Tigger Project

Raising the Cub

Distributed Real-Time Support in Tigger

C. Zimmermann V. Cahill
Distributed Systems Group
Department of Computer Science
University of Dublin

Distributed Systems Group
Department of Computer Science
University of Dublin
Trinity College, Dublin 2, Ireland.
Fax: +353-1-6772204

Document Status Final version
Distribution Public
Document # TCD-CS-94-44
Publication To appear in Proceedings of the Annual German Unix User Conference, 1994

Raising the Cub

Distributed Real-Time Support in Tigger

C. Zimmermann* V. Cahill†
Distributed Systems Group
Department of Computer Science
University of Dublin

Abstract

We present a proposal for an architecture supporting distributed objects exhibiting soft real-time behaviour. This support is aimed directly at the field of distributed multimedia applications. Since the architecture is designed in a modular fashion, we expect that this architecture can be easily extended to other application areas with similar demands such as distributed video games, a major future market. The design consists of a metalevel approach with four individual levels offering a clean separation between baselevel objects implementing application functionality, on one side and metalevel objects, responsible for control of behaviour of baselevel objects, on the other side.

1 Introduction

Object-oriented systems already represent a well-known and well-accepted environment for application development when applications are restricted to run on a single node of a network. With the advent of distributed systems there is now a growing demand for distributed, object-oriented systems allowing an application which uses objects to span several nodes.

In this paper we present a proposal for an architecture for the support of distributed objects suitable for applications with soft real-time demands like multimedia. This architecture will be implemented as an extension of the Tigger distributed object support platform.

The remainder of this paper is organized as follows: in section 2 we introduce the Tigger platform. In section 3 we discuss the design of our architecture based on this platform while in section 4 we describe the application of a metalevel approach to soft real-time object support within the overall architecture. The paper concludes with a section on the current status and future research issues.

*Email: Chris.Zimmermann@dsg.cs.tcd.ie
†Email: Vinny.Cahill@dsg.cs.tcd.ie

2 Tigger

The experience gained during the design and implementation of Amadeus which was part of the Comandos project [4] has had a great influence on its successor named Tigger . In this design iteration of our effort to support distribution of objects we aim at supporting a variety of different application needs including objects exhibiting soft real-time[1] behaviour. Tigger is designed to be modular, so that building customized versions of Tigger matching specific application needs is possible. A minimal subset known as the Tigger Cub [5] is the common denominator providing the basic services for higher Tigger layers.

This Cub represents the kernel of the Tigger system, supports distributed and persistent objects, and contains services for thread management and network-transparent interprocess communication (IPC). In order to support different operating system and network environments and to enhance portability, the Cub is subdivided into separate modules each responsible for encapsulating environment specifics such as the operating system interface and network access.

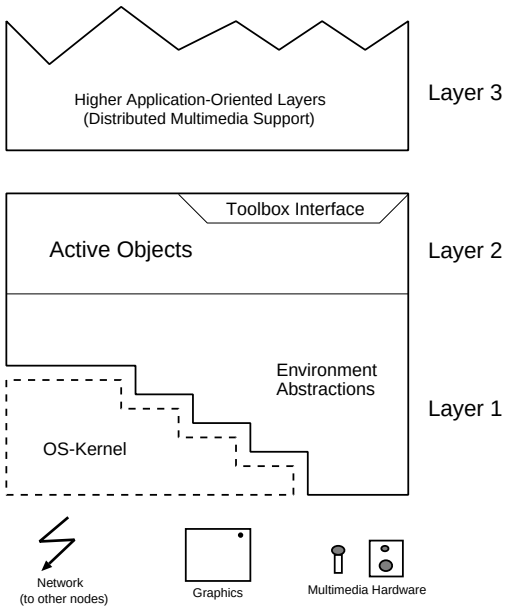

Figure 1: Software Architecture

3 Architecture Design

Our proposal for a suitable architecture for supporting distributed multimedia applications as an extension of the Tigger Cub is structured into three layers (see Fig. 1). In the following sections each layer is described in turn.

[1] *Soft* real-time in this context is defined in the usual way [9]: a failure to meet deadlines causes tolerable harm compared to the overall utility of the system.

3.1 Layer One: Environment Abstractions

This layer provides an abstraction from the underlying environment: operating system, network and multimedia hardware devices. Since the advent of new hardware is a frequent event in the field of multimedia computing and since Tigger is targeted to be ported to multiple operating system platforms such as Unix[2] [2] or Mach [13] [17] and network architectures, all changes that have to be made when introducing new system components should be restricted to this layer. Abstractions such as *streams* for high speed data transport, *links* to support multiple network architectures and *stream handlers* abstracting devices can be employed in this layer [16]. Where the underlying operating system and network environment fails to provide the necessary services, this layer uses the hardware directly as depicted in Fig. 1.

Since multimedia systems represent a kind of soft real-time environment, the Cub, which is responsible for thread management, has to be extended to support a suitable scheduling mechanism as discussed in section 4. This scheduling support must also be accessible to higher layers of the architecture, so that they have maximum control over the management of available CPU time. This soft real-time behaviour also applies to for the IPC part of the Cub which has to be modified accordingly.

3.2 Layer Two: Soft Real-Time Objects

Based on the services supplied by the bottom layer described above, the main task of this intermediate layer is to provide the notion of active objects supporting soft real-time behaviour. In this context the term *active object* denotes the fact that each object has at least one attached thread of control running inside of the object [1] [11]. The major advantage of this approach is that a change in the object's state can happen without any external triggering like a method call.

Furthermore, active objects allow a much finer scheduling granularity than passive objects, since each object has its own thread(s)[3] which can be scheduled independently from calling objects. Calling a method of an active object means delivering the parameters of the call, together with a selector for the method to be called, to the object. Threads running inside the object then accept the call and process the parameters inside of the method. This has the further advantage that calls without return values (so-called *asynchronous* calls) return much faster, since the calling thread need not wait for the callee to finish but can return immediately after delivering the parameters. Giving each object at least one thread of control also allows the independent scheduling of this object according to different (real-time) scheduling policies. This is important when objects deal with data which exhibits time-based behaviour as is the case with multimedia data.

In order to achieve soft-real time behaviour, this layer must interact closely with the bottom layer by using an appropriate scheduling interface, so that each thread of an active object is scheduled appropriately. Taking the realm of multimedia as an example, different multimedia data types like low-quality audio on one side and high-definition video on the other side have different throughput, and therefore different scheduling, demands.

This layer also has to extend the support for object distribution provided by Tigger since distribution has influence on both the local and the remote scheduling of objects. For example a local object interacting with a remote object must forward its timing characteristics to the remote one, otherwise deadlines may be missed (because the remote object may fail to react in a timely manner) and the real-time behaviour will break down.

These requirements are achieved through a toolbox interface which allows exact tailoring of the timing behaviour of objects at run-time by the higher layers discussed in the next section. This interface also

[2]Since we cannot guarantee *real* real-time behaviour when using Unix, it can only be provided on a best-effort basis.

[3]In case when there is more than one thread assigned to an object.

takes care of distribution aspects, since it allows the specification of the other objects on different nodes involved in the computational process.

3.3 Layer Three: Application Specific Support

In contrast to the two lower layers, which supply the basic mechanisms for distributed, active objects exhibiting soft real-time behaviour, this more application-oriented layer provides the support for distributed multimedia applications. Typical tasks of this layer include mixing of several multimedia data streams and synchronization of several data streams. The former is motivated by the requirement to support floor-passing mechanisms, where several multimedia data streams on the input side of the mixer are taken to form a single output data stream on the output side. This can be done on a percentage-basis (so that background noise can still be heard) or an individual stream can be chosen to be the output stream (in this case no background noise can be heard). Synchronization of multimedia data streams is used to provide the illusion of lip-sync, where the time gap between audio and video information must be kept to a minimum.

Other features not discussed above include, for example, sharing of multimedia devices by multiple applications or sophisticated naming schemes for locating multimedia hardware on remote nodes. In conclusion, this layer provides the basic facilities for application support. There are two possibilities for exploiting this functionality: either applications use it directly or some toolbox providing more abstract levels of functionality is placed between this layer and the actual application.

4 Design Issues

This section discusses the design of the toolbox character of Layer 2 taking scheduling and soft real-time behaviour of objects as an example. Due to the advantages of clean separation between application level functionality and the run-time characteristics of the objects (in this case scheduling), we chose a metalevel approach for the implementation of the toolbox interface. Since it cannot be assumed that the reader is familiar with the notion of metaobjects and metalevels, these are introduced before discussing our approach exploiting these techniques.

4.1 Metalevel Architectures

The design is based on the use of *metaobjects* which are members of a *metalevel*. Following the usual definition [7], a metaobject controls the behaviour of its corresponding baselevel (application) object. Metaobjects are grouped—at least conceptually—on a higher level, called a metalevel. Because metalevel objects directly control baselevel objects, one can think about this as the metalevel reasoning or *reflecting* upon the behaviour of the baselevel. Therefore systems implementing metalevel approaches are also called reflective systems [12].

Since metaobjects are also objects in their own right, they also have metaobjects on a corresponding metalevel. Since this could lead to a theoretically infinite tower of metalevels and metaobjects, real implementations of metaobjects like [15] typically restrict this tower to two or three levels in order to keep it manageable.

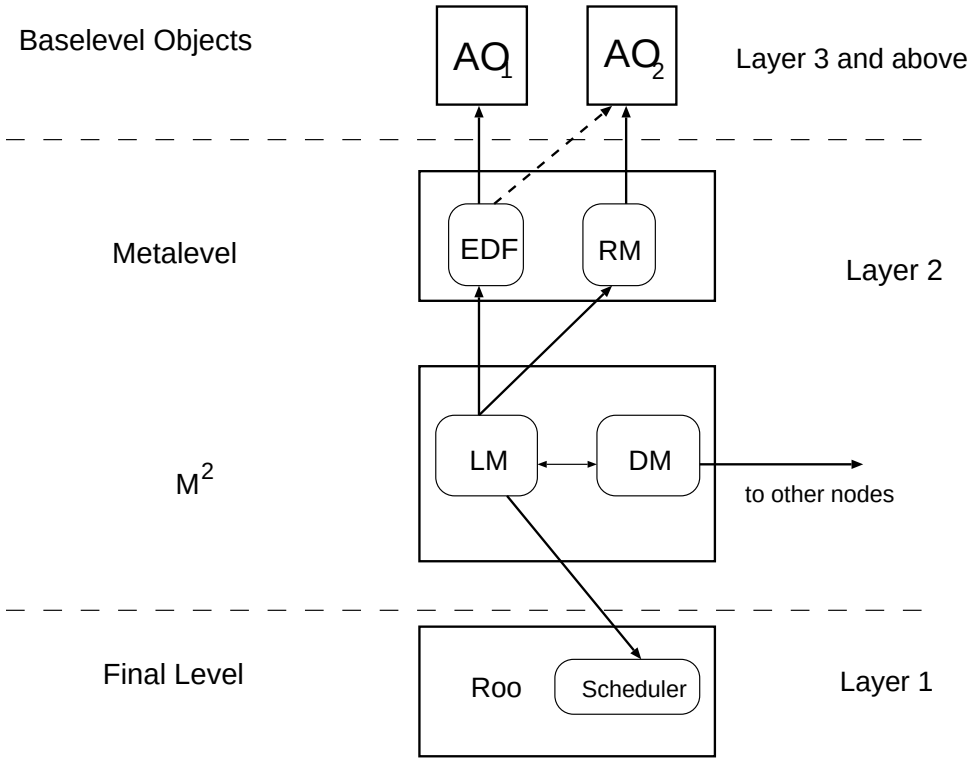

Figure 2: Metalevel Structure and Corresponding Layers

4.2 Scheduler Metaobjects

The proposed architecture will consist of a collection of metalevels[4] as depicted in Fig. 2: a baselevel, where application objects reside, a metalevel comprising different scheduling policies and controlling the scheduling of the baselevel-objects and a metametalevel (M^2) controlling the metalevel and implementing the notion of distributed real-time scheduling. The final level is a minimal real-time kernel, a real-time extension of the Cub providing scheduling support for threads.

- Baselevel objects: these are application-oriented objects (AOs), typically defined in Layer 3 or above, implementing multimedia functionality like MPEG video stream handling [6] or audio / video synchronization as described previously. Because they handle time-critical data (multimedia data with a demand for soft real-time), they must be scheduled accordingly.

- Metalevel scheduler objects: members of the first metalevel are metaobjects implementing different soft real-time scheduling policies like Earliest Deadline First (EDF) or Rate Monotonic (RM) [8]. They are directly responsible for the scheduling of methods (member functions in C++ [14]) of baselevel objects. If this rather fine scheduling granularity is not needed, a default policy for a whole object can be specified separately[5].

[4] In order to avoid confusion: *levels* in the following context denote the conceptual metalevel subdivision related solely to objects whereas *layers* reflect the structuring of the overall architecture as discussed in section 3.

[5] In fact, one specific scheduling policy (the *default* policy) is assigned to an object at creation time; this can be overridden at a later time by specifying a different scheduling policy for individual methods

- M^2 manager objects: since the metalevel objects are only concerned with the scheduling of baselevel objects and methods, a higher level has to take care of interaction with the real-time kernel scheduler (which is thread-based and knows nothing about active objects or object methods), distribution aspects and assignment of threads to objects and methods. This is done by manager objects on level M^2. Currently, M^2 has two members: a local manager (LM) metaobject which controls the different scheduler objects on the metalevel and—closely interacting with this local manager—a distribution manager (DM) responsible for interacting with other DMs running on different nodes. Since the scheduling of distributed objects must be coordinated in order to allow these objects to meet their timing requirements, this interaction is necessary. The local manager also provides the binding between the threads offered by the real-time kernel and the objects / methods supported by the metalevel.

- Final level: this final level consists of a minimal kernel offering the notion of soft real-time threads. This kernel is either part of the underlying operating system or is provided otherwise, i.e. as part of the environment abstractions. It is important to stress the fact that this kernel is a local one offering a thread-based interface and therefore does not know about distribution aspects or active objects. As discussed above, this mapping is left to the manager objects which are part of M^2.

Taking Fig. 2 as an example of the interaction between the different metalevels, here two application objects (AO_1 and AO_2) are scheduled by two scheduler objects on the metalevel: one implementing an earliest deadline first (EDF) scheduling algorithm, the other one responsible for rate monotonic (RM) scheduling. All methods of AO_1 are scheduled according to EDF[6]; whereas AO_2 is shared between the two scheduling metaobjects: the default policy is rate monotonic scheduling (the default policies assigned to each application object are denoted by a solid line); however, some of AO_2's methods are scheduled via EDF (as indicated by the dashed line). Both scheduling metaobjects are controlled by the local manager on M^2, which in turn interacts with the soft real-time kernel and the distribution manager.

5 Conclusion

This section serves two purposes: it discusses the status of the project and future research issues.

5.1 Status

Having laid out the road-map for the design of the above architecture we are now beginning to implement the proposal as an extension of the Tigger platform. Currently we are looking at the minimal real-time kernel as a basis for M^2. Since Tigger as a whole is aimed at supporting multiple platforms, the hardware dependencies in this real-time kernel have to be kept to a minimum. The first step in implementing this real-time kernel is the design and implementation of a generic threads package which supports multiple scheduling policies by offering a suitable scheduling interface to the local manager residing on M^2 on one side and which implements the necessary mechanisms for dealing with the notion of real-time[7] on the other side.

Following this, detailed evaluation of different application areas for distributed soft real-time objects including distributed multimedia systems will yield the requirements for the metalevel and M^2. More specifically, we expect for instance to gain knowledge about which exact scheduling policies to include on

---

of an object via an interface supplied by the scheduler metaobjects.

[6] This is the default policy for AO_1 as discussed above.

[7] These mechanisms include for example different locking protocols for mutual exclusion in real-time environments such as priority inheritance and priority ceiling [3].

the metalevel and how to model the interaction between the distributed managers on different nodes by looking at existing distributed multimedia systems such as [10]. However, it is already clear that we have to offer multiple, different policies like Rate Monotonic or Earliest Deadline First scheduling in order to cope with different demands that distributed, soft real-time applications may place on the architecture.

5.2 Outlook

Apart from the use of this architecture for distributed real-ime application support, other application areas with soft real time demands will also benefit from this approach. One example are simulations like pilot-training applications where the simulation must respond within a certain time so that the same real-time constrains are present as in the real system—the aircraft. Other examples include (distributed) video games: applications which are spread over several nodes with soft real-time requirements. This field is subject to major growth over the next years according to independent market surveys and forecasts. Since these games may also include multimedia aspects, a competitive application support environment must cater for both domains.

Since the lower two layers are intended to be generic enough, only the topmost layer has to be replaced in order to respond to the different needs of these applications. Since each of the layers has a well-defined interface, this replacement can be done quickly and without much effort. This leads to a chameleon-like approach: the application support policy (multimedia / games) and the corresponding interface (the colour of the beast) may change but underneath the skin the creature still remains the same—distributed soft real-time objects.

The scheduling architecture discussed above is only one specific aspect of customizing object behaviour at run-time. If our approach proves to be valid in this particular case, there is no reason why the toolbox interface should be restricted to distributed scheduling aspects only. One can imagine extending the tailoring process supported by the toolbox interface to notions including persistence and fault tolerance. One good example in this context is the notion of persistence: since there are several degrees of persistence ranging from non-persistent (the object's lifetime is bound to the lifetime of the application which created this object) to persistent (the object's lifetime is not bound the the application's lifetime) to durable (where an object even survives a disk crash because it has been backed up to a reliable archive system), it can be useful to allow an object to migrate from one domain to another dynamically.

6 Summary

We presented an application-support architecture for the use in distributed multimedia systems. Based on our experience with Amadeus and Tigger we proposed an approach consisting of three layers: one layer encapsulating all platform specifics like operating system, network and hardware; an intermediate layer providing the notion of distributed objects exhibiting soft real-time behaviour and, on top of that, an application-oriented layer providing application support like synchronization and mixing.

We argued that this topmost layer is subject to change in order to meet different applications demands like distributed games. Thus the lower two layers represent a kind of toolbox offering mechanisms for the management of distributed objects, from which one or more higher layers of the architecture can build customized objects which exactly suit their needs.

The design employs metalevel approach consisting of four levels altogether: a baselevel hosting application objects handled by applications themselves or application-oriented layers, a metalevel offering different soft real-time scheduling policies like RM or EDF. This metalevel is controlled by a M^2 implementing the interface to the local real-time kernel and coordinating scheduling activities with other nodes. Finally, a soft real-time kernel offers a thread-based interface, which is used by M^2 to assign threads to objects and individual methods resulting in the notion of active objects.

References

[1] P. America. POOL-T: A Parallel Object-Oriented Language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, 1987.

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

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

[4] V. Cahill et al., editors. *The COMMANDOS Distributed Application Platform.* Springer-Verlag, 1993.

[5] V. Cahill et al. Tigger family values. 1994. Talk at the CaberNet Workshop, Trinity College Dublin.

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

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

[8] H. Kopetz. Scheduling. In S. J. Mullender, editor, *Distributed Systems*, pages 491–509. ACM Press, second edition, 1993.

[9] H. Kopetz and P. Verissímo. Real Time and Dependability Concepts. In S. J. Mullender, editor, *Distributed Systems*, pages 411–446. ACM Press, second edition, 1993.

[10] B. Lamparter and W. Effelsberg. X-Movie: Digitale Filmübertragung und Darstellung im X-Window System. In *Proceedings of the GI-Jahrestagung*, pages 343–353. Springer-Verlag, 1991.

[11] K.-P. Löhr. Concurrency annotations. In *Proceedings of the 7^{th} Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 327–340, 1992.

[12] P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of the 2^{nd} Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 147–155, 1987.

[13] R. Rashid. Threads of a New System. *Unix Review*, 4(8):37–49, 1986.

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

[15] 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.

[16] C. Zimmermann. Das Switchboard: Ein Modell zur Multimedia-Programmierung in verteilten Systemen. *Offene Systeme*, 2(3):128–134, 1993.

[17] C. Zimmermann and A. W. Kraas. *Mach: Konzepte und Programmierung.* Springer-Verlag, 1993.