

A Distributed Object Platform for Multimedia Applications

Geoff Coulson and Shakuntala Baichoo
Lancaster University
Lancaster LA1 4YR
UK
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. The level of performance attained by the platform provides evidence of the feasibility of natively supporting continuous media in a distributed object platform environment

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 integrated* approach in which the two trends remain separate and orthogonal.

For example, the Internet community has defined the 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 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 integrated* approach 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 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, an 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 outlined in this paper is 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.

This paper focuses on the API personality aspect of our platform; GOPI core has been extensively described elsewhere [4]. The remainder of this paper is structured as follows. Section 2 describes the API personality. Section 3 then presents some performance results and section 4 discusses related work before section 5 offers some concluding remarks.

2. The CORBA API Personality

2.1 Computational Model and IDL

We have extended the CORBA Interface Definition Language (IDL) with *signals*, *flows* and *QoS groups* as described below. 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> ::= [ <interac_attr> ]
              [ <qosgroup_attr> ]

<interac_attr> ::= "in" | "out" |
                 "flowin" | "flowout"

<qosgroup_attr> ::= <identifier>

```

As can be seen, *op_dcl* (i.e. operation declaration) is generalised to represent an ‘interaction point’. Following RM-ODP [5] we recognise three types of interaction point:

- *operations*, which are the standard request/reply interactions supported by CORBA;
- *signals*, which 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*, which are like signals except that they emit/ receive continuous streams of data items according to a specified QoS defined over multiple emissions/ receptions.

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 *interac_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*. QoS groups are disjoint sets of interaction points which will share the same QoS at bind time (at bind time each QoS group is given a dedicated QoS specification and mapped to a dedicated GOPI core binding which is responsible for delivering this level of QoS).

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 GOPI core 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* [4], simple ‘meta data’ flow parameters such as integer frame counters can be used.

2.2 The Programmer's Perspective

Although it is not possible within the given space constraints to comprehensively describe the programmer's view of the platform, this section attempts to give an impression of this by means of a short example.

The programmer's first task is to *implement* the various interaction points specified in the IDL file. For signals and flows, implementations are provided for both the source and the sink ends of the flows. It is the responsibility of the programmer to provide implementations and of GOPI to upcall these implementations when a binding has been created.

In the following, a binding is created between the interface of a client object (*sink*) and a server object (*source*) that implements the `MediaServer` interface specified in section 2.1.

```
Binding *b = new Binding(sink, source);
b->set_QoS("videoqos", my_video_qos,
          new STRM_QoSM(b));
b->bind();
```

The first line creates a *binding object* [5] which encapsulates all the internal complexity of the underlying binding. Following this, `set_QoS()` method calls are made on the binding, one for each QoS group in the interface associated with the binding (i.e. `MediaServer`). It is possible to bind only a subset of the available QoS groups by omitting the calls related to unwanted QoS groups. The first argument to `set_QoS()` identifies the target QoS group as a string, the second is a *QoS specification* and the third is a *QoS manager* object. The QoS specification and QoS manager objects are of classes related to one or other of the underlying *application specific protocols* (ASPs) supported by GOPI core [4].

Following the configuration of each QoS group, the programmer enables the flow of data on the binding by calling the `bind()` method. For signal and flow QoS groups, this causes data to immediately start flowing between the programmer provided interaction point implementations (for operational QoS groups, the programmer initiates calls in the usual way).

The QoS manager object implements a `QoS_event()` method which is upcalled by GOPI core whenever the binding's ASP detects a deviation from the initially specified QoS. It is then possible for the `QoS_event()` method (which can be overridden by the programmer) to renegotiate QoS by re-calling `set_QoS()` and `bind()` on the binding object.

3. Performance

The performance of GOPI core has already been evaluated and reported in the literature [4]. In this section the performance of the full platform is evaluated and compared with the commercial market leader CORBA implementation (Iona's Orbix 2.3MT) as well as with GOPI core. The following tests were all run on a single machine: a SPARCstation 5 running SunOS 5.5. The tests for GOPI and Orbix involved IDL interfaces with a single interaction point. The operation invocation tests for GOPI core used (respectively) a void operation with no parameters, a 1K array parameter and an 8K array parameter. For GOPI with the CORBA API personality, the stream interaction tests used similar parameters applied to a flow interaction point. For Orbix, which of course does not support flows, oneway operations with similar parameters were used. None of the test applications touched their parameters in any way.

The figures in Table 1 clearly demonstrate GOPI's superiority over Orbix 2.3MT in terms of raw performance. More significantly, they also show, in the % row, the overheads of the full platform relative to GOPI core. These are acceptable for small messages (the higher overhead for the 0 byte flow interaction test is an unexplained anomaly) but become increasingly significant for larger messages. Clearly, the overhead of stubs/ skeletons is a major factor here. The fact that GOPI incurs a greater relative deterioration in performance between 1K and 4K message sizes than does Orbix is a further pointer to inefficient stubs/ skeletons. The relative deterioration here for GOPI is 42% (for operations) and 61% (for streams) as against 17%/ 54% for Orbix.

Packet size	Op. InvoCs (calls/sec)			Flow pkts (pkts/sec)		
	0	1K	8K	0	1K	8K
GOPI core	529	487	314	1149	1062	532
GOPI	506	447	255	1028	1014	395
%	4.5	8.9	23.1	11.7	4.7	37.6
Orbix	97	94	78	175	171	78

Table 1: Performance Measures

Although they are not yet implemented, we have 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. Data can then be marshalled directly into this buffer. The same trick could be used in skeletons where the parameters fit into single contiguous buffers. 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 as GOPI core bindings are connection-oriented and employ a connection handshake). 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.

4. 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 [6] is addressing the need for more general real-time functionality. As stated in section 1, 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 (which is 'partially integrated' in the terminology of section 1) is to transport stream data out of band and only use the ORB for control and management of streams.

TAO [7] 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. TAO adopts the partially integrated philosophy in dealing with multimedia; in particular, it implements the OMG's Control and Management of A/V Streams specification for multimedia streams [8]. In contrast to TAO, GOPI is designed to provide integrated soft real-time/ multimedia support in a conventional OS/ network environment and emphasises flexible user services rather than hard determinism.

The European Commission funded ReTINA project has designed two separate CORBA platforms featuring

streams and QoS extensions [9], [10]. The architecture of these platforms 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, the ReTINA platforms focus 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 [11].

5. Conclusions

This paper has described the design of an 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.

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 integrated multimedia distributed object platform 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 from http://www.omg.org/library/schedule/AV_Streams_RTF.htm
- [4] 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.

[5] ITU-T, ISO/IEC Recommendation X.902, International Standard 10746-2, "ODP Reference Model: Descriptive Model", January 1995.

[6] 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/>.

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

[8] 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.

[9] Dang Tran, F., Perebaskine, V., Stefani, J.B., Crawford, B., Kramer, A and Otway, D., Binding and Streams: the ReTINA Approach, Proc. TINA 96, Heidelberg, Germany, September 1996. Available at: <http://www.uk.infowin.org/ACTS/RUS/PROJECTS/ac048.htm>

[10] 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, Springer Verlag, London, England, pp141-156.

[11] 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.

