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

Rolando Martins
CRACS/EFACEC Sistemas de Electrónica, S.A.
Rua Engenheiro Ulrich - Apartado 3081
4471-907 Moreira da Maia
rolando.martins@efacec.pt

Luís Lopes, Fernando Silva
CRACS/Faculdade de Ciências, Universidade do Porto
Rua Campo Alegre 1021/1055
Portugal, 4169 - 007 Porto
{lblopes,fds}@dcc.fc.up.pt

Technical Report Series: DCC-2008-02

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Rua do Campo Alegre, 1021/1055,
4169-007 PORTO,
PORTUGAL
Tel: 220 402 900 Fax: 220 402 950
http://www.dcc.fc.up.pt/Pubs/

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 RTP_M, 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 infra-structure with the main focus being on providing a scalable, resilient, reconfigurable, highly available platform for real-time and QoS computing.

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[1], 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. [8, 14, 18, 21]), 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 [12, 13, 15, 20], arguably the standard by which most middleware implementations are developed.

We feel, however, that the way this support is introduced 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 [21]. For example,

[1] EFACEC, the largest Portuguese Group in the field of electricity, with a strong presence in systems engineering namely in public transportation systems, employs around 3000 people and has a turnover of almost 500 million euro; it is established in more than 50 countries and exports almost half of its production (c.f. **http://www.efacec.pt**).

TAO introduced different, configurable, execution models to minimize code paths both within and across layers, thus minimizing protocol overheads. The ICE [8] 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, 9], has provided a new tool to solve problems like reconfigurability, 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 centralized client-server architecture of the networking layer and replace it with a more flexible infra-structure with builtin 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, 17].

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 [4, 6, 10]. Finally, there are several frameworks that provide system developers with components and patterns to implement custom peer-to-peer systems [16, 24, 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 RTP$_M$ (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 with QoS support, in the line of peer-to-peer overlays such as P^3 [16] and Tapestry [27].

The RTP$_M$ networking layer relies on a modular infra-struc-ture 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 functionalities 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, RTP$_M$ 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 [21] 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 functionalities (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 RTP$_M$ 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 RTP$_M$ network model. Section 5 describes RTP$_M$'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

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

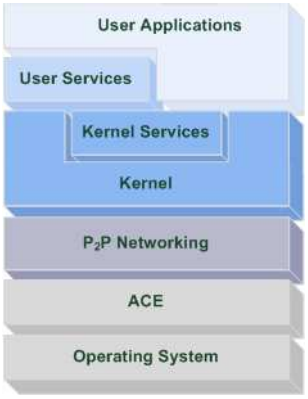

Figure 1. RTP$_M$ architecture overview.

The two bottom layers provide the interface with the underlying operating system, support interoperability and some degree of virtualization. In the **peer-to-peer net-**

working 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 RTP$_M$'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 customize their applications for minimal framework overhead.

3 Operating System Interface

One important design goal of RTP$_M$ is to provide a portable infrastructure capable of running in a multitude of operating systems. We decided to use the ACE framework [18] 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. The path for enabling real time performance in the Linux kernel demands efforts in reducing long code paths, better interrupt handling, and a real-time aware scheduler. The first two are being tackled by the Linux community, while the last is almost unexplored. In this context, the ARTiS [11] project explored scheduling techniques for the new multi-core processor architectures. More specifically, ARTiS tries 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.

RTP$_M$ goes one step further in that it introduces an admission control entity, implemented as a kernel module, that is responsible for verifying the executability of a given real-time task, 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 [24]. 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 infra-structure.

We use peer-to-peer overlays based on P^3 [16], 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.

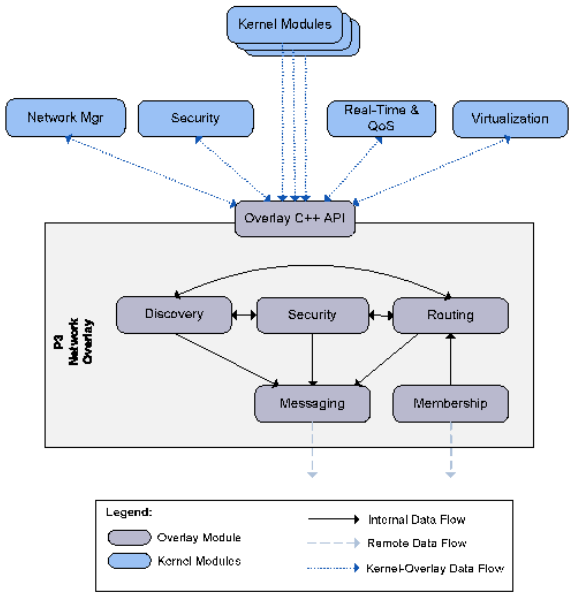

Figure 2. Peer-to-peer overlay architecture.

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 the 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 isn't 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 [21] demonstrated several software patterns that can be applied in the handling of

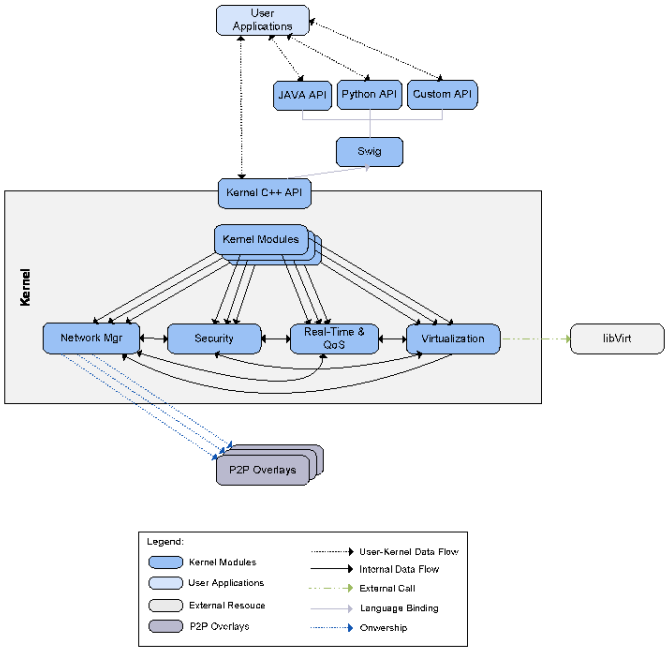

Figure 3. Kernel architecture.

concurrency in a real-time context providing in-layer and cross-layer reduction of overheads (e.g. the half-sync/half-async [19] and leader-followers [22] patterns); c) the **security** module contains all the information about security policies used to control access to resources throughout the framework. This module manages the security modules of every network overlay entity, allowing for a centralized enforcement of security policies; d) the **network manager** is responsible for maintaining every network overlay in the system and supports their management, e.g. dynamic (un)loading of network overlays; e) 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 RTP$_M$ 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 RTP$_M$ resources through the use of RTP$_M$'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 RTP$_M$'s application programming interface is naturally mapped into the C++ namespaces. The base namespace is country::company::rtpm which has the the following nested namespaces: service, with all user applications and services; kernel, with the core classes for RTP$_M$, 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 RTP$_M$'s runtime facilities a user must provide some authentication information (lines 31 to 34). Among other functionalities, 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).

```
1  namespace country { namespace company { namespace rtpm {
2    class RTPm{
3      public:
4        // RTPm entry point
5     RTPm(string& user, string& passphrase)
6          throw(RTPmException);
7     virtual ~RTPm();
8
9     // user services API
10    void insertUserService(UserService* us)
11         throw (RTPmException);
12    void removeUserService(UUID* uuid)
13         throw (RTPmException) ;
14     UserService* getUserService(UUID* uuid)
15         throw (RTPmException) ;
16     // kernel services API
17    void insertKernelService(KernelService* ks)
18         throw (RTPmException);
19    void removeKernelService(KernelService* ks)
20         throw (RTPmException);
21    KernelService* getKernelService(UUID* uuid)
22         throw (RTPmException) ;
23     // network related API
24    void insertNetwork (string& networkLibPath);
25         throw (RTPmException);
26    void removeNetwork (UUID* networkUUID)
27         throw (RTPmException);
28    Network* getNetwork (UUID* networkUUID)
29         throw (RTPmException);
30     // user related API
31    void loginUser(string& user, string& passphrase)
32         throw (RTPmException) ;
33    void logoutUser(string& user)
34         throw (RTPmException) ;
35  };
36 }}}
```

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 23), 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.

```
1  namespace country { namespace company { namespace rtpm {
2  namespace kernel  {
3   class Kernel {
4    public:
5     void insertUserService (User* user,
6           UserService* ks)
7           throw (KernelException);
8     void removeUserService (User* user,
9           UUID* uuid)
10          throw (User* user, KernelException);
11    UserService* getUserService (User* user,
12          UUID* uuid)
13          throw (KernelException);
14
15    void insertKernelService (User* user,
16          KernelService* ks)
17          throw (KernelException);
18    void removeKernelService (User* user,
19          UUID* uuid)
20          throw (User* user, KernelException);
21    KernelService* getKernelService (User* user,
22          UUID* uuid)
23          throw (KernelException);
24   protected:
25    KernelSecurity* getSecurity();
26    NetworkMgr* getNetworkManager();
27   };
28  }
29  }}}
```

Listing 2. RTP$_M$'s kernel interface.

The security module implements the general RTP$_M$ security policy, allowing the enforcement of service level permissions. The access is granted to privileged users with adequate permissions (lines 9 to 15).

```
1  namespace country { namespace company { namespace rtpm {
2  namespace kernel  {
3   class KernelSecurity: public Module {
4    public:
5     Permissions* getPermissions(User* user,
6           UUID* serviceUUID)
7           throw(SecurityException);
8
9     void addUser(User* user, Permissions* permissions)
10          throw(SecurityException);
11    void removeUserPermissions(User* user,
12          Permissions* permissions)
13          throw(SecurityException);
14    void removeUser(User* user)
15          throw(SecurityException);
16  };
17 }}}
```

Listing 3. RTP$_M$'s security module.

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 {
2  namespace kernel {
3   class NetworkMgr: public Module
4   {
5    public:
6     void insertNetwork(string& networkLibPath)
7          throw (KernelException);
8     void removeNetwork(Network* net)
9          throw (KernelException);
10    Network* getNetwork(UUID* netUUID)
11         throw (KernelException);
12    Network* getNetwork(string& netName)
13         throw (KernelException);
14    // search an UUID in all the active overlays
15    Network* findUUID(UUID* uuid);
16  };
17 }
18 }}}
```

Listing 4. RTP$_M$'s network manager module.

A network overlay is represented by the Network class (listing 5) which provides hooks for the **routing** (line 9), **messaging** (line 10), **membership** (line 11), **discovery** (line 12) and **security** (line 13) modules, as described in section 4.

```
1  namespace country { namespace company { namespace rtpm {
2  namespace network {
3   class Network: public Service {
4    public:
5     UUID* getUUID();
6     string& getName();

8     // network modules
9     virtual Routing* getRouting() = 0;
10    virtual Messaging* getMessaging() = 0;
11    virtual Membership* getMembership() = 0;
12    virtual Discovery* getDiscovery() = 0;
13    virtual Security* getSecurity() = 0;
14  };
15 }
16 }}}
```

Listing 5. RTP$_M$'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 }catch(RTPmException& ex){
6   // error handling ...
7 }
```

Listing 6. Dynamic loading of RPC service.

Listing 7 sketches the development of a kernel service, a simple RPC service. Being a kernel service, it can access reserved kernel facilities, such as threading policies, QoS and real-time policies.

The user services are managed by the interface implemented in lines 12 to 15. The sketched RPC service code does not rely on the generic messaging service (for demonstration purposes), instead it manages all the key interactions with the network layer. Lines 7 to 9 shows the network registration and client handling (with the addition of lines 57 to 60), through the Acceptor [18] class.

The oneWayInvocation method starts by assesing the user permissions to execute the requested invocation (lines 18 to 24), that is followed by a search to our local services list with the objective of checking for a possible local invocation (consult lines 25 to 30). If the wanted service is not local, we use the network manager to conduct a search across all the overlays present in the RTP$_M$'s runtime, in order to find a suitable network (lines 31 to 39).

If such a network exists, a RPC client is created that interacts with the remote Acceptor instance (see lines 40 to 44), and will be used to send the rpc request (line 46). Incoming RPC requests are dispatched by the network client to the service, through the interface sketched in lines 62 to 70.

Finally, lines 72 to 75 shows the interface stub for handling the replies for two way invocations.

```
1  namespace country { namespace company { namespace rtpm {
2  namespace kernel {
3   // RPC kernel service
4   class RPCService: public KernelService {
5    public:
6     RPCService() {
7      Network* net = getNetworkManager()->
8           register(RPCService::getUUID());
9      net->addAcceptor(new RPCAcceptor(this));
10    }
11    // service management ...
12    void registerRPCServer (RPCUserService* server)
13         throw (RPCException);
14    void unregisterRPCServer (RPCUserService* server)
15         throw (RPCException);

17    void oneWayInvocation (RPCRequest* request) {
18     User* user = request->getUser();
19     Permissions* perms =
20          this->getKernel()->security->
21            getPermissions (user, request->getUUID());
22     if (!perm->hasExecutionPermission ()) {
23      throw RPCException(NOT_ALLOWED);
24     }
25     Service* service;
26     if ((service =
27          isLocalRPCService (request->getUUID())) != 0) {
28      service->oneWayInvocation (request);
29      return;
30     }else {
31      Network* net = getNetworkManager->
32          findNetworkByObjectUUID (request->getUUID());
33      if( net != 0) {
34       UUID* peerUUID = net->getDiscovery()->
35          findUUID (RPC_SERVICE, request->getUUID());
36       if (peerUUID == 0) {
37        // peer no longer exists
38        throw RPCException(SERVICE_NOT_FOUND);
39       }else {
40        try {
41         RPCNetClient* rpcNetClient =
42          static_cast<RPCNetClient*>
43           (net->getClient (RPCService::getUUID(),
44                   peerUUID));

46         rpcNetClient->send (request);
47        }catch (NetwortException& ex) {
48         throw RPCException(SERVICE_NOT_AVAILABLE);
49        }
50       }
51      }else {
52       throw RPCException(SERVICE_NOT_FOUND);
53      }
54     }
55    }

57    int onNewClient(RPCSvcHandler* rpcSrvHandler) {
58      rpcSrvHandler->setRPCService(this);
59      return rpcSrvHandler->open();
60    }

62    int onRPCRequest(RPCSvcHandler* rpcSrvHandler,
```

```
63          RPCRequest* request) {
64      if (request->oneWayInvocation()) {
65       this->oneWayInvocation (request);
66      } else {
67       RPCReply* reply = this->twoWayInvocation (request);
68       rpcSrvHandler->send (reply);
69      }
70     }

72     int onRPCReply(RPCSvcHandler* rpcSrvHandler,
73          RPCReply* reply) {
74       // ...
75     }
76   };
77 }
78 }}}
```

Listing 7. Request dispatch in RPC kernel service.

Listing 8 sketches a user-level RPC service that basically registers itself with the kernel RPC service defined above and waits for requests issued by clients (lines 12 to 23).

```
1  namespace country { namespace company { namespace rtpm {
2  namespace services{

4   // a RPC user server
5   class RPCExampleServer: RPCUserService {
6    public:
7     static UUID* getServiceUUID();
8     enum Operations {
9      PING_OP = 1
10    };

12    void oneWayInvocation (RPCRequest* request)
13         throw (ServiceException) {
14     switch(request->getOperation()) {
15      case PING_OP: {
16       ping();
17       return;
18      }
19      default:
20       throw
21       ServiceException(OPERATION_NOT_IMPLEMENTED);
22     }
23    }

25    void ping() {
26     // the actual ping code goes here ...
27    }
28   };
29 }
30 }}}
```

Listing 8. RPC user service.

Listing 9 sketches a RPC client that handles the (un)registration of the server with the runtime (lines 9 to 16) and interfaces with the RPC kernel service from listing 7, the later acting as a proxy for the user service defined in listing 8 (lines 18 to 23 and 26 to 30).

```
1  namespace country { namespace company { namespace rtpm {
2  namespace services{

4   // a RPC client
5   class RPCExampleServiceClient: public RTPmNotifier {
6    public:
7     RPCExampleServiceClients(RTPm* runtime) ;

9     void registerRPCServer(RPCUserService* server)
10         throw (RPCException) {
11     getRPCService ()->registerRPCServer (server);
12    }
13    void unregisterRPCServer(UUID* rpcUserServiceUUID)
14         throw (RPCException) {
15     getRPCService ()->unregister (rpcUserServiceUUID);
16    }

18    void ping()  throw (RTPmException) {
19     RPCRequest* request =
20          new RPCRequest(RPCExampleServer::getUUID (),
21           RPCExampleServer::PING_OP);
22     getRPCService()->oneWayInvocation(request);
23    }
24  protected:
25    RTPm* getRuntime();
26    RPCService* getRPCService() throw (RTPmException){
27      KernelService* service = getRuntime ()->
28          getKernelService(RPCService::getUUID ());
29      return static_cast<RPCService*> (service);
30    }
31   };
32 }
33 }}}
```

Listing 9. RPC client.

8 Conclusions and Future Work

In this paper we present the architecture of RTP$_M$, 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 P^3 [16] overlays, supporting kernel services with efficient, highly scalable protocols, e.g. membership, discovery, routing and messaging. RTP$_M$'s kernel architecture includes several real-time and QoS features, namely at the level of service virtualization, thread scheduling and cross-layer optimizations and, multi-core aware scheduling.

RTP$_M$ 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 [5, 9], 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 RTP$_M$ runtime and virtualized services. Another interesting feature to include is real-time computing with lock-free shared objects [3, 23], for improved performance. Finally, thread scheduling, namely at the level of multi-core architectures [11], 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).

References

[1] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-Grid: A Self-organizing Structured P2P System. *SIGMOD Rec.*, 32(3):29–33, 2003.

[2] K. Adams and O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of 15th ACM SIGPLAN International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, pages 2–13. ACM, 2006.

[3] J. Anderson, S. Ramamurthy, and K. Jeffay. Real Time Computing with Lock-Free Shared Objects. In *IEEE Real-Time Systems Symposium*, pages 28–37, 1995.

[4] F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-Area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, volume 35, 5 of *ACM SIGOPS Operating Systems Review*, pages 202–215. ACM Press, October 2001.

[5] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'03)*, October 2003.

[6] P. Druschel and A. Rowstron. PAST: A Large-Scale, Persistent Peer-to-Peer Storage Utility. In *Proceedings of 8th Workshop on Hot Topics on Operating Systems (HotOS VIII)*, pages 75–80, 2001.

[7] A. Gokhale, B. Natarajan, D. Schmidt, and J. Cross. Towards Real-Time Fault-Tolerant CORBA Middleware. *Cluster Computing*, 7(4):331–346, 2004.

[8] M. Henning. A New Approach to Object-Oriented Middleware. *IEEE Internet Computing*, 8(1):66–75, 2004.

[9] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux Virtual Machine Monitor. In *Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07)*, June 2007.

[10] J. Kubiatowicz, D. Bindel, Y. Chen, S. E. Czerwinski, P. R. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX), ACM SIGPLAN*, pages 190–201. ACM Press, 2000.

[11] M. Momtchev and P. Marquet. An Asymmetric Real-Time Scheduling for Linux. In *Proceedings of the 16th International Parallel & Distributing Processing Symposium (IPDPS'02)*, pages 96–96, 2002.

[12] B. Natarajan, A. Gokhale, S. Yajnik, and D. Schmidt. DOORS: Towards High-Performance Fault Tolerant CORBA. In *Proceedings of International Symposium on Distributed Objects and Applications (DOA'00)*, pages 39–48, 2000.

[13] Object Management Group. Fault Tolerant CORBA Specification. OMG Technical Committee Document, June 2002.

[14] Object Management Group. The Common Object Request Broker: Architecture and Specification. OMG Technical Committee Document, June 2002.

[15] Object Management Group. Real-time CORBA Specification. OMG Technical Committee Document, January 2005.

[16] L. Oliveira, L. Lopes, and F. Silva. P^3: Parallel Peer to Peer – An Internet Parallel Programming Environment. In *Workshop on Web Engineering & Peer-to-Peer Computing, part of Networking 2002*, volume 2376 of *Lecture Notes in Computer Science*, pages 274–288. Springer-Verlag, 2002.

[17] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. Technical Report TR-00-010, University of California at Berkeley, 2000.

[18] D. Schmidt. An Architectural Overview of the ACE Framework. *;login: the USENIX Association newsletter*, 24(1), Jan. 1999.

[19] D. Schmidt and C. Cranor. Half-Sync/Half-Async: An Architectural Pattern for Efficient and Well-Structured Concurrent I/O. In *Proceedings of the 2nd Annual Conference on the Pattern Languages of Programs (PLoP'95)*, pages 1–10, 1995.

[20] D. Schmidt and F. Kuhns. An Overview of the Real-Time CORBA Specification. *IEEE Computer*, 33(6):56–63, 2000.

[21] D. Schmidt, D. Levine, and S. Mungee. The Design of the TAO Real-Time Object Request Broker. *Computer Communications*, 21(4):294–324, 1998.

[22] D. Schmidt, C. O'Ryan, I. Pyarali, M. Kircher, and F. Buschmann. Leader/Followers: A Design Pattern for Efficient Multi-threaded Event Demultiplexing and Dispatching. In *7th Pattern Languages of Programs Conference*, 2001.

[23] H. Sundell and P. Tsigas. Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems. In *Proceedings of the 17th International Parallel & Distributing Processing Symposium (IPDPS'03)*. IEEE press, 2003.

[24] J. Team. JXTA v2.0 Protocol Specification, October 2007.

[25] G. Vigna. *Mobile Code Technologies, Paradigms, and Applications*. PhD thesis, Politecnico Di Milano, Milan, Italy, 1997.

[26] X. Zhang, S. McIntosh, P. Rohatgi, and J. Griffin. XenSocket: A High-Throughput Interdomain Transport for Virtual Machines. In *Proceedings of the 8th ACM/IFIP/USENIX International Middleware Conference (Middleware 2007)*, volume 4834 of *Lecture Notes in Computer Science*, pages 184–203. Springer, 2007.

[27] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A Resilient Global-Scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 2003.