TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik

Technische Berichte
Technical Reports

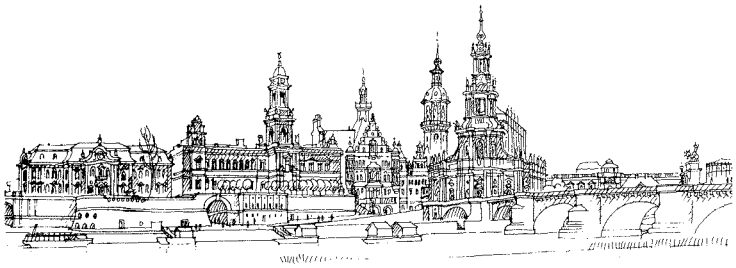
ISSN 1430-211X

TUD-FI04-01 - Februar 2004

Martin Pohlack, Ronald Aigner and Hermann Härtig

Institut für Systemarchitektur

Connecting Real-Time and Non-Real-Time Components



Technische Universität Dresden
Fakultät Informatik
D-01062 Dresden
Germany

Connecting Real-Time and Non-Real-Time Components

Martin Pohlack, Ronald Aigner, Hermann Härtig
Dresden University of Technology
Department of Computer Science
01062 Dresden, Germany
{pohlack,aigner,haertig}@os.inf.tu-dresden.de

Abstract

In this paper we describe a solution to the problem of communication between real-time and non-real-time components in a split container architecture. The split architecture carries forward an experience we gained in the *The Dresden Real-Time Operating System Project* (DROPS) [8]: Often, only small parts of applications need to be real-time capable. Furthermore, often these parts require only a small fraction of the complex services, which the remainder of the application needs.

Therefore, we proposed the splitting of applications in these two parts, whereas the small real-time part runs directly on our fast real-time capable L4 microkernel and the remainder runs on a off-the-shelf operating system [14]. Consequently, using this approach for a component-container architecture, we can support real-time components and standard components in one system.

In this paper we focus on the connections between both component types. We draft a buffer component which represents a non-real-time component in the real-time container. This component allows using non-real-time component's complex services from real-time components, without giving up the real-time.

1 Introduction

In the COMQUAD Project (COMponents with QUantitative properties and ADaptivity) we want to investigate technologies for components with non-functional properties [2, 5]. Examples for these properties are latency bounds for services, memory usage, or disk bandwidth.

Instead of developing 'yet another'™ component technology, we decided to adapt existing technology, in our case a subset of EJB [6], based on the JBoss implementation [10].

While we are able to reuse large parts of the code in JBoss, it is hard to impossible to add certain properties, such as real-time communication and real-time execution of components to the infrastructure layer — the container. On the other hand it is very unrealistic to re-implement the large component base existing for EJB to a real-time environment.

On the strength of past experiences from the development of DROPS we concluded that real-time capabilities are often not necessary for large applications, but just for small parts of them. So, we took the approach of implementing a small real-time capable microkernel (Fiasco, [17]), intended to run real-time servers, and a large off-the-shelf Linux server (L^4Linux, [15]) for all the other applications, which gives us a fairly large code base.

We transfered this approach to our container architecture which is now split in two parts. The non-real-time container is based on JBoss, running in a standard JVM on L^4Linux and providing complex services, such as a component repository. The real-time container is running directly on the L4 microkernel. The latter one will contain all the real-time–capable components and the necessary infrastructure.

By splitting our container in two parts the problem of connecting them arises, which shall be thoroughly discussed in this paper.

We propose a solution to the problem of communication from real-time to non-real-time components. We exploit know solutions from the literature and draft a buffer component, which encapsulates and hides the communication problem and which can be parameterized, to implement arbitrary request scheduling policies.

The remainder of this paper is organized as follows. In the next section we discuss related work. In Section 3 we introduce concepts, we based our work on. Section 4 describes our architecture and our solution proposals, as well as a complexity discussion. We conclude this paper with Section 5.

2 Related Work

During our work we looked at various similar real-time projects to investigate their approach to communication between real-time and non-real-time tasks. Most of these projects rarely mention how messages are transferred between real-time and non-real-time partners. Nonetheless all provide some sort of communication framework.

2.1 RT-Posix

Liu describes Real-Time POSIX communication in [18] (pg. 521ff., pg. 574) as priority-based and nonblocking using message queues. Due to the association with priorities, the use of a message queue is determined by the priorities of the message and the receiver. Therefore message-based priority inversion, where the receiving thread has a lower priority than the message, is a major problem. Example operating systems using message queues to communicate between real-time and non-real-time tasks are QNX and VxWorks.

Message delivery is bound to the priorities of the participating receiver. This does not imply that the receiver will get necessary processing time within a specified time period. Our approach uses priorities to implement resource reservation of CPU time, which can guarantee a minimal execution time for non-real-time tasks.

2.2 Real-Time Specification for Java

The Real-Time Specification for Java [21] describes the principal of "Asynchronous Transfer of Control". Imagine a complex algorithm, whose execution time is highly variable. If this algorithm has the additional property to refine its result with each iteration, it may be useful to interrupt the algorithm asynchronously and use its current result. This method has the advantage, that the algorithm can be granted a specific amount of time that is available.

However, this approach has also several drawbacks. The first is, that it is only applicable to iterative algorithms, which refine their result over time, that is, it can only be used for a limited number of algorithms. The second drawback is, that the algorithm must be interruptible asynchronously, that is, its internal state must not get corrupted by this and it must always have a valid (although not necessarily up-to-date) result available.

As a consequence, using the principal of "Asynchronous Transfer of Control" it is *not* possible to reuse existing components unmodified. Reusing arbitrary components does not make sense as well, as the algorithms must be able to refine results. We aim at a more universal solution.

2.3 Realtime Application Interface

The Realtime Application Interface (RTAI) [4] projects aims at providing a programming interface to Linux developers. It includes kernel modules, which provide real-time services. To allow the communication between real-time applications in the kernel and non-real-time applications in user land, FIFOs are available, which are similar to the FIFOs of RT-Linux. Another way to communicate between real-time and non-real-time tasks is the Linux Realtime Module (LXRT) which makes RTAI scheduler functions available to Linux processes. This allows a fully symmetric implementation of real-time services. It is possible to share memory, send messages, use semaphores and timings. And this can be done between two Linux processes, Linux and RTAI processes and, naturally between RTAI and RTAI processes.

RTAI allows to communicate between real-time and non-real-time tasks, but with the restriction of using RTAI constructs.This implies that non-real-time applications have to be modified so real-time application can use them.

2.4 RT-Linux

As mentioned in [25] and [24] real-time tasks communicate with non-real-time tasks using FIFO buffers called RT-FIFOs. These buffers are pinned into kernel memory. Reads and writes by real-time tasks are nonblocking and atomic. A RT-FIFO buffer has to be big enough to hold the data. If the buffer is full, no more data can be placed into it. And it is not possible to reorganize the content of the buffer. Our approach allows to implement flexible algorithms to reorganize the content of the FIFO buffer.

If a real-time tasks intends to use the service of a non-real-time tasks it cannot communicate with this task directly (for instance, by using RPC or sockets), but has to pipe its request to this task using a FIFO buffer. This implies that the used service has to be aware of this input variant and appropriately modified.

2.5 KURT

In the course of implementing a real-time Linux, the Kansas University Real-Time Linux (KURT) [7] group had to find ways to publish events from the real-time kernel to non-real-time applications. Therefor, KURT relies on the Data Stream Kernel Interface (DSKI), which uses data streams to publish kernel events to the user. To allow events to be buffered, queues are used. [3] makes no explicit statement about placing a message into a data stream, except that it has to be fast. Also do DSKI buffers have to be large enough to hold all messages. Otherwise new messages will be dropped.

The DSKI basically uses FIFO buffers to transfer data from the real-time kernel modules to non-real-time user applications. This implies the mentioned drawbacks of FIFO buffers. User applications relying on data from the kernel have to use the DSKI to obtain the data. Thus, modifications of application are necessary to use the real-time kernel modules.

2.6 Real-Time Event Service

In the course of the TAO (The ACE ORB) project [22], a Real-Time Event Service has been developed as an extension to the CORBA Event Service. The Real-Time Event Service [13] provides features required by real-time applications such as real-time event dispatching and scheduling,

periodic event processing, and efficient event filtering and correlation mechanisms. In contrast to the CORBA Event Service, which allows push and pull relationships between producers and consumers of events, the Real-Time Event Service only allows Push relationships. This allows suppliers of events to initiate the transfer of events to the consumer. Whereas a pull supplier would wait until an consumer pulls the event from it.

With respect to the mechanism of real-time to non-real-time event delivery we looked at the characteristics of the TAO Real-Time Event Service. Since the RT Event Service does not explicitly handle the real-time to non-real-time communication, we can regard a non-real-time consumer as a consumer with a lower priority than the real-time consumers. An incoming event is filtered and queued into a priority queue. Dispatching threads dequeue the events and deliver them to the consumers. A Run-Time Scheduler determines the appropriate priority queue for an event based on the event–consumer tuple. The implementation of the Dispatcher determines the use of dispatching threads.

The RT Event Service allows the decoupling of supplier–consumer relationships. Using the event service, suppliers or consumers can subscribe to events transparently, meaning that a supplier is not notified when there is a new consumer for its events and vice versa.

Since the dispatching of events depends on the priority of the dispatching threads, that is, the priority of the consumers, the event buffers can overflow. The Real-Time Event Service does not specify how the sizes of the buffers are determined. Since TAO is based on a off-line scheduling, we assume that the event buffer sizes can be determined off line as well. This fact also allows the implementation of *O(1)* (worst case: Timers *O(log N)* — N is the number of timers) event filtering by using lookup tables at run-time.

The TAO Real-Time CORBA implementation uses the same mechanisms as the TAO Real-Time Event Service to dispatch method invocations. Incoming requests are enqueued based on the priority of the invoked service (the consumer) and dispatched by dispatching threads.

One drawback of the RT Event Service is its complexity and required infrastructure, which makes it hard to port to DROPS quickly. The currently available implementation of the RT Event Service in TAO is fast for a fixed task set which can be scheduled off line. This includes feasibility analysis of message dispatching and it allows a off-line calculation of message priorities. Our approach allows to deliver messages on a dynamic real-time system.

2.7 DSI

The DROPS Streaming Interface (DSI) [19] is a mechanism to transfer large amounts of data unidirectionally using shared memory. An interesting aspect of DSI is the timeliness of packets in this stream. Packets are separated into packet headers, which contain meta-data, such as a logical timestamp, and packet data. This allows the packets to be delivered in a strict FIFO order, but the packet's content to be distributed arbitrarily in the data buffer. Based on the logical timestamp of a packet a consumer can decide to drop the packet prior to its processing. DSI is a custom solution for multi-media data stream transfer. A drawback is that the communication partners have to be DSI aware — communication is not simply exchanging data. Also the strict FIFO order of packets does not allow the usage of other replacement strategies then dropping packets.

3 Background

3.1 DROPS

DROPS, as depicted in Figure 1, is an real-time capable operating systems based on the L4 microkernel, which provides fast inter-process communication (IPC). I/O device drivers are implemented as user-level servers to provide isolation and fault-tolerance for the OS. To demonstrate the flexibility of L4, Linux has been modified to run as a user mode server on top of L4 — L^{4}Linux. This approach also demonstrates that in order to provide features, such as real-time or security, only few parts — device drivers — have to be ported, while the majority of the former application — Linux — runs unmodified.

DROPS includes a real-time window manager, DOpE [9], which can guarantee refresh rates for its real-time clients. Non-real-time clients of DOpE are drawn whenever there is time left. DOpE also contains features to hide the content of sensitive windows from untrusted windows.

Resources in DROPS are managed by resource managers [16]. Resource managers can form hierarchies to either combine different resources to form a higher level resource, or, if managing the same resource, to form resource domains. Resources can be reserved by an application in advance to ensure their availability when they are used. This includes the reservation of computation time, memory, etc.

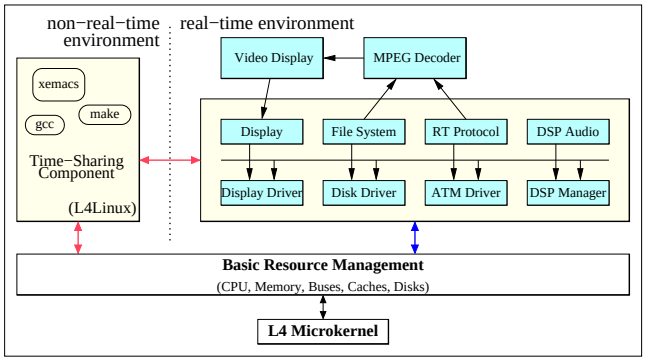


Figure 1: DROPS architecture

3.2 COMQUAD

The COMQUAD Project is implemented on top of DROPS. Goals of the COMQUAD Project are the specification of non-functional properties, such as resource demand, for components. We provide tools to transform such a specification into a run-time format, which is used to make resource reservations at run-time.

Another goal of the project is reuse of components and resource classifications. This implies that existing components, which are not real-time capable can be reused. Without a specification describing its run-time requirements, a component is regarded a non-real-time component. Such an specification includes the required services, which have to be provided by the run-time environment — the container.

3.3 Motivation

When a real-time task communicates synchronously with a non-real-time task it may loose its timeliness. This is due to the fact that the non-real-time task can be delayed arbitrarily when processing the request of the real-time task. And the real-time task can be delayed for a potentially unknown amount of time when placing its request at the non-real-time task.

Some services are complex and to make them real-time capable is practically impossible. Still, real-time applications may want to use those services, even though this implies disadvantages, such as data loss. Some approaches, for instance, the different real-time Linux flavors, use some sort of buffer to propagate information from real-time applications to non-real-time applications. Other approaches, such as the Real-Time CORBA, use event services to connect real-time applications with other real-time applications.

Our goal is to make non-real-time applications and especially components available for real-time components.

Some mechanisms mentioned in Section 2 could be used to implement our ideas. We decided to provide our own implementation for several reasons. Firstly, we did not want to port complex mechanisms and infrastructure, such as POSIX message queues, to DROPS. Secondly, other mechanisms are too simple to allow generic use cases. One such use case is the collection of sampling data from sensors. Using our approach, we can implement a policy which allows the reorganization of our message queue, for instance, by re-sampling the data (average values of samples next to each other). In contrast RT-Linux FIFOs cannot be used to achieve this, since no random access to elements is possible.

4 Connecting both worlds

4.1 Architecture

Applying our previous experiences with a small and simple real-time core and the big and feature-rich single-unix server L^{4}Linux in DROPS to the COMQUAD Project, we came up with a two-container approach, which is depicted in Figure 2. In practice real-time features mostly are not needed for a whole large application but only for a small and simple fraction of it. The splitting of the container in two parts gives us several advantages over comparable projects:

- It simplifies the implementation of the container itself as big parts can be reused from other projects.
- Not only is it harder to develop a complete real-time capable container, but also the development of software for such an environment is more complicated as the timing aspect has to be taken care of by the programmer. With our approach, this is only necessary for the parts that really need it.

The biggest part of the container resides on the L^{4}Linux side and consists of a stripped-down EJB Container, using JBoss on a Sun JVM, mostly off-the-shelf components. Of course this part of the container is not real-time capable and timing-critical jobs have to be relocated to the real-time container.

Components in the real-time container are L4 programs implementing special interfaces as to be managed by the real-time container. The functions comprising these interfaces correspond to the minimal set of functionality the real-time container has to fullfill, which are detailed below.

A basic means of information exchange is Inter-Process Communication (IPC), directly supported by the L4 microkernel. For every IPC a timeout can be specified. To ease and automate the very error-prone work of setting up the communication via IPC, we use our IDL compiler DICE [1], which generates the Server and Client stubs, and is also capable of

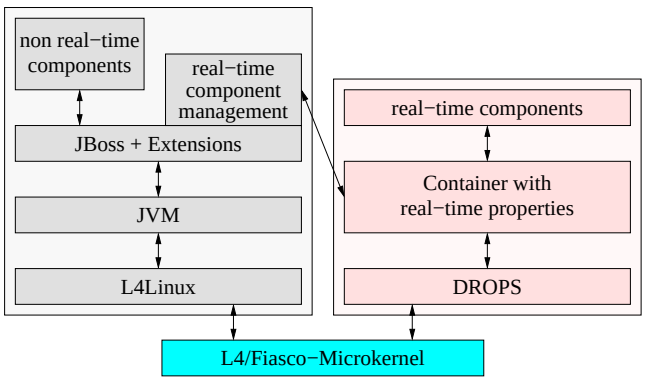


Figure 2: Container Architecture

marshalling information such that it becomes interchangeable between the two containers. On the real-time side we do not use a Dynamic Invocation Interface but use generated code, which is bigger but faster [12, 11, 13].

To maximize software reuse and to create a really small and fast real-time container, we tried to find a minimal set of functionality also necessary on the real-time side:

create The real-time container must be able to create new component instances.

destroy Also removing of created instances should be supported.

init Initialisation of components, so that they can be started, is necessary.

stop This functions stops the request delivery to a component. Buffered requests will still be processed.

connect The container must be capable of connecting component interfaces with other instances.

setParameter The container must be able to set components' properties. This method of modifying components' states is meant to be used for initialisation.

reserveResources The container must be able to make resource reservations on behalf of components.

callMethod The container must be able to forward calls to arbitrary component methods.

install/uninstall Specification/Implemenation The container must be able to accept and remove component specifications and implementations.

Fortunately, it is not necessary to implement all of these functions with real-time guarantees, such as the administration of components. In our development we focus on real-time communication between connected components, not the establishing of communication structures and setup in real-time, that means that only the callMethod operation must be carried out in real-time.

The proposed design principle could also be an interesting alternative to the "Real-Time Specification for Java" (RTSJ) [21] and its implementations, as not only large parts of today's software could be reused but also the real-time components run directly on the fast microkernel, can be written in pure C or C++, and can therefore be much faster (see [23]). We will describe our container architecture in detail in another paper.

Following from the two-container design methodology the problem of the coupling of both parts and the general communication between arbitrary components arises and shall be discussed in the following.

4.2 Communication

In this section we focus on the communication between components residing on different sides of the real-time/non-real-time border.

There are two kinds of communication partners, real-time components and non-real-time components. Using services from other components should not disturb the real-time capabilities. Synchronous communications is therefore feasible only in the following cases: real-time components using services from other real-time components and non-real-time components using services from both types of components, real-time and non-real-time.

Asynchronous communication is usable in all cases, but is necessary for real-time components using services from non-real-time components. Imagine a real-time component doing a synchronous call to a non-real-time logging service, which is arbitrarily delayed on the non-real-time side. RT-Linux has the concept of RT-FIFOs which can be accessed in a nonblocking and atomic way from the real-time side. However, if the FIFO's memory is full, the real-time part would have to wait, thereby possibly violating its timing constraints, or to drop data. We want to generalize the solutions for this problem using a buffer component, comprising the FIFO's storage function and extracting the complexity of the replacement policy in case of memory shortage.

Let us have an in-depth look at some solution attempts for such a buffer component:

JNI A non-real-time component in the Java container is willing to receive requests. It calls a special method implemented using the Java Native Interface (JNI), which can receive native L4 IPC. A real-time component sends its request to the non-real-time component, which is received and copied to a buffer in the JNI method. The next step would be to wake up threads waiting on the request queue. After this step the thread in the JNI method is able to receive the next call. During the time from the first message receipt until the buffer handling is completed no request could be received. As we are not willing to delay real-time components indefinitely, there must be an upper time bound after which the message sending must be completed, so that the real-time component can proceed with its job. Due to the time-imprecise nature of the non-real-time container, the upper bound, the non-real-time component is unable to receive messages, cannot be determined and lost requests are to be anticipated.

This solution attempt is depicted in Figure 3.

native buffer handling Moving much of the thread and buffer handling code, mentioned in the first solution above (JNI), from the JVM into JNI methods implemented in native C would reduce the amount of lost requests due to the elimination of the biggest source of

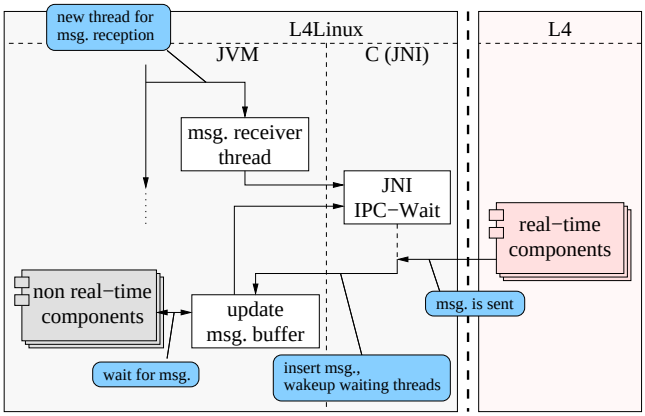


Figure 3: Communication from real-time components to non-real-time components

unpredictability: The Java garbage collector. Also, the code execution might be faster and therefore fewer request will be lost. On the other hand, code still runs on the non-real-time side (L^4Linux) and therefore is not perfectly predictable.

buffer component If one wants to eliminate the other sources of unpredictability (non-real-time operating system) in the solution attempt described above, request buffering code must be moved to the real-time container. So, the next logical step is a buffer component residing on the real-time side, comprising at least two threads. The receiving and storing of requests has to be implemented in a predictable way. A low-priority thread in the buffer component is responsible for delivering the request to non-real-time components. Both threads have to access shared data at one point, the stored requests. This operation — the removal of a request from the buffer — is the only part, where the delivering thread may interfere with the receiving thread's execution and is therefore the only part which must be implemented predictably in the sender thread. Unfortunately, also in this scenario data loss can occur as we only have limited memory to store requests. No assumptions can be made about the delivery rate of the requests as the receivers are non-real-time components.

This solution is depicted in Figure 4.

From the discussion of the three described solutions we conclude that lossless communication between real-time and non-real-time generally cannot be guaranteed. So, it is important to have at least some kind of control about the request losses. Using a buffer component with a replacement strategy eases the usability of our component architecture. Additionally, it is possible to use such a buffer component to connect real-time components with different throughput, using controlled request dropping in the buffer component.

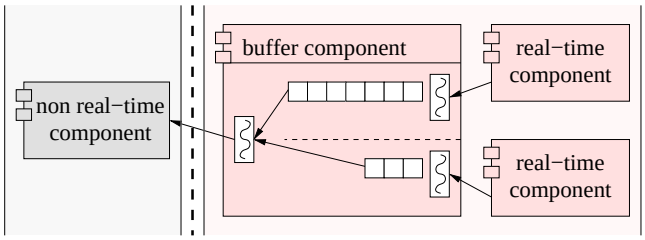


Figure 4: A closer view into a buffer component connecting two real-time components with one non-real-time component

Another example for a more complex version of the buffer component is the following example: A buffer component connects various real-time sensors with a costly evaluation component. It may not only drop single measuring points, but may re-sample data stored in the buffer (i.e. combine adjacent measurement points), thereby decreasing work but also accuracy.

4.2.1 Complexity Analysis

As of writing this paper a preliminary implementation of the buffer component exists. In this section we will discuss the principal complexity of basic operations of this component. The unit *buffer* is meant to hold one request and is organized in *pools*.

We designed the buffer component such that it contains at least two threads, whereas the first is responsible for receiving requests and the other for forwarding requests.

We use two memory pools, one for full buffers and one for empty ones. The sizes of the buffers and pools can be reserved at the time the communication is initialized. As with many other real-time problems, also the communication between the real-time component and the buffer component must be scheduled.

To receive a request, the receiver thread has to perform the following steps:

1. Receive IPC,
2. Decide whether to drop the request or not, based on reserved communication bandwidth,
3. Look for a buffer in the free-buffer pool,
   (a) if a buffer is found, remove it from the pool, copy the request into the buffer, and enqueue the buffer to the full-buffer memory pool,
   (b) if no buffer is found, either drop the request, or use a replacement policy to free a buffer and use it.

Depending on the data structures used to manage the memory pools, it is possible to easily select the first, the last,

or a random element from the pool to drop the request and reuse the buffer.

Both the receiving thread and the forwarding thread use shared data structures, which makes mutual exclusion necessary. The forwarding thread must be built such that blocking is bounded to one dequeue operation on the pool, so that message receipt can be predictable. Additionally, using a known solution from the literature to the priority inversion problem is advised here.

For example, by using simple lists the first element can be accessed with a time complexity of O(1) (see 2.6).

Another attempt to buffer management would be to use a real-time–capable memory-management library, such as TLSF [20].

4.2.2 Priorities

Priorities could have two different meanings in our communication schema, where the first is that high-priority requests overtake lower-priority requests, and the second is, that in case of buffer overrun no higher-priority request is dropped if lower-priority requests are still in the queue.

A buffer component has a one-to-one relationship with a non-real-time component, which should receive requests from a real-time component. In a situation where more than one real-time component sends requests to the same non-real-time component, it is possible to aggregate all senders on one buffer component. This is either possible using more receiver threads (which is depicted in Figure 4), or by scheduling one receiving thread accordingly. Data structures have to be instantiated for each sender to minimize interference. However, only one forwarding thread is necessary. The forwarding thread is also the location of choice to implement request-scheduling strategies. It could use a "fair" policy and check the queues in a round-robin manner, but it could also implement a hard priority policy, always checking high-priority queues first.

The buffer component is essentially a representative of the non-real-time component in the real-time container.

In our current implementation we do not consider request priorities explicitly, as we don't want to complicate buffer handling algorithms. Instead we want to keep them simple and real-time capable. However, priorities influence request handling indirectly as CPU time is necessary to send request and higher-prioritized components can therefore send request earlier and more often.

4.3 Results

We introduced a buffer component which:

- encapsulates all the low-level communication handling when communicating between real-time and non-real-time components;
- encapsulates the replacement policy in case of buffer overrun;
- may contain an easily parameterizable replacement policy;
- allows easy usage of complex non-real-time functionality from real-time components;
- allows the transfer of typed messages, with the help of tool-generation communication stubs.

We generalize the solutions proposed in the literature in Section 2 and provide an solution at component level.

5 Conclusions

In the COMQUAD Project our goal is to analyze non-functional properties in component architectures, where real-time capabilities of component's services are one important example. We wanted to focus on the research of non-functional properties, and therefore we tried to minimize implementation effort for infrastructure. We designed a system architecture which allows us to reuse large parts of existing non-real-time code. Our component container is based on JBoss running in a Sun JVM on Linux. Instead of porting the JVM to our real-time operating system DROPS, implementing all the RTSJ features, and finally adapting the JBoss container to use them, we took a different approach.

From the experience of our work in the DROPS project we carried forward the observation that often only small parts of large applications need to have real-time properties. We decided to adopt large parts of the existing solution (JBoss, JVM, and Linux) but use L^4Linux instead of a native Linux. Using L^4Linux we are able to have real-time programs running concurrently to the classic container on L^4Linux. One of these applications is a minimal real-time capable container, executing our real-time components.

Just having this split architecture does not yet allow reuse of large code parts or even whole components. It is necessary to be able to connect real-time and non-real-time components, so that both types of components can interact and use each other's functionality. Furthermore it is necessary to connect both containers, the small real-time–capable container and the huge standard container (JBoss).

This paper describes how these connections can be accomplished, what type of communication can be used in which situation, and describes the requirements, algorithms, and data structures for so called buffer components, which are actually used to connect components from both container types. This work reconsiders approaches know for a long time in the real-time community and tries to push this knowledge one step further towards component architecture, while still supporting common components. The described communication techniques open up an interesting alternative to

large and complex approaches like RTSJ and offer high reuse of existing code and components.

One interesting consequence — though not new — is that data flow from real-time to non-real-time components can not generally be realized without data loss, but only for special cases. Consequently protocols capable to cope with data loss must be used when sending data from real-time to non-real-time components.

5.1 Limitations and future work

Our current implementation is preliminary and does not yet offer all of in this paper drafted features. However, we are working on this and hope to be able to deliver measurement results soon.

We also want to extend the mentioned concepts towards streaming connections. Our attempt here is to base our work on DSI (see 2.7) and generalize its buffer handling. Also, a network-transparent implementation using the DSI interface looks interesting.

Finally, we want to realize more applications, using our component model, which will give us feedback on how to adapt the model to ease porting of existing and implementation of new applications. Additionally, we hope to practically verify the identified set of minimal functionality for the real-time container for a broader application scenario.

References

[1] R. Aigner. DICE Documentation, http://os.inf.tu-dresden.de/dice/.

[2] R. Aigner, H. Berthold, E. Franz, S. Göbel, H. Härtig, H. Hussmann, K. Meissner, K. Meyer-Wegener, M. Meyerhöfer, A. Pfitzmann, S. Röttger, A. Schill, T. Springer, and FrankWehner. COMQUAD: Komponentenbasierte Softwaresysteme mit zusagbaren quantitativen Eigenschaften und Adaptionsfähigkeit. *Informatik Forschung und Entwicklung*, 18:39–40, 2003.

[3] B. Buchanan, D. Niehaus, G. Dhandapani, R. Menon, S. Sheth, Y. Wijata, and S. House. The Data Stream Kernel Interface. Technical Report ITTC-FY98-TR11510-04, University of Kansas, Jun 1998.

[4] P. Cloutier, P. Mantegazza, S. Papacharalambous, I. Soanes, and S. Hughes. DIAPM-RTAI Position Paper. In *Workshop on Real Time Operating Systems and Applications and second Real Time Linux Workshop*, Lake Buena Vista, FL, Nov. 2000.

[5] Comquad Team. COMQUAD, http://www.comquad.org/.

[6] L. G. DeMichiel, L. Ü. Yalçinalp, and S. Krishnan. *Enterprise JavaBeans Specification, Version 2.0*. Sun Microsystems, final release edition, 14 Aug. 2001.

[7] W. Dinkel, D. Niehaus, M. Frisbie, and J. Woltersdorf. *KURT-Linux User Manual*. Univeristy of Kansas, Mar. 2002.

[8] DROPS Team. Drops - the dresden real-time operating system project. http://os.inf.tu-dresden.de/drops/.

[9] N. Feske and H. Härtig. Demonstration of DOpE — a Window Server for Real-Time and Embedded Systems. In *24th IEEE Real-Time Systems Symposium (RTSS)*, pages 74–77, Cancun, Mexico, Dec. 2003.

[10] M. Fleury and F. Reverbel. The JBoss extensible server. In M. Endler and D. Schmidt, editors, *International Middleware Conference*, volume 2672 of *Lecture Notes in Computer Science*, Rio de Janeiro, Brazil, 16–20 June 2003. ACM / IFIP / USENIX, Springer.

[11] A. Gokhale and D. Schmidt. The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks. In *Proceedings of GLOBECOM '96*, pages 50–56, London, England, IEEE, 1996.

[12] A. Haeberlen, J. Liedtke, Y. Park, L. Reuther, and V. Uhlig. Stub-code performance is becoming important. In *Proceedings of the First Workshop on Industrial Experiences with Systems Software (WIESS)*, pages 357–363, San Diego, USA, Oct. 2000.

[13] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of OOPSLA '97*, pages 184–200, Atlanta, GA, Oct. 1997.

[14] H. Härtig, R. Baumgartl, M. Borriss, C.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, Sept. 1998.

[15] H. Härtig, M. Hohmuth, and J. Wolter. Taming Linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART '98)*, Adelaide, Australia, Sept. 1998.

[16] H. Härtig, L. Reuther, J. Wolter, M. Borriss, and T. Paul. Cooperating resource managers. In *Fifth IEEE Real-Time Technology and Applications Symposium (RTAS)*, Vancouver, Canada, June 1999.

[17] M. Hohmuth. *Pragmatic nonblocking synchronization for real-time systems*. PhD thesis, TU Dresden, Fakultät Informatik, Sept. 2002.

[18] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[19] J. Löser, L. Reuther, and H. Härtig. A streaming interface for real-time interprocess communication. Technical Report TUD-FI01-09-August-2001, TU Dresden, Aug. 2001. Available from URL: `http://os.inf.tu-dresden.de/ jork/dsi_tech_200108.ps`.

[20] M. Masmano, I. Ripoll, and A. Crespo. Dynamic storage allocation for real-time embedded systems, 2003. Work in Progress, RTSS 2003, Cancun / Mexico.

[21] The Real-Time for Java Expert Group. *The Real-Time Specification for Java*, v1.0 edition.

[22] D. C. Schmidt, A. Gokhale, T. Harrison, and G. Parulkar. A High-Performance Endsystem Architecture for Real-Time CORBA. *IEEE Communications Magazine*, 14, Feb 1997.

[23] D. C. Sharp, E. Pla, and K. R. Luecke. Evaluating Mission Critical Large-Scale Embedded System Performance in Real-Time Java. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS)*, Cancun, Mexico, 2003.

[24] V. Yodaiken. The RTLinux manifesto. In *Proceedings of The 5th Linux Expo, Raleigh, NC*, Mar. 1999.

[25] V. Yodaiken and M. Barabanov. A Real-Time Linux. In *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*, Anaheim, CA, Jan. 1997. The USENIX Association.