European Research Seminar on Advances in Distributed Systems

Title: Open to Suggestions
On Adaptable, Distributed Application Support Architectures

Subject Area: Distributed Operating Systems
Principal author: Chris Zimmermann
Email: czimmerm@dsg.cs.tcd.ie
Phone: +353-1-608-1543
Fax: +353-1-6772204
Address: Distributed Systems Group
Department of Computer Science
Trinity College,
Dublin 2,
Ireland.

Open to Suggestions

On Adaptable, Distributed Application Support Architectures

Chris Zimmermann Vinny Cahill

Distributed Systems Group,
Department of Computer Science,
Trinity College, Dublin 2, Ireland
{czimmerm, vjcahill}@dsg.cs.tcd.ie
http://www.dsg.cs.tcd.ie/dsg_people/{czimmerm/czimmerm.html, vjcahill/vjcahill.html}

Abstract

In this paper we analyze the requirements that will be placed on future operating system architectures and conclude that most application areas will need different support from the operating system. What is needed for these applications is an operating system environment which can easily be adapted to application-specific needs. To address this problem we propose structuring the operating system as a collection of objects which allows the customization of the behaviour of application-level objects at run-time. This is achieved by using a metalevel architecture which allows the adaption of objects to application-specific needs dynamically. We illustrate our proposal using real-time environments as an example.

1 Introduction

Taking a close look at the requirements of emerging distributed application areas like distributed multimedia systems, the full range of the services that conventional, general purpose operating systems like Unix [1] offer, is not needed and as Mullender observes can even be an obstacle for this kind of application[1] [15]. What is needed is an operating system environment which can be adapted to application-specific needs easily. To motivate this, the following summarizes some of the the major requirements on future operating systems imposed by application demands:

- Distribution: With the advent of Information Highways on our desks [7], support for distribution will not be an add-on in future operating system architectures, but will be a prerequisite. Distribution cannot be taken on its own, but rather is a basis for other functionalities such as multimedia and transaction processing.
- Object-oriented design and implementation: Object-orientation in the design and implementation of application software is *the* de-facto standard in both industrial and academic environments nowadays. Due to this fact, the next generation of operating systems should provide support for objects as the units of computation. Furthermore, as the advantages of object-orientation in operating system design become obvious [12], operating system themselves will tend to be crafted using objects.
- Adaptability: Different applications have different demands on each functionality supported by the operating system. As an example consider persistence. Some applications for example do not need persistence, so the underlying operating system needs to support only volatile objects, whereas at the other end of the scale, other applications may demand a high degree of persistence where objects must survive system crashes. Thus the operating system has to support more than one option in order to satisfy these various demands[2].

[1] In the case of multimedia consider, for example, the lack of support for continuous media found in these operating system kernels.

[2] In terms of persistence the range could be: volatile (no support for persistence), persistent (an object's lifetime is not limited to the run-time of an application) and durable (the object is guaranteed to survive even system failures such as crashes).

In the remainder of this position paper we introduce our approach to supporting these demands. We offer an adaptable object model allowing applications to customize the behaviour of objects at run-time. This is achieved by employing a collection of distributed kernel objects providing minimal operating system functionality which can be tailored to application needs.

The rest of the paper is structured as follows: after an introduction to our distributed kernel framework named Tigger, we discuss an object execution model offering support for adaptable software. A section on related work and an outlook conclude this paper.

2 Tigger Overview

The Tigger project is developing a framework for the construction of a family—the Tigger Pride–of distributed object-support platforms suitable for use in distributed applications ranging from embedded soft-real time systems (actually 3-D arcade and console video games) to concurrent engineering frameworks [4]. Thus customizability, extensibility and portability are major design goals of Tigger in addition to distributed object support.

The baseline for the Tigger project is a set of minimal object-support platforms supporting at least four primitive abstractions: distributed objects; persistent objects; activities (i.e. distributed threads of control) and extents (i.e. protected collections of objects). A given object may be both distributed and persistent.

Members of the Tigger Pride may provide additional abstractions supporting, for example, security and transaction services. In addition, members of the Pride may be specialised to support different policies. For example, distributed objects will be supported using both RPC (function shipping) and/or DSM (data shipping) techniques. The result is that a Tigger which provides the necessary services for the target application domain can be constructed.

Each instantiation of the Tigger framework is intended to provide the necessary support for the use of some object-oriented language(s) for the development of distributed and persistent applications. Thus the fundamental interface provided by a Tigger is that provided for the language implementer. The interface used by an application developer is that provided by a supported language.

3 The Object Execution Model

An integral part of our strategy for supporting an adaptable and flexible operating system environment is the supported object execution model. This object execution model, which is layered above Tigger instantiations, allows the customization of object behaviour at run-time. Since different applications have different demands, for example, in terms of parallelism, synchronization and access control, the software-layer above Tigger offers an interface by which the execution of an object's code can be influenced.

By an *object* we mean the unit of computation consisting of slots [10] which denote either portions of executable code[3] or data[4] [3]. A *functionality* is a building block supported by individual instantiations of Tigger. Examples include support for transactions, security and real-time. The *behaviour* of an object is defined as the way that the code of the object, provided by the application programmer or language implementor, is executed. Applying different functionalities to an object changes its behaviour at run-time.

The following section motivates our approach by taking an MPEG-decoder object as an example.

3.1 Motivation

Imagine an object which is part of a distributed multimedia application and handling MPEG data [5]. This object fetches the compressed video stream from a network connection or a storage device like a hard disk, decompresses

[3] Called member functions in C++ [17] or methods in Smalltalk [6].
[4] Called member variables in C++ or instance variables in Smalltalk.

it and displays it in a window on the screen controlled by the local window manager. By doing so, this object has to cope with a range of constraints.

On one hand, if the individual frames of the video stream are large and the window in which they are displayed is fully exposed, it makes sense to distribute the work load onto multiple threads concurrently working on decompressing the frame in order to speed up the computational process. These threads then have to be synchronized accordingly when accessing shared data slots.

If, on the other hand, the individual frames are rather small and the output window is partially covered by another window, the decompression of the video stream may be handled by fewer threads and the synchronization may become less important so that different scheduling and synchronization protocols can be employed.

In conclusion depending on the parameters of the video stream, this object has different requirements in terms of parallelism, scheduling and synchronization. Unfortunately, the parameters are normally not known when the class[5] is coded during the application design process, but become obvious at run-time when the final parameters of the MPEG stream can be determined. This motivates the need for a means to influence and control certain aspects of the real-time behaviour of the object at run-time.

Continuing the example of the MPEG-decoding object, we can identify the three major building blocks which are used to achieve the distributed real-time behaviour:

- Parallelism: When dealing with real-time, concurrency inside an object is often desirable which leads to the notion of active objects [24]. A variety of object models regarding the number of threads concurrently active inside an object are possible depending on the point in time at which a thread is created.
- Scheduling: Since real-time requires different scheduling compared to systems not dealing with real-time aspects, the real-time functionality supports typical real-time scheduling algorithms like Earliest Deadline First (EDF) and Rate Monotonic (RM) [13]. The individual threads active inside an object are scheduled according to these policies.
- Synchronization: As Sha observes [16], applying ordinary synchronization mechanisms like mutexes without proper scheduling leads to a variety of problems in the realm of real-time. The third building block therefore offers different real-time synchronization mechanisms such as priority inheritance, priority ceiling [16] and stack reservation protocols [2] in order to overcome these problems.

Together with the specification of an object in terms of code and data slots, an application programmer coding the MPEG-object includes a set of rules stating how the behaviour should be adapted according to the final parameters of the MPEG video stream. This can be done on a per-object basis (by *default*) or for individual slots.

3.2 Discussion

The discussion above motivates the notion of an open implementation [12, 11]: certain details of the implementation are revealed and the application programmer is allowed to change these. The potential benefits of doing so are countered by two obstacles: one has to be careful about what to expose and the details must still be abstract enough in order to be manageable. If the wrong aspects of an implementation are exposed, the danger of crashing the system by altering the wrong parameters is obvious. If too much detail is offered to the application programmer (i.e. the open implementation is not sufficiently abstract), it is possible that the programmer is overloaded with unimportant details making the open implementation too complex to be really useful.

From a conceptual point of view, controlling the behaviour of an object's implementation in this way is done by a metaobject hierarchy [10]. A metaobject, allowing the control of various aspects of a baselevel or application object, resides on the first metalevel (M^1). Since this metaobject is an object in its own right, itself is controlled by a metametaobject residing on the metametalevel (M^2). Theoretically, this allows an infinite tower of metaobjects

[5] From which the object is instantiated at run-time.

[23]. For practical purposes the number of metalevels is limited to two or three at most [21] in order to cope with the growing complexity of such hierarchies.

This metalevel approach represents a means of structuring a system; it does not say much about the actual control of a system during *run-time*. In order to allow this, a system must be able to reason about its behaviour [14]. This involves dynamically changing certain aspects of the system in order to cope with different demands.

To conclude this discussion, our approach to allowing the customization of objects at run-time is a reflective metalevel architecture of an open implementation. The programmer specifies the behaviour of a baselevel object by using an *Object Meta Interface* (OMI). For each functionality provided by the Tigger, the metalevel offers its own OMI. Taking the example from above, the OMI allowing the control of real-time aspects is structured further into interfaces dealing with parallelism, scheduling and synchronization.

4 Implementation

This section sketches some of the details of the implementation. Due to space constrains, we will discuss these issues only briefly, a more detailed review can be found in another paper [25].

Since real-time is not the only functionality provided by the underlying Tigger, a layer named *genCore* below the individual OMIs allows a mapping of functionality onto objects and slots by offering a generic interface. This genCore is also responsible for managing dependencies between the individual functionalities. In this sense, the different OMIs provide a form of syntactic sugar, because strictly speaking the genCore interface could be offered to the application programmer directly. But doing so would be error-prone and tedious: instead of using the functionality-related OMI, the programmer would use an interface like `map(object#12, slot#3, function#14)`.

The following metaobject hierarchy can be identified (see Fig. 1)[6]:

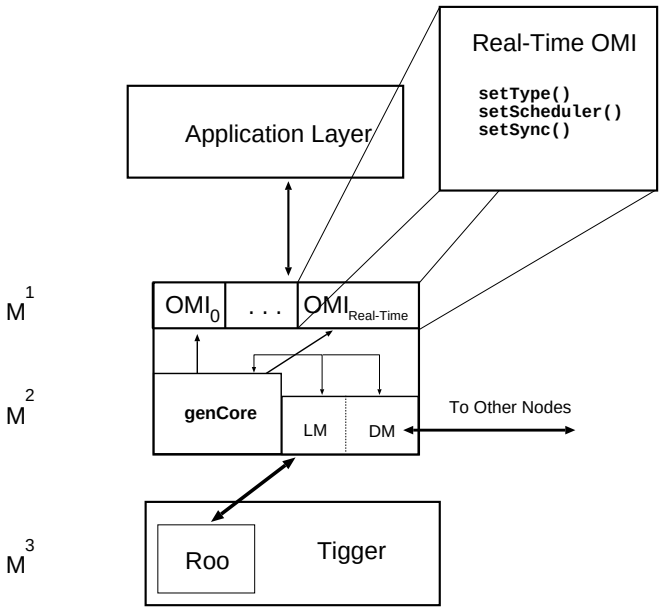

Figure 1: Implementation as a Hierarchy of Metalevels

- M^1: The real-time OMI which is used by the baselevel object programmer resides on this level. This OMI allows the specification of different active object models in regard to parallelism (`setType()` in Fig.

[6] For clarity we continue our real-time example.

1), different real-time scheduling policies (`setSchedule()`) and synchronization mechanisms (`setSync()`). These routines in turn use the genCore interface to map the specific functionality onto slots and objects.

- M^2: As discussed above, the genCore which controls the individual OMIs resides on M^2. In the case of distributed real-time, the genCore deploys the real-time functionality which consists of a Local Manager (LM) and a Distribution Manager (DM). The LM in turn controls a small local real-time kernel. Since we are dealing with the notion of distributed real-time, it must be guaranteed that real-time constrains are not only met in the local case but also in the distributed case. The DM, which interacts with DMs on other nodes to make sure that real-time constrains in the distributed case are met, employs mechanisms like time fences [20].
- M^3: On the metametametalevel, a small real-time executive named Roo, which is part of the underlying real-time instantiation of the Tigger, essentially provides real-time threads. These real-time threads contribute to the different active object models as discussed above [26].

5 Related Work

The use of metalevel architectures to design and implement operating systems or part thereof has been discussed in approaches like Apertos [22] and DROL [18]. In contrast to our work, these approaches typically offer only *one* mechanism[7], whereas our approach offers the programmer a variety of choices to select from.

Also, the capability of altering behaviour *dynamically* (i.e. during run-time) is normally not found in object-oriented operating systems. For example, Choices [9, 8] offers a means to change the scheduling characteristics of an operating system but this cannot be done on the fly, i.e. at run-time.

6 Conclusion and Outlook

As Kiczales shows, there can be significant advantages in exposing certain details of operating systems to application programs using these operating systems [12]. By providing applications with a means to alter these details, they are able to adapt themselves to changing requirements.

We presented a minimal operating system architecture allowing the customization of object behaviour as one the the major features offered by the object execution model supported by this kernel framework. Besides the discussed real-time example this architecture supports the alteration of every functionality the underlying Tigger offers.

Since the OMI allows easy changing of an object's behaviour, the full set of parameters of the target environment need not to be known in advance. Taking the MPEG example from above, it means that the final configuration in terms of throughput of the MPEG data stream can be determined at run-time and the single aspects of the behaviour such as parallelism and scheduling can be set accordingly on the fly.

Taking this idea one step further leads to a software system that can adapt *itself* to the changing needs of the application environment. A software layer residing above the OMI measures the behaviour of application objects resulting from the executed code and identifies hot spots where a customization of object behaviour would lead to an optimization. Retaining the real-time example from above, this software would scan the number of active threads inside an object and compare it to the overall CPU utilization and missed deadlines of this object. If this rate reaches a certain threshold[8] it increases the number of threads allowing a finer granularity of concurrency and hence a better overall throughput. Again, this strongly relates to the notion of reflection as discussed above.

The status of this work in progress is that we are currently designing and integrating this metalevel architecure within the existing Tigger framework. Roo is implemented and stable on multiple platforms like standard Intel PC Hardware. The genCore and OMIs are still in the design phase while we investigate the requirements for the

[7] For example, Apertos just allows the programmer either to make an object volatile or persistent [19].

[8] I.e. there are too many missed deadlines and not enough threads to spread the workload on.

RT-OMI which will be our first OMI to implement, taking distributed multimedia application environments as our main source of input. From this target environment we expect sufficient contribution in order to derive the final shape of the RT-OMI.

References

[1] M. J. Bach. *The Design of the Unix Operating System.* Prentice Hall International, 1987.

[2] T. P. Baker. Stack-Based Scheduling of Realtime Processes. *Real-Time Systems*, 3(1):67–99, 1991.

[3] G. Booch. *Object-Oriented Design with Applications.* Benjamin/Cummings, second edition, 1993.

[4] V. Cahill, C. Hogan, A. Judge, D. O'Grady, B. Tangney, and P. Taylor. Extensible Systems—The Tigger Approach. In *Proceedings of the SIGOPS European Workshop*, pages 151–153. ACM SIGOPS, Sept. 1994.

[5] D. L. Gall. MPEG: A Video Compression Standard for Multimedia Applications. *Communications of the ACM*, 34(4):46–58, 1991.

[6] A. Goldberg and D. Robson. *Smalltalk-80: The Language.* Addison-Wesley, 1980.

[7] A. Gore. Remarks by Vice President Al Gore at National Press Club. Washington DC, December 21, 1993.

[8] N. Islam and R. Campbell. Uniform Co-Scheduling Using Object-Oriented Design Techniques. In *International Conference on Decentralized and Distributed Systems*, Palma de Mallorca, Spain, 1993.

[9] R. E. Johnson and V. E. Russo. Reusing Object-Oriented Design. Technical Report UIUCDCS 91-1696, Department of Computer Science, University of Illinois, 1991.

[10] G. Kiczales et al. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[11] G. Kiczales et al. Open Implementations: A Metaobject Protocol Approach. In *Proceedings of the 9^{th} Conference on Object-Oriented Programming Systems, Languages and Applications*, 1994. Tutorial notes.

[12] G. Kiczales and J. Lamping. Operating Systems: Why Object-Oriented? In *Proceedings of the 3^{rd} International Workshop on Object-Oriented Operating Systems*, 1993.

[13] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[14] P. Maes. Computational Reflection. Technical Report 87.2, Artifical Intelligence Laboratory, Vrije Universiteit Brussel, 1987.

[15] S. J. Mullender. You and I are past Our Dancing Days. In *6^{th} ACM SIGOPS European Workshop on "Matching Operating Systems To Application Needs"*, pages 95–99, 1994.

[16] L. Sha et al. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 9 1990.

[17] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Second edition, 1992.

[18] K. Takashio and M. Tokoro. DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems. In *Proceedings of the 7^{th} Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 276–294, 1992.

[19] T. Tenma et al. Implementing Persistent Objects in the Apertos Operating System. In *Proceedings of the 2^{nd} International Workshop on Object-Oriented Operating Systems*, 1992.

[20] H. Tokuda and C. W. Mercer. ARTS: A Distributed Real-Time Kernel. *ACM Operating System Review*, 23(3):29–52, 1989.

[21] Y. Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proceedings of the 7^{th} Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 414–434, 1992.

[22] Y. Yokote. Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach. In *Proceedings of the 1^{st} International Symposium on Object Technologies for Advanced Software*, pages 145–162. Springer Verlag, 1993.

[23] A. Yonezawa, editor. *ABCL*. MIT Press, 1990.

[24] A. Yonezawa and M. Tokoro, editors. *Object-Oriented Concurrent Programming*. MIT Press, 1989.

[25] C. Zimmermann and V. Cahill. Raising the Cub. In *Proceedings of the Annual German Unix Users Group Conference*, pages 79–86, 1994.

[26] C. Zimmermann and V. Cahill. Roo: A Framework for Real-Time Threads. In *Proceedings of the Workshop on Distributed and Parallel Real-Time Systems, held at the 9^{th} International Processing Symposium*, 1995.