A Framework to Support Large-Scale Active Applications

John Bates
University of Cambridge Computer Laboratory
Pembroke Street, Cambridge UK
Email: job@cl.cam.ac.uk

Many applications are emerging which share the characteristic of being *active*, i.e. driven by asynchronous run-time events. The application areas include multimedia, cooperative working (such as conferencing), intelligent agent architectures, active database and support for user mobility. Prototype active applications are often developed from scratch as monolithic entities and are thus inflexible. Platforms which offer generalised support only usually deal with one application class in isolation.

The thesis of this work is that active applications have generic requirements of a supporting platform. Rather than building each application as a monolithic entity, they can be composed of inter-connected sub-components called *active objects*. These objects can be used interchangeably in a plug-and-play architecture. This encourages on-the-fly application development and assists fast prototyping.

This paper is supported by our implementation of a prototype distributed active application support platform. Our software is used to provide examples throughout. Section 1 describes the application requirements which a platform must be able to support. Section 2 describes the model for composing applications. Section 3 describes how a distributed programming platform has been enhanced for run-time object support and to allow new active object classes to be constructed easily. Finally, section 4 concludes.

1 Requirements of a Platform

In order to support world-wide applications, a platform must be able to support applications that are

- *active* - i.e. event-driven rather than sequential in nature. Asynchronous run-time occurrences, such as user interaction, reaching a point in the presentation of a media item or a participant's movement to another room, can be used to initiate further actions within an application.

- *distributed and scalable* - many active applications are inherently distributed, for example multimedia conferencing in which different people exist in geographically separate locations. It must be possible to create applications involving any number of users in any inter-connected locations.

- *mobile* - as users move around computing environments it is beneficial if their applications follow them. If they are involved in cooperative applications then the sessions should be reconfigured transparently.

- *easily constructed* - distributed applications are often unattractive to application developers as they are difficult to build.

- *accessible* - applications are often unattractive to users if they are difficult to use.

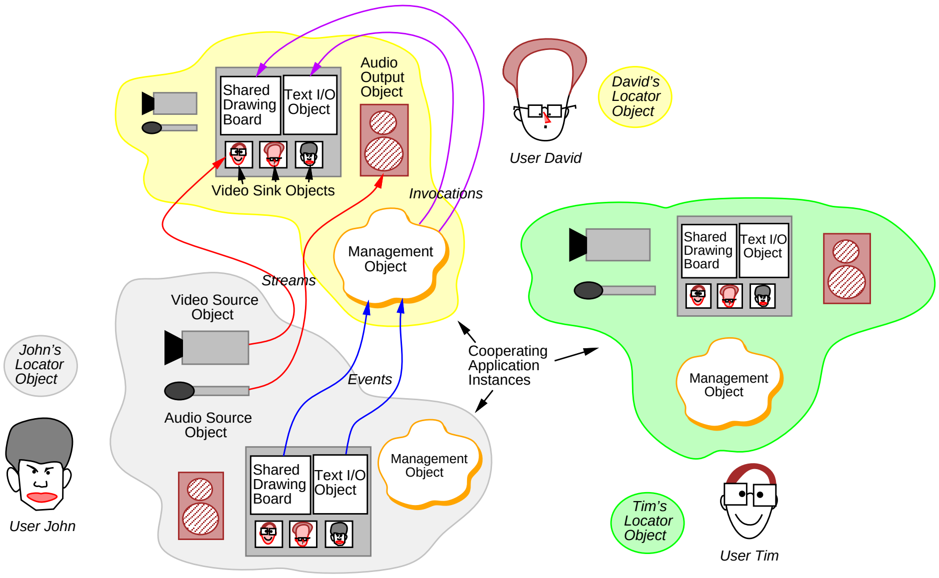

Figure 1: A cooperative multimedia application built with active objects

2 Composing Active Applications

Our framework supports the construction of active applications from *active objects*. An active object is an autonomous entity, such as an encapsulation of a media processing function. Each can concurrently receive and send messages, as well as receiving streams of data, processing them and outputting them. Communicating active objects can be distributed across multiple machines and domains. Figure 1 shows an application built as part of our experiments, which is used throughout this paper as an illustration.

2.1 Uses of Active Objects

There is a distinction between pre-built and bespoke active object classes. Modularising functionality as pre-built active objects facilitates reusable application *building blocks*. Well-known classes of object can be created on workstations which can support the relevant functionality. Example functions are as follows:

- *media sources and sinks* of varying types, both live (such as video cameras and microphones) and persistent (abstractions of storage services).
- *active user interface components* of various types, e.g. video windows, drawing boards and text boxes.
- *intelligent agents*, e.g. to monitor for faces in a video stream.
- *location monitoring objects*, to notify clients when users change their physical location. These abstract location sensing devices, e.g. *active badges* [8].
- *interpreter objects*, which are a special case. They allow code to be executed on-the-fly inside an active object wrapper. Most object classes are compiled for efficiency and because they represent core abstractions. Interpreter objects provide for fast prototyping of object functions.

To construct applications out of these existing classes requires objects which act as glue to plug existing object classes together and which can manage run-time interaction and movement. Such *management objects* represent the bespoke aspect of an application. They are also responsible for communication with other application instances, as in the example of figure 1. Interpreter objects are useful for building such management objects, since applications can be built on-the-fly and existing scripts may require dynamic code modification. In our prototype implementation the management objects are implemented in this way. A new class of conferencing program can be devised and quickly prototyped as a script. The script can be sent to colleagues anywhere and each can run an instance of it.

2.2 Active Object Functions

Active objects are able to access the facilities of an enhanced distributed programming platform, the implementation of which is described in the next section. Each object has an interface specification which describes how it can communicate with other objects. In traditional object-oriented distributed programming only *method invocation* is used as a communication mechanism. Methods specify what messages can be sent to an object, e.g. it is possible to start and stop a video source object by invoking method `play` or `stop`. In our model the following mechanisms are also supported:

- *Typed stream endpoints* - Two stream endpoints of the appropriate type can be connected. This allows continuous streams of typed data to be sent between objects. In the example, to connect a video camera object to a video window object, a management object performs a *join* on a video *socket* stream endpoint in the source and a video *plug* stream endpoint in the sink.

- *Events* - The event mechanism is a generic way of registering interest in and receiving notification of run-time occurrences, such as user interaction and points in data presentation [1]. Each object declares the event classes that it can monitor for, in its interface. Each event has a parametrised class which is used in notification (operating in the same way as a parametrised invocation). Objects which need to be informed of a particular class of event *register interest*, using parameters to specify their range of interest. If the object monitoring for the event detects an event in the range of interest, it *notifies* all interested parties.

In the example of figure 1, events are used as a tailoring glue to create shared drawing and text conferencing tools from stand-alone components. The text object and drawing object are able to notify clients of drawing strokes or text sentences from a particular user. Taking the drawing as an example, an event class `StrokeDrawn(person, x1, y1, x2, y2)` is defined in the interface of a drawing board. Tim and David can register interest in `StrokeDrawn("John", A, B, C, D)` with John's drawing board. If John draws a stroke then an event such as `StrokeDrawn("John", 123, 234, 456, 345)` is notified. This can be used by Tim and David's management objects to update their own local boards with a consistent view. Such functionality can be encoded as an *event-driven rule*. These specify one, or an ordered combination of events as an *event expression*. If the expression is detected at run-time then the rule fires and a series of invocations can be performed.

Support for Quality of Service (QoS) in network and operating systems is emerging. It is a requirement that the system inform an application if it can no longer use the same level of resources. These notifications can be treated in the same way as user-specified events. Event-driven rules can be used to encode policy, such as if the current level of bandwidth cannot be supported then turn off video.

The application in figure 1 follows the replicated (as opposed to the centralised) model in that each user has his/her own copy of the controlling management object. A centralised approach could also be supported easily. The management objects are responsible for joining streams, registering events and managing event-driven rules. In this case each management object has an interface to allow a new user to connect to any existing instance and get a list of all other active instances. It can then register interest in all events and bind to the video and audio sources within the instances. In turn, the new management object must inform the other management objects that it has joined the session so they can add new video and audio sinks and register interest in its events.

2.3 Object Creation

In this framework, each domain (e.g. `cl.cam.ac.uk`) has its own local set of services. One is an object creation service. This can create an object of a specified class and, if necessary, on a specified machine. Although the system-dependent functions of objects differ between machines and environments, the system-independent functionality should remain the same. Thus an operation to create an object of class `VideoLiveSource` (for example) is possible on any machine with a camera and frame-grabber. The underlying representation may differ but, by providing a consistent system-independent interface, inter-domain inter-operation is possible.

A useful technique is to be able to author adaptive applications, i.e. have the application not only alter QoS parameters in response to system resource availability but also alter media usage based on available hardware in a particular location. So, for example, a conferencee using a workstation without camera and microphone will only be able to communicate to others using text and shared drawing. The object creation service is linked to a location information database which enables information about the capabilities of particular hardware to be ascertained. Policies can be encoded to select the appropriate object classes in a certain location.

2.4 Object Mobility

This framework supports the fact that users may want to move their physical location and have their application sessions follow them. Rather than just moving the user interface [10] or migrating an entire monolithic application [4, 5], this architecture supports moving the objects relevant to a particular user. For example, if John moved room then his video and audio source objects as well as his user interface objects would go with him. After movement, they rebind to the objects they were previously connected to. The ability to determine the hardware and resource availability in a particular location is useful here since as objects move they must adapt to the machine on which they are being run. An advanced architecture has been developed to support such mobility [3].

3 Platform Support for Active Objects

This section outlines how a distributed programming platform has been enhanced to support active objects. Our experiments are built on an ODP-ANSA [9] based platform but equally apply to other models, such as CORBA [6].

3.1 An Extended IDL

The facilities to define an object's interface in an IDL have been extended. Traditionally only composite types and methods have been declared. We have added to this the ability to specify events and stream endpoints. Figure 2 illustrates how the IDL can be used to specify interfaces for live video and video windows.

The extended IDL yields several benefits. Firstly that it is possible to communicate with unfamiliar objects by examining their interfaces. Secondly, it allows a third party (e.g. a management object) to explicitly re-establish a communication link after movement. Tools for IDL parsing and stub generation have been enhanced to support these extensions.

3.2 Extended Programming Support

In order to provide programming support for the extended IDL and other active object facilities, the following library modules have been integrated with the platform:

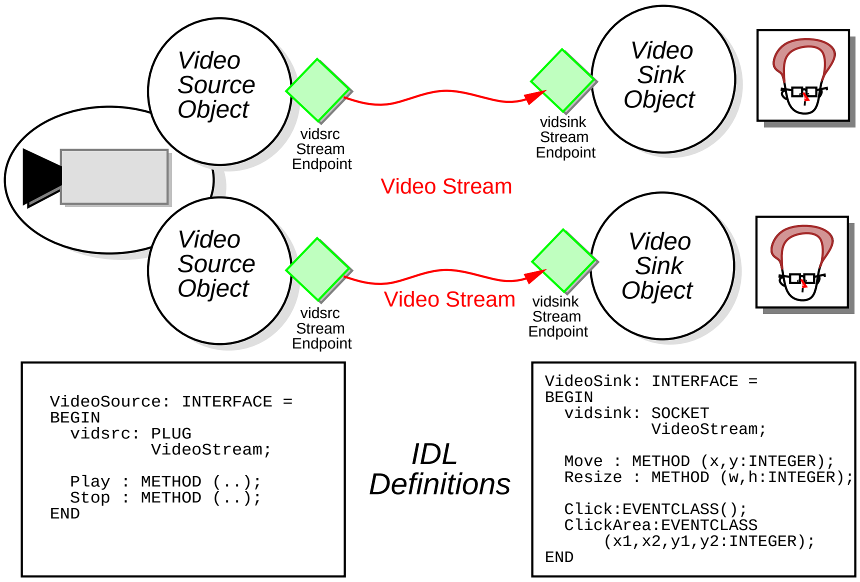

Figure 2: Example of active object interfaces

3.2.1 Stream Library

The stream library implements generic stream management facilities. Endpoints can be used to read and write typed data down a stream; for example a video source will write video frames and a video window will read them in. Endpoints are capable of multicasting when writing to multiple clients. Various types of stream are available depending on transmission reliability and efficiency requirements. QoS parameters can adjust the characteristics of a stream on-the-fly. Operations are provided so that third party objects can connect two named objects together with reference to typed stream endpoints.

3.2.2 Event Library

The event library provides support for registration and notification of events. Stubs to marshall and unmarshal relevant classes of event are generated automatically by event stub generation tools operating on an object interface. Both client and server have these stubs.

Every object inherits an interface for registration. The event library handles the receipt and storage of event registrations. This means that an object can internally signal events and the management of sending out notifications to relevant clients is done by the event library. Objects also inherit interfaces for handling notifications. An object developer can set up their own event handlers which are called by the event library in response to events of a certain class.

Interest in events can be deregistered explicitly by a client. An object can choose also to discard registrations after a certain period of time for garbage collection purposes or if a charging mechanism is in place.

3.2.3 Composite Event Library

Often, activity within applications is triggered not just by a single event but by a complex pattern of events. Such composite detection can involve parameter checking of event occurrences, state lookup and monitoring for particular event orderings. The composite event library provides support for composite event monitoring. It uses a model of finite state automata, with an active *token* for each possible evaluation path in an expression. The detection of composite events can be used to trigger invocations on objects in the same way as single events (see [1] for more details).

3.3 Factories and Creation Service

Two models of object execution are provided. Firstly objects can run in their own address space. Secondly, objects can run within a *factory's* address space to optimise access to a shared resource. Factories are object managers which can create objects of certain classes on a particular machine. They often abstract shared devices or services, such as a video camera or an interpreter.

A creation service uses the facilities of a trader to ascertain whether a relevant factory exists on a requested machine. It is also linked up to a location information database which contains information about machine hardware configurations. Objects in another domain which wish to create objects remotely and then link to them can do so. An alternative approach is to transmit an interpreter object into a domain and have it create objects there. Having done so, it can rendezvous with instances in other domains (as in the example of figure 1).

For each system-independent class, there are various system-dependent sub-classes of this class depending on the operating system and hardware combinations which are available on a particular host. By posting a request to the object creation service for an instance in a specific location, the appropriate system-dependent class will be created.

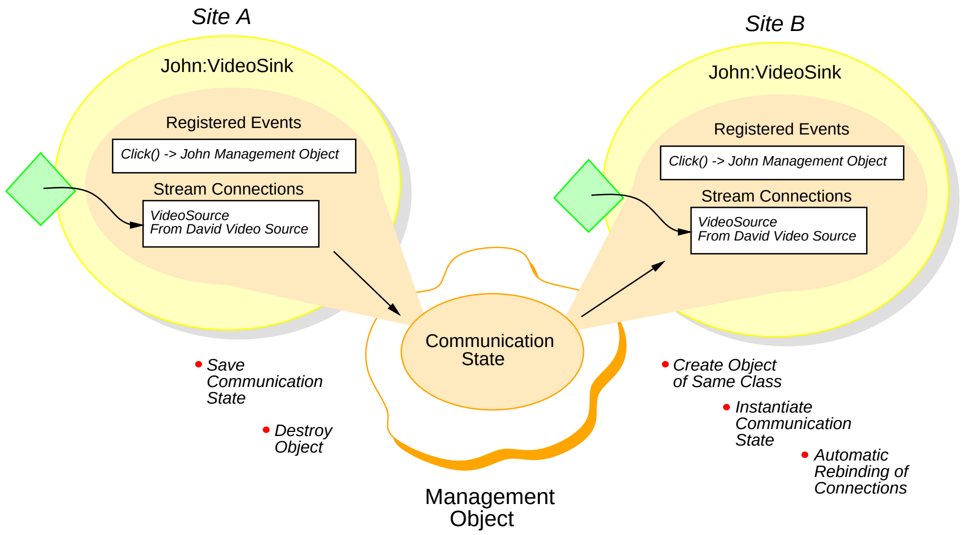

Figure 3: Saving of state for mobile objects

3.3.1 Mobility Support

For a media object to "move", the old object's communication state is stored by a third party, an object of a compatible class is created in the new location and the communication state is used to instantiate the new object. The old object is destroyed or suspended. All of the state associated with an object's communications can be accessed by an authorised third party (see figure 3). This allows an object to be suspended and restarted elsewhere without any heavyweight migration facilities. Other state can also be saved before movement. The third party keeps the state and uses it to reinstantiate the object elsewhere. The correct system-dependent version for a particular location is instantiated via a request to the creation service.

Objects are named explicitly to allow them to participate in communication, even after movement. The object's details are registered with a trading service, which can provide details of physical location (an interface reference) to interested parties. For example, to register interest in an event with Jean's video source object first requires the interface reference for this object. This can be retrieved from the trader by asking for an object of class `VideoSource` with attribute owner set to "Jean".

In our prototype system, special facilities exist for the transmission of interpreter objects [7]. They can save their own execution state and transmit themselves to a new location. This enables the interpreter objects to save the state of other objects, transmit themselves to the new location and reinstantiate the objects there. This factor enhances the benefits of using them as high level managers.

3.4 Session Management

As part of our initial platform, we developed an interpreted language to support the construction of management objects. This language allows the specification of configuration, event-driven rules (including composite events) and policies for groups of objects (see [2] for more detail).

As part of our ongoing work we are investigating a language to build management objects which take into account mobile objects as well as cooperative application instances. This is involving the development of a flexible conference management module to allow new management object instances to join in and leave shared sessions dynamically. It is also involving facilities for management objects to control the process of moving other objects.

3.5 Prototype Objects and Experiments

We have implemented all of the facilities discussed in this paper, including the extensions to the distributed programming platform, the object and mobility support services and a range of active object classes. The object classes we have built include video and audio sources and sinks (both live and persistent), the active text box and drawing board and the locator objects (based on active badge technology). To support all of the concurrent functions within objects required multi-threading. Kernel threads are available as part of the platform. Our prototype platform runs on Dec Alpha and Mips architectures and uses PThreads. The library modules described above incorporate threads to manage their own tasks autonomously.

An application's user interface is the key to its popularity with users. The active user interface components are designed so that they can be re-parented within a user interface management object. This reassures users that they are using a single application by retaining a consistent look, rather than making them use various ad hoc separate windows. The X forms widget set has been extended and integrated with the platform to provide support for this.

The user interface components are also designed flexibly to allow *overlaying*. For example, it is possible to overlay a whiteboard onto a video window so that the drawing strokes appear on top of the video.

We have also integrated our framework with the World Wide Web. The current model of the Web cannot support active applications. We have developed a plug-in for the Netscape browser to allow it to receive an interpreted object as a MIME type and forward it to the correct factory for execution. This object can manage the set-up of a complex application session. The ability to re-parent active user interface components has enabled the application to be displayed within the browser's window. Making applications available via the Web addresses the requirement of accessibility. Browsers, such as Netscape, are becoming the universally accepted user interface for an ever-growing number of computer users.

Because it is easy to prototype a new application, a variety of different applications has been experimented with. These range from simple "follow-me" video demonstrations to the complex cooperative application shown in figure 1.

4 Conclusions

This paper has illustrated a platform which can support state-of-the-art distributed, active and mobile applications. The emphasis has been that applications can be constructed elegantly, scalably and quickly using active object components.

Each active object fulfils an encapsulated function. Instances can be plugged together on the fly using streams. Objects can monitor for certain classes of event and clients can register interest in a particular range of these. They are notified if such events occur. In turn these notifications can be used to initiate invocations on other objects, thus facilitating an event-driven model. Bespoke objects to manage an application session composed of pre-existing classes can be constructed easily.

The facilities of a distributed programming platform have been extended to support active objects. An IDL has been enhanced to provide typed stream endpoints and event classes. Libraries have been added into the platform to provide event registration and notification functions, flexible stream management and state saving and reinstantiation for mobility. Each domain provides services to support the creation and mobility of objects. The ubiquitous deployment of such platforms can provide the basis for flexible world-wide support for active applications.

Acknowledgements

I would like to thank the EPSRC for supporting this work under grant GR/J42007. I am very grateful for the support of Michael and Morven Heller and St Catharine's college, Cambridge. Thanks also to ICL for general support of our research group. I would like to acknowledge all the hard work David Halls has put into the support for mobility. Finally, I am indebted to my colleague Jean Bacon for much assistance and advice.

References

[1] J. Bacon, J. Bates, R. Hayton, and K. Moody. Using Events to build Distributed Applications. In *Proc IEEE SDNE Services in Distributed and Networked Environments*, pages 148–155, June 1995.

[2] J. Bates and J. Bacon. Supporting interactive presentation for distributed multimedia applications. *Multimedia Tools and Applications*, 1(1), 1995.

[3] J. Bates, D. Halls, and J. Bacon. A Framework to Support Mobile Users of Multimedia Applications. *To appear in ACM Mobile Networks and Nomadic Applications (NOMAD)*, 1996.

[4] H. Cejtin, S. Jagannathan, and R. Kelsey. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems*, 17(5):704–739, September 1995.

[5] M. Fuchs. *Dreme: For Life in the Net.* PhD thesis, Department of Computer Science, New York University, September 1995.

[6] Object Management Group. The Common Object Request Broker: Architecture and specification. Technical Report 91.9.1, Object Management Group, December 1991.

[7] D. Halls. Remotely Executable Programs. Internal Report, University of Cambridge Computer Laboratory, June 1995.

[8] A. Harter and A. Hopper. A distributed location system for the active office. *IEEE Network*, 8(1), 1994.

[9] Architecture Projects Management Limited. *Advanced Networked Systems Architecture Testbench Implementation Manual*, 1993.

[10] T. Richardson, F. Bennett, G. Mapp, and A. Hopper. Teleporting in an X Window System Environment. Technical Report 94-4, Olivetti Research Limited, 1994.