FLEXIBLE EVENT-BASED THREADING FOR QOS-SUPPORTING MIDDLEWARE

Thorsten Kramp and Rainer Koster

Department of Computer Science
University of Kaiserslautern
P. O. Box 3049, 67653 Kaiserslautern, Germany
{kramp | koster} @informatik.uni-kl.de

Abstract: Distributed computing platforms have made their way out of the research lab nowadays, yet support for some key application areas such as real time in general and multimedia in particular is still found lacking. Since a single platform is unlikely to be able to satisfy the widely diverse requirements of such QoS-sensitive application domains, we propose a low-level foundation called COOL JAZZ instead, which is designed for easy customisation and on which more specific middleware platforms for distributed heterogeneous environments with QoS-requirements can be built. COOL JAZZ combines support for concurrency, communication, and signal handling within a unified event-based processing model inspired by SDL. In this paper we outline the overall architecture of COOL JAZZ and present the concurrency mechanisms in detail.

Keywords: Event-based processing, user-level threads, user-defined scheduling, real-time, quality of service

1 INTRODUCTION

Research in distributed computing has made considerable progress during the last decade, successfully providing selective levels of distribution transparency while masking the heterogeneity of hardware architectures, operating systems, and programming languages. Run-time platforms based on international standards such as CORBA [14] and DCE [6] have even enabled widespread deployment of distributed applications in various fields of industry and commerce.

In certain key application areas, however, distributed processing is still dominated by ad-hoc solutions, specifically in the realm of QoS-sensitive environments. The reasons for this are twofold. Firstly, the structuring of QoS-supporting distributed run-time systems for optimal performance, predictability, and configurability is still not well understood [2]. Secondly, since the term QoS covers all kinds of non-functional aspects, a single platform would have to cope with requirements from such diverse areas as reliability, real time, and multimedia. It is therefore unlikely that a single platform or standard will be able to meet all criteria equally well. Instead, we expect a number of domain-specific middleware platforms and standards to emerge.

As a consequence, we propose a minimalistic approach in providing only an object-oriented low-level foundation for the development of more specialised QoS-supporting run-time systems for heterogeneous environments. For such a foundation to be successful, it is crucial that the abstractions provided are versatile, efficient, and predictable. Specifically, we concentrate on combining support for concurrency, communications, and signal handling — arguably the most important system services from a platform developer’s perspective — within a unified processing model similar to that of SDL [9]. In this model, threads resemble state machines that perform transitions in response to events only, during which further events are possibly sent to other threads.

The benefits of such an approach are fourfold. Firstly, an event-based processing model is versatile enough to efficiently cope with synchronous *and* asynchronous inter-thread communication within a single address space *and* across address-space and node boundaries. The more common abstraction of inter-thread communication exclusively based on shared memory, in contrast, becomes more and more complex to uphold efficiently in distributed environments, particularly in heterogeneous ones. Secondly, events underpin both discrete interaction *and* continuous media interaction (i. e., streams) according to RM-ODP [8]. Discrete interaction is composed of two events for announcements and four events for interrogations, whereas a stream is modelled as a sequence of event emissions and receptions from a producer to a consumer, respectively. Thirdly, the notion of explicit bindings is easily integrated. Explicit bindings are a major prerequisite for QoS negotiation and appropriate resource reservation. And fourthly, a processing model similar to that mandated by SDL eases the transition from SDL specifications to working code. To this end, current research on real-time extensions to SDL is of particular interest.

Even such a low-level foundation, however, cannot be expected to be equally well-suited for all QoS-sensitive environments likewise. A number of critical design decisions will inevitably show up during design and implementation. For these no single “best” solution can be identified and trying to be clairvoyant will only lead to users coding between the lines. This is particularly true in the areas of scheduling and interaction protocols. COOL JAZZ therefore systematically exposes critical design decisions via meta interfaces, applying the *open implementation* design methodology [10]. This way, the designer of a specific run-time system is free to override the default behaviour whenever needed. Considering concurrency, for instance, all scheduling code has been factored out and, instead, a user-defined scheduler is called whenever an event might induce a scheduling decision. Furthermore, in view of a potentially

large number of media types and yet unforeseen protocol functionality required by future applications, a communication framework allows to integrate new interaction protocols at user level.

The remainder of this paper is structured as follows. In Section 2 an architectural overview of COOL JAZZ is given, before Sections 3 and 4 focus on the event-based processing model and the flexible scheduling framework, respectively. Thereafter the performance of our threads implementation is evaluated in Section 5, demonstrating the general viability of this approach. Related work is considered in Section 6, before Section 7 finally presents our conclusions and plans for future work. Due to space constraints a description of the communication framework — although briefly sketched in Section 2 — had to be omitted and is discussed in detail in a companion paper.

2 ARCHITECTURAL OVERVIEW

As mentioned before, COOL JAZZ offers support in the areas of concurrency, communication, and signal handling. The corresponding mechanisms are cleanly integrated within an event-based processing model and, thus, are handled in a unified way.

2.1 Concurrency

Threading can facilitate programming on middleware platforms in general, yet is particularly useful for developers of servers or real-time applications [15]. COOL JAZZ therefore offers *user-level threads*, which in comparison with kernel-level threads promise even lower context-switching and thread-creation times, scale to a large number of concurrent threads, and particularly allow the scheduler to be tuned per application.¹ These threads adhere to the event-based processing model in that a thread is only activated when it receives a message from another thread. Thus, a thread behaves similar to a state machine, performing state transitions in response to messages.

Threads are addressed via globally unique identifiers that are meaningful across node boundaries. Albeit, we distinguish between threads within the same address space and threads that are possibly located in another address space, perhaps even on another machine. Messages may be sent to local threads directly, while for possibly remote threads communication is based on *messengers*. A messenger establishes a binding to another thread when initialised with the peer thread's unique identifier. The basic semantics of sending a message to either a local thread directly or to a possibly remote thread via a messenger is identical in accordance with the event-based programming model. Yet, a messenger may also provide for QoS negotiation and the more complex failure model of remote communication, for instance. Location transparency is therefore achieved in the sense that messengers hide the actual location of their possibly remote peer threads.

System-level programmers, however, are commonly discouraged from relying on readily available threads packages due to lack of control over scheduling [7]. Black-

¹User-level threads may be mapped onto kernel-level threads to exploit multi-processing on SMP machines. At the time of writing, though, support for multi-processing is not implemented in COOL JAZZ.

box implementations allow the user to choose a scheduler from a number of predefined ones at best, which is clearly insufficient in view of the highly diverse requirements imposed by QoS-sensitive environments. Consequently, in COOL JAZZ all scheduling code has been factored out and, instead, a user-defined scheduler is called whenever some incident might induce a scheduling decision. Typical examples are thread state transitions or timer interrupts. In real-time environments, where priority inversion usually must be bounded, this also includes the blocking of threads at synchronisation primitives. To allow for extensibility, user-defined schedulers are not only activated by COOL JAZZ itself, but may also be called within code added by the developer of a specific middleware platform (e. g., additional synchronisation primitives).

A detailed description of the processing model, the scheduling framework, and the overall performance of our implementation is given in Sections 3, 4, and 5, respectively.

2.2 Communication

In QoS-sensitive environments fine-grained control over communication is crucially important — particularly, in view of the large diversity of media types with their individual QoS parameters [3]. COOL JAZZ therefore integrates a flexible communication framework with support for explicit bindings. Thus, new interaction protocols can be easily added in user space as also suggested in [17].

The communication framework distinguishes four components, each contributing to a different level of abstraction. *Transport protocol wrappers* encapsulate any transport protocols available (e. g., TCP, UDP, ATM-based protocols). These wrappers are used by various *interaction protocols* such as IIOP, while some interaction protocols (e. g., shared-memory communication between address spaces on the same machine) may operate independently from any transport protocols. Both transport protocol wrappers and interaction protocols are assumed to process marshaled data in contrast to messengers, which rely on separate *marshalers* to do the actual marshaling according to CDR or ASN.1, for instance. *Messengers*, as mentioned before, represent bindings to threads that are possibly located within another address space, possibly even on another node. Aside from implicit bindings, messengers also allow explicit bindings to be established, for which QoS parameters over an ongoing sequence of discrete or continuous media interactions might be specified (e. g., latency, bandwidth reservation).

The composition of actual protocol stacks from components is primarily performed at compile time by choosing appropriate classes from the component class hierarchies. The decision for or against certain components will be based on their different functional properties (e. g., connection-oriented or connection-less) as well as on different non-functional properties (e. g., the ability to consider certain QoS parameters). As long as the functional interface is identical, however, components may also be replaced at run time. With such a late binding it becomes possible to take the current state of the system or the actual location of the peer thread into account, for instance. The choice (and composition) of a compatible protocol stack at the peer site is further supported by a typing scheme.

A detailed description of the communication framework in general and the readily provided base classes in particular is given in a companion paper.

2.3 Signal Handling

Signals are used by an operating system to notify code in user space of asynchronous events such as alarm expiry, periodic timer signals, or I/O availability. QoS-supporting middleware in particular relies on signals to become aware of such events in a timely fashion without the need and overhead of polling with system calls.

In COOL JAZZ signals are transformed into messages to threads and, thus, are cleanly integrated with the event-based processing model. To receive a message when the next signal of a certain type occurs, a thread first sends a request message to a core thread of COOL JAZZ, which is responsible for the type of signal requested. It is exactly this message that will be returned to the user thread as soon as the signal occurs and, thus, threads only must take care of messages. Furthermore, since a thread must register anew for each occurrence of a signal, signal messages are not created in an uncontrolled fashion by the COOL JAZZ core, enhancing predictability.

On signal reception, all messages registered for that particular signal are returned. Next, depending on whether preemption of the active thread is currently enabled, the scheduler is invoked immediately or at the very next instant that allows an interruption. In either case, signal notification is therefore performed with minimum latency.

2.4 Requirements on the Operating Environment

In general, COOL JAZZ imposes no special requirements on the underlying operating system and network, apart from signals for asynchronous notifications by the operating system. Context switches are based on the *setjmp/longjmp* mechanism of the standard C library.

Considering multimedia environments, however, we believe that predictability can be considerably enhanced with respect to process and network scheduling. To this end, we have already added a transport protocol with bandwidth reservation and clock synchronisation for ETHERNET LANs to the LINUX kernel [11]. Furthermore, we will investigate the effects of replacing the LINUX kernel scheduling policy by a proportional-share policy such as *lottery scheduling* [18], for instance, as also suggested in [4]. Such an approach is attractive because proportional-share scheduling promises to be a far superior basis on which predictable user-level schedulers can be built.

3 EVENT-BASED THREADING

The most common processing model for (user-level) threads associates a code function with each thread that is automatically called as soon as the thread is created and eventually left when the thread terminates. Such a code function is similar to the *main* function of a process. Within this model inter-thread communication is usually based on shared memory alone. A thread waiting for some data to become available from another thread either has to poll some shared variable or has to block at a consumer/producer semaphore, for instance. Furthermore, efficiently upholding the abstraction of shared memory in distributed environments in general and heterogeneous ones in particular becomes quite complex.

Figure 1. Anatomy of a COOL JAZZ thread.

With COOL JAZZ, in contrast, we argue for an event-based processing model. Again, a code function is associated with each thread, yet this function is called anew for every message a thread receives. In the state-machine model the code function can be seen as the state transition function. Such a code function is therefore similar to the core of an implicit event loop, which performs a series of steps in response to each message received. These steps possibly include sending messages to other threads and receiving further messages, before finally the thread's function is left with a return value. The return value indicates whether the thread awaits further messages and should remain in the system, or has terminated eventually and should be removed.

The resulting set of interacting threads is initially set into motion by a *root thread* created during system startup, which usually spawns a first set of user threads, sends some initial messages to a subset of these, and finally awaits termination. For all practical matters, messages can be assumed to originate from another thread, since signals are transformed into appropriate messages by dedicated core threads (cf. Section 2). These messages also re-animate the web of interacting threads when the system temporarily went idle.

Aside from the code function each thread is associated with at least two message queues. The *new-queue* holds incoming messages not yet inspected, while one or more *save-queues* store messages that have been inspected but cannot be processed immediately due to an outstanding response from another thread, for example.

Messages may either be sent synchronously if there remains nothing to do for a thread until an answer is given, or asynchronously whenever the reply is not needed immediately to continue or no response is required at all. Synchronous messages are termed as *calls*, whereas asynchronous messages are *sent*. If not explicitly stated otherwise, the term *send* is used to indicate that both asynchronous and synchronous communication is allowed within a particular context.

The default message class consists of five fields, namely *from*, *reply to*, *id*, *value*, and *constraint*; more specific message types can be derived from this class. The first two fields describe the originator of a message and to which thread a reply should be sent. The *from* field is automatically set as part of any send operation and *reply to* defaults to *from*. The semantics of *id* and *value*, in contrast, are completely user-defined. The *constraint* field finally describes further properties such as a deadline, which may be of interest to a scheduler. Its actual semantics, however, is user-defined again and, of course, strongly tied to the user-defined scheduler.

At each instant, a thread is either preemptable and may be interrupted in favour of another thread, or non-preemptable. A non-preemptable thread processes undisturbed until it blocks, explicitly allows a reassignment of the processor, or returns from its associated function. The preemption status can be dynamically toggled on a per-thread basis.

3.1 *Handlers and Core Threads*

The resulting web of threads and bindings is inherently asynchronous in the sense that processing and communication take a certain and sometimes unpredictable amount of time to be carried out. The goal of QoS management is to constrain such asynchronous behaviour, prescribing worst-case execution times or imposing limits on the variance of inter-frame delivery times, for instance.

Adding QoS management threads to this model for monitoring and control, however, raises the difficulty that these controlling threads would be operating asynchronously as well and, thus, are themselves part of the environment they control. To break this recursive management, the notion of *reactive objects* has been introduced by Blair and Stefani [2]. Reactive objects are state machines whose state transitions in response to messages are assumed to be performed instantaneously in no time, that is, their processing is atomic with respect to the environment. Provided the execution time of a reactive object is at least one of order of magnitude faster than that of the threads controlled, these assumptions are valid even in the real world and reactive objects can be used for all kinds of real-time control.

COOL JAZZ provides *handlers* for the implementation of reactive objects. Handlers are identical to threads, yet are always invoked instantaneously and execute non-preemptively at the invoking thread's priority. Since sending a message to or calling a handler always preempts the active thread in favour of the handler, it is not necessary for handlers to own a context of their own. Instead, a handler borrows the context of the active thread, effectively avoiding context switches.

Aside from their role as reactive objects, handlers are also used as *core threads* within COOL JAZZ to notify the run-time system of asynchronous signals of the operating system with minimum latency (cf. Section 2). Using handlers as core threads, only the unavoidable context switch from the active user thread to the signalled user thread is performed; otherwise, with plain user threads instead of handlers, two context switches would be needed.

Although a handler is always invoked without delay, the semantics of asynchronous and synchronous communication is retained. A request to be notified at some instant in the future, for instance, which is sent asynchronously to the core thread responsible for timer signals, invokes the timer handler immediately and the message is saved within its save-queue. However, the sending thread is not blocked thereafter until the notification is due, as it would be the case if the request had been sent synchronously via a call.

3.2 Synchronisation

While the event-based processing model eases inter-thread communication, threads are still assumed to access shared data. As a simple, yet sometimes perfectly reasonable solution to synchronise concurrent accesses from threads on the same processor, pre-emption can be turned off on a per-thread basis. This way, a thread's function becomes the unit of mutual exclusion.

Preemption, however, relieves the programmer from manually taking care of more “urgent” threads during long-running calculations. Real-time system designers, in particular, try hard to keep periods of non-preemption at a minimum to tame priority inversion. Consequently, COOL JAZZ also provides fine-grained synchronisation via semaphores. These semaphores keep track of all threads within a critical section and optionally report blocked threads to the user-defined scheduler. The scheduler, in response, might perform some kind of priority inheritance, effectively bounding priority inversion [16].

The developer of a specific middleware platform, of course, is free to add more elaborated synchronisation primitives on his own, such as monitors with condition variables.

3.3 Memory Management

Since COOL JAZZ is targeting QoS-sensitive environments in general and (soft) real-time environments in particular, its memory management has been designed to be predictable with respect to utilisation as well as worst-case execution times.

Utilisation predictability means that there are no hidden memory allocations, which cause an application to become surprised by an out-of-memory error. Each memory allocation is only done in response to a user request. As already mentioned in Section 2, even signals sent by the operating system are reported by merely returning the message with which a thread had requested the notification and, thus, no memory allocation is caused at all. Memory utilisation is therefore highly predictable and allows the programmer to easily keep track of the available memory.

Predictability with respect to worst-case execution times, in contrast, is concerned with allocation and deallocation operations. While deallocation including any necessary coalescences of adjacent free blocks always can be done in constant time (i. e., $O(1)$ steps), allocation can be worst-case bounded to $O(\log n)$ steps at best. COOL JAZZ therefore obeys a user-defined bound k on the number of elements in the free list that are inspected as part of each allocation operation. If no sufficiently large memory block can be found in the first k elements of the free list, memory allocation fails. This way, a worst-case bound on memory allocation and deallocation can be derived, particularly since COOL JAZZ optionally disables paging for all its memory.

4 USER-DEFINED SCHEDULERS

As already stated in Section 2, lack of control over scheduling is commonly at the root of the cause if an implementation of user-level threads is not useful to system-level programmers. Consequently, we refrained from adding a covert scheduler to

Figure 2. Thread state transitions.

COOL JAZZ, yet systematically transformed all incidents that might induce a scheduling decision into calls to an user-defined scheduler. Foremost this includes (but is not limited to) thread state transitions such as when a thread becomes ready (marked as 1 in Fig. 2), when a reassignment of the CPU occurs due to preemption or at the will of the active thread (2), or when the active thread runs out of messages to process and becomes idle (3).² In QoS-sensitive environments, however, the choice of a scheduling policy also affects the message queues and synchronisation primitives, likewise.

4.1 Queues and Parameters

With each thread a *priority* template-parameter is associated, which allows the user to configure a thread with any kind of priority and priority-related information at compile time. The structure and semantics of these fields are completely user-defined and may represent a deadline for use with earliest-deadline-first (EDF) scheduling or a fixed priority for rate-monotonic-like policies, for example.

The user also has complete control over the internal organisation of the ready queue, using the term ‘queue’ in its broadest sense. A ready queue only must adhere to a minimal common interface such as the insertion of new threads and the removal of the first one. Specifically, a scheduler may employ any number of user-implemented ready queues in parallel, activating the one for COOL JAZZ to use on demand.

In addition, COOL JAZZ associates with each thread a *profile* template-parameter to describe general thread characteristics, which are used for admission tests prior to thread creation. Such a profile might include information on the thread’s worst-case execution time, or whether a thread is periodic or sporadic and its period or minimum inter-arrival time, for instance. Again, structure and semantics of a profile are completely user-defined and depend solely on the scheduling policy actually employed.

With respect to messages, COOL JAZZ further allows the assignment of user-defined constraints to each message (e. g., a deadline), as already mentioned in Section 3. These constraints may be evaluated by user-defined new-queues that replace the default implementation (e. g., performing an admission test prior to accepting a message) and to re-evaluate a thread’s priority on a per-message basis. Similarly, the implementation of a thread’s default save-queue may be substituted whenever appropriate.

²State transitions due to asynchronous termination have been omitted from Fig. 2 for readability.

4.2 Schedulers and Scheduler Invocations

In COOL JAZZ all schedulers are derived from a basic scheduler class, which provides some basic mechanisms useful to all schedulers (e. g., to dispatch the first thread in the ready queue). Furthermore, it prescribes the interface for scheduler invocations by the COOL JAZZ core when a thread becomes ready and, thus, should be inserted into the ready queue or when a time slice has run out, for instance. At each instant exactly one active scheduler is known to the COOL JAZZ core, yet the user may dynamically replace the scheduler if a mode change is needed, for example.

To start with, the core scheduler is invoked for the thread state transitions marked in Fig. 2. In addition, the priority of a thread may depend on the constraint or type of the active message. These are termed *dynamic* threads opposed to *static* ones, whose priority is not affected of the active message. For dynamic threads the scheduler is also invoked whenever such a thread reads the next message from its new-queue, while a thread's new-queue additionally even might call the scheduler for every message just inserted.

This approach is suitable for fixed-priority scheduling as well as dynamic scheduling, likewise. With a fixed-priority scheme such as rate-monotonic scheduling, message constraints are primarily used to order the processing of messages within a thread (although the implementation of the message might still call the scheduler). With a dynamic scheme such as EDF, in contrast, the constraint put on the thread's current message is also reflected in the priority of the thread. Both schemes can also be used in parallel, each one for disjoint subsets of all threads, while a thread may switch from being classified as dynamic to static and vice versa at run time.

Furthermore, the scheduler may also be called whenever a thread is blocked at a semaphore or while calling another process (marked as 4 in Fig. 2)—this behaviour can be toggled per P operation and per send operation, respectively. The scheduler might perform some priority-inheritance protocol in response, effectively bounding priority inversion. To request being called again as soon as the blockade is raised, the scheduler must set a flag within the thread's data structure. Nested or multiple priority inheritance is also possible based on internally assigned unique identifiers reported to the scheduler, which link corresponding inherit/reset requests.

Finally, the scheduler is consulted when a new thread is spawned to perform an admission test based on the thread's profile. If the admission test fails, the scheduler may weaken the profile in accordance with the currently available resources. This modified profile is returned to the user as a hint for further thread allocations.

Aside from these pre-defined scheduler invocations, the developer may add functionality in concert with further synchronisation primitives, for instance.

4.3 Default Policies

By default, thread priorities are merely integers, while lower numbers denote higher priorities to allow an easy mapping of deadlines. Similarly, message constraints are plain integers intended as deadlines. Message queues are ordered by constraint.

<i>schedule</i>	1.2 (+ 9 %)	1.1			
<i>semaphore</i>	1.3 (+ 8 %)	1.2			
<i>send</i> → <i>thread</i>	0.9 (± 0 %)	0.9	<i>send</i> → <i>handler</i>	1.1 (± 0 %)	1.1
<i>send</i> ↔ <i>thread</i>	3.2 (+ 3 %)	3.1	<i>send</i> ↔ <i>handler</i>	1.4 (± 0 %)	1.4
<i>call</i> ↔ <i>thread</i>	3.0 (+ 3 %)	2.9	<i>call</i> ↔ <i>handler</i>	1.2 (± 0 %)	1.2

Table 1. COOL JAZZ performance figures in μs (P200MMX, Linux 2.0.35).

The default scheduler implements a multi-level policy featuring preemptive round-robin scheduling within each priority level. In addition, a simple priority inheritance protocol is used to bound priority inversion.

5 PERFORMANCE MEASUREMENTS

To demonstrate the viability of such an open event-based processing model, a number of benchmarks have been performed, the results of eight of which are given in Table 1. The values listed have been averaged over 100 000 iterations each.

Of course, the added flexibility cannot be expected to come for free. Yet to exactly quantify the overhead caused by opening the threads package a black-box variant of COOL JAZZ has been implemented. Numbers in bold face denote the measured results for the open implementation, giving an impression of the absolute performance of COOL JAZZ. The relative performance in percent compared to those measured for the black-box implementation and the absolute results of the black-box implementation are shown in plain text. For all experiments the default scheduler has been used.

It should be noted that all overhead denotes a *worst-case loss* in the sense that an application's overall performance will suffer from such a slow down only if it does virtually no processing besides sending messages and causing context switches. Consequently, the real-world overhead imposed by factoring out the scheduler will be considerably less than the percentages denoted in Table 1, while exact figures are inevitably application-dependent.

The *schedule* benchmark denotes the performance of a single context switch as caused by yielding the processor, while the *semaphore* experiment couples two threads in a producer/consumer relationship, which is synchronised via a semaphore. The round-trip time therefore includes a P and V operation as well as two context switches. The '*send* →' benchmarks quantify the time needed for sending a message to another user thread or handler, respectively. This operation merely includes allocating a message and inserting this message into the recipients queue of new messages in case of user threads. For handlers, the handler invocation and the message deallocation is also included. The remaining four experiments, in contrast, use a single message per round trip, that is, no allocation and deallocation is needed. Within '*send* ↔' a thread asynchronously sends an initial message and afterwards await the reply via a blocking read, while within '*call* ↔' an initial call is performed instead, in which the sending thread is blocked until the reply is received. The performance figures given always denote the round-trip time, including two context switches.

6 RELATED WORK

COOL JAZZ represents the third generation of threads packages developed along the lines of the event-based processing model here at the Distributed Systems Group. The first implementation called SDE was an undocumented black-box threads package written in C by Peter Buhler. In SDE all the basic abstractions were already present, yet, neither inter-thread communication across address-spaces boundaries nor support for real time were provided. The viability of the event-based processing model was demonstrated by using SDE in a number of other projects such as a fault-tolerant distributed shared memory and the core of a platform for mobile agents. Next, FREE JAZZ built on SDE by adding support for real-time threading and user-defined schedulers to a proof-of-concept implementation [12]. COOL JAZZ, finally, is a complete rewrite of FREE JAZZ in C++, utilising the experience made and adding support for remote inter-thread communication as well as a flexible framework for user-defined interaction protocols.

The work most similar to our approach is GOPI, the General Object Platform Infrastructure [4]. GOPI also aims at providing “a low-level platform, which offers generic middleware services useful for the implementation of a range of multimedia capable distributed object systems.” Yet, no unified processing model is proposed. Similar to COOL JAZZ, GOPI allows for user-defined schedulers and application-specific interaction protocols. In contrast to COOL JAZZ, however, the principles of soft real-time are not directly supported by GOPI (e. g., the problem of priority inversion is not explicitly addressed) and its user-defined schedulers may not be called from user code.

The notion of an open implementation is also exploited in OPEN THREADS [7]. Again, the scheduler is only called whenever a thread state transition occurs. But neither OPEN THREADS nor any other flexible threads packages known to us has been developed specifically with real-time environments in mind, while real-time user-level threads packages, in contrast, were black-box solutions up to now. RT THREADS [5], for instance, is such a real-time user-level threads package. Its synchronisation primitives, however, do not care about priority inversion and the scheduling policy — a multi-level approach featuring EDF within each priority — is buried.

Finally, our work is related to the real-time CORBA effort [1]. Since standards concentrate on and try to establish one-size-fits-many solutions, a significant number of potentially critical design decisions are necessarily resolved once and for all instead of being left open. A more flexible platform such as COOL JAZZ, in contrast, is not designed along the lines of a single standard, yet facilitates the implementation of different middleware personalities likewise, which on their part may adhere to a standard such as RT-CORBA or else may be highly application-specific.

7 CONCLUSIONS AND FUTURE WORK

In this paper we have argued for a low-level foundation, on top of which more specialised QoS-supporting middleware for heterogeneous environments can be built. Specifically, we have proposed an event-based processing model to integrate support for concurrency, communication, and signal handling — arguably the most important system services from a platform developer’s perspective.

Since it is hardly feasible to define a generic set of QoS parameters and resource management functions, which perform adequately for arbitrary QoS requirements, COOL JAZZ delegates QoS management to user-defined schedulers and communication protocols. This way, we sidestep the problem of finding a single QoS mapping policy, while trying to provide as much generic support as possible. The performance of our prototype has shown to be promising and in our opinion the flexibility gained is well worth the worst-case overhead reported in Section 5, although, admittedly, it needs developers skilled in writing efficient system-level components to exhaust the facilities.

As part of our future work, we will add thread groups and thread pools to the event-based processing model. Thread groups and multicast communication are particularly important for safety-critical systems, while the notion of thread pools is a mechanism especially for multi-threaded servers [15]. Afterwards, we will consider flexible support for QoS-supporting distributed objects, which again will be primarily designed to ease the development of more powerful object models.

Finally, COOL JAZZ will be used within other research projects by our group, primarily providing the foundation for our own multimedia middleware platform [13]. One sample application in this context will be interactive multi-party document editing with integrated video conferencing.

Acknowledgements

The authors are indebted to the anonymous reviewers for their constructive comments, to Lothar Baum and Holger Peine for their valuable feedback on earlier versions of this paper, and to Georg Molter for some insightful discussions.

References

- [1] ALCATEL, HP, HIGHLANDER, IONA, INPRISE, LOCKHEED MARTIN, LUCENT, NORTEL, OBJECTIVE INTERFACE SYSTEMS, OBJECT-ORIENTED CONCEPTS, SUN, AND TRI-PACIFIC. Real-time CORBA: Joint Revised Submission, October 1998.
- [2] BLAIR G. AND STEFANI J.-B. *Open Distributed Processing and Multimedia*. Addison Wesley, 1997.
- [3] CAMPBELL A., COULSON G., GARCÍA F., HUTCHINSON D., AND LEOPOLD H. Integrated quality of service for multimedia communications. *Proceedings of the IEEE Infocom*, pages 732–739, March 1993.
- [4] COULSON G. *A configurable multimedia middleware platform*. Technical report, Distributed Multimedia Research Group, Lancaster University, 1998.
- [5] FINKELSTEIN D., HUTCHINSON N. C., MAKAROFF D. J., MECHLER R., AND NEUFELD G. W. *Real-time threads interface*. Technical Report TR-95-07, Department of Computer Science, University of British Columbia, Canada, 1995.
- [6] THE OPEN GROUP. *Introduction to OSF DCE 1.2.2*, November 1997.
- [7] HAINES M. On designing lightweight threads for substrate software. *Proceedings of the 1997 Annual Technical Conference*, pages 243–255. USENIX, 1997.
- [8] ITU. *Recommendation X.901–X.904, Open Distributed Processing – Reference Model*, 1997.

- [9] ITU-T. *Recommendation Z.100: CCITT Specification and Description Language SDL*, 1994.
- [10] KICZALES G., DES RIVIERES J., AND BOBROW D. G. *The Art of the Metaobject Protocol*. MIT Press, 1991.
- [11] KOSTER R. *Design of a real-time communication service for local-area networks*. Master's thesis, Department of Computer Science, University of Kaiserslautern, May 1998.
- [12] KRAMP T. *FREE JAZZ: A real-time user-level threads package designed for flexibility*. SFB 501 Report 9/98, Department of Computer Science, University of Kaiserslautern, October 1998.
- [13] KRAMP T. AND KOSTER R. A service-centred approach to QoS-supporting middleware. Work-in-Progress Paper presented at *Middleware '98 (IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing)*, September 1998.
- [14] OMG. *The Common Object Request Broker: Architecture and Specification (Release 2.2)*, February 1998.
- [15] SCHMIDT D. C. Evaluating architectures for multi-threaded CORBA object request brokers. To appear in a *CACM Special Issue on CORBA*, edited by K. Seetharaman, 1998.
- [16] SHA L., RAJKUMAR R., AND LEHOCZKY J. P. Priority inheritance protocols: An approach to real-time synchronisation. *IEEE Transactions on Computers*, pages 1175–1185, September 1990.
- [17] THEKKATH C. A., NEUGEN T. D., MOY E., AND LAZOWSKA E. Implementing network protocols at user level. *IEEE Transactions on Networking*, 1(5):554–565, October 1993.
- [18] WALDSPURGER C. A. AND WEIHL W. E. Lottery scheduling: Flexible proportional-share resource management. *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, November 1994.

Biographies

Thorsten Kramp received his M.Sc. in computer science in 1996 from the University of Kaiserslautern. Since then he is working as a member of the Distributed Systems Group at the University of Kaiserslautern. His interests and research focus on quality of service in distributed object computing.

Rainer Koster received his M.Sc. in computer science in 1997 from the Oregon Graduate Institute of Science and Technology. He is currently working as a member of the Distributed Systems Group at the University of Kaiserslautern. His interests and research focus on quality-of-service support and distributed multimedia systems.