetime of the binding to report on current levels of QoS. The initiators of these QoS reports are the ASP entities

⁷ Note that this model does *not* imply that all data transfers within the group must be channeled through the centralised group manager; ASPs are free to implement data transfer in any appropriate manner (e.g. through network level mechanisms such as IP multicast).

involved in the binding⁸. In implementation, a signal iref with *qos_handler* attached is transparently created by the *bind_bindreq()* call and forwarded to the involved end-systems by the binding protocol (unless a null *qos_handler* argument is given, in which case QoS management is disabled). The internal send/ receive/ call routines of the ASP entities at these end-systems are then able to initiate the sending of a QoS report by returning, as an “out” parameter, a non-null buffer containing a CharSeq encoded instance of the ASP’s QoS schema. Whenever a non-null buffer is returned, the bind module forwards the report using the *bind_invoke()* primitive.

Typically, the GOPI core user will react to the delivery of a QoS report by initiating a *bind_invoke()* call on the signal iref that was returned as the result of *bind_bindreq()*, thus closing the QoS management ‘loop’. Operations are available on this signal iref to start and stop the binding, to renegotiate the QoS of the binding and to destroy the binding.

3.3.3 Interaction Styles and Binding Topologies

A rich set of interaction styles and binding topologies can be supported using appropriate combinations of the interaction primitives and binding types discussed above. Some examples follow.

Signal interaction is trivially implemented using *bind_send()* on a binding between signal irefs. Handlers can be attached to the iref at the receiving end of the binding, or *bind_receive()* can be used as desired. Asynchronous two way communication is also possible as signal bindings are bi-directional. Pipelines of signal bindings (or indeed arbitrary graph topologies) can be created by binding both the customer and provider roles of intermediate signal irefs. Handlers attached to intermediate irefs can use *bind_send()* to forward messages downstream.

Request/ reply interaction is also built by binding signal irefs; a handler is attached at the provider end, and *bind_call()* is used at the customer end. The extension to pipeline and graph topologies is achieved by having the handlers of intermediate irefs issue further downstream *bind_call()* or *bind_mcall()* calls before returning. In the case of *bind_mcall()*, the handler must derive a single aggregate result before returning; this is achieved by collating the multiple replies from the downstream bindings in some application specific manner.

Stream interaction is achieved by binding flow irefs. If a handler is attached to the iref at the *flowout* end of the binding, GOPI periodically upcalls this as explained above. If a handler is not attached to the *flowout* end, the flow rate is simply determined by the frequency with which *bind_send()* is called by the user. The extension to stream pipelines and graphs is achieved, as above, by binding both the customer and provider roles of intermediate irefs. Intermediate irefs with handlers are driven by the rate of message arrival on their incoming bindings rather than by the ASP specified rate as described above (although the periodic behaviour automatically restarts when all upstream bindings have been destroyed). Where there are multiple upstream bindings, messages are received in a first-come-first-served order.

Regarding group/ multicast topologies, although the above examples illustrate one means of achieving these (i.e. by creating multiple point to point bindings), it is clear that in most situations multicast is more efficiently implemented at the ASP level as

⁸ Note, however, that ASPs are not *required* to use the QoS reporting facility. Furthermore, ASPs that do use the facility are entirely unconstrained as to the frequency of reporting.

outlined in section 3.3.1.

4. The CORBA API Personality

4.1 IDL Extensions

We have extended the CORBA Interface Definition Language (IDL) with signals, flows and QoS groups as described above. The extensions are realised as additional operation attributes and backward compatibility with standard CORBA IDL is retained. In the following BNF definition of the extensions, original IDL specification fragments are shown in bold:

```
<op_dcl> ::= [ <op_attr> | <gopi_attr> ]  
            [ <op_type_spec> ] <ident> <param_dcls>  
            [ <raises_expr> ] [ <context_expr> ]  
  
<gopi_attr> ::= [ <qosgroup_attr> ] [ <interaction_attr> ]  
  
<qosgroup_attr> ::= <identifier>  
  
<interaction_attr> ::= "in" | "out" | "flowin" | "flowout"
```

As can be seen, *op_dcl* (i.e. operation declaration) is generalised to represent an ‘interaction point’; i.e. an operation, a signal or a flow. Syntactically, an *op_dcl* is defined as an optional attribute followed by an optional return type (*op_type_spec*) followed by the interaction point name and parameter declaration followed by optional ‘raises’ and ‘context’ expressions. The optional attribute is either the original CORBA attribute (i.e. “oneway”) or *gopi_attr* which subsumes all the extensions (see below). *op_type_spec* has been made optional as signals and flows do not return any value.

The new *in*, *out* and *flowin*, *flowout* attributes respectively identify the associated interaction point as a signal or a flow (as opposed to a conventional operation which has no interaction type attribute). The semantics of signals and flows require that the *param_dcls* section be compatible with the *interaction_attr*. That is, all the parameters of an *in* or *flowin* interaction point must be “in” parameters and all the parameters of an *out* or *flowout* interaction point must be “out” parameters. In addition, *op_type_spec* must be empty for signals and flows.

The *qosgroup_attr* attribute is used to specify which interaction points in the interface should be placed in common QoS groups (see section 2). If no *qosgroup_attr* identifier is given for some or all of the interaction points in an interface, then these are themselves considered to constitute a single ‘default’ QoS group. Backward compatibility is thereby supported as the default case (i.e. multiplexing all operations onto a single QoS group and hence a single underlying binding). For operations and signals, the semantic of multiplexing multiple interaction points on to a binding is first-come-first-served. For flows, a ‘time division multiplexing’ semantic is adopted; i.e. the multiple *flowout* interaction points are upcalled in round robin order so that given a rate of *r* for the binding as a whole, each of *n* multiplexed interaction points sees a rate of *r/n*. The general semantic of QoS groups requires that all the interaction points in a given QoS group have the same interaction type attribute, although operations with the same interaction type attribute may, of course, be placed in separate QoS groups.

To illustrate the use of the IDL extensions, consider the following example.

```

// IDL
interface MediaServer {
    typedef char Title [512]
    sequence<Title> Titles;
    sequence<char> Frame;
    sequence<char> AudioPkt;

    void getTitles(out Titles t);
    void selectTitle(in Title t);
    command in start();
    command in stop();
    event out newTitleAdded(out Title t);
    videoqos flowout video(out Frame v);
    audioqos flowout audio(out AudioPkt v);
};

```

This describes the interface of a ‘media server’ which emits video and audio streams. There are conventional operations to browse the available video titles (*getTitles()*) and to select a particular title (*selectTitle()*). These operations have no associated QoS group and therefore use a default QoS. In addition, there are two *in* signals, *start()* and *stop()*, in a common QoS group (*command*) which are used to turn the flow of video on or off, and there is an *out* signal, *newTitleAdded()*, in its own dedicated QoS group (*event*) which asynchronously notifies when a new title has been added to the server (titles are assumed to be added via a separate management interface). Finally, there are out flows, *video()* and *audio()*, in their own dedicated QoS groups (*videoqos* and *audioqos*) for the transmission of the video and audio themselves.

Two important points are illustrated by this example. Firstly, it is possible to freely mix interaction types in an interface; i.e. the specification of interaction points is orthogonal to their partitioning into interfaces. Secondly, flows and signals use typed parameters in just the same way as normal operations. The choice of parameter types for flow interactions is governed by the degree of access required by the application to the real-time media. The above example only permits the application to access video frames as unstructured character arrays, but the full power of IDL syntactic structure definition is available to applications that require detailed access to complex media structures such as MPEG packets. Alternatively, in cases where the application wishes to delegate the production/ consumption of media to GOPI core direct connections, simple ‘meta data’ flow parameters such as integer frame counters can be used (see section 3.2.2).

4.2 Implementing Interfaces

Given an input file *Foo.idl*, the IDL compiler generates the following output files: *pFoo.cc*⁹, *pFoo_i.cc*, *cFoo.cc*, *cFoo_i.cc* and *Foo.h*. *pFoo.cc* and *pFoo_i.cc* together implement the provider side of a binding (by implementing a class *pFoo*) while *cFoo.cc* and *cFoo_i.cc* together implement the customer side (by implementing a class *cFoo*). *Foo.h* contains standard definitions and is included by all the other files.

Because signal and flow directionality is orthogonal to interface role, both provider and customer side files must implement both ‘stub’ and ‘skeleton’ functionality. On the provider side, the application programmer implements all the interaction points specified in the IDL file as methods of class *pFoo*. This is done by expanding IDL compiler generated templates in *pFoo_i.cc*. Additional methods of

⁹ C++ is currently the only language supported.

class *pFoo* are then defined automatically by the IDL compiler in *pFoo.cc*. These methods are as follows:

- i) There is one ‘demultiplexing’ method for each QoS group of operational interaction points (including a ‘default’ demultiplexing method for any operational interaction points not associated with any QoS group). These methods contain code to demultiplex incoming messages to the target IDL method implementation together with skeleton code to unmarshal requests and remarshal replies. They are used as GOPI core handlers and are attached to the QoS group’s dedicated iref (each QoS group has a dedicated iref as there is a one to one mapping from QoS groups to GOPI core bindings).
- ii) There is one demultiplexing method for each QoS group of *in* and *flowin* interaction points. These are similar to the above except that they do not need to contain remarshalling code as there are no replies.
- iii) There is one method for each QoS group of *flowout* interaction points. These methods are again used as GOPI core handlers and are upcalled periodically at the *flowout* end of the binding as described in section 3.3.2. Each method calls the associated flow implementations (one flow implementation call per upcall in round robin order) and marshalls the resulting “out” parameter values into GOPI core buffers before returning.
- iv) There is a further method for each QoS group of the form: *set_<qosGroupName>(QoS *qos)*. These methods are used to associate a default QoS with the provider class (see section 4.3).
- v) There is a method *getInterfaceRef()* which returns a location independent *interface reference* relating to *pFoo* (again, see section 4.3).

As an example, the automatically generated provider side class methods produced from the above *MediaServer* interface would be as follows:

```
// C++
interfaceRef *MediaServer::
getInterfaceRef();

// Handler methods...

// upcall getTitle(), selectTitle()...
static void MediaServer::
default_op_demux(Iref *iref, int op1, void *op2, Buffer **b);

// upcall start(), stop()...
static void MediaServer::
command_demux(Iref *iref, int op1, void *op2, Buffer **b);

// upcall newTitleAdded()...
static void MediaServer::
event_upcall(Iref *iref, int op1, void *op2, Buffer **b);

// upcall video()...
static void MediaServer::
videoqos_upcall(Iref *iref, int op1, void *op2, Buffer **b);

// upcall audio()...
static void MediaServer::
audioqos_upcall(Iref *iref, int op1, void *op2, Buffer **b);

// QoS set calls for each QoS group...
```

```

void MediaServer::set_command(QoS *qos);
void MediaServer::set_event(QoS *qos);
void MediaServer::set_videoqos(QoS *qos);
void MediaServer::set_audioqos(QoS *qos);

```

Customer side code generation is broadly similar to the provider side scheme just described but differs in the following ways. Firstly, the programmer only provides methods in *cFoo_i.cc* for flow interaction points and for *out* signal interaction points. In the case of *in* signals and operations, appropriate proxy methods are automatically defined in *cFoo.cc*. Secondly, the *set_<qosGroupName>(QoS *qos)* methods are not generated.

4.3 QoS Specification and Binding

4.3.1 QoS Classes

To allow the specification of QoS at the API personality level, ASP implementers provide per-ASP C++ classes, conventionally named *<ASP name>_QoS*, that wrap and extend the basic QoS specification routines provided at the core level (see section 3.2.1). This is in conformance with the general GOPI approach of associating individual QoS schemas with individual ASP implementations. More specifically, each ASP implementation provides a C++ class which derives from the following base class:

```

class QoS {
    QoS();
    ASP getAsp();
    CharSeq *toCharSeq();
    static QoS *toQoS(CharSeq *c); // class method
};

```

The given constructor method is used to create QoS objects of a suitable default type. It is expected that derived QoS classes will additionally provide ‘custom’ constructors which take parameters relating to the QoS schema of their associated ASP. For example, the (single) custom constructor for the *STRM* ASP (see section 3.2.1) is as follows:

```

STRM_QoS(int pkt_size, int rate);

```

It is also conventional to provide associated *get* and *set* methods which can later be used to retrieve and reset these parameters.

The remaining methods in the QoS class function as follows: *getAsp()* returns the identifier of the ASP associated with the QoS object, *toCharSeq()* returns a character sequence representation of the QoS specification, and *toQoS()* is a class method that generates a new QoS object corresponding to the given character sequence representation.

QoS objects are used in two contexts: i) to specify a ‘default provider QoS’ for the provider object and ii) to potentially override the default provider QoS at bind time (see section 4.3.3). To specify a default provider QoS, the programmer passes appropriately configured QoS objects to the provider class’s *set_<qosGroupName>(QoS *qos)* methods. For example:

```

// C++
pMediaServer *ms = new pMediaServer();
STRM_QoS *my_video_qos = new STRM_QoS(8192, 24);
ms->set_video(my_video_qos);

```

Internally, the provider class's `set_<qosGroupName>()` methods call the QoS object's `toCharSeq()` and `getAsp()` methods to obtain values suitable for passing to GOPI core's `bind_irefcreate()` routine. In this way, an appropriately configured core level iref is created for each QoS group in the provider class which has its `set_<qosGroupName>()` method called. If one or more `set_<qosGroupName>()` methods remain uncalled before the provider class is bound, then GOPI core bindings for the related QoS groups will not be created at bind time and thus interactions on interaction points in those QoS groups will not be possible.

4.3.2 InterfaceRefs

InterfaceRefs are location transparent 'proxies' used to represent their parent customer and provider objects in remotely created bindings (including third party bindings). To permit InterfaceRefs to be freely passed around the distributed system, their IDL definition is transparently available to all IDL files. This means that stubs and skeletons for InterfaceRefs are freely available to all applications.

The IDL definition of InterfaceRefs is as follows:

```
// IDL
struct Irefrec {
    Iref iref;
    int asp;
    CharSeq provider_qos; // null for customer irefs
    String qosgroupname;
};
struct InterfaceRef {
    String type; // used for semantic checking of bind requests
    sequence<Irefrec> iref;
};
```

InterfaceRefs are obtained from their parent customer or provider objects by calling the `getInterfaceRef()` method defined on these objects.

4.3.3 The Binding Class

Bindings are represented by objects of class *Binding* which is defined as follows:

```
// C++
class Binding {
    Binding(InterfaceRef *customer, InterfaceRef *provider);
    ~Binding(); // destroy the binding
    int set_QoS(String qosgroupname, QoS *qos, QoSManager *qos_manager);
    QoS *get_QoS(String qosgroupname);
    int bind();
    int pause();
};
```

Bindings are established in three phases: firstly, a Binding object is created for (type-compatible) customer and provider InterfaceRefs; then the Binding object is customised in various ways using `set_QoS()` calls for each QoS group; finally, `bind()` is called to allocate resources for the binding and enable the flow of data. The `set_QoS()` method is used to override the default provider QoS and, optionally, to install a *QoS manager* (see section 4.4).

The implementation of `bind()` invokes GOPI core's `bind_bindreq()` routine once for each QoS group that the customer and provider InterfaceRefs have in common to create per QoS group GOPI core bindings. More precisely, `bind_bindreq()` is invoked on the first call only; subsequent calls of `bind()` transparently invoke the GOPI core

QoS renegotiation machinery (see section 3.3.2) on already existing GOPI core bindings.

Once a binding has been successfully established, its QoS can be dynamically altered at any time by re-calling *set_QoS()* to propose new levels of QoS and then re-calling *bind()*. The *pause()* method is used to stop/ restart the flow of data on a binding.

4.4 QoS Management

The GOPI approach, both at the core and API personality levels, is to associate individual QoS management policies with individual ASP implementations. To extend this approach from the core level to the API personality level, per-ASP *QoS manager* classes are derived from the following base class:

```
// C++
class QoSManager {
public:
    QoSManager(Binding *b); // create QoS manager for a specific binding
    void qos_event(QoS *current_qos);
    Binding *this_bind; // attribute to remember associated binding

private:
    static int
    qos_handler(Iref *iref, int op1, void *op2, Buffer *b);
};
```

The derived classes, which are conventionally named as *<ASP name>_QoSManager*, may be provided by ASP implementers or, alternatively, they may be provided by application programmers to implement application specific QoS management policies for a given ASP.

Instances of derived QoS manager classes are registered with Binding objects using the *set_QoS()* method in the Binding class (a null *QoSManager* argument is passed if the Binding does not require QoS management). At registration time, the private static method *qos_handler()* is installed as a QoS handler with the QoS group's associated iref. Subsequently, when a GOPI core level QoS report arrives at the Binding object, *qos_handler()* translates the CharSeq contained in its buffer argument to a QoS object using the *<ASP name>_QoSManager.toQoS()* method and then passes this QoS object to an upcall of *qos_event()*. *qos_event()* implementations embody some policy on how to react to notifications of altered QoS. As with QoS handlers at the GOPI core level, they typically close the QoS management 'loop' by re-calling *set_QoS()* and *bind()* and thus renegotiating the QoS of the binding.

4.5 Binding and QoS Management Example

This section offers a simple example of the use of the API personality's binding and QoS management facilities. In the following, a binding is created between a customer object and a provider object that implements the *MediaServer* interface specified in section 4.1. The *STRM* ASP is used to support the *videoqos* QoS group, and an application specific QoS manager is defined to manage the *video()* flow that uses this QoS group. The other QoS groups use the default provider QoS.

Firstly, the *STRM*/ application specific QoS manager class is implemented in C++:

```
// C++
class STRM_QoSManager : QoSManager {
    STRM_QoSManager(Binding *b) : QoSManager(b);
};
```

```

    void qos_event(QoS *current_qos);
};

void STRM_QoS::qos_event(QoS *current_qos)
{
    Binding *b = this_bind; // initialise from internal attribute
    STRM_QoS *oldq = b->get_QoS("videoqos"); // last config'd QoS
    STRM_QoS *newq = new STRM_QoS();
    STRM_QoS *nowq = (STRM_QoS *)current_qos;

    if (nowq->get_rate() < oldq->get_rate() / 2) {
        newq->set_rate(nowq->get_rate() - 1);
        // adjust video encoding to use smaller packets...
        newq->set_pkt_size(nowq->get_pkt_size() / 2);
        b->set_QoS("videoqos", newq, this); // modify QoS
        b->bind(); // rebind
    }
}

```

The body of *qos_event()* contains code to react in some appropriate way to *STRM* QoS reports (note that the above code just illustrates the principle; it is not intended to represent a realistic policy).

Having implemented the QoS manager, all the machinery is in place to establish and manage the binding. Firstly, we assume that *InterfaceRefs* have been obtained from *cMediaServer* and *pMediaService* objects (which are assumed to be already in existence) and delivered to the site at which the binding is to be created and managed. The next step is then to create a suitably configured *STRM_QoS* object:

```
STRM_QoS *my_video_qos = new STRM_QoS(8096, 24);
```

Following this, the binding and its associated QoS manager can be created, initialised and set in motion. We use the default provider QoS and no QoS management for the *command*, *event*, *audioqos* and *default_op* QoS groups:

```

Binding *b = new Binding(cust, prov);
b->set_QoS("videoqos", my_video_qos, new STRM_QoS(b));
b->bind();

```

At this point, the binding is in existence, resources have been allocated, and data is flowing on the *video()* flow. The binding is also ready to have the QoS of its *video()* flow renegotiated by the *my_video_qosm* object when the *STRM* ASP delivers an appropriate QoS monitoring report.

4.6 Extended Bindings

The binding facilities described above support the binding of any pair of compatible *InterfaceRefs* with a specified ASP stack and QoS, and also support QoS management (in cooperation with ASPs). However, as currently implemented, the facilities do not fully exploit the architectural potential of explicit bindings. For example, it should be possible to encapsulate media filtering or sophisticated QoS management functionality *inside* an explicit binding object and thereby provide higher level functionality to applications [5]. It should also be possible to provide *open bindings* which allow applications to access internal components (objects) and individually replace or adapt them at run time [9].

With the current platform, it is possible to implement such 'extended' bindings, but only at the GOPI core level. For example, given the existence of a set of distributed 'factories' which create instances of *irefs* associated with handlers encapsulating filtering code, a specialised *Binding* class could internally instantiate a

GOPI core pipeline consisting of such irefs. The problem with this approach is that, in a way reminiscent of the partially platform-integrated architecture discussed in the introduction, it requires application programmers to rely on advanced “systems” programmers to provide suitable filter handlers/ binding classes. To enable ordinary application programmers to provide such facilities, it would be necessary to extend the API personality to support features such as group bindings and pipelines. This would permit extended bindings to be internally structured in terms of CORBA level objects and would also provide the necessary support for open binding implementation.

5. Performance

In this section the performance of the full platform is evaluated and compared with i) the commercial market leader CORBA implementation, Iona’s Orbix 2.3MT (to evaluate the relative performance of operation invocation), ii) a minimal TCP/IP socket program (to evaluate the relative performance of stream interaction) and iii) GOPI core (to evaluate the overhead of the API personality). The tests were all run on a single machine, a SPARCstation 5 running SunOS 5.5.

The Orbix/ GOPI operation invocation tests involved IDL interfaces with a single operational interaction point. The three tests undertaken employed an operation with, respectively, no parameters, a 1K array parameter and an 8K array parameter. GOPI was configured to use IOP (i.e. the *GIOP* ASP running over a TCP/IP transport) so as to be directly comparable with Orbix. In the socket/ GOPI stream interaction tests, GOPI used similar parameters to the above but with a flow interaction point. GOPI used the *FRAG*¹⁰ ASP in the stream tests. *FRAG*, which was layered on top of TCP/IP, simply fragments large messages into packets of a size defined by its QoS specification. In all the tests, the packet size QoS parameter was configured so that the whole message fitted into a single packet and no fragmentation took place. To make it more directly comparable to the functionality of GOPI, the socket program was written to be able to send and receive multiple streams (by employing the *poll()* system call before each call of *recv()*). However, only a single stream was involved in the tests.

	Operation invocations (calls/sec)			Stream interactions (pkts/sec)		
	0 bytes	1K bytes	8K bytes	0 bytes	1K bytes	8K bytes
GOPI core	529	487	314	1149	1062	532
GOPI	506	447	255	1028	1014	395
<i>overhead</i>	<i>4.54%</i>	<i>8.94%</i>	<i>23.13%</i>	<i>11.77%</i>	<i>4.73%</i>	<i>37.68%</i>
Orbix	97	94	78	N/A	N/A	N/A
Socket	N/A	N/A	N/A	1539	1210	579

Table 1: Performance Measures

¹⁰ For efficiency, *FRAG* demultiplexes incoming messages on the basis of an integer in the packet header as opposed to a string as is used in *GIOP*. Its functionality is essentially similar to that of *GIOP* except that it also supports stream mode communications.

In terms of the operation invocation tests, the results in Table 1 clearly show GOPI's superiority over Orbix 2.3MT. This demonstrates that the additional facilities offered by GOPI are not negatively impacting its fundamental functionality. The results are also encouraging in the stream interaction case. Despite its far richer functionality and correspondingly higher overhead, GOPI is not far behind the baseline socket program in terms of stream throughput. The overhead for 0 byte payloads is 49% but in the more relevant 1K case it is only 19%. The 8K case, however, is anomalous as we would expect the percentage overhead to be even further reduced whereas in fact is higher at 46%. The cause of this anomaly can be traced to the similarly anomalous figure of 37.68% for the GOPI/ GOPI core comparison in row three of the table which, in turn, can be traced to the effects of inefficient stubs/ skeletons which do not scale in performance with the size of their data. Evidence for this diagnosis can be seen in the fact that GOPI incurs a greater relative deterioration in performance between the 1K and 4K cases than does Orbix. The relative deterioration (for the operation invocation tests) is 42% for GOPI and only 17% for Orbix. The conclusion seems to be that GOPI core is performing excellently but that there is room for improvement in the stub/ skeleton area.

Although they are not yet implemented, we have therefore identified a number of potential stub/ skeleton optimisations which should address these deficiencies, mainly by avoiding redundant copy operations. Firstly, when dealing with arrays of basic types, stubs can allocate a GOPI core buffer of the (*a priori* known) required size and simply pass pointers up to the *out* flow implementation. The same strategy could be used in skeletons where the parameters fit into single contiguous buffers¹¹. Secondly, structured data type transfers can be optimised in cases where the sender and receiver share a common machine endian type and language environment (this can easily be determined at bind time in our connection-oriented design). In such cases, skeleton copies can be avoided by simply assuming that the memory layout of the datatype in the buffer is directly usable by the receiver. It should even be possible to apply this approach to pointer based datatypes by appropriately patching pointers.

Another, already available, means of reducing/ eliminating stub/ skeleton overheads is to employ the *direct connection* approach for flows as discussed in section 3.2 and 4.1. This, however, does not address cases where applications need to process real-time media which, of course, is a major motivation for the design of GOPI.

6. Related Work

Recent work in the OMG's CORBA forum has recognised the need to support continuous media 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' [3] has addressed the need for streams in CORBA, and the Realtime SIG's (ongoing) 'Realtime CORBA' specification [10] 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

¹¹ Clearly, however, there are semantic implications here for the application programmer who must be aware of ownership conventions for the buffer memory.

streams. The Realtime CORBA activity defines a number of concepts also found in GOPI core (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 yet exists.

TAO [11] is a CORBA 2.0 compliant platform that runs on real-time operating systems and has been a significant influence on the OMG's Realtime CORBA activity. TAO has been primarily designed and optimised for hard real-time applications such as avionics. In particular, it features a real-time message scheduling service that can provide deterministic temporal guarantees for operation invocations (given a real-time OS infrastructure) by avoiding priority inversion and non-determinism. It adopts the *partially platform-integrated* philosophy in dealing with multimedia; in particular, it implements the OMG's Control and Management of A/V Streams specification for multimedia streams [12]. In contrast, GOPI is designed to provide fully platform-integrated soft real-time/ multimedia support in a conventional OS/ network environment and emphasises flexible user services rather than hard determinism.

The European Commission funded ReTINA project has designed a CORBA platform featuring streams and QoS extensions [13]. The architecture is based on a clean separation between i) ORB support mechanisms such as interface reference management, threads, buffers etc., and ii) *binding classes* which provide communications services tailored to particular applications. 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 CNET, France Telecom is continuing the main themes of the ReTINA research in the context of a new Java ORB called Jonathan [14].

The DIMMA project [14] at APM Ltd., Cambridge, UK has also addressed 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 core is achieved largely through the ASP and application scheduler context frameworks which are lacking in DIMMA. DIMMA also lacks a run-time QoS management facility.

7. Conclusions

This paper has described the design of a platform-integrated multimedia distributed object platform as defined in the introduction. The platform features IDL-level multimedia flows and QoS specification/ management facilities for both flows and standard request/ reply invocations. To support QoS in an extensible way, the platform natively provides only low level resource management services (in GOPI core) and relies on 'plug in' ASPs to build application tailored QoS support on top of these services. ASPs define their own QoS schema, QoS-to-resource mapping strategy, QoS monitoring strategy and QoS renegotiation strategy. They are visible both at the GOPI core level (which provides significant run-time support for ASP implementations) and at the API personality level where they are configured through

per-ASP *QoS* objects.

Currently, we are extending the API personality to incorporate more of the services available at the GOPI core level, particularly group bindings and pipelines, to facilitate the implementation of 'extended' bindings in the sense of section 4.6. Simultaneously, we have begun to develop an alternative API personality based on the Python language [16]. Because Python is an interpreted language with advanced reflective capabilities [17], using it as a wrapper around GOPI core services has great potential for exposing the flexibility of the platform in a principled manner.

Finally, although our ORB does not fully implement the CORBA architecture (e.g. it omits Interface and Implementation repositories, the Dynamic Invocation Interface and the Portable Object Adapter) our implementation and performance evaluation provide evidence that the platform-integrated approach is at least feasible. However, a more detailed performance analysis of the platform following optimisations, particularly in the stub/ skeleton area, would be required before any firm conclusions can be drawn. It would also be useful to be able to report on experiences with non-trivial multimedia application development in the GOPI environment. All these issues will be addressed in our future work.

References

- [1] The Common Object Request Broker: Architecture and Specification 2.2, available at <http://www.omg.org/>
- [2] Schulzrinne, H., Casner, S., Frederick, R., Jacobson, V., Internet RFC 1889: "RTP: A Transport Protocol for Real-Time Applications", 1996.
- [3] Control and Management of Audio Visual Streams, OMG Document number telecom/97-05-07; available at <http://www.omg.org/>.
- [4] Anderson, D.P., Tzou, S.Y., Wahbe, R., Govindan R. and M. Andrews, "Support for Continuous Media in the DASH System", Proc. 10th International Conference on Distributed Computing Systems, Paris, May 1990.
- [5] ITU-T, ISO/IEC Recommendation X.902, International Standard 10746-2, "ODP Reference Model: Descriptive Model", January 1995.
- [6] 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.
- [7] Coulson, G., "A Configurable Multimedia Middleware Platform", IEEE Multimedia, Vol 6, pp 62-76, No 1, January - March 1999.
- [8] Coulson, G., Campbell, A and P. Robin, "Design of a QoS Controlled ATM Based Communication System in Chorus", IEEE Journal of Selected Areas in Communications (JSAC), Special Issue on ATM LANs: Implementation and Experiences with Emerging Technology, 1995.
- [9] Blair, G.S., Coulson, G., Davies, N., Robin, P. and Fitzpatrick, T., "Adaptive Middleware for Mobile Multimedia Applications", Proc. 7th International Conference on Network and Operating System Support for Digital Audio and Video (Nossdav'97), St Louis, Missouri, USA., 1997, pp 259-273.

- [10] 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/>.
- [11] Schmidt, D.C., "The Design of the TAO Real-Time Object Request Broker", <http://www.cs.wustl.edu/~schmidt/new.html#corba>.
- [12] 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.
- [13] 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.
- [14] Dumant, B., Horn, F., Dang-Tran, F. and Stefani, J.-B., "Jonathan: an Open Distributed Processing Environment in Java", Proc. Middleware '98, The Lake District, England, November 1998.
- [14] 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.
- [16] Watters, A., van Rossum, G., and Ahlstrom, J., "Internet Programming with Python", Henry Holt (MIS/M&T Books), 1996.
- [17] Blair, G.S., Coulson, G., Robin, P. and M. Papathomas, "An Architecture for Next Generation Middleware, Proc. Middleware '98, The Lake District, England, November 1998.