A Peer-to-Peer Middleware Platform for Fault-Tolerant, QoS, Real-Time Computing

Rolando Martins
CRACS/EFACEC Sistemas de Electrónica, S.A.
rolando.martins@efacec.pt

Luís Lopes, Fernando Silva
CRACS/Faculdade de Ciências, Univ. do Porto
{lblopes,fds}@dcc.fc.up.pt

ABSTRACT
In this paper we present the architecture of RTPM, a middleware framework aimed at supporting the development and management of information systems for high-speed public transportation systems. The framework is based on a peer-to-peer overlay infrastructure with the main focus being on providing a scalable, resilient, reconfigurable, highly available platform for real-time and QoS computing.

Categories and Subject Descriptors
D.2.11 [Software Engineering]: Software Architectures—data abstraction, domain-specific architectures, languages, patterns

Keywords
Peer-to-Peer, Middleware, Fault-Tolerance, QoS, Real-Time

1. INTRODUCTION AND MOTIVATION
The development and management of information systems for application domains that require real-time and QoS computing is pushing the limits of the current state-of-the-art in middleware frameworks. At EFACEC\(^3\), our particular case-study is that of information systems used to manage public, high-speed, transportation networks. Such systems typically transfer large amounts of streaming data; have erratic periods of extreme network activity; are subject to relatively common hardware failures and for comparatively long periods, and; require low jitter and fast response time for safety reasons (e.g. vehicle coordination). These characteristics put stringent constraints on the software used for management and therefore on the underlying middleware framework. Real-time, QoS and fault-tolerance mechanisms are clearly essential features for middleware platforms that support such extreme information systems.

Despite the large body of research on middleware systems (e.g. [19, 22, 9, 15]), the integration of resilient services, soft real-time and QoS in these systems remains a hard problem. Several major steps have been taken in recent years to introduce fault-tolerance and real-time mechanisms in CORBA [13, 21, 16, 14], arguably the standard by which most middleware implementations are developed.

We feel, however, that most known strategies to introduce this support in CORBA may get in the way of supporting QoS and real-time features. For example, fault-tolerant CORBA resorts to services that implement mechanisms for object replication, fault detection and fault recovery. These services are implemented at the ORB level and follow moderately complex protocols that introduce a considerable amount of overhead, that conspires to make QoS and real-time more difficult to support. Moreover, the service infra-structure that supports fault-tolerant CORBA must, naturally, in itself be fault-tolerant, thus shifting part of the problem to the underlying networking layer. This poses a problem if we realize that most middleware platforms are based on classic client-server network layer architectures which have little or no builtin support for fault-tolerance mechanisms, besides other limitations relevant for QoS and real-time implementations such as: single point of failure associated with servers; limited load-balancing and reconfiguration capabilities, and; permeability to denial-of-service attacks. In addition, fault-tolerant CORBA does not handle byzantine and partial failures, and does not guarantee that two implementations that conform to the real-time and fault-tolerant CORBA specifications will result in the same QoS properties [7].

Real-time and QoS issues have been given attention in the context of the TAO project [22]. For example, TAO introduced different, configurable, execution models to minimize code paths both within and across layers, thus minimizing protocol overheads. The ICE [9] project trimmed the CORBA standard in order to produce a middleware framework optimized for scalability and efficiency. Lately, the introduction of computational mobility [25] and especially of virtualization techniques [2, 5, 10], has provided a new tool to solve problems like dynamic reconfiguration, load-balancing and availability, all fundamental issues in fault-tolerant, real-time, QoS middleware.

In this paper, we argue for a more integrated approach to the problem of introducing fault-tolerance, QoS and real-
time in middleware platforms. The key idea is to drop the centralised client-server architecture of the networking layer and replace it with a more flexible infra-structure with built-in support for the above mentioned middleware features. Peer-to-peer infra-structures stand out as good candidates since their decentralized nature promotes scalability and resilience, and their architectures may be optimized to address QoS and real-time constraints [1, 18].

There is a considerable amount of work on architecture, protocols and algorithms for peer-to-peer systems that addresses fault-tolerance, QoS and some aspects of real-time. A particularly good example of this work comes from distributed file-sharing systems [6, 4, 11]. Finally, there are several frameworks that provide system developers with components and patterns to implement custom peer-to-peer systems [8, 17, 27].

Despite these a priori advantages, mainstream peer-to-peer middleware systems soft real-time and QoS computing are, to our knowledge, unavailable. This is partly due to the proliferation of distinct peer-to-peer architectures. Motivated by this state of affairs, by the limitations of the current infra-structure for the information system we are managing (based on CORBA technology) and, last but not least, by the comparative advantages of flexible peer-to-peer network architectures, we have designed a service-oriented peer-to-peer middleware framework we call RTPM (Real-Time Peer-to-Peer Middleware). The framework aims to provide deterministic behavior, suitable for real-time systems, with an underlying extensible resilient network infra-structure capable of running in a multitude of operating systems [3, 17, 27].

The RTPM networking layer relies on a modular infra-structure with multiple peer-to-peer overlays. The support for fault-tolerance and, in part, for QoS and real-time features is provided at this level through the implementation of efficient and resilient services for, e.g., resource discovery, messaging and routing. The kernel of the middleware system (the ORB) is implemented on top of these overlays and uses the above mentioned peer-to-peer services to provide developers with APIs for fault-tolerance and customization of QoS and real-time policies. The kernel also provides support for service migration by integrating virtualization techniques with the peer-to-peer networking layer. However, QoS and real-time support is not just provided at the network level. For example, RTPM provides APIs to the operating system layer that allow the definition of dynamic scheduling policies for CPU resources, taking advantage of the current trend in multicore technology. This can be seen as a further extension to the approach taken in TAO [22] by defining distinct strategies for the execution of tasks by threads.

The main contribution of this work is on the use of peer-to-peer overlays to implement the networking layer of the middleware platform and on the implementation of the kernel (the ORB) based on the facilities provided by the above mentioned peer-to-peer infra-structure, allowing the developers to customize and optimize the fault-tolerance, QoS and real-time features of the framework to the needs of each specific application.

The remainder of this paper is structured as follows. The next section gives an overview of the RTPM software architecture. Section 3 describes the low level interface with the operating system. Section 4 describes the peer-to-peer infra-structure that supports the RTPM network model. Section 5 describes RTPM’s kernel and its core services. Sections 6 and 7 describe the service and application development APIs. Finally, the last section describes the current state of the work and the prospects for future development.

2. ARCHITECTURE OVERVIEW

RTPM is based on a modular layered architecture. Figure 1 gives an overview of the different layers provided by the framework.

![Figure 1: RTPM architecture overview.](image)

The two bottom layers provide the interface with the underlying operating system, support interoperability and some degree of virtualization. In the peer-to-peer networking layer we have the underlying peer-to-peer overlays that support scalable, efficient protocols for networking such as membership, discovery and routing. Each computational node may belong to several overlays simultaneously. The kernel is the central piece of RTPM’s architecture, its responsibilities include the management of the network resources, task scheduling, including real-time policies, and QoS enforcing. Atop the kernel we have the kernel services, that have access to in-kernel facilities, just as Linux’s kernel modules. The top layers are the user application and user services, both having access to several entry points in the framework: user services; kernel services; kernel; and network (network access is guarded and monitored by the kernel, thus for added simplicity, this connection is omitted in figure 1). This allows users to implement their applications with minimal framework overhead.

3. OPERATING SYSTEM INTERFACE

One important design goal of RTPM is to provide a portable infrastructure capable of running in a multitude of operating systems. We decided to use the ACE framework [19] that provides patterns and components for the seamless development of portable high-performance real-time services and applications. ACE simplifies the development of network services by providing mechanisms for inter-process communication, event demultiplexing, explicit dynamic linking, and concurrency.

Linux is our most used OS and we dedicate special attention to it. In this context, work on providing real-time and QoS focuses on ensuring deterministic code paths and
preemptive interrupt handling, while trying to minimize the length of the code paths. Another important feature, that so far has deserved less attention, is real-time aware scheduling. For example, the ARTiS [12] project studied scheduling techniques for the new multicore processor architectures. The basic idea was to reserve a set of cores for real-time operations and another set for generic operations. The real-time reservation, while guaranteed, is not exclusive and does not imply a waste of resources. A migration mechanism of non-preemptive tasks ensures a latency level on these real-time processors. Furthermore, load-balancing strategies take full advantage of the full power of the SMP systems.

RTT₉ goes one step further in that it introduces an admission control entity, implemented as a kernel module, that is responsible for verifying that a given real-time task is executable, under the chosen scheduling policy (e.g. EDF – Earliest Deadline First, or RM – Rate Monotonic).

4. PEER-TO-PEER NETWORKING

The networking layer is based on the notion of network overlay introduced in JXTA [8]. However, JXTA’s rigid architecture, text based messaging and lack of QoS support make it inadequate to be considered as starting point for an implementation. Thus we have opted to design and implement our own peer-to-peer infrastructure.

We use peer-to-peer overlays based on P³ [17], resulting in a hierarchical peer-to-peer topology, where the bottom peers have less knowledge than the upper peers, the later also called super-peers. A set of these super-peer is called a cell, and they cooperate in order to maintain the coordination of lower rank peers. Lower rank peers can be promoted, or promote themselves in certain circumstances, to become a super-peer and join a given coordination cell. This characteristic of the networking layer effectively excludes single point of failure problems higher up in the framework.

Each network overlay (figure 2) includes the following modules: membership, that handles the overlay’s dynamic topology; messaging, a QoS aware messaging infrastructure; security, that enforces the security policies; routing, that maintains the routing information, and; discovery, that controls dynamic resource searching and publish/subscribe. This modular approach makes it possible to implement distinct peer-to-peer overlays by simply adjusting the behavior implemented by these modules.

The membership module is responsible for joining, maintaining and leaving the peer-to-peer overlay. The bootstrap process is done by means of a multicast advertisement, where the peer announces its wish to join the overlay. The nearby available super-peers reply to this request by sending a message, using the messaging service, containing all relevant information on service access points (SAP) and their respective real-time and QoS policies.

Message passing is supported by the messaging module, that is optimized for throughput rather than for low latency. We have chosen this approach in order to ease the implementation of the prototype. More elaborate strategies will be implemented in the future. This module has several SAPs, each one reflecting an implicit QoS policy. By doing this we can avoid multiplexing requests of different QoS values, decreasing overheads and thus promoting real-time behavior.

The routing information is maintained by the routing module, that handles all the necessary operations needed to maintain the overlay organization, with full support for direct and indirect routing.

All the security features are implemented in the overlay’s security module, that in turn is coordinated by the kernel. The main goal of this module is to enforce security policies on the access and usage of local resources.

The discovery service handles dynamic resource searching and resource publishing, using the messaging service for peer-to-peer communication. The search capability uses the routing and messaging services. When a peer is not known, a dynamic search is made in order to find that specific peer in the overlay. In this context, the discovery service uses the messaging service to send a message to the coordination cell of the current peer, requesting the wanted routing information. If none of the super-peers have the requested information, the request is propagated until the information is found or the root cell is reached (and the resource is found). When a peer wants to publish a new SAP, it uses the publishing capabilities built in the discovery module, that acts as a proxy for the peer and contacts the peer’s coordination cell by sending an advertisement message.

5. THE KERNEL

The kernel layer is composed of five main modules (figure 3), namely: a) the core module acts as the controller for the middleware, and is responsible for enforcing the desired real-time, QoS, fault-tolerance and load-balancing policies; b) the real-time and QoS modules manage all aspects of the real-time and QoS related policies. It is through these modules that the real-time behavior of the framework is configured, allowing for example, to define a balance between throughput and latency. Middleware tasks are processed based on the chosen real-time model and QoS parameters. Here there are multiple opportunities for optimization. For example, TAO [22] demonstrated several software patterns that can be applied in the handling of concurrency in a real-time context providing in-layer and cross-layer reduction of overheads (e.g. the half-sync/half-async [20] and leader-
The use of RTP over overlays and local virtualization resources (e.g., virtualization module) allows for a centralized enforcement of security policies; d) the virtualization module manages all the aspects of the migration of services, coordinating the efforts of the network overlays and local virtualization resources (e.g., libVirt in Linux); f) finally, a set of kernel modules provides the entry points for kernel and user level services to access privileged resources.

6. KERNEL AND USER-LEVEL SERVICES

The tight integration of the fault-tolerance, QoS and real-time support in the network and kernel layers might lead to a monolithic framework architecture that would be difficult to maintain and unreliable. Thus, despite the focus on performance, modularity and operation safety are greater concerns in the RTPM design. To promote modularity and efficient resource management, the framework supports the dynamic (un)loading of services on demand, inspired in ACE’s service framework.

There are basically two types of services: kernel services and user services. The kernel services are similar to Linux’s kernel modules in that they have access privileges to kernel facilities, and network layer resources, providing a cross-layer pathway for interaction with key software components with a negligible performance loss. The remote procedure call (RPC), for example, is a primitive kernel service in the current prototype. User services, on the other hand, have a limited access to the underlying RTPM resources through the use of RTPM’s user API. An event service, a common middleware facility that is responsible for notifying consumers of specific system events, can be straightforwardly implemented as a user-level service that uses the discovery service provided by the peer-to-peer networking.

7. PROGRAMMING API

In this section we present a general view of the API. Several facilities are not reflected in the code samples that follow for the sake of clarity and due to space restrictions.

The RTPM’s application programming interface is naturally mapped into the C++ namespaces. The base namespace is country::company::rpm which has the following nested namespaces: service, with all user applications and services; kernel, with the core classes for RTPM; and, network, with the classes responsible for the management of the peer-to-peer overlays.

The main entry point for the API is class RTPM (listing 1). To access the RTPM’s runtime facilities a user must provide some authentication information (lines 31 to 34). Among other functions, the runtime manages the kernel services (lines 17 to 22), user services (lines 10 to 15) and controls the network overlays (lines 24 to 29).

```cpp
1 namespace country { namespace company { namespace rpm {  
2     class RTPM {  
3         public:  
4             // RTPM entry point  
5             RTPM ( string& user, string& passphrase ) throw ( RTPmException );  
6             virtual ~RTPM ();  
7             // user services API  
8             void insertUserService ( UserService* us ) throw ( RTPmException );  
9             void removeUserService ( UUID uuid ) throw ( RTPmException );  
10             UserService* getUserService ( UUID uuid ) throw ( RTPmException );  
11             // kernel services API  
12             void insertKernelService ( KernelService* ks ) throw ( RTPmException );  
13             void removeKernelService ( KernelService* ks ) throw ( RTPmException );  
14             KernelService* getKernelService ( UUID uuid ) throw ( RTPmException );  
15             // network related API  
16             void insertNetwork ( string& networkLibPath ) throw ( RTPmException );  
17             void removeNetwork ( UUID networkUUID ) throw ( RTPmException );  
18             Network* getNetwork ( UUID networkUUID ) throw ( RTPmException );  
19             // user related API  
20             void loginUser ( string& user, string& passphrase ) throw ( RTPmException );  
21             void logoutUser ( string& user ) throw ( RTPmException );  
22         }  
23     }  
24 }  
25 }  
26 }  
27 }}
```

Listing 1: Main interface.

The Kernel class (listing 2) encapsulates all the core entities present in the runtime, ranging from kernel and user service handling (lines 5 to 34), security (line 25) and, network management (line 26). More details on the security and network management modules are shown in listings 3 and 4, respectively.

```cpp
1 namespace country { namespace company { namespace rpm {  
2     namespace kernel {  
3         class Kernel {  
4             public:  
5                 void insertUserService ( User* user, UserService* ks ) throw ( KernelException );  
6                 void removeUserService ( User* user, UserService* ws ) throw ( KernelException );  
7             }  
8         }  
9     }  
10 }  
11 }}
```

Listing 2: Kernel class.
The security module implements the general RTPM security policy, allowing the enforcement of service level permissions. The access is granted to privileged users with adequate permissions (lines 9 to 15).

```
namespace country { namespace company { namespace rtpm {
  namespace kernel {
    class KernelSecurity { public Module {
      Permissions+ getPermissions(User+ user, 
      UUID+ serviceUUID) {
        throw (SecurityException); 
      }
      void addUser(User+ user, Permissions+ permissions) {
        throw (SecurityException); 
      }
      void removeUserPermissions(User+ user, 
      Permissions+ permissions) {
        throw (SecurityException); 
      }
      void removeUser(User+ user) {
        throw (SecurityException); 
      }
    }
  }
}}
```

Listing 2: RTPM’s kernel interface.

Another runtime kernel module is the network manager that implements the NetworkMgr class (shown in listing 4), responsible for the dynamical (un)loading of peer-to-peer overlays.

```
1 namespace country { namespace company { namespace rtpm {
  namespace network {
    class NetworkMgr { public Service {
      public:
      UUID+ getUUID();
      string8 getName();
    }
    virtual Routing+ getRouting() = 0;
    virtual Messaging+ getMessaging() = 0;
    virtual Membership+ getMembership() = 0;
    virtual Discovery+ getDiscovery() = 0;
    virtual Security+ getSecurity() = 0;
    }}
}}
```

Listing 5: RTPM’s network interface.

For example, listing 6 shows the code to dynamically load an RPC kernel-service onto the runtime.

```
1 try{
2 RTPm+ runtime = new RTPm("user", "passphrase");
3 RPCService+ rpcService = new RPCService();
4 runtime-›insertKernelService (rpcService);
5 return (RTPmException& ex){
6   // error handling ...
7 }}
```

Listing 6: Dynamic loading of RPC service.

8. CONCLUSIONS AND FUTURE WORK

In this paper we present the architecture of RTPM, a middleware framework based on a peer-to-peer infra-structure and aimed at fault-tolerant, real-time, QoS computing. The modular peer-to-peer infra-structure is based on P2P [17] overlays, supporting kernel services with efficient, highly scalable protocols, e.g. membership, discovery, routing and messaging. RTPM’s kernel architecture includes several real-time and QoS features, namely at the level of service virtualization, thread scheduling and cross-layer optimizations and, multicore aware scheduling.

RTPM is an ongoing work. We are currently finalizing the implementation of the first system prototype which, as yet, has no support for virtualized services. This work will be followed by a thorough architecture and performance evaluation. All future developments are, naturally, dependent on this evaluation. However there are some issues that we envision will be important in future work.

First we plan to add the aforementioned service virtualization support by integrating existing technology [10, 5], providing pathways for implementing dynamic load-balancing, highly availability of services and fault-tolerance. A further QoS enhancement will be provided by the use of the concepts brought by the XenSockets [26], allowing for optimized communication paths between RTPM runtime and virtualized services. Another interesting feature to include is real-time computing with lock-free shared objects [3, 24], for improved performance. Finally, thread scheduling, namely at the level of multicore architectures [12], is another important aspect to explore given the ubiquity of these systems in our days and presumably in the future.

Acknowledgments.

Rolando Martins is supported by the SFRH/BDE/15644/2006 grant from the Fundação para a Ciência e Tecnologia and by EFACEC Sistemas de Electrónica, S.A.. Luís Lopes is partially funded by project CALLAS of the Fundação para a Ciência e Tecnologia (contract PTDC/EIA/71462/2006).

9. REFERENCES