Supporting Adaptive Multimedia Applications through Open Bindings

Tom Fitzpatrick, Gordon S. Blair, Geoff Coulson, Nigel Davies and Philippe Robin

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

E-mail: [tf, gordon, geoff, nigel, pr]@comp.lancs.ac.uk

Abstract

In order to support multimedia applications in mobile environments, it will be necessary for applications to be aware of the underlying network conditions and also to be able to adapt their behaviour and that of the underlying platform. This paper focuses on the role of middleware in supporting such adaptation. In particular, we investigate the role of open implementation and reflection in the design of middleware platforms such as CORBA. The paper initially extends CORBA with the concept of explicit binding, where path of communication between objects is represented as first class objects. We then introduce the concept of open bindings which support inspection and adaptation of the path of communications. An implementation of open bindings is described, based on the Ensemble protocol suite from Cornell University.

1. INTRODUCTION

Future computer systems will consist of end-systems which will be either disconnected, weakly connected by low speed wireless networks such as GSM, or fully connected by fixed networks ranging from Ethernet to ATM. Furthermore, the level of connectivity will vary over time as a consequence of the mobility of the modern computer user (see figure 1). Even when connected to a particular network, fluctuations in throughput and delay may be experienced due to congestion, e.g. as witnessed in the Internet.

To cope with such variations, it is important that systems can *adapt* to the quality of service (QoS) offered by the network. Such adaptation can take place at a variety of levels in the system, e.g. in the communications stack, in the operating system or in the application. In this paper, we are concerned with support for adaptation in the middleware platform, i.e. the layer of software above the operating system, which offers a platform independent programming model and hides problems of heterogeneity and distribution. More specifically we consider the design of middleware platforms which (i) allow the application to *inspect* the current level of QoS at various points of the system, and (ii) enable applications to dynamically *adapt* their behaviour or the behaviour of the underlying platform in response to changes in QoS. We are particularly interested in adaptation as required by multimedia applications.

A range of middleware technologies is now available, including CORBA, DCE, and DCOM. In addition, ISO have recently completed an international standard defining a Reference Model for Open Distributed Processing (RM-ODP); this standard provides a framework for the development of middleware platforms. In this paper, we focus on the CORBA platform from OMG, although many of the arguments could be applied to other platforms. CORBA provides an environment whereby objects can interact in a distributed environment. Objects are defined in a language and platform independent manner through an Interface Definition Language (IDL). An Object Request Broker enables clients to issue requests on an object; the ORB locates the object, transmits the request, prepares the object implementation for receiving and processing the request, and conveys results back to the client. (Further details of CORBA can be found in [2].)

A major problem with CORBA is that the architecture adopts a traditional black box approach whereby the implementation of the platform is hidden from the application. Until recently, this has not been a great problem. Indeed, it could be argued that this is a highly desirable property of a middleware platform. With the advent of mobile (multimedia) computing, however, such an approach is untenable; in such environments, it is essential to have (selective) access to the underlying implementation. To achieve this, we adopt concepts from open implementation [14] and reflection [16]. In this paper,

we focus on the use of such techniques to enable inspection and adaptation of the path of communication between interacting objects. In other research at Lancaster, we also consider the use of such techniques in other aspects of a middleware platform (e.g. concurrency control, thread scheduling and real-time synchronisation).

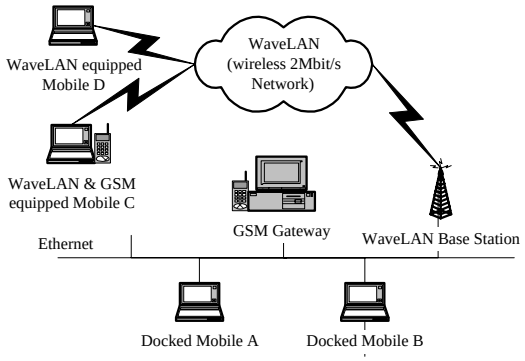

Figure 1: Mobility Scenario

The paper is structured as follows. Section 2 presents some background material on open implementation and reflection. Section 3 then considers our approach to supporting multimedia in CORBA, focussing on the concept of explicit bindings. Following this, section 4 presents an extension to explicit bindings, referred to as open bindings. The role of open bindings in supporting inspection and adaptation is then considered in section 5, with section 6 presenting a short example. Section 7 examines the implementation of the extended middleware platform, featuring the use of the configurable protocol stack, Ensemble. Section 8 examines some related work and section 9 presents some concluding remarks.

2. BACKGROUND ON OPEN IMPLEMENTATION AND REFLECTION

The concept of open implementations has recently been investigated by a number of researchers, most notably Kiczales et al at Xerox PARC [14]. The goal of this work is to overcome the limitations of the black box approach to software engineering and to open up key aspects of the implementation to the application. This must however be achieved in such a way that there should be a *principled* division between the functionality they provide and the underlying implementation. The former can be thought of as the base interface of a module and the latter as a meta-interface [20].

The role of reflection is then to provide a *principled* means of achieving open implementation. In a reflective system, the meta-level interface provides operations to manipulate a *causally connected self-representation* of the underlying implementation. According to Maes [16], a system is said to be causally connected to its domain if "the internal structures and the domain they represent are linked in such a way that if one of them changes, this leads to a corresponding effect on the other". Such a system has the benefits that, firstly, the self representation always provides an accurate representation of the system, and that, secondly, a reflective system can bring modifications or extensions to itself by virtue of its own computation. In other words, a reflective system naturally supports inspection, and adaptation:

i) Inspection

Reflection enables applications to observe the occurrence of arbitrary events in the underlying implementation. Such an approach can be used to implement functions such as QoS monitors or accounting systems in a portable manner.

ii) Adaptation

Similarly, reflection allows applications to adapt the internal behaviour of the system either by changing the behaviour of an existing service (e.g. tuning the implementation of message passing to operate more optimally over a wireless link), or dynamically reconfiguring the system (e.g. inserting a filter object to reduce the bandwidth requirements of a communications stream). Such steps are often the result of changes detected during inspection (see above).

Most of the early research in reflection focussed on the field of programming language design [14, 24]. More recently, the work has diversified with applications of reflection in areas such as windowing systems [20] and operating systems [25]. There is also growing interest in the use of reflection in distributed systems. Pioneering work in this area was carried out by McAffer [18]. More recently, researchers at APM have developed reflective extensions to Java with a view to supporting distributed applications. However, there has been much less activity to date in the design of reflective middleware. Campbell at Illinois has carried out initial experiments on reflection in Object Request Brokers (ORBs) [22]. The level of reflection however is coarse-grained and restricted to invocation, marshalling and dispatching. In addition, the work does not consider key areas such as support continuous media interaction. Manola has carried out work in the design of a “RISC” object model for distributed computing [17], i.e. a minimal object model which can be specialised through reflection. A PhD student of Cointe is also investigating the use of reflection in proxy mechanisms for ORBs [15].

3. INTRODUCING EXPLICIT BINDING

3.1. What is Explicit Binding?

In order to address our requirements, it is necessary to extend the programming model offered by CORBA [2] (thereby aligning it more with ISO RM-ODP). In particular,

we introduce the concept of *explicit binding*. In the current CORBA programming model, binding is implicit in that, when objects interact, an appropriate communications path is created by the underlying ORB. In our approach, we suggest that bindings should be created explicitly by the programmer; the result of the binding is then an *object* representing the underlying *end-to-end* communications path (see figure 2). One consequence of this approach is that bindings can be created by a third party (in CORBA, the client always initiates an interaction).

The importance of explicit binding is twofold:

i) the act of creating a binding can subsume static QoS management functions such as negotiation, admission control and resource reservation, and

ii) the interface on the binding can be used for dynamic QoS management functions such as inspection and adaptation.

In this paper, we are particularly interested in this second aspect of explicit bindings, i.e. support for inspection and adaptation.

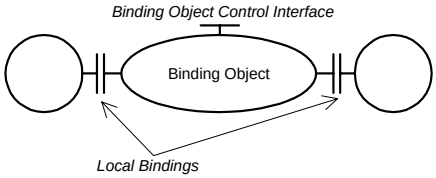

Figure 2: Explicit Binding

We introduce two different styles of binding, namely *operational bindings* and *stream bindings*: operational bindings support the traditional style of interaction in CORBA, namely operation requests. Stream bindings are then required to support continuous media interaction. A given stream consists of one or more *flows* where each flow represents the unidirectional transmission of a continuous media type (e.g. audio or video). As a result of this change, it is also necessary to distinguish between operational interfaces and stream interfaces as end-points of operational bindings and stream bindings respectively.

The architecture is *open* in that bindings are created by an extensible set of *binding factories*. Factories in CORBA are objects that support the creation of a particular class of object; binding factories are therefore responsible for creating a new binding between a target set of objects. One binding factory could provide the semantics of standard CORBA requests whereas another could provide real-time guarantees in terms of end-to-end latency (perhaps exploiting meta-level functionality to meet the required guarantees). Similarly, for continuous media interactions, one factory could provide a best effort service for video transmission whereas another factory could provide guarantees through an appropriate resource reservation strategy. In addition, programmers are free to develop their own binding classes, perhaps in terms of existing classes.

Explicit bindings provide one step towards a more open architecture in that communication becomes both visible and controllable. This is necessary but, in our view, not sufficient for mobile multimedia applications. We therefore extend this concept further by introducing open bindings.

4. THE CONCEPT OF OPEN BINDINGS

4.1. General Approach

To support mobile computing, it is necessary for the application to be able to exert some control over bindings. One way of achieving this is for binding interface to offer QoS management operations to monitor the current levels of QoS and to adapt to perceived changes. The problem with this approach is that it is very difficult to design a general means of achieving adaptation. This is especially problematic when mobility is introduced due to the proliferation in possible actions. Our approach is for bindings to offer a meta-interface providing access to a causally connected self-representation. This self-representation is provided by an *object graph*, representing the underlying end-to-end communications path. This equates to a procedural as opposed to a declarative approach to QoS management [3]. We argue that this approach offers the level of flexibility required by mobile computing.

This procedural approach is influenced by our experiences with the use of logic or QoS attributes to specify QoS requirements [6]. We have found that this is a perfectly valid approach for dealing with static QoS properties but the approach cannot easily be extended to deal with adaptation. In Adapt, we still allow the association of some simple QoS attributes with bindings as a means of checking consistency between interfaces in the bindings. However, the main mechanism for dealing with more dynamic aspects of QoS management is to directly manipulate graphs. Note that this does not preclude the use of declarative techniques which can be built on top of the basic procedural facilities provided by the platform.

4.2. Object Graphs

An object graph consists of processing objects and binding objects which are connected together by *local bindings*. Communication across a local binding is assumed to be instantaneous and reliable, normally implying that local bindings are located in a single address space or a single machine. All other interactions are represented explicitly by the binding objects in the graph. Processing objects then either perform computations on the data flowing through the graph or are responsible for a particular management function. Examples of processing objects include QoS

filters and mixers, QoS monitors, or rate control components. The concept of an open binding is illustrated in figure 3 below.

To control visibility of interfaces within a binding, we introduce the concept of *interface mapping*. (illustrated as dotted lines in fig.3) Interface mapping allows an external interface to map on to the interface of an internal component. The external interface acts as a proxy for the internal interface; all interactions occur at the internal interface via the external interface.

As a further refinement, binding objects can themselves be open bindings and hence also be composed in terms of object graphs. The nesting bottoms out by offering a set of *primitive bindings* whose implementation is closed. For example, a particular platform might offer RTP or IP services as primitive bindings (depending on the level of openness in the platform). This nested structure provides access to lower levels of the implementation (if required). At a finer granularity, each object in the graph can offer an interface to control its individual behaviour. In addition, each object is expected to provide an interface for event notification; to use this, programmers register their interest in particular events and then receive call-backs when the events occur (see section 5.2). The concept of nested open bindings is illustrated in the forthcoming example fig. 4.

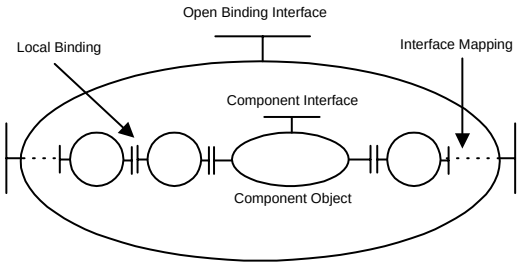

Figure 3: Open Bindings

5. USING OPEN BINDINGS

5.1. The Component Class

In implementation, open bindings are created as a subclass of a Component object class. This class enhances the normal CORBA object class in three important ways. Firstly, the Component class enables CORBA objects to be created with multiple interfaces (stream or operational interfaces) as required by the extended programming model (see section 3 above). Secondly, the class provides access to a MetaComponent interface. This meta-interface supports operations to inspect and adapt the associated object graph structure (see sections 5.2 and 5.3 below). Thirdly, the Component class allows the programmer to raise queries about the various interfaces supported by an object, and to ascertain whether a given stream interface will accept a local binding, i.e. will it accept a given media type. Note that processing objects are also descended from the Component class. In this way, it is also possible to recursively open up the implementation of, for example, a filter object as an object graph. Again, however, some processing objects will be primitive and will not support inspection and adaptation.

5.2. Support for Inspection

Object graphs represent a powerful and intuitive way for the programmer to inspect the implementation of open bindings. The meta-interface provides one operation, getConfiguration, which returns a representation of the object graph. By traversing this graph structure, the programmer can identify how the individual components are connected together. For example, one could ascertain that a buffering component was connected up between the output stream interface of a transport binding object and the input a decoder object. This level of information allows more educated decisions on how adaptation mechanisms are to be applied. An example in this case would be the removal of the buffering component to reduce end-to-end latency on detecting a drop in the level jitter. To convey the functional representation of a binding object in a declarative way would certainly be a non-trivial task. Note that every component in the graph also has a unique name, referred to as its role. This role can be used to access the object directly, using the getObjectByRole method.

Once accessed, a component in the graph can be queried with regard to its type (for example DelayBuffer), its location and the number of stream interfaces that it possesses. In addition the various control interfaces that these nodes support can be interrogated in an interface-specific manner. As described above, components also support an event notification mechanism. An application can register for events from a particular component and then receive a back-call when this event occurs. This feature can be used for example to allow an application to monitor the level of packet loss experienced over the current network.

Finally, the mechanisms described below can be used to insert arbitrary QoS monitors into an object graph giving further information on the behaviour of the binding. An example of this type of object would be one that measured the drift in synchronization between paired audio and video streams.

5.3. Support for Adaptation

5.3.1. Overview

As mentioned above, there are two approaches to adaptation. Firstly, an application can modify the behaviour of a component without altering the structure of the object

graph, e.g. to counteract mild fluctuations in QoS such as the modification of the cutoff point of a media filtering component residing within a binding object. Secondly the programmer can *dynamically reconfigure* the object graph, changing its structure in some way by adding or removing components, or by reconnecting the existing components in a different manner. This type of change would be used to combat more significant changes in QoS such as a change in network type from Ethernet to a wireless GSM link. In this case, it is more sensible to remove inappropriate components and replace them with new components more suited to the new environment. We will now examine these two classes of change in turn.

5.3.2. Modification of Behaviour

This is the simplest form of adaptation. To modify the behaviour of a component, the application must first obtain the interface for this component (e.g. through the role of the component). Once the application has obtained this interface, the programmer can make changes to that component such as increasing the size of a buffering component or altering the compression strategy of an MPEG component. The precise interface is clearly dependent of the object class of the component.

Note that the meta-interface of the corresponding binding object is free to decide upon the level of adaptation that it will allow. It may choose to make all components within its object graph visible to change, or it may choose to restrict changes to certain components, such as a QoS filter above

5.3.3. Dynamic Reconfiguration

As a component's implementation is modelled as an object graph the obvious types of changes that may be applied are the addition and removal of nodes and arcs to/from this graph. New component objects may be added and local binding arcs added to the graph as this component is connected to the others. Similarly it may be useful to remove whole components and their associated local bindings when they are no longer needed, an example being a jitter buffer object which would be removed when moving from wireless to fixed networks.

The basic operations for manipulating object graphs are as shown in table 1.

Operation	Effect
addObject	A new object to be added to the object graph (initially unbound).
removeObject	Removes an object from the object graph, deleting any associated local bindings.
LocalBind	Creates a new arc in the graph between two specified interfaces.
BreakBind	Removes an arc from the graph.

Table 1: Basic graph operations.

The above operations provide a means of changing the functional structure of an implementation object graph. However unhindered changes could result in the functional characteristics of the binding object being destroyed. For example all component objects could be unbound leaving a disconnected graph with no path from input to output interfaces. For this reason, all changes to an implementation object graph are requested through the meta-interface of the parent component, thereby allowing it to reject those which it deems undesirable.

In some cases, however, this may prevent perfectly reasonable changes from being effected. For example, it may be desirable to replace one processing object with another. One reasonable course of action would be: (i) to add the new object to the graph, (ii) destroy the local bindings connecting the old object to other objects, and (iii) create new local bindings connecting the new processing object into the media flow allowing the graph to operate as before. However the meta-interface may reject step (ii) on the grounds that it will create a hole in the graph preventing media from flowing. There is no way for the meta-interface to know that eventually the new processing component will be connected up, therefore allowing the graph to flow again.

To remedy this situation, we provide *compound* operations on the Component meta-interface - see Table 2.

Operation	Effect
insert	Allows one component (offering an incoming and outgoing interface) to be inserted between two existing components in the graph, e.g. to insert a filtering object between a codec and a transport binding object.
cut	Performs the opposite action to insert, removing a component from a pipeline and connecting up the predecessor and successor components directly.
replace	Replaces one component with a similar component (the new component inherits the role of the original component).
redirect	Redirects a local binding to an alternative interface, e.g. to send continuous media data down a new path of control more suited to new network conditions

Table 2: Compound graph operations.

The use of compound operations allows a meta-interface to cushion or reduce the undesirable effects of these changes where appropriate. For example the replacement of a boundary component will destroy the corresponding interface mapping, therefore resulting in the local binding of an external interface being broken. The effects of this filter upwards through the hierarchy of object graphs. If the compound Replace operation was used to replace the existing object with a new one then this replacement could be done atomically allowing the local binding between

external stream interfaces to remain intact.

By defining a variety of operations, the Component meta-interface is able to choose the level of access that it will allow. For example a particular binding object may open up its implementation for complete inspection but only allow one particular QoS filter component to be replaced after it has ascertained that the replacement object is acceptable.

6. A SIMPLE EXAMPLE

As an example of the use of object graphs to support adaptation, we consider a simple unidirectional point-to-point video binding object. The binding object is intended to encapsulate the end-to-end delivery of multimedia data and so it performs the compression, transmission and decompression of video data: at one end it consumes raw video, at the other raw video is produced for consumption by the sink object. Initially the computer is connected by a high performance fixed network and so full broadcast quality MPEG video may be sent across the network with little or no perceived degradation in quality.

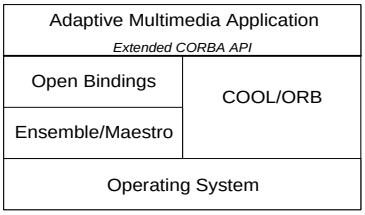

Figure 5: The extended CORBA platform

At some point however the computer may move to a wireless LAN, such as WaveLAN, with greatly reduced bandwidth and increased jitter in traffic. To address this change in connectivity it is necessary to use the meta-interface of the binding object to locate the video compression object and cause it to reduce the bandwidth of compressed video. Similarly to address the increase in perceived jitter we insert a jitter compensation buffer object between the transport and video decompressor objects.

At some later stage the computer moves out of range of the WaveLAN base station and accordingly a GSM dialup link is used to reach the fixed partition of the network. Rather than change the MPEG compression parameters, it is now more prudent to replace the video compressor object with one more suited to low-bandwidth operation such as an H.263 encoder object. Similarly the decompressor object would be changed from an MPEG decompressor to a H.263 decompressor. This full range of adaptation mechanisms is illustrated in figure 4.

More extreme adaptations would involve opening up the transport binding object to alter protocol characteristics

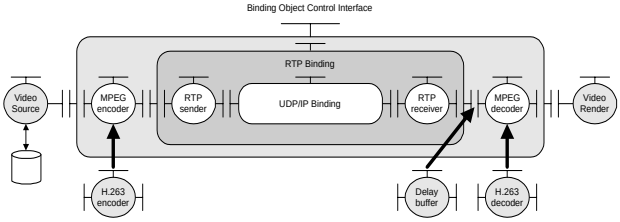

Figure 4: Adaptation using Open Bindings

using the aforementioned mechanisms in Ensemble. This type of adaptation would be used to achieve maximum performance over a network such as GSM where packet headers can consume a significant percentage of available bandwidth.

Obviously there is a wide variation in the perceived quality of video produced by the binding object: from broadcast quality down to H.263. Nonetheless, the example shows that a constant uninterrupted video stream is possible regardless of network connectivity.

7. IMPLEMENTATION APPROACH

We have implemented an experimental middleware platform featuring the concept of explicit open bindings. This platform is based on a CORBA implementation from Chorus Systems, called COOL-ORB [9]. This runs over a variety of computers and operating systems; our experimental testbed consists of laptop PCs running Windows NT 4.0. These are interconnected by a variety of networks, including Switched Ethernet, WaveLAN and GSM.

In order to support open bindings, the COOL platform has been extended as shown in figure 5.

Currently COOL-ORB supports two basic communications infrastructures: TCP/IP and CHORUS IPC. In order to support higher degree of configurability, we have extended COOL with a third communications infrastructure based on Ensemble/ Maestro [10]. Ensemble, developed at Cornell University, is the successor to Horus [23]. The software is written in Objective Caml (a dialect of ML) and provides a modular framework for constructing protocol stacks, specifically protocol stacks for group communication (point-to-point communication is treated as a special case). The role of Maestro is to offer C++ bindings for the Ensemble implementation.

In Ensemble, a *protocol profile* is built from a stack of modules. Ensemble offers a library of such modules including UDP, packet loss detection, data encryption and flow control. The software enables the programmer to select a particular protocol profile at *bind time* by providing a list of the component modules. For example, the following protocol profile provides a stack offering virtual synchronous group communication:

vsync= [Gmp;Sync;Heal;Migrate;Switch;Frag;Suspect;Flow]

Ensemble also supports *run-time* adaptation, in terms of modification of modules and also dynamic reconfiguration. For example, the parameters for flow control can be modified at run-time. Similarly, Ensemble allows the programmer to *switch* to an alternative protocol stack at any point during an interaction.

We exploit this configurability by using Ensemble to provide a set of low level open bindings (operational and stream). The internal details of such open bindings are presented as object graphs, thus providing a consistent style of adaptation throughout the architecture.

For operational bindings, the standard CORBA bind call has been extended to enable the protocol stack to be specified at bind-time. The format of the new bind call is as follows:

```
void COOL_bind(const ServerImpl& server,
Server_ptr& inter, COOL_ComCtrl_ptr& ctrl,
char *comName, CORBA_Environment& _env);
```

The first two parameters specify the object implementation and pointer to be used to invoke this object respectively. The third parameter is then the control interface for the resultant open binding. The comName parameter specifies the preferred protocol suite to be used, followed by a specification of a protocol stack (if Ensemble). The final parameter is a standard CORBA environment variable.

As an example, the following call establishes an explicit binding between to the object designated by serverImpl:

```
COOL_bind(serverImpl, server, ctrlIntf,
"Ensemble::Gmp:Sync:Heal:Switch:Frag:
Suspect:Flow", env);
```

For stream bindings, we provide a set of binding factories offering pre-configured Ensemble stacks for continuous media interaction. Through the control interface of these bindings it is possible to modify a component object or switch to an alternative stack (as for operational bindings). Note that it is relatively straightforward to extend the range of bindings by constructing new binding factories from existing components. Prominent among the components are a range of filter objects for common media formats.

8. RELATED WORK

There is currently considerable ongoing research in the area of extensible and adaptable operating systems. Key examples include Spin [1], Exokernel/Aegis [8] and Spring [19]. The aim of this work is to introduce flexibility in operating system structures to allow, for example, the addition of new services. In general, however, this research has not considered the requirements of mobile multimedia applications. In addition, we prefer to implement adaptation at a different level, i.e. in middleware. This offers a platform independent means of achieving adaptability.

The specific concept of object graphs was introduced by researchers at JAIST in Japan [11]. As with our approach, a system is decomposed into a graph of objects. The system also supports a model of information flow between objects. Adaptation is handled through the use of control scripts written in TCL. Although similar to our proposals, the JAIST work does not provide access to the internal details of communication objects. Furthermore, the work is not integrated into a middleware platform. The concept of object graphs is also used in Microsoft's ActiveMovie software. This software, however, does not address distribution of object graphs. In addition, the graph is not re-configurable during the presentation of a media stream. Finally, object graphs feature in a DEIMOS, a related project at Lancaster University investigating adaptable and extensible operating systems [4].

A number of researchers have considered the impact of multimedia on middleware [5, 12, 13] and the impact of mobility on middleware [7, 21]. In general, these activities do not provide as comprehensive an approach to adaptation as we feel is necessary. However researchers at CNET have developed an extended CORBA platform to support multimedia [2]; this platform features the concept of recursive bindings and has been highly influential in our research.

Finally, recent work in the OMG forum has addressed the need for multimedia streams in CORBA. In particular, a revised proposal has recently been developed in response to a Request for Proposals (RFP) for the Control and Management of Audio/ Visual Streams (issued by the Telecommunication Special Interest Group). However, the current proposals do not explicitly address the issues of openness and adaptation that are the central concerns in our research.

9. CONCLUSIONS

This paper has considered the design of middleware platforms to support mobile multimedia applications and has suggested that future middleware platforms should be adaptive in order to address the diverse requirements imposed by such applications. The paper has also outlined the design of an adaptive middleware platform, based on CORBA, but extended with the concepts of open bindings and object graphs.

The advantages of this approach are that, firstly, applications can be made aware of arbitrary events within the platform, and, secondly, applications have a high degree of flexibility in the way they respond to events. There are also potential disadvantages with the proposed approach. For example, the programmer can be faced with added complexity in responding to events although this can be controlled by the provision of standard policies in

application libraries. Secondly, there is a danger of compromising the integrity of systems by providing low level access although this problem is less serious at the middleware level than at the operating system level. We also believe that object graphs in open bindings provide a sufficiently constrained style of interaction to avoid this problem. Finally, it is arguable that the flexibility provided by object graphs is gained at the expense of efficiency. The overheads of object graphs can however be minimised by careful engineering. This is aided by the user level implementation where most inter-object interaction is by procedure calls.

Ongoing research in Adapt is looking in more detail at the support required by operational and stream bindings and also at the provision of multi-party bindings exploiting the facilities offered by Ensemble.

ACKNOWLEDGEMENTS

The work described here is being carried out in the Adapt Project, a collaboration between Lancaster University and BT Labs sponsored by the EPSRC (Research Grant GR/K72575). BT Labs are also contributing towards the cost of a PhD studentship attached to the project. Particular thanks are due to our collaborators at BT Labs namely Andrew Grace, Alan Smith and Steve Rudkin.

REFERENCES

[1] Bershad, B.N., S. Savage, P.Przemyslaw, E.G. Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, S. Eggers, S., "Extensibility, Safety and Performance in the SPIN Operating System". Proc. 15th ACM SOSP, pp267-284, Copper Mountain CO, USA, December 1995.

[2] Blair, G.S., J.B. Stefani, “Open Distributed Processing and Multimedia”, Addison-Wesley, 1998.

[3] Blair, G.S., Coulson, G., Davies, N., Robin, P. and Fitzpatrick, T., "Adaptive Middleware for Mobile Multimedia Applications", Proc. NOSSDAV, St Louis, Missouri, USA., pp 259-273, May 19-21, 1997.

[4] Clarke, M., Coulson, G., "An Architecture for Dynamically Extensible Operating Systems". To appear in Proc. ICCDS'98, Annapolis MD, USA, May 1998.

[5] Coulson, G., Blair, G.S., Horn, F., Hazard, L, Stefani, J.B., "Supporting the Real-Time Requirements of Continuous Media in Open Distributed Processing", Computer Networks and ISDN Systems, Vol. 27, No. 8, 1995.

[6] Coulson, G. and Waddington, D.G., "A CORBA Compliant Real-Time Multimedia Platform for Broadband Networks", Proc. TRENDS 96, Aachen, Germany, September 1996.

[7] Davies, N., A. Friday, G.S. Blair, and K. Cheverst, “Distributed Systems Support for Adaptive Mobile Applications”, ACM Mobile Networks and Applications, Special Issue on Mobile Computing - System Services, Vol. 1, No. 4, 1996.

[8] Engler, D.R., M.F. Kaashoek, J. O'Toole jnr, J., "Exokernel: An Operating System Architecture for Application-Level Resource Management". Proc. 15th ACM SOSP pp 251-266, December 1995.

[9] Habert, S., L. Mosseri, V. Abrossimov, “COOL: Kernel Support for Object-Oriented Environments”, Proceedings of ECOOP/OOPSLA Conference, Ottawa, Canada, October 1990.

[10] Hayden, M., "The Ensemble System", PhD Dissertation, Dept. of Computer Science, Cornell University, USA, 1997.

[11] Hokimoto, A., T. Nakajima, “An Approach for Constructing Mobile Applications using Service Proxies”, Proc. 16th ICDCS’96, IEEE, May 1996.

[12] Interactive Multimedia Association, "Multimedia System Services - Part 1: Functional Specification (2nd Draft)", IMA Recommended Practice, September 1994.

[13] Interactive Multimedia Association, "Multimedia System Services - Part 2: Multimedia Devices and Formats (2nd Draft)", IMA Recommended Practice, September 1994.

[14] Kiczales, G., J. des Rivières, D.G. Bobrow, “The Art of the Metaobject Protocol”, MIT Press, 1991.

[15] Ledoux, T., “Implementing Proxy Objects in a Reflective ORB”, Proc. ECOOP’97 Workshop on CORBA: Implementation, Use and Evaluation, Jyväskylä, Finland, June 1997.

[16] Maes, P., “Concepts and Experiments in Computational Reflection”, In Proc. OOPSLA’87, Vol. 22 of ACM SIGPLAN Notices, pp147-155, ACM Press, 1987.

[17] Manola, F., "MetaObject Protocol Concepts for a "RISC" Object Model", Technical Report TR-0244-12-93-165, GTE Laboratories, 40 Sylvan Road, Waltham, MA 02254, USA, December 1993.

[18] McAffer, J., “Meta-Level Architecture Support for Distributed Objects”, In Proceedings of Reflection 96, G. Kiczales (ed), pp39-62, San Francisco, 1996.

[19] Mitchell, J.G., J.J. Gibbons, G. Hamilton, P.B. Kessler, Y.A. Khalidi, P. Kougiouris, P.W. Madany, M.N. Nelson, M.L. Powell and S.R. Radia, "An Overview of the Spring System". Proc. IEEE COMPCON'94, February 1994.

[20] Rao, R., “Implementational Reflection in Silica”, Proceedings of ECOOP’91, Lecture Notes in Computer Science, P. America (Ed), pp251-267, Springer-Verlag, 1991.

[21] Schill, A., and S. Kümmel, “Design and Implementation of a Support Platform for Distributed Mobile Computing”, Distributed Systems Engineering Vol. 2, No. 3, pp128-141, 1995.

[22] Singhai, A., Sane, A., Campbell, R., “Reflective ORBs: Supporting Robust, Time-critical Distribution”, Proc. ECOOP’97, Jyväskylä, Finland, June 1997.

[23] van Renesse, R., K.P. Birman, S. Maffeis, “Horus: A Flexible Group Communications Service”, Communications of the ACM, April 1996.

[24] Watanabe, T., A. Yonezawa, “Reflection in an Object-Oriented Concurrent Language”, Proc. OOPSLA’88, Vol. 23 of ACM SIGPLAN Notices, pp306-315, ACM Press, 1988.

[25] Yokote, Y., “The Apertos Reflective Operating System: The Concept and Its Implementation”, Proc. OOPSLA’92, Vol. 28 of ACM SIGPLAN Notices, pp414-434, ACM Press, 1992.