Active Objects: a Paradigm for Communications and Event Driven Systems

G. Caal, A. Divin, C. Petitpierre
Laboratoire de Téléinformatique
Ecole Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland

Abstract

Current techniques for handling events, such as the "callback functions" approach, present some limitations to the development of Network Software, Communication Layered Protocols, Graphical Interfaces for Real-Time Applications and Multimedia. These techniques lack the mechanisms for data flow structuring, for the interaction between different events and for Finite State Machine design. This paper presents a paradigm, based on the concept of Active Objects, that provides a new way of designing event driven applications. The concurrent and object-oriented capabilities of this approach make it particularly useful for facilitating the construction of complex event driven systems.

1 Introduction

Network servers, control systems, communication protocols, windowing services, and multimedia applications are all event driven systems. As those systems become prevalent, more and more interactions with the outside world will have to be handled. Efficient and easy ways of treating I/O events are thus very important.

In this article, we first analyze current techniques for handling events. The most used techniques follow the concept of "callback functions" which presents some limitations to the development of complex applications. An example of communicating systems is provided, showing the difficulties encountered.

Secondly, we present a new approach to handling events based on the concept of Active Objects. Our proposal extends the C++ [6] programming language while respecting its philosophy. Thus it contains all OO features and can be acquired very quickly by C++ programmers.

Finally, we show how our approach resolves the limitations and we refer again to the example of the communicating systems.

2 Event Handling

2.1 Different UNIX techniques

UNIX offers several different techniques to support applications that perform I/O on multiple channels "simultaneously" [2].

One method is "polling". Polling operates by cycling through a set of open descriptors, checking each one for pending I/O activity. The primary disadvantage with polling is that it consumes excessive CPU cycles by making unnecessary system calls while performing "busy-waiting".

Another approach considers the application as a concurrent server, which creates (fork) a separate UNIX process to manage a communication channel connected to a distinct client. One disadvantage with the multiple process design is that it consumes excessive OS resources. The cost is even higher if additional communication or data sharing between the separate processes are required.

A third approach utilizes the I/O multiplexing facilities available via the BSD select() and System V UNIX poll() system calls. These functions allow network applications to wait for different types of I/O events occurring on multiple I/O descriptors by suspending the caller until an event arrives. The event can then be handled by the right procedure. These services solve several limitations of the alternative approaches but, when the application is complex, they become very complicated to use. In addition, they are weakly typed and not extensible.

2.2 Callback Functions

The usual way of developing applications that handle I/O events is the following. The main program consists of an infinite loop (in some systems it is hidden inside the system) that waits for the events and calls a particular function when a particular event arrives. This function, defined by the programmer, is called a "Callback Function". An application is thus segmented into functions that are called when events occur.

In this paper, we call this method of Handling Events the "Callback Functions" method. It is used, with some variations, by the following approaches.

2.2.1 The Callback Functions concept using the BSD UNIX select

Using the BSD UNIX select, the main program of an application that follows the concept of "Callback Functions" would be (using pseudo-code):

```
loop forever{
select(ListOfEnabledDescriptors);
   if (Descr1 = IsSet) TreatMessageFromDescr1();
   if (Descr2 = IsSet) TreatMessageFromDescr2();
   if (Descr3 = IsSet) TreatMessageFromDescr3();
   ...
}
```

Each descriptor represents an access point. Messages coming from these descriptors are treated or dispatched to the objects or procedures that will treat them, by the "TreatMessageFromDescr" functions which contain the whole application.

2.2.2 An object oriented approach

An object-oriented approach can simplify the use of UNIX I/O multiplexing facilities by encapsulating the select and/or poll facilities into classes that insure reusability, extensibility and type-security.

The work realized by Schmidt [7] [8] (C++ Wrappers) is an example. A C++ class, called the Reactor, integrates the multiplexing of I/O-based events together with timer-based events. When these events occur, the Reactor automatically dispatches them to previously registered application-defined member functions to handle the events.

In this case, the main program consists of the Reactor, which hides the infinite loop, and the callback functions are the previously registered application-defined object methods.

2.2.3 Graphical User Interfaces approach

Toolkits [10], in the sense of the X-toolkit, provide objects (called Widgets) such as editable or non-editable text items, different kinds of buttons, text windows, etc. These objects communicate with the application using the same mechanism described before, ie. callback functions. At the creation of the widget, the user interface programmer specifies a procedure to be called automatically when the widget is activated. After defining all its widgets, the application must enter an infinite loop (XtMainLoop) that is responsible of calling the previously defined procedure callbacks when the widget events arrive.

2.3 Limitations of Current Techniques on Building Complex Systems

The "Callback Functions" approach is useful for simple applications. By simple, we mean:

- each callback executes a series of instructions that never block;
- objects or procedures responsible for treating the incoming messages do not interact with information coming from other sources (other clients or servers, or graphical user interface for instance);
- when a specific message arrives, it is always treated in the same manner.

The "callback functions" concept fails if the execution of a function must be temporarily suspended to wait for another event. If that happens, the function must be divided in two subfunctions. When the second subfunction is called, it must check that the subfunction containing the first part of the code has actually been executed. This becomes quickly unaffordable when the complexity of the application grows.

In network servers, real-time control systems, applications for factory automation, layered protocols, events from various sources must be handled. Moreover, these events can interact with each other and, depending on the state of the system, a particular event can be treated in different ways. In addition, a predefined order of treating the events is often required. Using the current techniques, the implementation of such systems presents several difficulties because they lack the mechanisms for data flow structuring, for the interaction between callback functions and for Finite State Machine [3] design.

2.3.1 A Communications System Implementation

In order to illustrate the difficulties of the different Event Handling techniques, consider a protocol application composed of several Communication Entities. Our example examines a subpart of the application: two communication entities CE1 and CE2 work in parallel assembling messages from less complex messages that come from different sources (see figure 1). Messages "a1", "a2" and "c" come from the network, message "b" is produced by another entity. Entity CE1 is responsible of assembling "aba" messages composed by two "a" messages (an "a" message can be "a1" or "a2") and one "b" message.

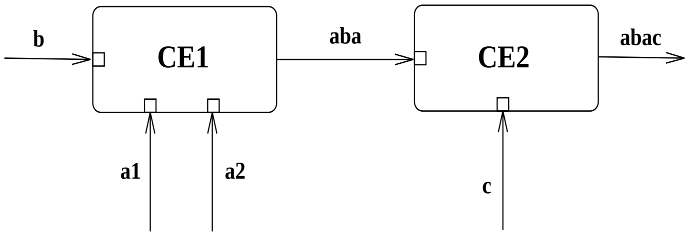

Figure 1: Communication Entities of a Protocol

The Finite State Machine (FSM) of the CE1 entity is represented in figure 2. In order to work correctly, CE1 must first receive an "a" message, then a "b" and finally another "a". Afterwards, it must send the "aba" message to CE2 and then, begin again with its cycle.

Communication entity CE2 works in a similar way, assembling "abac" messages.

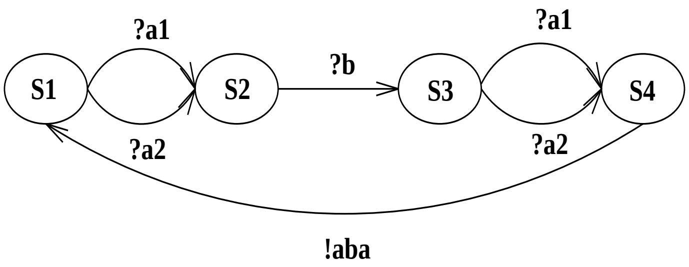

Figure 2: Finite State Machine of Communication Entity **CE1**

Using the current Event Handling techniques, one can develop this example, but not without a good amount of effort. The possible solutions using polling or various processes will not be treated since they are very expensive in OS resources and thus not realistic. We will concentrate on the possible solutions using the "callback functions" concept.

One possible approach is to always be ready for any event: all the communication channels ("a1","a2" and "c") are enabled. The main problem with this approach is that we do not know what to do with the messages that arrive when we are in a state in which we cannot treat them. In this case, it is necessary to create and manage fifos for the incoming messages and the actions to be taken must be postponed. The application becomes complicated very quickly when the number of events increases, since each time an event occurs we must check the fifos to know if postponed actions must also be executed.

Another approach is to enable/disable the channels from which the application can receive messages. The list of enabled channels will depend on the state of the system. In this case, the callback functions will update the list of channels. This approach is simpler. However, there is still a problem with the internal events ("b" or "aba" messages) since in that case we have no channel to disable. Again the only possibility is to create and manage fifos.

3 Description of our Extensions to C++

3.1 Active and Passive Objects

In our conception, the events do not come from the main loop only (or the system), but they can also have their origin in the surrounding objects. Hence, the only addition we made to C++ is the possibility for the objects to run concurrently. Although the objects communicate with the others by synchronous method calls only, all the synchronization problems that can be solved by any other multi-tasking language [13], can also be solved with our proposal.

In our proposal, there are two kinds of objects:

Passive objects: These are the usual C++ objects. Once an object has been instantiated, its methods can be called at any time.

Active objects: An active object introduces concurrency. Its behavior is defined by the programmer inside a function called the **body**. At the instantiation of an *active* object, a new thread of execution carrying the body of the instantiated object, is started. This **body** runs in parallel with the instantiator. An active object is protected in such a way that only one method can be executed at a time [9].

In the same way as passive objects are instances of classes, active objects are instances of **active** classes. Active objects are created using the same mechanisms as those used for passive ones. Thus static and dynamic instantiations of active objects are also available.

The syntaxes used to declare active classes and usual C++ classes are very similar. The differences are that, in an active class, the keyword *active* precedes the keyword class and that a new member function can be added, ie. the body.

3.2 Call And Accept Statements

At the lexical level, calling a passive or an active object is absolutely identical. The syntax used to express a method call is:

object->method(parameters)
or
object.method(parameters)

The caller of an active object method is suspended until the called method has been explicitly enabled.

In order to enable a method for one call, an object uses the following instruction (usually located within the body) (see Figure 3):

```
accept method;
```

As soon as the synchronization takes place, the method is disabled, and its code is executed. In order to accept further calls the *accept* statement must be re-executed. The code that executes the *accept* statement is suspended until the method has been completely executed.

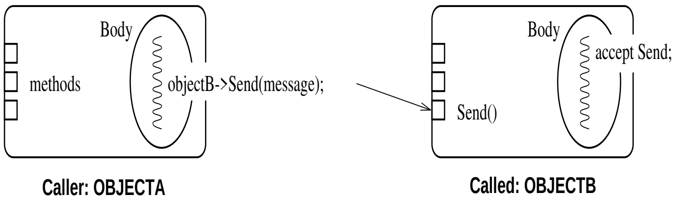

Figure 3: Communicating Objects

3.3 Interaction Choice, Select Statement

In order to express the choice between multiple synchronization alternatives, an active object can execute a *select* statement. This statement is very close to the `select` statement found in Ada [5]. However the semantics of our *select* statement are based on the CCS choice operator [3], allowing as many acceptances, as many calls and as many timeouts as needed.

Here is an example of the usage of our *select* statement:

```
select {
    accept send;
      ...instructions...
  ||
    waituntil(date1);
      ...instructions...
  ||
    objectA->deliver(x,y,z);
      ...instructions...
}
```

This *select* statement is composed of three **alternatives**. The execution of the *select* is suspended until either one of the **events** in the following subset happens:

- method *send* is called by another object
- or another object *"objectA"* accepts method *deliver*
- or time reaches *date1.*

Only one of the events is selected and fired by the kernel. Which one is selected depends on the behavior of the surrounding objects. The execution of the active object is resumed only when the event is completed (i.e. the called or the accepted method has returned or the timeout has been fired) and the instructions of the alternative triggered by the selected event, are executed.

4 Handling Events with our Approach

Our approach presents all the advantages offered by object orientation ie. encapsulation, reusability, maintainability, extensibility, etc., and goes far beyond. Using our previously explained Active Objects, the problems mentioned in section 2 are solved in a very simple way. The first reason is that each object has its own life and works independently from the others. Each object is responsible of its own incoming and outcoming events. The events can come from the normal I/O channels or from the surrounding objects.

Another reason is that Active Objects can accept and send messages just when they need to do so. Hence, the order of treating the events can be easily established making our approach very useful for building Finite State Machines.

Furthermore, using our **select** statement, an object can handle multiple events of very different nature in real time: a message coming from network, a keyboard input, a mouse click, etc. The **body** of such an object would be:

```
select {
  ActiveSocket1->recv(message,sizeof(message));
    AnalyseAndTreat(message);
||
  WindowText1->read(message);
    Evaluate(message);
||
  Button1->pressed();
    exit;
}
```

ActiveSocket1 represents an active object that treats a connection with the network. WindowText1 is an active object that has a window on the screen in which we can read and write. Button1 is a button that can be pressed using the mouse. Each of the latter objects is responsible for its own task, works independently from the others and satisfies services only when it is ready to satisfy them and when they are asked by an application.

4.1 The Communications System Implementation with our Approach

Using our proposal, the example of the communicating entities explained in section 2 is very easy to implement. CE1 and CE2 (figure 1) are implemented as Active Objects whose behaviours are programmed in their bodies.

In our proposal the problem of fifos does not exist because Active Objects explicitly decide when to accept internal or external events.

The CE1 **body** is the following; notice that it represents exactly the FSM of the figure 2. With the approach of an infinite loop that dispatches events, one can hardly know which is the state at a given time.

body of CE1

```
while(1) {
  select {                                    // state S1
    ActiveSocket1->recv(MessageA,sizeof(Message));
  ||
    ActiveSocket2->recv(MessageA,sizeof(Message));
  }
  TreatFirstA(MessageA);

  accept ReceiveBs;                           // state S2
  TreatB();

  select {                                    // state S3
    ActiveSocket1->recv(MessageA,sizeof(Message));
  ||
    ActiveSocket2->recv(MessageA,sizeof(Message));
  }
  TreatSecondA(MessageA);

  R2->ReceiveABAs;                            // state S4
}
```

5 Conclusions

Current Event Handling techniques suffer from some limitations when developing complex software systems. Although object oriented approaches present several advantages compared to normal techniques, these advantages are often not sufficient to overcome all the limitations.

We believe that these limitations can be overcome by integrating concurrency within objects. We propose an environment that nicely merges the object orientation and the multitasking approach. Our proposal remains very close to the C++ philosophy while integrating the concept of Active Objects. Active Objects have their own life and work independently from the others.

Since our Active Objects can accept and send messages only when they need to do so, they are very useful for developing Finite State Machines and offer a good method for data flow structuring. Furthermore, our proposal is particularly useful for handling multiple events of very different nature in real time.

We think that the use of toolkits to develop User Interfaces [11] for complex applications is far from being efficient. Our conception merges the worlds of Network Software and Graphical User Interfaces (GUI) by handling the different events in the same way. Hence, the integration of GUIs in networking applications becomes trivial.

Our extensions have been successfully used in the development of Communication Network Protocols, Teleconferencing and Multimedia Applications. Our experience [4] [14] [15] demonstrates that these extensions do not affect the efficiency of the systems.

The current version of our extensions, containing a preprocessor, a run time library and two libraries that implement UNIX sockets and Motif widgets as active objects, is freely available on the anonymous server: ftp@ltisun.epfl.ch. (Please state your e-mail address as password).

References

[1] G. Agha (Ed.), *Research Directions in Concurrent Object-Oriented Programming* The MIT Press, 1993.

[2] M.Gabassi and B.Dupouy, *L'informatique répartie sous UNIX*, Collection de la Direction des Etudes et Recherches d'Electricite de France. Editions Eyrolles, 1992.

[3] R. Milner, *Communication and concurrency*, International series on computer science. Prentice Hall, 1989.

[4] G. Conti, *Méthodologie d'implantation des protocoles de communication*, Doctoral thesis, Department of Computer Science, Ecole Polytechnique Fédérale de Lausanne, CH-1015 Lausanne, Switzerland, May 1990.

[5] R.J.A. Buhr, *System Design with ADA*, Prentice Hall, 1984.

[6] B. Stroustrup and M.A. Ellis, *The Annotated C++ reference manual.* Addison-Wesley Publishing Company, 1991.

[7] D.Schmidt, *The Reactor. An Object-Oriented Wrapper for Event-Driven UNIX I/O Multiplexing*, C++ Report, February 1993.

[8] D.Schmidt, *Object-Oriented Techniques for Developing Extensible Network Servers*, C++ World Conference, Dallas, 1993.

[9] C.A.R. Hoare, *Communicating Sequential Processes*, International series on computer science, Prentice Hall, 1985.

[10] P. Asente and R. Swick, *X Window System Toolkit: the complete programmer's guide and specification*, Digital Press X-motif series, 1990.

[11] D.A. Duce, M.R. Gomes, F.R. Hopgood and J.R. Lee, editors *User Interface Management and Design*, EurographicsSeminars, Springer-Verlag, 1991.

[12] S. Khoshafian and R. Abnous, *Object Orientation. Concepts, Languages, Databases, User Interfaces*, Wiley, 1990.

[13] B. Wyatt, K. Kavi, S. Hufnagel, *Parallelism in Object-Oriented Languages: A survey*, IEEE software, November 1992.

[14] A. Divin and C. Petitpierre, *An Object Oriented Method for Implementing Layered Protocols*, Formal Description Techniques FORTE, Boston, 1993.

[15] G. Caal and A. Divin, *Implementing Real-Time Applications with Concurrent Objects*, Proceedings of the 6th Euromicro Workshop on Real-Time Systems, Sweden, 1994.

[16] Yonezawa and Tokoro, editors, *Object-Oriented Concurrent Programming*, The MIT Press Series in Computer Systems, 1987.