Universidade do Minho
Escola de Engenharia

Diogo Pereira Ribeiro

Implementation of an API for Distributed Communication Between Processes in Closed Contexts

Junho de 2012

Diogo Pereira Ribeiro Implementation of an API for Distributed Communication Between Processes in Closed Contexts

UMinho | 2012

Universidade do Minho
Escola de Engenharia

Diogo Pereira Ribeiro

Implementation of an API for Distributed Communication Between Processes in Closed Contexts

Dissertação de Mestrado
Ciclo de Estudos Integrados Conducentes ao
Grau de Mestre em Engenharia de Comunicações

Trabalho efectuado sob a orientação do
Professor Doutor Bruno Alexandre Fernandes Dias

Junho de 2012

Acknowledgements

This dissertation is the product of many hours of effort and dedication and it represents the turning point of a goal in my life I had long hoped to achieve. This would have been next to impossible without the help and support from many people, namely:

My family, and my parents in particular, whose support and sacrifice made this a possibility.

My supervisor, Bruno Dias, for his knowledge, readiness and dedication towards helping me achieve my objectives.

My friends and colleagues, whose presence, care and patience kept me motivated and going through some less easy times.

Everyone else not mentioned and who contributed, directly or not, to the development of this dissertation.

I would like to express my sincere gratitude towards everyone mentioned above, whom without this work wouldn't either be possible or as thorough as it is.

Abstrato

Existem atualmente diversas *Application Programming Interfaces* (APIs) que ajudam na programação de aplicações distribuídas. Na maior parte dos casos, estas utilizam de forma inflexível um único tipo de protocolo aplicacional e interface, ficando dependente dos protocolos de transporte já existentes e do sistema operativo. Para o programador, a *stack* de protocolos e o tipo de interface têm que ser decididos explicitamente antes do estabelecimento da comunicação entre os processos.

Algumas APIs facilitam a programação ocultando alguns aspetos específicos dos mecanismos e protocolos de comunicação utilizados, disponibilizando uma interface mais homogeneizada. No entanto, a programação continua a não ser totalmente transparente e independente dos protocolos de comunicação utilizados, dos sistemas operativos e da localização relativa dos processos comunicantes. Além disso, estas APIs não tomam decisões sobre o mecanismo de comunicação a utilizar quando existem várias alternativas possíveis, sendo esta decisão da responsabilidade do programador.

Num contexto de implementação de simuladores distribuídos e modulares para protocolos de redes de computadores e sistemas de comunicação, seria vantajoso poder-se utilizar uma API para comunicação dos processos de simulação que disponibilizasse apenas um único interface de programação e que decidisse de forma transparente o mecanismo ou protocolo comunicacional mais eficiente, tendo em conta a localização relativa dos processos.

Nesta dissertação são abordadas as soluções semelhantes já existentes e é estudada uma API que pretende preencher estas lacunas. A arquitetura desta API será depois apresentada, assim como uma solução com base na investigação realizada. Por fim, os resultados dos testes serão analizados e a conclusão apresentada.

Esta dissertação foi desenvolvida no contexto do projeto RoutUM, um simulador de redes de computadores atualmente a ser desenvolvido pela Universidade do Minho.

Palavras-chave: Computação distribuída, Comunicação inter-processos, Simulação de redes, API, RoutUM

Abstract

There are several *Application Programming Interfaces* (APIs) available to simplify the development of distributed applications. In most cases, they invariably use one type of application protocol and interface, being dependent of the existing transport protocols and operating system. To the programmer, the protocol stack and the type of interface must be explicitly chosen before initiating communication between processes.

Some APIs simplify programming by hiding some specific aspects about the communication protocols and mechanisms, revealing a more homogenized interface. However, the programming is still not completely transparent and independent from the communication protocols, the operating system and the relative location of the communicating processes. Also, they are unable to decide which communication mechanism to be used when there are several available possibilities, leaving that responsibility to the programmer.

On the context of implementation of distributed and modular simulators for network protocols and communication systems, it would be desirable to be able to use an API that would allow communication between the processes while providing only one programming interface. It would then transparently decide the most efficient mechanism or communication protocol from the relative location of the communicating processes.

This dissertation will present and discuss the currently available solutions and the problems associated with the development of an API which attempts to fill the missing features indicated above. The API's architecture will then be shown and developed into a solution based on the results from the investigation. In the end, this solution will be tested and the final results will be presented.

This dissertation was developed in the context of the RoutUM project, a network simulator being currently developed in the University of Minho.

Keywords: Distributed Computing, Inter-Process Communication, Network Simulation, API, RoutUM

Contents

List of Figures

List of Tables

Glossary

AVL binary tree An AVL tree is a self-balancing binary search tree. In an AVL tree, the heights of the two child subtrees of any node differ by at most one. Lookup, insertion, and deletion all take O(log n) time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. 87, 97

Base64 An encoding standard to store (binary) data in ASCII format, using only a printable sub-set of the 7-Bit characters that ASCII comprises. 90

bash A shell, or command language interpreter, for the GNU operating system. It was written by Brian Fox for the GNU Project. 26, 27

CPU scavenging A technique to create a computing grid from the unused resources in a network of computers. Typically this technique uses desktop computer instruction cycles that would otherwise be wasted when it is idling. In practice, these computers also donate some supporting amount of disk storage space, RAM, and network bandwidth, in addition to raw CPU power. 9, 21

daemon A a computer program that runs as a background process, rather than being under the direct control of an interactive user. 18, 81, 83–85, 88, 107

deadlock A a situation which occurs when a process enters a waiting state because a resource requested by it is being held by another waiting process, which in turn is waiting for another resource. If a process is unable to change its state indefinitely because the resources requested by it are being used by other waiting process, then the system is said to be in a deadlock. 15

DiffServ Differentiated Services (or DiffServ) is a computer networking architecture that specifies a simple, scalable and coarse-grained mechanism for classifying and managing network traffic and providing Quality of Service (QoS) on modern IP networks. DiffServ can, for example, be used to provide low-latency to critical network traffic such as voice or streaming media while providing simple best-effort service to non-critical services such as web traffic or file transfers.. 75

hash table A data structure that uses a hash function to map identifying values, known as keys, to their associated values. In other words, a hash table implements an associative array. The hash function is used to transform the key into the index (the hash) of an array element (the slot or bucket) where the corresponding value is to be sought. 51

IPsec Internet Protocol Security (IPsec) is a protocol suite for securing Internet Protocol (IP) communications by authenticating and encrypting each IP packet of a communication session. IPsec also includes protocols for establishing mutual authentication between agents at the beginning of the session and negotiation of cryptographic keys to be used during the session. 78

microkernel In computer science, a microkernel is the near-minimum amount of software that can provide the mechanisms needed to implement an operating system (OS). These include low-level address space management, inter-process communication and thread management. 25

mutex In concurrent programming, mutual exclusion algorithms (abbreviated to mutex) are used to avoid the simultaneous use of a common resource (such as a variable) by pieces of computer code called critical sections. A critical section is a piece of code in which a process or thread accesses the resource in common. 103, 104, 110

node In computer networking, a node is an active device attached to a computer network or other telecommunications network, such as a computer or a switch, or a point in a network topology at which lines intersect or branch. 8–10

process An instance of a computer program that is being executed. It contains the program code and its current activity. Its working memory is isolated from other processes and it contains an arbitrary number of threads. Communication between processes is only possible through IPC mechanisms. 1, 2, 12, 14, 15, 24, 26, 27, 43, 57, 68, 69

resource starvation A multitasking-related problem, where a process is perpetually denied necessary resources. Without those resources, the program can never finish its task. 15

software library In computer science, a library is a collection of resources used to develop software. These may include pre-written code and subroutines, classes, values or type specifications. Libraries contain code and data that provide services to independent programs. This encourages the sharing and changing of code and data in a modular fashion, and eases the distribution of the code and data. Another advantage is that the executables are typically smaller in size, since a portion of their code is contained in the library itself. Executables and libraries make references known as links to each other through the process known as linking, which is typically done by a linker. 107

syslog A standard for logging program messages. It allows separation of the software that generates messages, the system that stores them and the software that reports and analyzes them. It also provides devices which would otherwise be unable to communicate a means to notify administrators of problems or performance. Because of this, syslog can be used to integrate log data from many different types of systems into a central repository. 83, 95, 108

System V Unix System V is one of the first commercial versions of the Unix operating system. It was originally developed by American Telephone & Telegraph (AT&T) and first released in 1983. During the period of the Unix wars System V was known for being the primary choice of manufacturers of large multiuser systems, in opposition to BSD's dominance of desktop workstations. With standardization efforts such as POSIX and the commercial success of Linux, this generalization is no longer as accurate as it once was. 32, 36, 62–64

thread A thread of execution is the smallest unit of processing that can be scheduled by an operating system. The implementation of threads and processes differs from one operating system to another, but in most cases a thread is contained inside a process. Multiple threads can exist within the same process and share resources such as memory, while different processes do not. 36

X.509 In cryptography, X.509 is a standard for a public key infrastructure (PKI) for single sign-on (SSO) and Privilege Management Infrastructure (PMI). X.509 specifies, amongst other things, standard formats for public key certificates, certificate revocation lists, attribute certificates, and a certification path validation algorithm. 80

Acronyms

ACL Access Control List. 78

API Application Programming Interface. iv, v, 1–3, 13, 16, 30, 36, 37, 57–60, 62, 73–85, 87–93, 95–99, 101, 103–111, 113, 116, 119, 120, 125, 127–131, 133, 135–137, 139–141

CLI Command-Line Interface. 26, 62, 82, 93, 95, 96

CORBA Common Object Request Broker Architecture. 37

CPU Central Processing Unit. 9, 33, 34, 43, 55, 61, 69–71, 130, 132, 133, 137, 140

DBMS Database Management System. 82, 85, 89, 90, 92, 108

DNS Domain Name Service. 13, 85, 97

DoS Denial-of-Service. 13

FIFO First In, First Out. 27, 43, 62, 63, 67, 73, 87, 102

GCC GNU C Compiler. 68

GSI Grid Security Infrastructure. 23

GUI Graphical User Interface. 19, 46

IDE Integrated Development Environment. 46, 81

IP Internet Protocol. 13, 20, 28, 29, 65, 66, 68, 74–76, 83, 85, 87, 90, 95, 97–99, 121, 128, 130–135

IPC Inter-process communication. 2, 3, 5, 6, 12, 13, 24, 25, 28–30, 32–37, 47–49, 55–66, 68, 70, 73–76, 79, 83, 84, 87, 90, 92, 97–99, 101, 105, 107–110, 119, 121, 128, 130, 132–135, 137, 140–142

LAN Local Area Network. 75

LDAP Lightweight Directory Access Protocol. 23, 92

LP Logical Process. 43, 53

MAC Medium Access Control. 79

NAT Network Address Translation. 75

NED NEtwork Description. 47

NIC Network Interface Controller. 79

OMG Object Management Group. 37

OS Operating System. 79

OSI Open Systems Interconnection. 29

OTcl Object Tool Command Language. 44

PDES Parallel Discrete Event Simulation. 42

PDU Protocol Data Unit. 55, 140

PKI Public Key Infrastructure. 77, 83

POSIX Portable Operating System Interface for Unix. 26, 32, 34, 36, 60–63, 67, 68

QoS Quality of Service. 75

RAM Random-Access Memory. 33

RDC RoutUM's Distributed Computing. 81, 88, 90, 91, 95, 97, 106–109, 116, 130, 133, 135, 137, 140

RMI Remote Method Invocation. 37

RNG Random Number Generator. 40, 45

RPC Remote Procedure Call. 13, 37

SDK Software Development Kit. 16, 37

SMP Symmetric multiprocessing. 110

SOAP Simple Object Access Protocol. 37

SQL Structured Query Language. 82

SSL Secure Sockets Layer. 78, 83

SVID System V Interface Definition. 61

Tcl Tool Command Language. 44, 45

TclCl Tcl with classes. 44

TCP Transmission Control Protocol. 13, 20, 29, 65, 66, 68, 75, 76, 83, 87, 90, 95, 97–99, 121, 128, 130–135

UDP User Datagram Protocol. 66, 70, 71, 75, 76, 137

UID Unique Identifier. 79, 80

UML Unified Modeling Language. 88

VLAN Virtual Local Area Network. 115

WAN Wide Area Network. 75

XML Extensible Markup Language. 37

Chapter 1

Introduction

1.1 Motivation

There are several APIs currently available to simplify the development of distributed applications. In the majority of cases, they invariably use only one type of applicational protocol and interface, becoming dependent from the already existing transport protocols and operating system. To the programmer, the protocol stack and the type of interface must be explicitly decided before initializing the communication between processes.

Some APIs simplify programming by hiding some specific aspects about the communication protocols and mechanisms, revealing a more homogenized interface. However, the programming is still not completely transparent and independent from the communication protocols being used, the operating systems and the relative location of the communicating processes. Also, these APIs are unable to decide between the communication mechanism to use when there are several available possibilities, turning the responsibility towards the programmer to make an anticipated and conscious choice.

In the context of the implementation of distributed and modular simulators for network protocols and communication systems, it would be desirable to be able to use an API that could allow the communication between the simulation's processes while providing only one programming interface and transparently deciding the most efficient mechanism or communication protocol, having in consideration the relative location of the communicating processes.

1.2 Overview

This dissertation attempts to create a feasible solution for the problems posed above. In other words, it should be able to transparently decide between the most efficient communication mechanism and/or protocol when there is more than one available. This should be accomplished by having in consideration the overall performance of the transmission medium and the relative location of the communicating processes. All the specifics related to the transmission method (e.g. the address) or operating system should be hidden away from the programmer.

In theory, these features should result in an easy, transparent and homogenous programming interface. Also, the clear division between the communication subsystem and all of the other parts of an application should increase the modularization of the code and simplify the creation and maintenance of distributed applications. If properly developed, another advantage should arise from the increased performance, through dynamically choosing the best available method for communication (fastest, less error-prone, etc). This is a particularly important feature to have in distributed applications working in closed contexts, where seemingly minimal performance gains might come to have a severe impact on long lasting, computationally intensive and complex practical scenarios.

This dissertation was developed in the context of **RoutUM**, a network simulator currently being developed on the University of Minho. Some particular functionality might be implemented in order to satisfy the needs of this system. It should be noted, however, that this additional functionality should not compromise the versatility of the API, being purely optional.

1.3 Objectives

The work in this dissertation is split into three phases: investigation, development and testing.

The first phase consists of the state-of-the-art research and an analysis of existing implementations.

The second phase involves the investigation of the major problems posed by the API. These include:

- Choosing a programming language which allows the best possible performance in the context of distributed systems. Cross-platform support is preferable;
- Choosing between several Inter-process communication (IPC) mechanisms and/or associated protocols for the ones which offer the best performance. This phase is further split between local and remote mechanisms;
- Specification of an application-level communication protocol;
- Specification of a central database server responsible for the storage and distribution of information relative to processes, communication

interfaces, contexts, etc;

- Identification of the security and access policy needs and its respective application to all the communication modules;
- Definition of methods and metrics for performance testing of the API;
- Specification of an API that provides a normalized interface for connection-oriented communications, maximizing performance and reliability in a transparent and dynamic way to the programmer.

The third phase consists on the implementation and evaluation of the API. This also includes the development of a test application, in order to ensure the correct working of the API and to assess its reliability and performance.

1.4 Restrictions

Due to time limitations and the potential extensibility of this dissertation, the investigative and developmental phases need to be restricted. Therefore, only two types of interfaces will be taken into account for the API (one local and one remote IPC mechanism), with only one type of protocol (if applicable) for each interface type. Only Linux or similar UNIX-based operating systems will be supported by the API's prototype. However, it is still desirable to be taken into consideration the modularization and portability of the code.

1.5 Dissertation Structure

This document contains a total of six chapters.

The first chapter is dedicated to the introduction, indicating the motivation and the goals of this dissertation.

The second chapter describes the current state-of-the-art in the most relevant areas for this dissertation. A theoretical background of distributed systems, IPC mechanisms, and network simulators (RoutUM in particular) is presented.

The third chapter contains the identification and resolution of the main problems behind the development of an API which attempts to attain the established goals.

The fourth chapter describes an implementation based on the knowledge acquired from the previous chapter. A test program is also included, in order to test the resulting API.

The fifth chapter presents some tests and evaluations made to the aforementioned API.

The sixth chapter states the final conclusions taken from this dissertation.

Chapter 2

Background and Related Work

2.1 Distributed Computing

The use of distributed systems has its roots in IPC mechanisms studied during the development of operating system architectures in the 1960s. These technologies were further developed and expanded during the 1970s to include different computers through local area networks, such as *ethernet.* [3] Distributed computing allows to overcome some fundamental limitations associated with the traditional (centralized) model of computing, such as:

- The ability to solve a given problem that requires more resources than those available on a single machine (e.g., storage, memory or processing limitations).
- Reducing the time needed to solve a given problem, which would otherwise be impractical with traditional solutions (e.g. distributed rendering in computer graphics).
- The use of several interconnected computers in order to provide redundancy and high availability of a given service (e.g. electronic aircraft control systems).
- The integration of several geographically dispersed services and systems (e.g. the Word Wide Web).

With the widespread use of the internet, distributed systems became part of everyday computing experience. Some popular websites such as Facebook or YouTube can only exist due to the presence of mechanisms that enable the distribution of load and storage demands between hundreds, or even thousands of machines that communicate with each other. The constant increase in software complexity also demands an increased level of integration between different service providers and/or middleware. Enterprises are witnessing an increased demand for collaboration and data sharing among different entities. Some enterprises use and outsource some of their services in order to simplify and increase profit to their business. In conclusion, the computing world nowadays is progressing towards ubiquity of services, through the concept of distributed computing.

2.1.1 Definition

The concept of a distributed system has broadened over the years. While there isn't a single, commonly accepted definition of what a distributed system is, there are some basic characteristics shared between them:

- A distributed system consists of multiple autonomous computational entities.
- All of these entities interact with each other through IPC mechanisms.
- These entities work together in order to achieve a common goal.

[4] also includes the following properties in the definition:

- **Heterogeneity:** A key characteristic of distributed systems is the heterogeneous nature of the entities involved. The heterogeneity may lie in the type of system or user, underlying policies and/or the data/resources that the underlying subsystems consume. The heterogeneity of distributed systems can be best observed in the Internet, where multitudes of systems, protocols, policies and environments interact to create a scalable infrastructure.
- **Concurrency:** Another important characteristic that distinguishes any distributed system from a centralized one is concurrency. Different components of distributed systems may run concurrently as the components may be loosely coupled. Therefore, there is a need to understand potential synchronization issues during the design of distributed systems.
- **Resource sharing:** Sharing of resources is another key characteristic of distributed systems.

2.1.2 Types of Distributed Systems

There are several major types of distributed systems. A distributed system may possess properties from more than one type of system.

Cluster computing

A cluster consists of several computers connected to each other, usually (but not always) over fast local area networks. Functionally, this is the closest distributed equivalent of a single computer. This type of systems have very little geographical dispersion, are more homogeneous and tend to work in closed contexts. They are usually deployed to improve performance and/or availability over that of a single computer, while being typically cheaper than a single computer of comparable speed or availability. There are several types of cluster computing:

- **High-availability clusters** - This type of clusters are usually employed to increase the availability of a given service. They operate by having redundant nodes, which can be used to provide service when one or more nodes fail.
- **Load-balancing clusters** - This type of clusters are used to distribute the workload through several machines on a network, making it viable to provide a service which otherwise would be impossible with a single machine due to lack of computational resources. To achieve this, multiple computers are linked together and virtually work as a single computer.
- **Compute clusters** - This kind of clusters are the closest functional equivalent to a single computer. They are typically employed to work with very heavy workloads (which would otherwise be impractical for a single machine), and tend to use internode communication very intensively (sometimes through specialized computer buses). Some uses for compute clusters include weather prediction, rendering farms and simulations.

Figure 2.1 shows an example of a distributed cluster system.[1]

Parallel computing

Since the 1950s, parallelism has been employed mainly in high performance computing. Interest in the area has grown lately due to the dominance of multicore processors in consumer level hardware, as a consequence of physical constraints preventing further frequency scaling.

As with a distributed system, there is no general consensus as to what defines a parallel system. [2] describes parallel computing as "*a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel").*" Parallel computing can be implemented at several different levels: bit, instruction, data, and task level.

There also isn't a clear distinction between parallel and distributed computing. Both types of systems might inherit characteristics from each other.

[1]`http://www.nas.nasa.gov/About/Projects/Columbia/columbia.html`

Figure 2.1: Example of a cluster computer: Supercomputer Columbia.

For instance, a given problem might be split into smaller parts and solved in parallel through the use of several computers connected by a network. Such system would fit in the definition of parallel and distributed system.

[20] distinguishes these two computing paradigms in the following way:

- In parallel computing, all processors have access to a shared memory. Shared memory can be used to exchange information between processors.
- In distributed computing, each processor has its own private memory (distributed memory). Information is exchanged by passing messages between the processors.

Figure 2.2 shows a graphic representation of this distinction.[2]

Grid computing

The term grid computing originated in the early 1990s as a metaphor for making computer power as easy to access to as an electrical power grid. [8] In simple terms, the Grid can be though of as a distributed system with non-interactive workload that involves a large number of files. In comparison to cluster computing, it can be seen as a more heterogenous, geographically dispersed and loosely coupled form of distributed computing.

The nodes which a Grid is built upon tend to be complete computers

[2]`http://en.wikipedia.org/wiki/File:Distributed-parallel.svg`

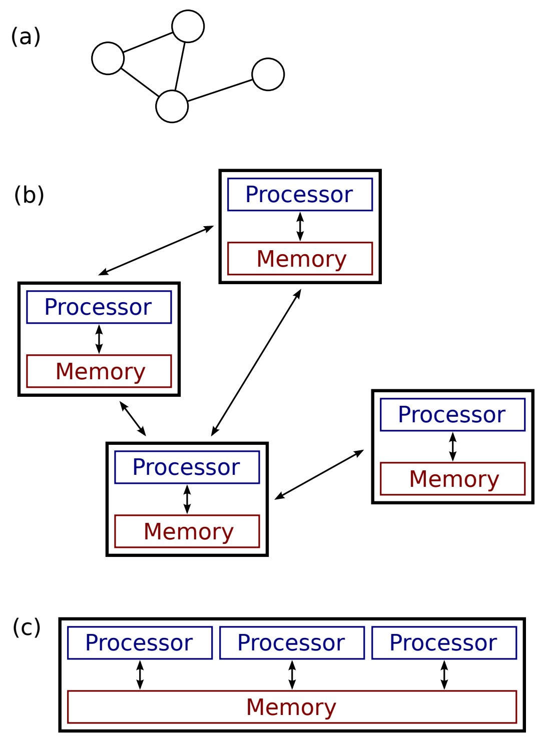


Figure 2.2: Distinction between parallel and distributed systems: (a)-(b) represents a distributed system; (c) represents a parallel system.

(with Central Processing Unit (CPU), storage, power supplies, network interfaces, etc) based on simple commodity hardware. Unlike some types of computer clusters, these computers are connected using conventional network interfaces (such as Ethernet interfaces). One of the main advantages from using this type of hardware is that it offers equivalent performance of a supercomputer at a fraction of the price. On the other hand, the relatively low speed of the network interfaces makes grid computing more appropriate for applications where multiple parallel computations can take place independently, without the need to communicate intermediate results between nodes. For the same reason, grid computing offers the highest degree of scalability between all types of distributed systems.

An example of a grid computing system is shown in Figure 2.3.[3] A more in-depth example is analyzed in Section 2.1.5.

There are some well-known public projects based on Grid computing, such as SETI@Home[4] or Folding@Home.[5] These exploit the concepts of volunteer computing and CPU scavenging, allowing anyone with a relatively

[3]`http://www.maxi-pedia.com/Grid+computing+distributed+computing`
[4]`http://folding.stanford.edu/`
[5]`http://setiathome.ssl.berkeley.edu/`

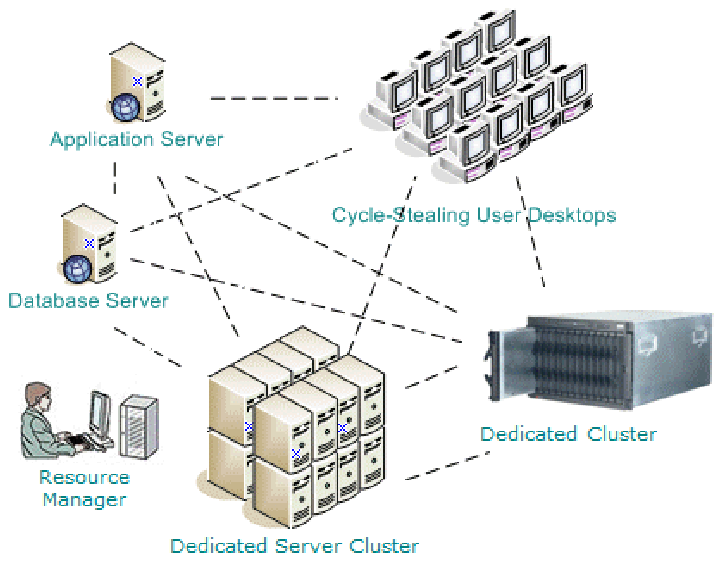


Figure 2.3: Architectural example of a grid computing system

modern computer and a working internet connection to join a virtual "supercomputer". By downloading a proprietary program, one can "donate" unused cycles from the idle time of their computers, which would otherwise be wasted. In theory, the combined processing power from all the nodes of a grid computing system can surpass even the fastest cluster supercomputers in existence.[6]

Cloud computing

Cloud computing can be seen as a web-based shift of the traditional client-server paradigm, as a consequence of the popularization of internet, the introduction of always-on broadband access and the improvement of web technologies (also coined by the term *Web 2.0*). It's a web-based type of distributed computing, whereby shared resources, information and software are provided by the cloud to computers and other devices (the clients) on demand. Figure 2.4 shows a conceptual diagram of cloud computing[7] and Figure 2.5 shows a detail of the stack that comprises a cloud service[8].

There are three types of cloud services that can be provided to the end-user:

- **Infrastructure as a service -** The service provider bears all the cost of servers, networking equipment, storage, and back-ups. The clients

[6]`http://www.museumstuff.com/learn/topics/GRID_computing::sub::Fastest_Virtual_Supercomputers`

[7]`http://www.incose.org/chesapek/mailings/2011/2011_05_Feature.html`

[8]`http://andromida.hubpages.com/hub/cloud-computing-architecture`

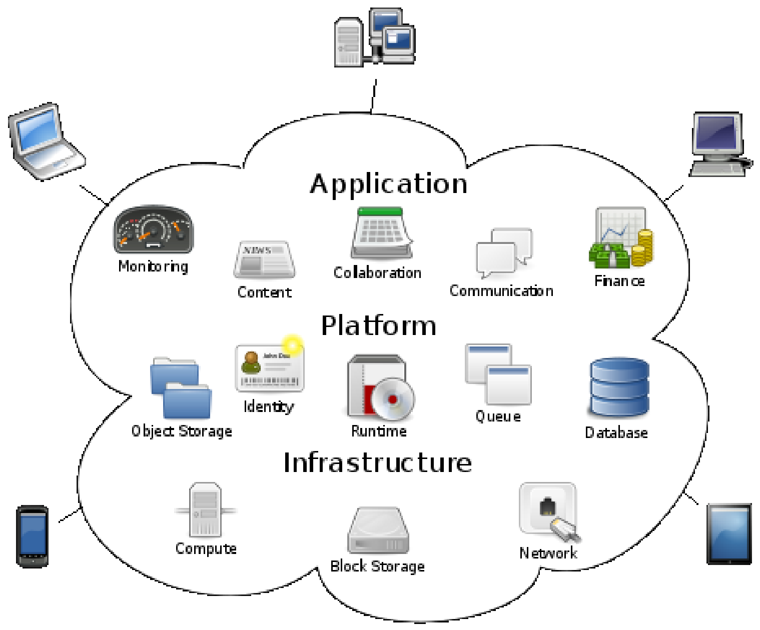


Figure 2.4: Cloud computing conceptual diagram

pay to use the computing service and they build their own application software on top of it. Example: Amazon's Elastic Compute Cloud (Amazon EC2).[9]

- **Platform as a service -** Here the service provider only provides the platform or a stack of solutions for its clients, thus helping them to save the investment on hardware and software. Example: Force.com.[10]

- **Software as a service -** The service provider offers its clients the use of their software, especially any type of application software. Example: Google Docs.[11]

2.1.3 Characteristics and Design Challenges

This section presents characteristics and the challenges that should be taken into account during the design of distributed systems. Due to the extensibility of this subject, only the relevant topics for this dissertation are

[9]`http://aws.amazon.com/ec2/`
[10]`http://www.salesforce.com/platform/`
[11]`http://docs.google.com/`

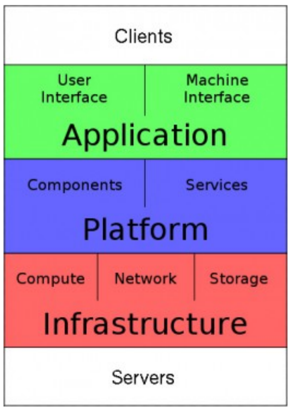


Figure 2.5: Cloud computing stack detail

addressed here.

Note that the word "process" is used here to refer to an instance of a computer program that is being executed on a machine. A process can communicate with others through local or remote IPC mechanisms. Section 2.2 further discusses the latter subject.

Security

Distributed systems introduce some additional security threats and vulnerabilities, in addition to the existing host and application-level threats inherited from local application design (such as viruses, spyware and other types of malicious software). [4] identifies the common security issues and technologies as:

- **Authentication** - Making sure that the individual/entity is indeed who he/she/it claims.
- **Authorization** - Providing different levels of access (e.g. deny and permit) to different parts of or operations in a computing system, dictated by the identity of the person or entity requesting the access.
- **Data integrity** - Making sure a piece of data arrives at the target destination without having been tampered with, during its transmission from one location to another.
- **Confidentiality** - Restricting access of information to authorized persons only, and preventing others from having access to that information.
- **Availability** - Ensuring that a piece of information is available to authorized users when they need it.

- **Trust** - Trust has always been one of the most significant influences on customer confidence in services, systems, products and brands. Generally an entity can be said to *trust* a second entity when the first entity makes the assumption that the second entity will behave exactly as the first one expects.

- **Privacy** - A broader issue than confidentiality, it is about the provision for any person, or any piece of data, to keep information about themselves from others, revealing selectively.

- **Identity management** - This is a process in which every person or resource is provided with unique identifying credentials, which are used to identify that entity uniquely.

Infrastructure-level threats and vulnerabilities such as Denial-of-Service (DoS) and Domain Name Service (DNS) attacks also need to be taken into account, as well as application-level threats particular to this type of systems (such as cross-site scripting or code injection).

Fault tolerance

Distributed systems tend to involve a large number of software, hardware, other physical components and sometimes even users. As such, it is to be expected a higher probability of failure at some point in the system by any of these entities. Issues such as buggy software, wear of hardware components, user error, unreliability of power supplies and data networks, all account for a higher probability of service failure. Because of this, it is imperative to introduce fault detection and recovery mechanisms on these systems. Fault detection mechanisms are justified due to the complexity of the systems and the increased difficulty in isolating errors, while both types of mechanisms are justified by the reduction of down time of the service in the event of a failure.

There are several different mechanisms that can be used to provide fault-tolerance to a system. Some of these include process resilience, reliable communication, distributed commits, checkpointing and recovery, agreement and consensus, failure detection and self-stabilization.

Communication

Appropriate IPC mechanisms must be used (or designed) for data communication in distributed systems. These is generally done over high-level APIs built over primitive IPC mechanisms and protocols, such as Transmission Control Protocol (TCP)/Internet Protocol (IP) sockets or message queues. Some examples include Remote Procedure Calls (RPCs) and Remote Object Invocation (ROI).

Network limitations

Besides being unreliable and insecure, computer networks introduce additional concerns to distributed systems. These include:

- **Limited bandwidth -** On a purely local system, the exchange of data between processes occurs through the local bus. On a distributed system, the use of an external computer network almost always translate to a much narrower bandwidth available. This can be particularly problematic for highly parallel/cluster systems, since they need to transmit an high amount of data between machines.

- **Latency -** The latency is constituted by the processing, transmission and propagation times involved with the transmission of a message. In a two-way communication, this introduces a performance hit when a distributed application awaits for data. Latency is particularly problematic with applications that exchanges data with high frequency (particularly in a request-reply scenario).

- **Jitter -** Variation of latency over time introduces problems related with predictability of performance (and synchronization mechanisms) for distributed applications. Just like latency, this is a problem that becomes more visible as the frequency of the exchanges of data is increased.

- **Heterogeneity -** A computer network almost always consists of a different combination of transmission mechanisms, protocols, networking devices (such as routers and switches) and topologies. Some of these entities can even change as the system is running (dynamic topology), thus making the distributed system less reliable and predictable performance-wise.

Naming

Devising easy to use and robust schemes for names, identifiers and addresses is essential to be able to locate resources and processes in a transparent and scalable manner. Naming in mobile systems introduces additional challenges because it cannot easily be tied to a static geographical topology.

Synchronization

Due to the parallel and unsynchronized nature of distributed systems, multiple forms of synchronization or coordination among the processes or machines are essential. Mutual exclusion, leader election, physical clock synchronization or global state recording algorithms are some of the mechanisms that take part in establishing the correct synchronization between resources and processes.

The implementation of synchronization mechanisms adds complexity to the system and introduces some potential problems, such as deadlocks and resource starvation. As a general rule, more processes on a system also means an increase in synchronization-related overhead.

Load balancing

The goal of load balancing is to achieve an higher throughput and reduce the user perceived latency. Load balancing may be necessary because of a variety of factors such as high network traffic, high request rate (causing the network connection to become a bottleneck) or an high computational load. A common situation where load balancing is used in are server farms, where the priority is to service incoming client requests with the shortest turnaround time.

Scalability and modularity

Scalability and modularity refers to a system's expansibility and its ability to handle growing amounts of work. To achieve this, the algorithms, resources (data) and services must be as distributed as possible. Various techniques such as replication, caching, and asynchronous processing help to achieve an high degree of scalability.

Consistency and replication

To avoid bottlenecks, fast access to data and scalable replication of data objects is highly desirable. This leads to issues of managing the replicas and dealing with consistency among the replicas/caches on a distributed setting. An example of this issue is deciding the level of granularity (i.e., size) of data access.

Debugging of distributed applications

Debugging distributed applications can be a very complex task due to concurrency and the uncertainty that comes from the large number of possible sequences of execution. Adequate debugging mechanisms and tools need to be used to meet this challenge.

2.1.4 Implementations

There are three level of implementations of distributed applications:

- **Operating system -** A cluster-unaware application can be converted to a distributed one through the use of a specialized operating system. This works by providing an abstraction layer from the cluster's

hardware, in which the application is only aware of being run in a multi-processor computer. The operating system's kernel is responsible to replicate and synchronize the application's memory space between computers. The main downside of this approach is that it usually involves a very high amount of network and computational overhead.
This solution is generally used only when there is no access to the source code of an application, or for some other reason it cannot be modified. An example of such operating system is Mosix[12] (for compute clusters).

- **Compiler/Linker -** This is a somewhat identical implementation to the one above, except that it requires re-linking to specialized libraries or a recompilation of the application through specialized, distributed-aware compilers and/or Software Development Kits (SDKs). In this case, the presence of a specialized, distributed-aware kernel is not required. [16] describes a possible approach to this problem.

- **Application/Source code -** This is the "native" method used to implement a distributed system. Here, the application is truly distributed-aware since it implies the integration from the source code (APIs are often used to achieve this). When properly designed, this is the most efficient type of implementation of a distributed system.

2.1.5 Related Work

Berkeley Open Infrastructure for Network Computing (BOINC)

The Berkeley Open Infrastructure for Network Computing[13] (BOINC) is an open-source middleware system for volunteer and grid computing. It was originally developed in 2002 as an improvement of the original SETI@home project, which, at the time, was suffering from several security breaches. Since then, the BOINC project has matured and nowadays is used to power several grid and volunteer computing projects in various science fields. A full list of projects and related statistics can be found in `http://boincstats.com`.

General architecture

The BOINC project is based on a client-server architecture. It provides three different software components:

- **Libraries** allowing the development of applications that will run on volunteers' computers;

[12]`http://www.mosix.org/`
[13]`http://boinc.berkeley.edu/`

- **Client software** that allows volunteers to join projects, download work units, run them and upload the results;
- **Server software** that allows the distribution of work to clients and the collection the results.

There are three types of entities defined in the BOINC's system: the BOINC system core, the computing projects, and the participant volunteers. Figure 2.6 shows BOINC's architecture and how these entities relate to each other.[14]

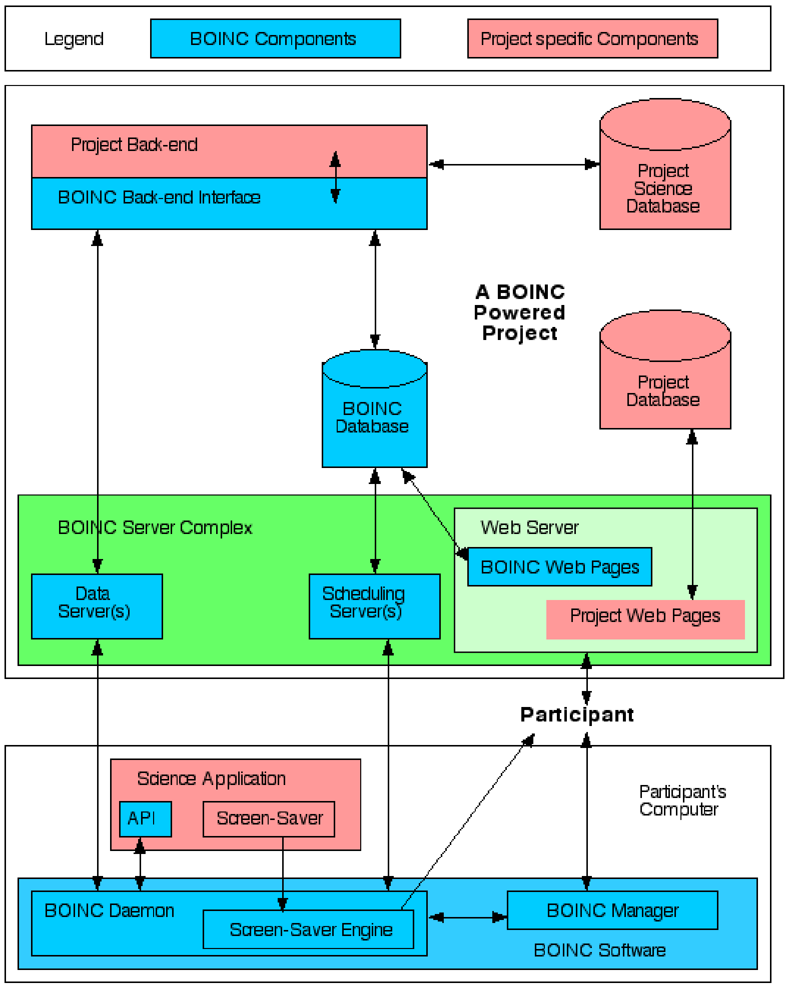


Figure 2.6: BOINC's system architecture

All computing projects being executed in BOINC are connected to a central server. Each of these projects has its own database, application and

[14]`http://www.boinc-wiki.info/w/images/d/dc/Boinc-project-interconnect.png`

project back-end. When a volunteer installs the BOINC software, s/he is given the option to choose the projects s/he may want to participate in. The server-client exchange works roughly in the following way:

1. The volunteer chooses one or more projects to participate in;
2. The client software downloads project software;
3. The client software downloads work units (a portion of data that a project needs to be analyzed);
4. The client software executes work units while the computer is idle;
5. The client software uploads results to the server as work units are completed;
6. Steps 3-5 are repeated as long as there are works units available for processing and the user does not opt out of the experiment.

Server architecture

The BOINC system uses databases at two different levels of the system. On the first level there is the BOINC database. It contains information about the participants, the participants' computers, teams, results, work units and more. The second level database is used specifically by a project and stores the results from the concluded work units after they've been returned by the participant and have passed validation.

A BOINC-powered project includes a set of server-side daemons, namely:

- **Assimilator** - Handles the work units that have been completed. If the work unit's results are deemed credible, then the information is recorded to the project's database.
- **Database purge** - Removes work-related database entries when they are no longer needed.
- **File deleter** - Deletes input and output files when they are no longer needed in the project's data servers.
- **Transitioner** - Handles the state transitions of the work units and results. It generates initial results for work units and generates more results when timeouts or errors occur.
- **Validator** - Compares redundant results and selects a canonical result representing the correct output.
- **Work generator** - Creates work units for the project's client application.

The BOINC project uses a credit system with the goal of keeping volunteers motivated to contribute to the project. Credits are awarded every time a work unit is completed by the client. Statistics are then sent to the main BOINC server and displayed on the publicly accessible BOINC Stats webpage.[15] For several reasons, these statistics may contain errors and some users have even attempted to abuse the credit system by returning bogus results. For the projects where result validation is computationally inexpensive, simply verifying them before attributing credit is enough to avoid this issue. Other projects implement a redundancy system, where two or more computers execute the same work unit. Credit is only given if the results match with each other.

Client architecture

Figure 2.7 shows BOINC's architecture from the client side.[16] The client package is made up of several software applications:

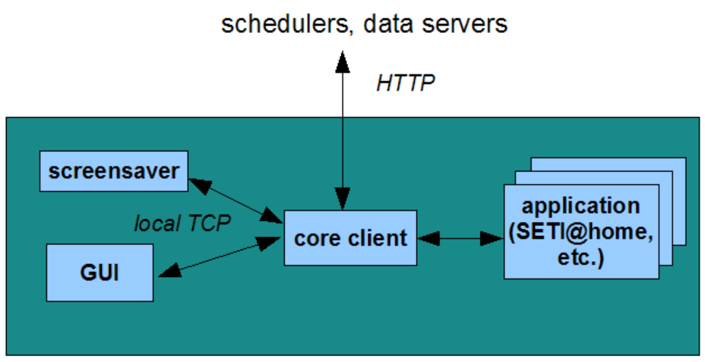


Figure 2.7: BOINC's client architecture

- **The application** - This is the program that processes the work units for a BOINC project. Applications are attached to an instance of `boinc-client`.

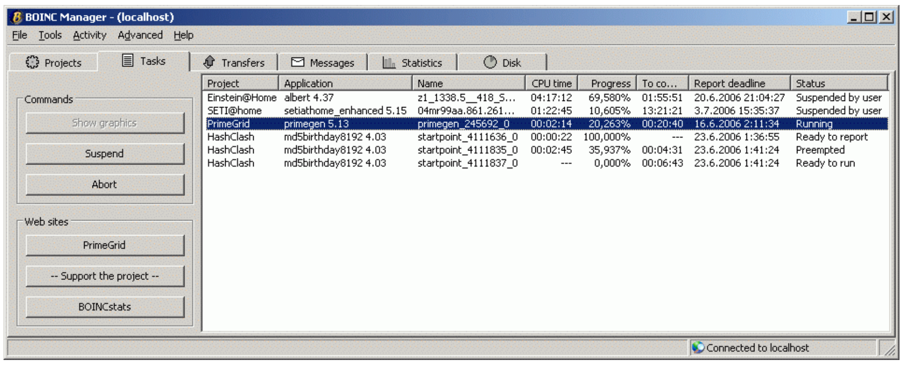
- `boinc-client` - Runs as a demon on the computer that processes the work units, providing the communication and processing platform that applications require to run. `boinc-client` interprets the user settings and runs the application appropriately.

- `boinc-manager` - A Graphical User Interface (GUI) program that is used to manage an instance of `boinc-client`. It allows the control

[15]`http://boincstats.com`
[16]`http://boinc.berkeley.edu/w/images/a/ae/Client.png`

of local or remote instances of `boinc-client`. It communicates with `boinc-client` through a local TCP/IP socket. `boinc-manager` is not necessarily required when an account manger is used. Figure 2.8 shows a screenshot of this application.[17]

- **Account manager** - A program or service (often web based) used to manage multiple instances of `boinc-client` at once. Although not strictly required, it can be useful when a user has many computers set up to run BOINC simultaneously.
- **Screensaver (optional)** - A screensaver can be provided with the application for aesthetic reasons. It communicates with `boinc-client` through a local TCP/IP socket.



Project	Application	Name	CPU time	Progress	To co...	Report deadline	Status
Einstein@Home	albert 4.37	z1_1338.5__418_S...	04:17:12	69,580%	01:55:51	20.6.2006 21:04:27	Suspended by user
SETI@home	setiathome_enhanced 5.15	04mr99aa.861.261...	01:22:45	10,605%	13:21:21	3.7.2006 15:35:37	Suspended by user
PrimeGrid	primegen 5.13	primegen_245692_0	00:02:14	20,263%	00:20:40	16.6.2006 2:11:34	Running
HashClash	md5birthday8192 4.03	startpoint_4111636_0	00:00:22	100,000%	---	23.6.2006 1:36:55	Ready to report
HashClash	md5birthday8192 4.03	startpoint_4111835_0	00:02:45	35,937%	00:04:31	23.6.2006 1:41:24	Preempted
HashClash	md5birthday8192 4.03	startpoint_4111837_0	---	0,000%	00:06:43	23.6.2006 1:41:24	Ready to run

Figure 2.8: Screenshot of `boinc-manager`

The application has to be specifically coded using the BOINC's libraries. Take the following "Hello World" application as an example:

```
#include <stdio.h>

int main()
}
    FILE *f;
    int j, num;

    f = fopen("out.txt", "a");
    fprintf(f, "Hello, BOINC World!\n");
    fprintf(f, "Starting some computation...\n");
    for(j = 0; j < 1234567890; j++)
      num = rand()+rand();
    fprintf(f, "Computation completed!\n");

    fclose(f);
    return 0;
}
```

[17] `http://upload.wikimedia.org/wikipedia/commons/f/f1/BOINC_screenshot.png`

The BOINC equivalent to this application would be:[18]

```
#include <stdio.h>
#include "boinc_api.h"

int main()
}
    FILE *f;
    char resolved_name[512];
    int j, num;

    boinc_init();

    boinc_resolve_filename("out.txt", resolved_name,
        512);

    f = boinc_fopen(resolved_name, "a");
    fprintf(f, "Hello, BOINC World!\n");

    fprintf(f, "Starting some computation...\n");
    for(j = 0; j < 1234567890; j++)
      num = rand()+rand();
    fprintf(f, "Computation completed!\n");

    fclose(f);
    boinc_finish(0);
    return 1;
}
```

Nearly all BOINC's applications implement CPU scavenging, where they run as a very low priority process so as to take advantage of unused resources. Quite often, they also implement some sort of visualization of the work unit's progress (usually as a 3D screensaver) as a way to be more enticing to the user. Figure 2.9 shows a screenshot of folding@home's screensaver visualization.[19]

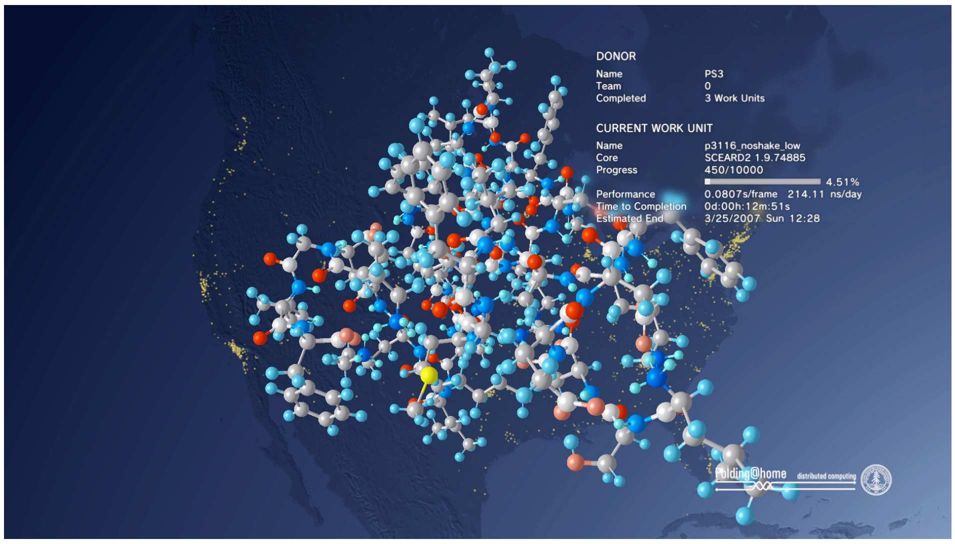
Globus Toolkit

The Globus Toolkit[20] is an open source software used to build computing grids. It allows the sharing of computing power, databases and other tools securely across a network. The toolkit includes software for security, information infrastructure, resource management, data management, communication, fault detection and portability. It's packaged as a set of components that can be used either independently or together to develop distributed applications.

[18] `http://www.kuliniewicz.org/boinc/html/img21.html`

[19] `http://research.scea.com/2006-09-folding@home/folding@home_files/shot-00008.jpg`

[20] `http://www.globus.org/toolkit/`



```
DONOR
Name                PS3
Team                0
Completed           3 Work Units

CURRENT WORK UNIT
Name                p3116_noshake_low
Core                SCEARD2 1.9.74885
Progress            450/10000
                                        4.51%
Performance         0.0807s/frame  214.11 ns/day
Time to Completion  0d:00h:12m:51s
Estimated End       3/25/2007 Sun 12:28
```

Figure 2.9: Fold@home screensaver visualization

General architecture

[7] defines the Globus architecture as being built on three pyramids. As shown in Figure 2.10 (from [7]), these are:

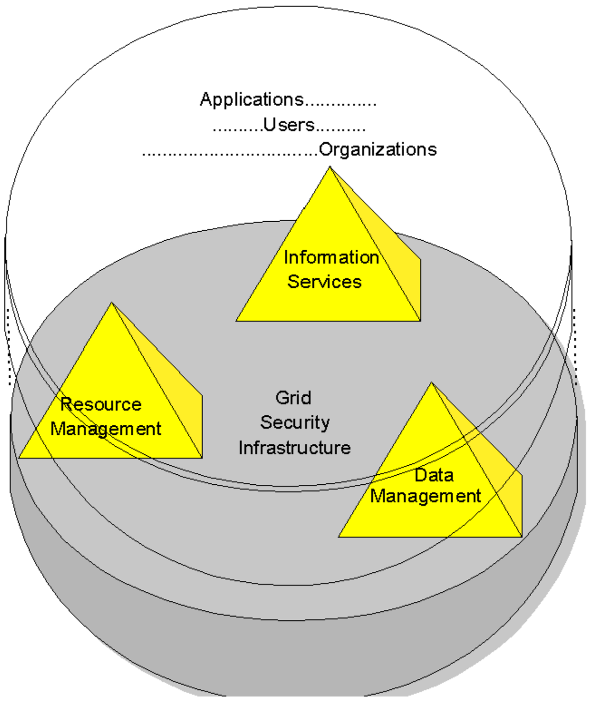

Applications..............
..........Users..........
.................................Organizations
Information Services
Resource Management
Grid Security Infrastructure
Data Management

Figure 2.10: Globus Toolkit's three pyramids

- **Resource management** - Provides support for:
  - Resource allocation;
  - Submission of jobs (remotely running executables and receiving results);
  - Managing job status and progress.
- **Data management** - Provides support and management for file transfers among machines in the grid.
- **Information services** - Based on the Lightweight Directory Access Protocol (LDAP), it provides support to collect and query information on the grid.

The three pyramids are built on top of the underlying Grid Security Infrastructure (GSI), which is responsible to provide security functions, including single/mutual authentication, confidential communication, authorization, and delegation. Figure 2.11 (from [7]) shows a detailed view of all the main components of the Globus Toolkit. These are:

- **GRAM/GRASS** - The primary components of the resource management pyramid are the Grid Resource Allocation Manager (GRAM) and the Global Access to Secondary Storage (GASS);
- **MDS (GRIS/GIIS)** - Based on LDAP, the Grid Resource Information Service (GRIS) and Grid Index Information Service (GIIS) components can be configured hierarchically to collect the information and distribute it. These two services are called the Monitoring and Discovery Service (MDS). The information collected can be static information about the machines as well as dynamic information showing current CPU or disk activity.
- **GridFTP** - The key component for secure and high-performance data transfer. The Globus Replica Catalog and Management is used to register and manage both complete and partial copies of data sets.
- **GSI** (not shown) - All of the above components are built on top of the underlying GSI.

Jobs

A job can be defined as a program that a user wishes to execute on a known remote machine. A job has to be validated before it can be executed, and any additional resources needed for the job noted by the remote machine. Therefore a job is sent with a job request which can specify a number of things, such as:

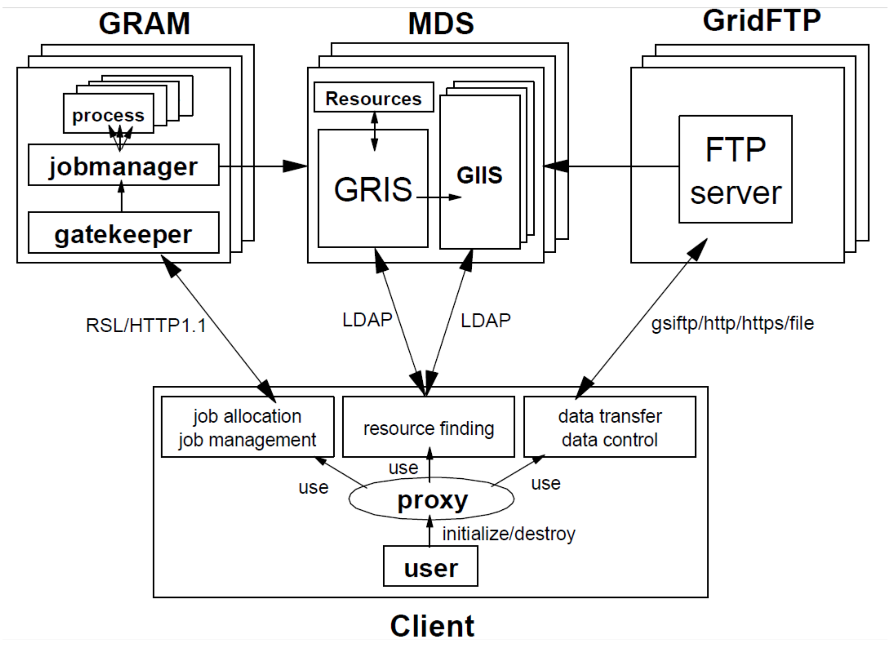


Figure 2.11: Globus Toolkit system components

- Name of program(s) to submit;
- Machine(s) to submit to;
- Method of result retrieval;
- Access to files required;
- Maximum execution time;
- Minimum/Maximum memory.

Job requests are sent to and retrieved from GRAM (through the gatekeeper). Globus supports a variety of ways of retrieving results once a job has completed. The default being that results should be sent back to the screen of the user that sent the request. However a number of alternatives are available, for instance:

- Send to screen of the local machine;
- Send to file of the local machine;
- Store in a file at a remote `ftp`/`http` server;
- Wait until retrieve command is given from local machine;
- Don't do anything with the results.

2.2 Inter-Process Communication Mechanisms

IPC mechanisms allow the transmission of data between computer processes, either located in the same machine or on different ones. On modern operating systems, each process has its own memory space, which is isolated from every other process on the system. Because of this, they rely on IPC mechanisms to communicate between each other. High-performance, low overhead IPC mechanisms are also an essential part of microkernels.

Local IPC mechanisms are used to transmit data and other information (such as concurrency handling primitives) between two processes located in the same machine. The exact IPC mechanisms that are available and their implementation are dependent on the OS.

Remote IPC mechanisms are used to transmit data between processes located on different machines and/or operating systems. They generally operate over a specialized type of hardware interface and are tightly coupled to the protocols used in their respective network types. Some remote mechanisms can also be used for local communication.

This section describes the IPC mechanisms typically available for Unix-based systems. A performance evaluation of each of these mechanisms is presented in Section 3.4.

2.2.1 IPC Mechanisms on Linux Systems

In this section, the IPC methods supported by the Linux kernel are discussed and implementation examples in C programming language are shown. The mechanisms analyzed here are: file, pipe, named (FIFO) pipe, network and unix domain sockets, message queues, shared memory, memory map, semaphores, signals and message passing mechanisms. Most of these are available for other UNIX-based operating-systems.

File

Although not purposefully developed as an IPC mechanism, files in the filesystem can provide a simple way of exchanging data between applications. This method possesses some limitations, most notably the very low speed (when compared to the other mechanisms) and concurrency issues (since it generally works through *polling*). The data included is also limited to the available disk space (or memory, for virtual disks).

A file is a stream-based resource. Programmatically, the C functions for file handling are included in the `stdio.h` header. These include:

Opening a file:

```c
FILE *fp;
fp = fopen("file.txt", "rw");
```

Writing to a file:

```c
char* msg_w = "Hello world!";
fwrite(msg_w, sizeof(char), strlen(msg_w), fd);
```

Reading from a file:

```
const int n = 14;
char msg_r[n];
fread(msg_r, sizeof(char), n, fd);
```

Closing the file:

```
fclose(fp);
```

Pipe

A software pipeline enables two or more processes to connect to each other through their standard streams, so that the output of a process (*stdout* or *stderr*) feeds directly as an input (*stdin*) to the other. Since there is no ambiguity between the pipes accessible to an application, they need not to be identified. For this reason, they are also named *anonymous pipes*.

Pipes can be created by the user as the programs are launched through a Command-Line Interface (CLI), and they exist only as long as the respective processes are running. A pipe in *bash* can be declared in the following way:

```
diogo@ubuntu:~$ cat textfile.txt | grep "Hello world!"
```

In this example, `cat` dumps the contents of the file named `textfile.txt` to the *stdout* and redirects this output to the *stdin* of another program (`grep`), which in turn displays (i.e., outputs to *stdout*) only the lines that contains the string "Hello world".

Programatically, a software pipe can be created through a *fork*. In other words, pipes can only be established between parent-child processes or processes that share a common ancestry. As such, it's not possible to establish a pipe between two arbitrary running process. The Portable Operating System Interface for Unix (POSIX) implementation of a pipe defines it as an *half-duplex* communication channel, meaning that two pipes need to be defined in order to establish bidirectional communication.

The C functions used to implement a pipe are located in `sys/types.h` and `unistd.h`. They are used in the following manner:

Forking a process:

```
pid_t pid;
pid = fork();
if(pid == 0) {
    /* Code for child process */
} else {
    /* Code for parent process */
}
```

Establishing the pipe:

```
/* One file descriptor for each stream:
   fd[0] : Input
   fd[1] : Output */
int fd[2];
pipe(fd);
```

Writing to the pipe:

```
char* msg_w = "Hello world!";
write(fd[1], msg_w, strlen(msg_w));
```

Reading from the pipe:

```
const int n = 14;
char msg_r[n];
read(fd[1], msg_r, strlen(msg_r));
```

Closing the pipe:

```
close(fd[0]); /* Input */
close(fd[1]); /* Output */
```

Named pipe

A named pipe, also known as First In, First Out (FIFO) pipe, can be seen as an extension of the anonymous pipe. Unlike the latter, named pipes allow unrelated processes to communicate with each other.

A named pipe is system-persistent and is identified in the filesystem as a (virtual) file. They exist beyond the life of the attached processes and as such they must be deleted once they are no longer being used. As with any other file, its permissions can be changed. Unlike anonymous pipes, the standard streams are not redirected, leaving them available to be used for other purposes.

Named pipes can be created and used in a command line interface, as the following example (in bash) demonstrates:

```
diogo@ubuntu:~$ mkfifo my_pipe
diogo@ubuntu:~$ gzip -9 -c < my_pipe > out.gz
diogo@ubuntu:~$ cat file_a > my_pipe
diogo@ubuntu:~$ cat file_b > my_pipe
diogo@ubuntu:~$ rm my_pipe
```

In the first line, a named pipe is created. Then `gzip` is attached to the pipe in order to compress the data that arrives through the `stdin`. The compressed data is sent to `stdout`, and then piped into a file called `out.gz`. In lines 3 and 4, two different files are sent through the pipe. Finally, the pipe is closed and removed.

An application can have complete control over named pipes, and running processes can freely connect to existing pipes (and thus to other running processes). Each pipe is a byte-oriented, half-duplex transmission medium. Programmatically, the functions used to handle named pipes are present in `unistd.h`, `sys/types.h` and `sys/stat.h` header files. They are used in the following way:

Open read (and write) pipe:

```
/* Read pipe */
const char rpath[] = "/tmp/rpipe";
int rfd;
rfd = open(rpath, O_RDONLY);
if(rfd <= 0) {
    /* Pipe doesn't exist. Declare it: */
    mkfifo(rpath, S_IRWXU | S_IRWXG | S_IRWXO);
}

/* Write pipe */
const char wpath[] = "/tmp/wpipe";
int wfd;
wfd = open(wpath, O_WRONLY);
if(wfd <= 0) {
    /* Pipe doesn't exist. Declare it: */
    mkfifo(wpath, S_IRWXU | S_IRWXG | S_IRWXO);
}
```

Write to the pipe:

```
char* msg_w = "Hello world!";
write(wfd, msg_w, strlen(msg_w));
```

Read from the pipe:

```
const int n = 14;
char msg_r[n];
read(rdf, msg_r, strlen(msg_r));
```

Close the pipes and remove (unlink) them from the filesystem:

```
close(rfd);
close(wfd);
unlink(rpath);
unlink(wpath);
```

Network domain socket

An Internet socket or network socket is an endpoint of a bidirectional IPC flow across an IP-based computer network, such as the Internet. Local IPC communication is also possible, since most systems allow the communication between processes on the same OS through a *loopback* address from the network stack (such as ::1 on IPv6 networks).

Since network sockets were not purposely built for local transmission of data, there is usually a performance penalty when used as such. Even within the same OS, the data transmitted between processes has to pass through the same stack of protocols as the ones used in remote communications, including unnecessary data (such as error correction, multiplexing, routing data, etc). The advantage of using this method for local IPC comes from the ease of implementation, since the same code can be shared between local and remote processes.

A socket is a stream-based, full-duplex communication mechanism and it uses a client-server model to establish connections between the processes. Several protocols from layers 2 and 3 of the Open Systems Interconnection (OSI) network layer model are supported by the Linux (Berkley) sockets implementation.

Programmatically, socket functions are present in the `sys/socket.h` header. The following example shows how to establish and transmit data through a TCP/IP socket:

Open the socket (client side):

```
int cl_sockfd;
struct sockaddr_in cl_adr;
struct hostent* cl_host;

/* Prepare the client's socket address */
cl_adr.sin_family = AF_INET;
cl_adr.sin_port = htons(1234);
cl_host = gethostbyname("192.168.0.2");
bcopy((char*)cl_host->h_addr, (char*)&cl_adr.sin_addr.
    s_addr, cl_host->h_length);

/* Open socket and connect to server */
cl_sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
connect(cl_sockfd, (const struct sockaddr*)&cl_adr,
    sizeof(cl_adr));
```

Open the socket (server side):

```
int cl_sockfd;
int srv_sockfd;
struct sockaddr_in srv_adr;

/* Configure the socket's adress */
srv_adr.sin_family = AF_INET;
srv_adr.sin_port = htons(1234);
srv_adr.sin_addr.s_addr = INADDR_ANY;

/* Create a socket and configure it to listen for
    clients on any local address */
srv_sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)
    ;
```

```
bind(srv_sockfd, (struct sockaddr*)&srv_adr, sizeof(
    srv_adr));
listen(srv_sockfd, 1);

/* Wait for clients to connect: */
while(connected) {
    cl_sockfd = accept(cl_sockfd, NULL, 0);
    /* Do something with the new client here... */
}
```

Read from the socket:

```
const int n = 14;
char msg_r[n];
recv(cl_sockfd, msg_r, strlen(msg_r), 0);
```

Write to the socket:

```
char* msg_w = "Hello world!";
send(cl_sockfd, msg_w, strlen(msg_w), 0);
```

Close the socket (client side):

```
shutdown(cl_sockfd, SHUT_RDWR);
close(cl_sockfd);
```

Close the socket (server side):

```
shutdown(cl_sockfd, SHUT_RDWR);
close(cl_sockfd);
close(srv_sockfd);
```

Unix domain socket

As the name suggests, a Unix domain socket (or IPC socket) is a traditional socket applied to the operating system's local domain. Functionally, it is somewhat similar to a named pipe, but with full-duplex communication and the option to choose between stream or datagram transmission modes. The API is similar to that of a network socket, but functionally a network protocol is not used for communication. This way, the typical overhead associated with network communication is avoided, including processing time associated to network-specific applications that might be running on the system (such as packet sniffers and firewalls).

As with named pipes, Unix domain socket use the file system as name space, and the communication occurs entirely within the operating system's kernel.

Programmatically, the functions used to manage and transmit data through unix-domain sockets are the same as the ones used in network domain sockets. These are located in `unistd.h`, `sys/socket.h` and `sys/un.h` headers. It's main functions can be used as the following example demonstrates:

Open the socket (client side):

```
int cl_sockfd;
struct sockaddr_un cl_adr;

/* Prepare the client's socket address */
strcpy(cl_adr.sun_path, "/tmp/sock_test");
cl_adr.sun_family = AF_UNIX;

/* Open socket and connect to server */
cl_sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
connect(cl_sockfd, (const struct sockaddr*)&cl_adr,
    sizeof(cl_adr));
```

Open the socket (server side):

```
int len;
int cl_sockfd;
int srv_sockfd;
struct sockaddr_un srv_adr;

/* Configure the socket's adress */
strcpy(srv_adr.sun_path, "/tmp/sock_test");
srv_adr.sun_family = AF_UNIX;

/* Unlink the virtual file if it exists (optional): */
unlink(srv_adr.sun_path);

/* Create a socket and configure it to listen for
    client processes */
srv_sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
len = strlen(local.sun_path) + sizeof(local.sun_family
    );
bind(srv_sockfd, (struct sockaddr*)&srv_adr, len);
listen(srv_sockfd, 1);

/* Wait for client processes to connect: */
while(connected) {
    cl_sockfd = accept(cl_sockfd, NULL, 0);
    /* Do something with the new client process here
        ... */
}
```

Read from the socket:

```
const int n = 14;
char msg_r[n];
recv(cl_sockfd, msg_r, strlen(msg_r), 0);
```

Write to the socket:

```
char* msg_w = "Hello world!";
send(cl_sockfd, msg_w, strlen(msg_w), 0);
```

Close the socket (client side):

```
shutdown(cl_sockfd, SHUT_RDWR);
close(cl_sockfd);
```

Close the socket (server side):

```
shutdown(cl_sockfd, SHUT_RDWR);
close(cl_sockfd);
close(srv_sockfd);
unlink(srv_adr.sun_path);
```

In addition to data, processes may also send file descriptors across a Unix domain socket connection using the `sendmsg()` and `recvmsg()` system calls. The function `socketpair()` can be used to create a pair of connected sockets between related processes.

Message queue

A message queue is an asynchronous IPC mechanism that allows the transmission of data structures known as messages. A process creates the message which is then placed on the incoming message queue of another process. Once ready, the receiving process retrieves it using the appropriate system call.

Message queues are managed by the operating system's kernel. Each message queue has a persistent identifier, meaning that they can exist beyond the lifetime of the process. They must be explicitly removed (unlinked) when no longer needed.

It's possible to know the message queues being currently used in a Linux system. For the System V message queue implementation, the command `ipcs` can be used. For the POSIX implementation, the queue file system must be mounted first, using the following commands:

```
root@ubuntu:~$ mkdir /dev/mqueue
root@ubuntu:~$ mount -t mqueue none /dev/mqueue
```

All POSIX message queues (if present) are shown in this directory as virtual files.

Programmatically, there are two different implementations of message queues available for Linux: System V IPC messages and POSIX message queue (supported in Linux since kernel 2.6.6). The POSIX implementation uses the `mqueue.h` header. Its main functions include:

Creating the message queue:

```
int fd;

/* Prepare atributes of the message queue */
struct mq_attr attr;
attr.mq_maxmsg = 20;
attr.mq_msgsize = 14;
attr.mq_flags = 0;

/* Open queue */
fd = mq_open("my_queue", O_CREAT|O_RDWR, PMODE; &attr)
    ;
```

Sending data into the queue:

```
char* msg_w = "Hello world!";
int priority = 1;
mq_send(fd, msg_w, strlen(msg_w), priority);
```

Reading data from the queue:

```
const int n = 14;
char msg_r[n];
mq_receive(fd, msg_r, (size_t)n, 0);
```

Closing and removing the message queue:

```
mq_close(md);
mq_unlink("my_queue");
```

Message queues are the preferred IPC mechanism for real time operating systems, due to the tight integration between message passing and CPU scheduling. All implementations include a message priority field, allowing a given higher-priority message to take precedence to all lower-priority ones in a process's message queue.

Memory mapping/Shared memory

Shared memory is an IPC mechanism that uses a large block of Random-Access Memory (RAM) memory allocated outside the area that is reserved for running processes. This memory area can then be accessed by multiple running processes, whether they are related or not. This particular method of IPC is commonly used as a way to conserve memory, since the use of a shared memory space for two or more applications avoids the need of duplicate information. This mechanism is often used for shared libraries.

Shared memory segments are managed by the kernel. Once the memory is mapped into the address space of the processes that are sharing the memory region, no kernel involvement occurs in passing data between the processes (unlike most IPC mechanisms). Each segment is persistent and has its own identifier, meaning that it can exist beyond the process's lifetime. A segment must be explicitly removed (unlinked).

Shared memory poses some particular problems not present in other IPC mechanisms. These issues originate from the fact that CPUs need fast access to memory and will likely cache it, which has two implications:

- CPU cache coherence: whenever one cache is updated with information that may be used by other processors, the change needs to be reflected on the other processors as well, otherwise they will be working with incoherent data. Some efficient mechanisms can be employed to resolve this, but they can occasionally be overloaded and become a bottleneck.
- CPU-to-memory concurrency becomes a bottleneck: when working with shared memory, scalability is generally poor.

As with message passing, it's possible to know the shared memory segments being currently used on a Linux system through the command `ipcs`.

Memory mapping is the process of assigning a direct byte-for-byte correlation of a file or a file-like resource into memory. This method can be applied to a shared memory object as well, allowing the programmer to access the shared memory object as an ordinary memory segment in the application's memory space.

Programmatically, shared memory does not contain any method to handle concurrency, so the programmer must use another IPC mechanism (such as semaphores) for this purpose. After being declared, the shared memory segment is memory-mapped and can be accessed as an ordinary segment of memory. The POSIX implementation of this mechanism uses the `sys/mman.h` header and its main functions are used as the following example shows:

Create the shared memory object:

```c
int shm_fd;

/* Remove existing shared memory object, if any */
shm_unlink("shm_test");

/* Create shared memory object */
shm_fd = shm_open("shm_test", O_CREAT|O_RDWR, 0666);
```

Establish the memory mapping:

```c
char* map_mem;
int pagesize = sysconf(_SC_PAGESIZE);

/* Set shared memory segment size */
ftruncate(shm_fd, pagesize * 2);

/* Establish the mapping */
map_mem = (char*)mmap(NULL, pagesize * 2, PROT_READ |
    PROT_WRITE, MAP_SHARED, shm_fd, (off_t)0);
```

Any change on the `map_mem` array is reflected on the shared memory object. So after writing on the array...

```
map_m = "Hello world!";
```

...the change would be instantly reflected on all the other processes:

```
printf(map_m);

/* Outputs "Hello world!" */
```

Removing the shared memory object and mapping:

```
munmap(map_mem, pagesize * 2);
shm_unlink("shm_test");
```

Memory-mapped file

A memory-mapped file is a type of memory mapping (described above), but a file is used instead of a shared memory object. When opened, the file is copied into virtual memory and any process can access it. Once the file is closed, all modifications are copied back from the memory into the file.

Signals

A signal is a simple asynchronous notification sent to a process to notify it of an event that occurred. When a signal is sent to a process, the operating system interrupts the process's normal flow of execution. If the process has previously registered a signal handler, that routine is executed. Otherwise the default signal handler is executed instead.

Signals were not designed to transmit data, and as such it is a very limited form of IPC. Some of its common uses include notifying the process when a QUIT, INTERRUPT or KILL requests occurs (for instance). In addition, signal handling is an asynchronous process, so race conditions must be taken into account.

Programmatically, linux signals implementation leaves two of all possible signal values as *user defined*.[21] As the name suggests, these can be used to signal custom events. Signals can be implemented using the `signal.h` header, as the following example demonstrates:

```
/* Declare signal handler function */
void sig_hdlr(int signal) {
  if(signal == SIGINT) {
    /* Do something... */
  }
}
```

[21] `http://linux.die.net/man/7/signal`

```

int main(int argc, char* argv[]) {
    ...
    /* Register signal handler for interruption */
    (void)signal(SIGINT, sig_hdlr);
    ...
}
```

Semaphores

An IPC semaphore can be seen as a protected variable that provides access control between multiple processes and/or threads. In the context of IPC, they are generally used to coordinate access to other local, thread-unsafe IPC mechanisms (such as shared memory).

Semaphores have their own ID and are system-persistent, so they live beyond the process's lifetime and must be explicitly declared and removed. It's possible to know the semaphores present on a system through the `ipcs` command.

Programmatically, there are two different implementations of IPC semaphores available for Linux: System V Semaphores and POSIX Semaphores. The latter uses `semaphore.h` and its main functions are:

Opening and initializing a semaphore:

```
sem_t *sem;

/* Remove existing semaphore first */
sem_unlink("sem_test");

/* Create and initialize a binary semaphore */
sem = sem_open("sem_test", O_CREAT);
sem_init(sem, true, 1);
```

Locking a semaphore:

```
sem_wait(sem);
```

Unlocking a semaphore:

```
sem_post(sem);
```

Closing and removing (unlinking) a semaphore:

```
sem_close(sem);
sem_unlink("sem_test");
```

Message Passing

A message passing mechanism refers to several IPC APIs that are tightly coupled to the operating system or programming language. These APIs work

IPC Mechanism	Scope	Data type	Resource ID
File	Local & remote	Byte stream	File
Pipe	Local	Byte stream	N/A
Named pipe	Local	Byte stream	File
Network socket	Local & remote	Byte stream/message	File desc.
Unix Socket	Local	Byte stream/message	File
Message Queue	Local	Message	File desc.
Mmap/Shared memory	Local	Memory segment	File desc.
Memory-mapped file	Local	Memory segment	File
Signals	Local	Event	Process ID
Semaphores	Local	Sync primitives	File

Table 2.1: Feature comparison of IPC mechanisms in Linux (1 of 2)

by sending messages to one or more recipients through function calls, signals or data packets. Message passing refers to higher-level mechanisms, in the sense that they are built over the other IPC technologies described in this section. For this reason, they are also inherently slower and they won't be discussed with the same detail as the other mechanisms.

Probably the most commonly used method for message passing is the RPC. It allows a program to execute a subroutine or procedure on another address space without the programmer explicitly coding the details for this remote interaction. Some existing implementations include:

- **D-Bus (Desktop Bus) -** A simple IPC system that allows several processes to connect to each other.

- **Java Remote Method Invocation (RMI) -** A subset of the Java SDK that adds support for RPC.

- **Common Object Request Broker Architecture (CORBA) -** A standard defined by the Object Management Group (OMG) that enables software written in different languages and/or for different operating systems to communicate with each other.

- **Simple Object Access Protocol (SOAP) -** A protocol specification for structured information exchange in web services, based on Extensible Markup Language (XML).

Many other APIs, standards and protocols are available.

2.2.2 Summary

Tables 2.1 and 2.2 shows a feature comparison between IPC mechanisms in Linux.

IPC Mechanism	Thread safe	Process relationship
File	N/A	None
Pipe	Yes	Parent-child
Named pipe	Yes	None
Network socket	Yes	Server-client
Unix Socket	Yes	Server-client
Message Queue	Yes	None
Mmap/Shared memory	No	None
Memory-mapped file	No	None
Signals	N/A	None
Semaphores	N/A	None

Table 2.2: Feature comparison of IPC mechanisms in Linux (2 of 2)

2.3 Network Simulation

As the name suggests, Network Simulators are pieces of software designed to simulate data networks. They allow to model an hypothetical network topology in order to test new or existing protocols, topologies and several other aspects related to computer networks. With the explosion of data networking, these simulators are becoming an integral part of the rapid and stable development of computer networks.

Research about simulation of packet communication systems has been referred since the end of the 70's. Network simulation tools began to be effectively used during the 90's, mainly for simple validation of communication protocols. In the last decade, these tools have evolved and their usage has spread to testing and validation of communications technologies, network protocols, services and distributed applications, among others. These days, several commercial and free open-source simulation packages are available to satisfy different purposes.

Nowadays, network simulators are also used as educational and test bench tools for network protocols and configurations. They allow the test of potentially faulty scenarios without the negative consequences of doing so on production-ready systems. They also tend to allow a much closer and easier inspection of the inner workings of a network, with event-driven time shifting controls of the simulation. Since the network's hardware is simulated and it doesn't require the manipulation of existing hardware, these solutions have proven themselves to be relatively cheaper to use.

2.3.1 Network Simulation and Emulation

Network simulation allows the modeling of the behavior of a network by calculating the interaction between different network components using algorithms and mathematical formulas. These components can be end-hosts or network entities such as routers, links or packets. The simulated network can be computationally modeled by capturing and playing back experimental observation from real production networks. The behavior of the network and supported protocols can then be analyzed, allowing to infer the simulation model. In practice, this model must provide a low enough margin of error when compared to a real system, so as to provide usable simulation results.

Network emulation, on the other hand, implies an exact simulation of the network under planning in order to assess its performance or to predict the impact of possible changes or optimizations. The major difference between emulation and simulation is that a network emulator allows end-systems (such as computers) to be attached to the emulator. These will then act exactly as if they were connected to a real network. In the end, the network emulator's job is to emulate the network that connects end-hosts, but not the end-hosts themselves.[19]

Another major difference is related to how time progresses in emulation and simulation. Time in emulation progresses linearly and in real-time. For this reason, emulators aren't generally used for long duration network scenarios. Simulation time, on the other hand, can progress arbitrarily. Since it isn't dependent on external devices and/or accurate emulation, time can be slowed down, sped up, rewound, etc. For instance, a simulation time of three months could be compressed into just a couple of hours, if the complexity and resources available to the simulator allow it to do so. An emulator, on the other hand, would take the same amount of time as the network being that's being emulated.

Typical network simulation tools include NS2,[22] which is a popular network simulator that can also be used as a limited functionality emulator. In contrast, a typical network emulator such as WANsim[23] is a simple bridged WAN emulator that utilizes some Linux functionality.

Both network emulators and simulators share some common characteristics. In the context of this dissertation, both will simply be referred as *simulators*, unless explicitly described as such.

2.3.2 Basics of Computer Network Simulation

Elements of simulation

A network simulator is built upon a set of basic structural elements. [10] defines them as:

- **Entities -** These are the objects that interact with each another in a network simulator, causing changes to the state of the system. Entities might be computer nodes, routers, packets, flows of packets, or non-physical objects such as simulation clocks. To distinguish different types of entities, unique attributes are assigned to each of them. For

[22]`http://www.isi.edu/nsnam/ns/`
[23]`http://code.google.com/p/wansim/`

instance, a packet entity may have attributes such as packet length, sequence number, priority, etc.

- **Resources -** In general, there is a limited supply of resources available for the simulator. These are part of a complex system and have to be shared among a certain set of entities. This is usually the case of computer networks, where bandwidth, air time or the number of servers represent network resources that have to be shared among the network entities.

- **Activities and events -** Entities engage in activities from time to time. Events are generated when this happens, triggering changes in the system's state. Examples of such activities include delay and queuing. So, for instance, when a packet is waiting for transmission while the transmission medium is busy, it is said to be engaged in a waiting activity.

- **Scheduler -** A scheduler maintains the list of events and their execution time. During a simulation, the scheduler creates and executes events.

- **Global variables -** Global variables are used to keep track of some useful values related to the simulation, and they are accessible by any function or entity on in. These variables might include, for instance, the simulation time, the length of the packet queue in a single-server network, the total busy air time of the wireless network, or the total number of packets transmitted.

- **Random number generator -** A Random Number Generator (RNG) is required to introduce randomness in a simulation model.

- **Statistics gatherer -** The main function of the statistics gatherer is to collect data from the simulation so that meaningful inferences can be drawn from such data.

Types of simulation

The *core* of a network simulator refers to the piece of code responsible to advance the state of the simulation. This is usually done with one of the following techniques [6]:

- **Time-driven simulation -** With this technique, the network is simulated between consecutive time intervals (Figure 2.12, from [11]). This is also known as a "fluid" type of simulation due to the seemingly continuous way data packets are transmitted. This technique presents good overall scalability.

- **Event-driven simulation -** In this type of simulation, a series of events are created and executed in chronological order (Figure 2.13, from [25]). Some events may trigger other additional events (for instance, a packet departure event might generate another event to signal its arrival at the other end of the line). Most available network simulators implement this technique. This approach yields fine-grained results (in relation to both time and data), but suffers from major scalability problems. Some event-driven simulators implement optimization techniques in order to attenuate them.

- **Hybrid simulation -** As the name suggests, this is an intermediary of time-driven and event-driven simulation. These models can deal with fluid data and packet trains on the same simulation (but not in the same simulated component), and allow the use of efficient analytical techniques of traffic fluxes to improve the computational efficiency of the simulation.

- **Analytical simulation -** This type of simulation uses mathematical models [12] to predict network and application behavior. These models allow fast calculation of performance metrics and other results, but they have very limited application for computer network simulators. They are currently used to simulate only very specific parts of a computer network.

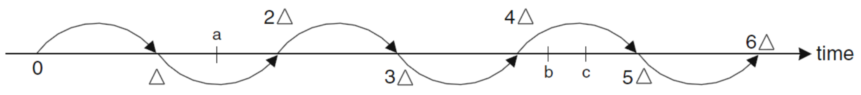


Figure 2.12: Simulation state advance in time-driven simulation

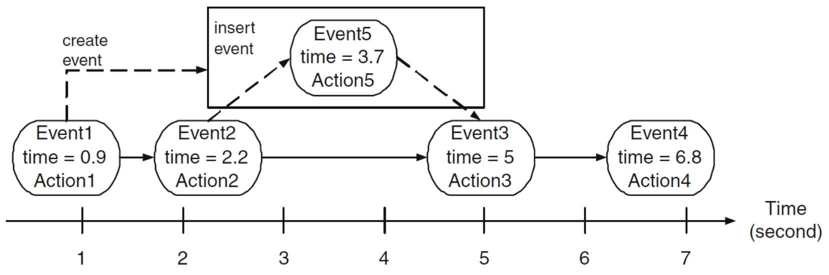

create event
insert event
Event5 time = 3.7 Action5
Event1 time = 0.9 Action1
Event2 time = 2.2 Action2
Event3 time = 5 Action3
Event4 time = 6.8 Action4
1
2
3
4
5
6
7
Time (second)

Figure 2.13: Simulation state advance in event-driven simulation

Parallel event simulation

The increasing computational complexity of simulation models and modeled scenarios motivates the demand for parallelism of the simulation. Hence a wide range of research has been conducted on Parallel Discrete Event Simulation (PDES) [25], which allows developers to draw benefits from executing a simulation on multiple processing units in parallel. This is an essential step to achieve before a network simulator can run successfully as a distributed system.

The approach taken by PDES is to divide a simulation model into multiple parts which are then executed on independent processing units in parallel. The central challenge is to maintain synchronization and correctness of the simulation results.[25]

The specific solutions for the challenges posed by parallelization differs according to the type of simulation being used. The main issue is related with *causality violation*. As the simulation progresses, the events with the smallest timestamps are removed from the queue and executed sequentially by the event handler. While the handler function is running, events can also be added to or removed from the list. This introduces a problem for parallelization, since it is necessary to synchronize the correct sequence of events between all processing units. If two events do not interfere with each other, then the simulation can be run in parallel. Otherwise, they must be executed sequentially. Parallel simulation frameworks employ a wide variety of synchronization algorithms to solve this issue.

A parallel simulation model is composed of a finite number of partitions which are created in accordance to a specific partition scheme. The three main partitioning schemes are: [25]

- **Channel parallel partitioning -** This is based on the assumption that transmissions over different (radio) channels, mediums, coding, etc. do not interfere. Thus, events on non-interfering nodes are considered independent and the simulation model is split between them. This type of partitioning scheme is not generally applicable to every simulation model, thus leaving it only for specialized simulation scenarios.

- **Time partitioning -** This scheme subdivides the simulation time of a simulation run in time-intervals of equal size. The simulation of each interval is considered independent from the others under the premise that the state of the simulation model is known at the beginning of each interval. However, the state of a network simulation is usually very complex and not known in advance. Due to this fact, this partitioning scheme cannot be applied to all simulation models as well.

- **Space parallel partitioning -** This scheme splits the simulation model along the connections of simulated nodes. The resulting partitions constitute clusters of nodes. This type of division is used very often, since it doesn't present the drawbacks of the other two.

The run-time component of the simulation that handles the simulation of a partition is named a Logical Process (LP). Each partition is mapped exactly into one LP. Every LP resembles a normal sequential simulation, containing state variables, a timestamped list of events and a local clock. Additionally, inter-LP communication is done through the exchange of timestamped messages via FIFO channels. The latter are used to preserve local FIFO characteristics, thus avoiding causality violations. Figure 2.14 (from [25]) shows the typical structure of an LP.

In practice, the number of LPs (i.e., partitions) is generally equal to the number of CPUs provided by the host's hardware. Consequently, LPs tend to directly map to processes on the operating system.

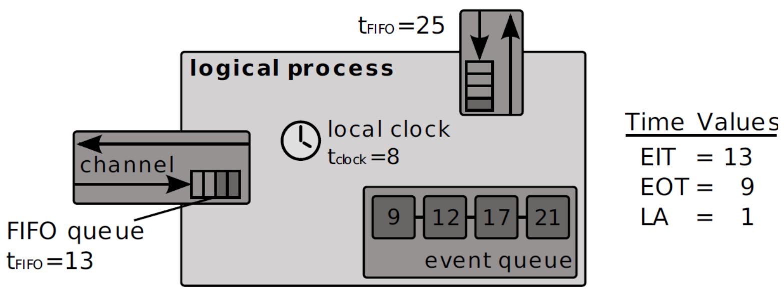


Figure 2.14: Logical process of a network simulation.

The challenges associated with the design of a distributed network simulator are identical to the ones identified in section 2.1.3.

2.3.3 Related Work

Network Simulator 2

Network Simulator 2 (NS2) is an open-source event-driven simulator designed specifically for research in computer communication networks. Since its inception in 1989, NS2 has gained interest from industry, academia, and even government entities. Having been under constant investigation and development for years, NS2 now contains numerous modules for network components, routing protocols, transport protocols, etc.

NS2 has been succeeded by Network Simulator 3 (NS3). NS3 was written entirely from scratch, and it is focused on improving the core architecture, software integration, models, and educational components of NS2. Nevertheless, NS2 is still used nowadays, mainly due to the lack of support for certain protocols in NS3 (which is not backwards-compatible with NS2). It's expected that the latter will eventually replace NS2 in most universities that are currently using it.[24]

General architecture

Figure 2.15 (from [11]) shows the basic architecture of NS2. It provides users with an executable command `ns`, which is usually followed by the name of the Tool Command Language (Tcl) simulation script to open. In most cases, a trace file is generated containing the results of the simulation.

NS2 works in two key languages: C++ and Object Tool Command Language (OTcl). The C++ portion of the code contain the definitions of the internal mechanisms (i.e., a backend) of the simulation objects, while OTcl is used to set up the simulation by assembling and setting each object's configuration, as well as scheduling discrete events (i.e., a frontend). The C++ and OTcl sections of the code are linked together using Tcl with classes (TclCl).

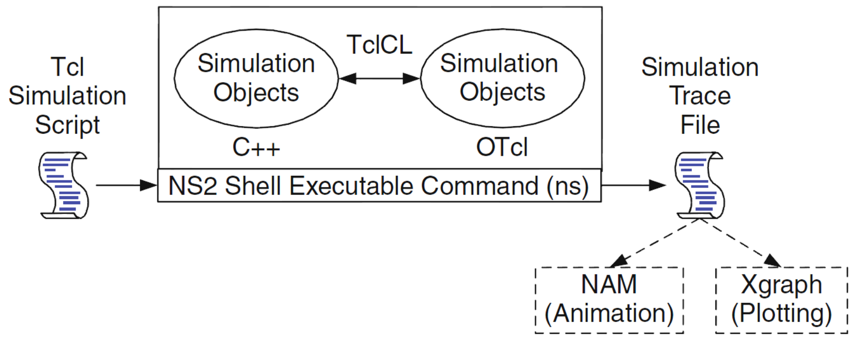


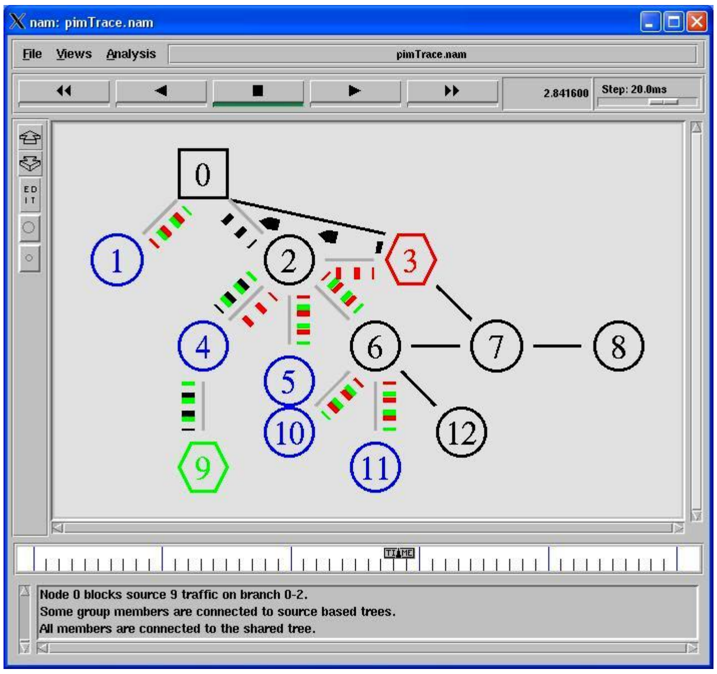
Figure 2.15: Basic architecture of Network Simulator 2

After the simulation is finished, NS2 outputs either text-based or animation-based simulation results. Tools such as Network AniMator (NAM) (Figure 2.16) and XGraph[25] can be used to interpret these results graphically and interactively. To analyze a particular behavior of the network, users can extract a relevant subset of text-based data and then format it into a more convenient way for better visualization it (such as graphs).[11]

Simulation details

NS2 is a discrete event simulator. An event contains an execution time, a set of actions and a reference to the next event. All events connect to each other, forming a chain on the simulation's timeline (Figure 2.13). NS2 does

[24] `http://www.nsnam.org/tutorials/NS-3-LABMEETING-1.pdf`
[25] `http://www.xgraph.org`



```
Node 0 blocks source 9 traffic on branch 0-2.
Some group members are connected to source based trees.
All members are connected to the shared tree.
```

Figure 2.16: Graphical representation of a network in Network AniMator (NAM)

not support any form of simulation parallelism.

A simulation scenario is set up in a Tcl simulation script and goes through two phases when it is executed. The first phase, the *Network Configuration Phase*, defines and configures the network components and how they link to each other. A chain of events is created by connecting all generated events chronologically. The second phase, the *Simulation Phase*, chronologically executes (or dispatches) the created events until the simulator is halted or there are no more events to execute.

There are four main classes involved in an NS2 simulation [11]:

- Class `Simulator` is responsible for the supervision of the simulation. It contains simulation components such as the Scheduler, RNG, etc., and also information of the objects that are shared by other (simulation) components.

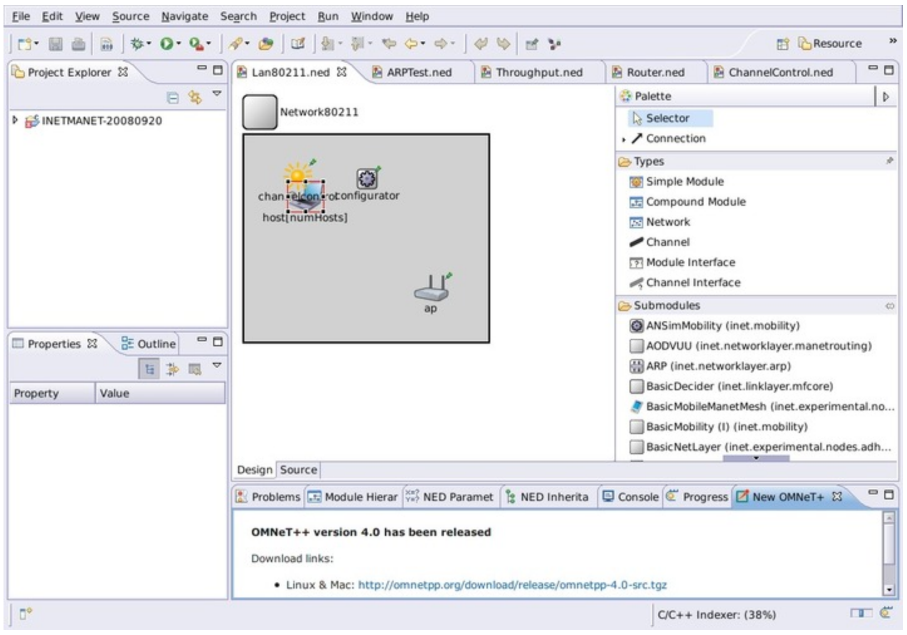
- Class `Scheduler` maintains the chain of events and chronologically dispatches them.

- Class `Event` contains the definition of an event on the simulation. It consists of the trigger time and the associated handler. Events are put together to form a chain of events, which are dispatched one by one by the Scheduler.

- Class `Handler`: When associated to an event, a handler specifies default actions to be taken when the event is dispatched.

OMNeT++

OMNeT++ is a modular, open-source, component-based C++ simulation library and framework. Although its primary application area is communication networks, OMNeT++ has a generic and flexible architecture, making it successful in other areas like IT systems, queuing networks, hardware architectures, or business processes. OMNeT++ has GUI support (Figure 2.17) and is available for Unix-like systems and Windows.



OMNeT++ version 4.0 has been released

Download links:

- Linux & Mac: http://omnetpp.org/download/release/omnetpp-4.0-src.tgz

Figure 2.17: OMNeT++'s Graphical User Interface

General architecture

OMNeT++ uses an Eclipse-based[26] simulation Integrated Development Environment (IDE) with GUI support built in. At its core, OMNet++ provides a component architecture based on *simulation modules*. In network simulations, these may represent user agents, traffic sources and sinks, protocols, network interfaces, etc., and data structures, such as routing tables. These modules, written in the C++ language, can be connected to each

[26]`http://www.eclipse.org`

other via *gates* and then combined together in order to form *compound modules*. These compound modules may represent more complex systems, such as routers and switches. Connections are created within a single level of the module hierarchy, meaning that a submodule can be connected with another submodule or with the containing compound module.[19]

Every simulation model is an instance of a compound module type. These models (components and topology) are described in NEtwork Description (NED) files. The NED language defines only the model structure (topology), and leaves behavior and a subset of module parameters open. This behavior can then be defined via the C++ code behind simple modules. Module parameters left unassigned in NED files will get their values from `ini` files (used to describe simulation parameters).[27]

Simulation models created with OMNeT++ are compiled using `make`. For convenience, a tool is provided to create the respective makefiles, named `opp_makemake`. The result of this compilation is an executable file. When executed, the simulation is ran and its results are saved in *output vector* (`.vec`) and *output scalar* (`.sca`) files. Output vectors capture behavior over time, while output scalar files contain statistics (such as the number of packets sent or peak throughput). The capability to record simulation results has to be programmed into the modules.

Several tools are provided to allow real-time simulation visualization, result collection and analysis. These tools enable the user to, for instance, watch an animation of the running network simulation (Figure 2.18),[28] collect event logs, sequence diagrams (Figure 2.20),[29] or plot scalar and vector data (Figure 2.19, from [25]) acquired from the simulation.

Simulation details

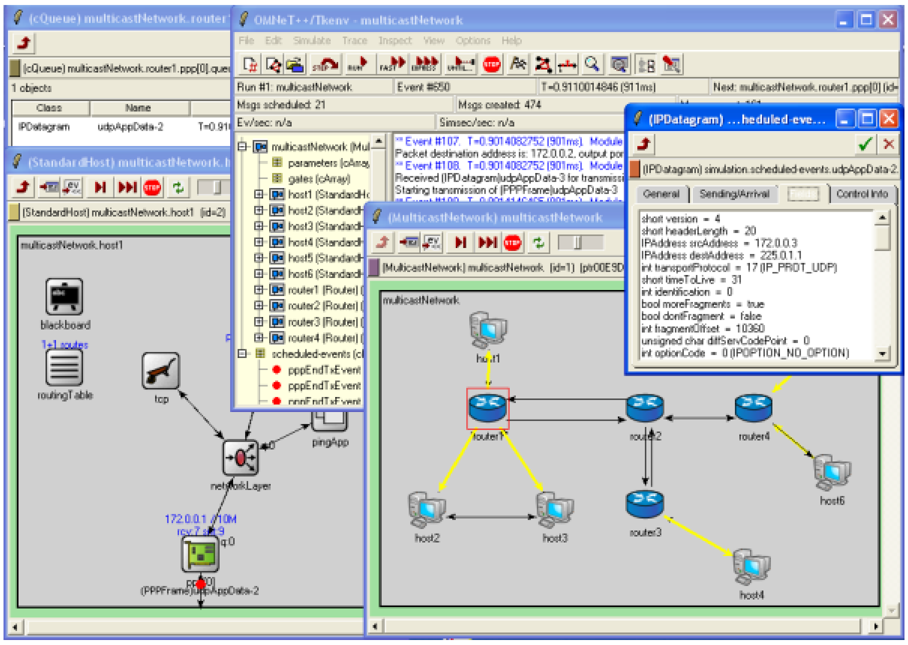
OMNeT++ is a discrete event simulator. It also provides support for network emulation, through packet capture from real networks. Parallel simulation is also possible via a conservative implementation of the *Null Message Algorithm* (see [21] for more details).

Unlike NS2, OMNeT++ uses live simulation. In other words, the user is able to observe the simulation as it progresses and can even use C++ debugging tools to examine the network elements in detail.

2.4 RoutUM

RoutUM is a network simulator being developed by the Department of Informatics of University of Minho since 2007. The subject of this dissertation originated as a direct result of the IPC requirements appointed during

[27]`http://www.omnetpp.org/pmwiki/index.php?n=Main.OmnetppInNutshell`
[28]`http://www.cs.wustl.edu/~jain/cse567-08/ftp/simtools/fig-5.jpg`
[29]`http://omnetpp.org/doc/omnetpp/ide-overview/pictures/img16.png`



(cQueue) multicastNetwork.router
OMNeT++/Tkenv - multicastNetwork
File Edit Simulate Trace Inspect View Options Help
(IPDatagram) ...heduled-eve...
(StandardHost) multicastNetwork.h
(MulticastNetwork) multicastNetwork
General
Sending/Arrival
Control Info
blackboard
routingTable
tcp
pingApp
networkLayer
host1
host2
host3
host4
host6
router1
router2
router3
router4

(cQueue) multicastNetwork.router1.ppp[0].que

1 objects

Class	Name	
IPDatagram	udpAppData-2	T=0.91

Run #1: multicastNetwork Event #650 T=0.9110014846 (911ms) Next: multicastNetwork.router1.ppp[0] (id=

Msgs scheduled: 21 Msgs created: 474

Ev/sec: n/a Simsec/sec: n/a

```
** Event #107.  T=0.9014082752 (901ms).  Module
Packet destination address is: 172.0.0.2, output por
** Event #108.  T=0.9014082752 (901ms).  Module
Received (IPDatagram)udpAppData-3 for transmissi
Starting transmission of (PPPFrame)udpAppData-3
```

```
multicastNetwork (Mul
  parameters (cArray
  gates (cArray)
  host1 (StandardH
  host2 (StandardH
  host3 (StandardH
  host4 (StandardH
  host5 (StandardH
  host6 (StandardH
  router1 (Router)
  router2 (Router)
  router3 (Router)
  router4 (Router)
scheduled-events (c
  pppEndTxEvent
  pppEndTxEvent
  pppEndTxEvent
```

(StandardHost) multicastNetwork.host1 (id=2)

multicastNetwork.host1

1+1 routes

172.0.0.1 / 10M

rcv:7 snt:9

(PPPFrame)udpAppData-2

(MulticastNetwork) multicastNetwork (id=1) (ptr00E9D

multicastNetwork

(IPDatagram) simulation.scheduled-events.udpAppData-2

```
short version = 4
short headerLength = 20
IPAddress srcAddress = 172.0.0.3
IPAddress destAddress = 225.0.1.1
int transportProtocol = 17 (IP_PROT_UDP)
short timeToLive = 31
int identification = 0
bool moreFragments = true
bool dontFragment = false
int fragmentOffset = 10360
unsigned char diffServCodePoint = 0
int optionCode = 0 (IPOPTION_NO_OPTION)
```

Figure 2.18: OMNeT++'s Tkenv Graphical Runtime Environment

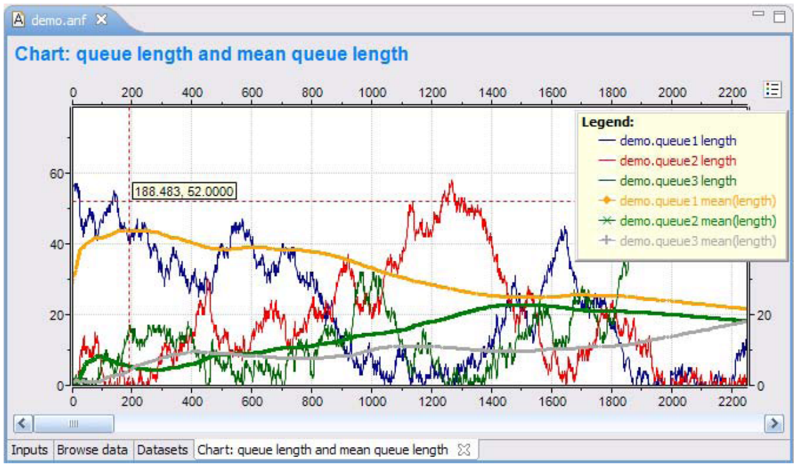
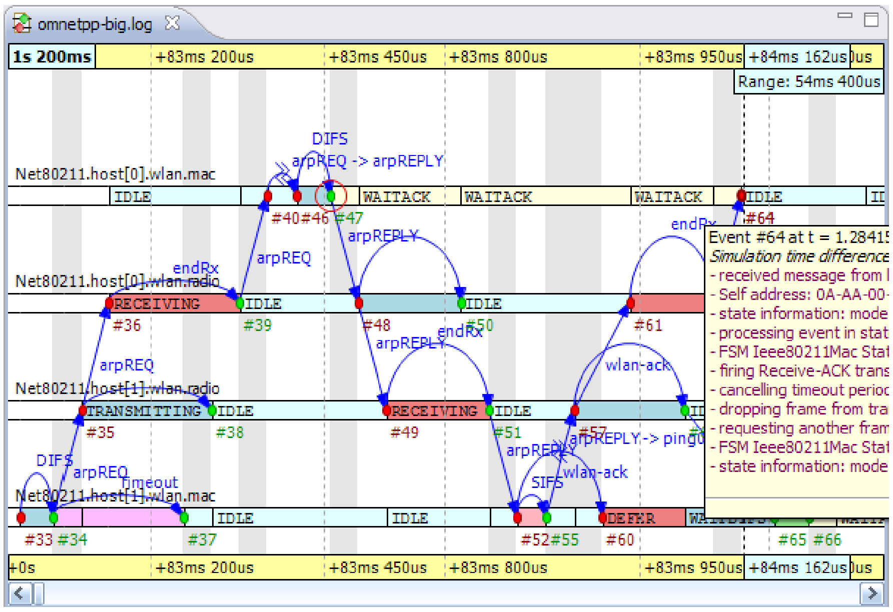


Figure 2.19: OMNeT++'s Result Analysis Tool

the RoutUM project's planning phase. Because of this, an understanding of the simulator's architecture and IPC needs are essential to achieve the best results.



Event #64 at t = 1.28415
Simulation time difference
- received message from
- Self address: 0A-AA-00-
- state information: mode
- processing event in stat
- FSM Ieee80211Mac Stat
- firing Receive-ACK trans
- cancelling timeout period
- dropping frame from tra
- requesting another fram
- FSM Ieee80211Mac Stat
- state information: mode

Figure 2.20: OMNeT++'s Sequence Diagram example

2.4.1 Goals

The main goal of the RoutUM project is to define a new implementation model for computer network simulators, using state of the art techniques and technologies. This project is an attempt to surpass the limitations of existing major public domain tools, while maintaining an high level of appeal for first time users of network simulators. The underlying model is based on new and recent paradigms, relying on simple processes for an high level of extensibility in future component upgrades (including the core). Cooperation is a key aspect for the evolution and long term success of this project.[6]

2.4.2 Architecture

The RoutUM's architecture is based on an hybrid simulation core, and parallelism is achieved through a space parallel architecture. Each logical process that results from this division translates into a different process on the operating system. These processes are then free to be located on the same machine or in different ones (or a combination of both), communicating with each other through local and remote IPC mechanisms, respectively.

The RoutUM's architecture defines a central simulation client. This client is responsible to receive a simulation request from a user, execute it, and then return the simulation's results to the user.

As previously stated, the simulation can be executed on the same machine (centralized model) or in different ones (distributed model). If the user opts for a distributed model, two different classes of processes are used: *leader* and *worker*. A *leader* process is the only one that interacts with the central simulation client. Its task is to receive simulation work and distribute it between all *workers*. This distribution can be specified by the user, or it can be done by an external module. A *worker* process executes the simulation tasks requested by the *leaders*. If no workers are registered on the system, then the *leader* does all the work instead.

Figure 2.21 represents RoutUM's general architecture. For the sake of example, each node represents a machine with only one process.[23]

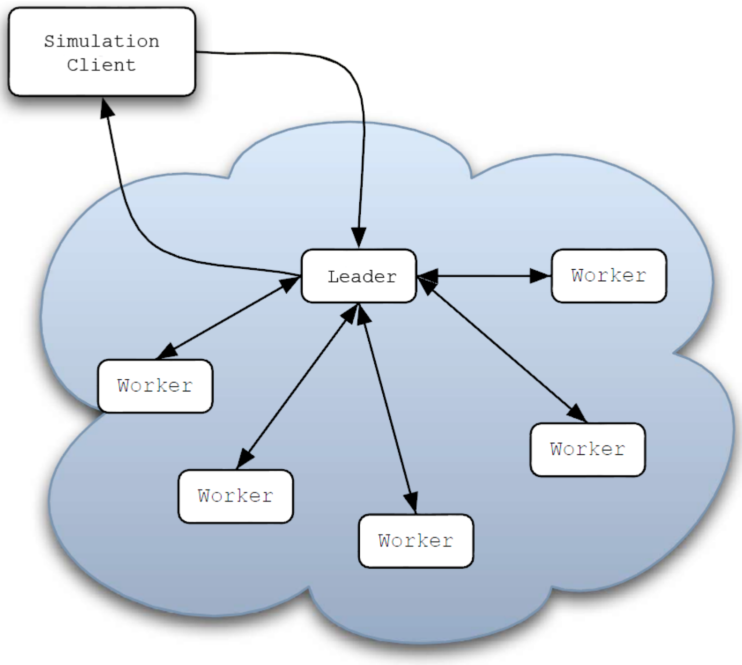


Figure 2.21: RoutUM's general architecture

Initialization of the simulation

Figure 2.22 represents the steps taken by RoutUM during the initialization of a new simulation.

In the first step, the simulator receives and analyzes the simulation scenario the user intends to run. Two lists originate from this analysis: one containing the simulation modules and another with their connections. A hierarchical list of components required for each simulation module is then determined. An example of such list can be seen in Figure 2.23.

The second step consists of two tasks. First, the simulator verifies if the computing resources available (memory, storage space, etc) are enough for the simulation to run. Secondly, it registers the simulation's modules

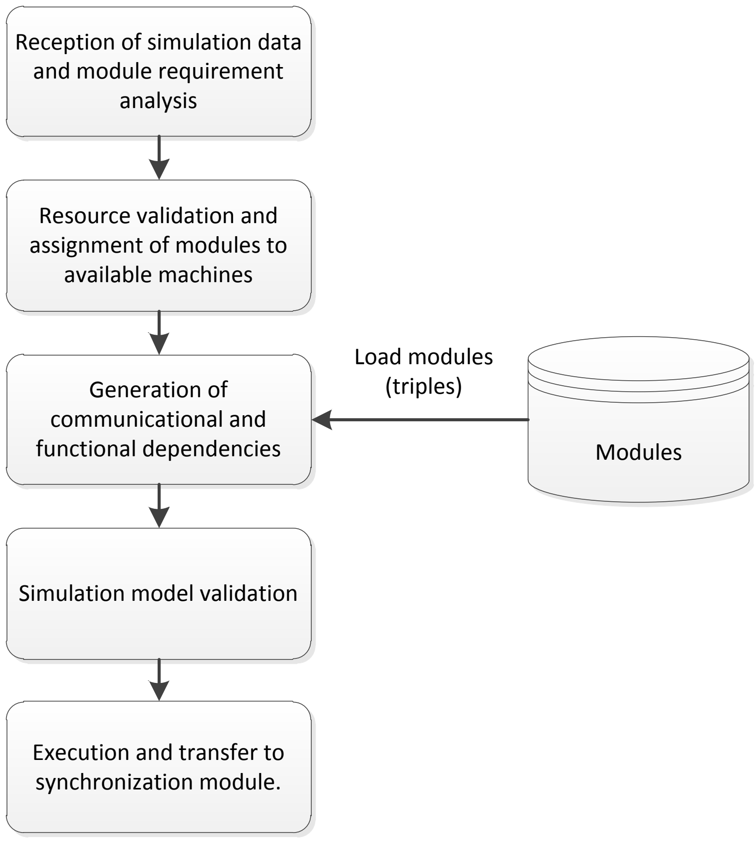

Reception of simulation data and module requirement analysis
Resource validation and assignment of modules to available machines
Generation of communicational and functional dependencies
Load modules (triples)
Modules
Simulation model validation
Execution and transfer to synchronization module.

Figure 2.22: RoutUM's initialization procedure

throughout the available machines (the *workers*). As previously stated, this distribution can be achieved either by attempting to balance the workload automatically between the available workers, or by appointing specific tasks to specific workers (through manual configuration by the user).

During the third step, the simulator generates functional and communicational dependency information from the data generated in the previous steps. The functional dependencies are established as an hierarchy of *nodes*, each one containing two objects. The first object stores the information pertaining to that node, while the other stores a branch list (each identified by their respective key in the *hash table*). This is exemplified on (the left side of) Figure 2.24. All objects in all nodes are then instantiated.

The communicational dependencies are established at two different levels: The first level pertains to communications along the branches of the module hierarchy. In other words, these are the communications between a module and its immediate parent or child module. The second level of dependencies are the ones established between unrelated sub-modules. Each

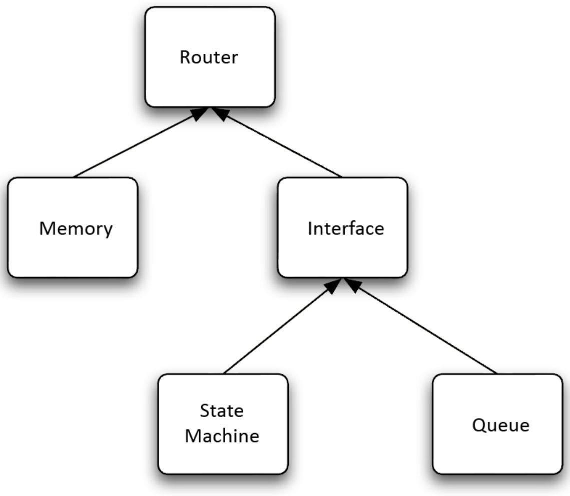


Figure 2.23: RoutUM's module hierarchy (example)

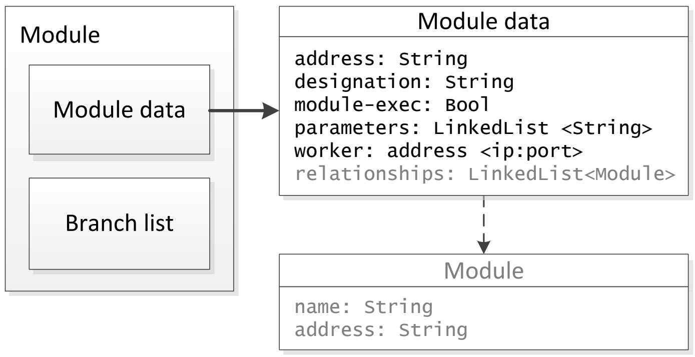


Figure 2.24: RoutUM's internal module structure

module contain variables used to identify these connections, as seen in (the right side of) Figure 2.24. The variables (and objects) shown in gray are empty if no connections exist between unrelated sub-modules.

The fourth step consists of the logical validation of the simulation environment. The semantics and syntax of the network topology are evaluated, and the simulation progresses to the next step if no errors were detected.

The fifth and last step consists on the execution of the simulation. The lowest-level modules on the module hierarchy are executed first, followed by their parent modules and so forth. The simulation process is then transferred to the *synchronization module*.[23]

Synchronization module

RoutUM uses an hybrid approach for its simulation, meaning that it can simulate both at a packet and data flow level. The synchronization module keeps control the various components of the simulation, namely its temporal advancement.

Simulation contexts

RoutUM uses a space parallel architecture to achieve simulation parallelism. Each group of LPs (known internally as *contexts*) receives an instance of a temporal synchronization module, which in turn is part of an hierarchical structure of these modules (Figure 2.25). A submodule communicates with its "parent", and so forth until the main synchronization module is reached.

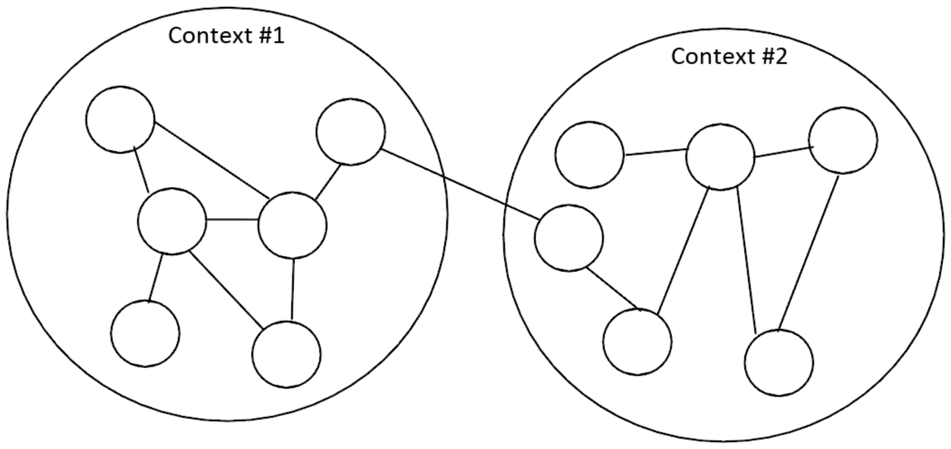


Figure 2.25: RoutUM's simulation contexts

Every simulation context has its own clock, which progresses in fixed time intervals. The simulation time is only allowed to progress (*tick*) if there are no events left to process in the next interval. An advantage provided by simulation contexts is that they allow the usage of different time intervals for each context. Without it, every module would progress with the same time interval, thereby potentially wasting resources with idle parts of the network. The contexts concept makes it possible to immediately advance a particular context up to the interval with the next event. So, for instance, it would be possible to have a context A progressing with a fixed time unit of 1, while context B (which has a lot less activity than A) would progress with a time unit of 10. This way, context B would only need to check for events (with each of its modules) with 10 times less frequency than A, therefore increasing efficiency.[22]

Hierarchical synchronization system

The division of a network into contexts introduces issues related with the synchronization of events between them. RoutUM attempts to solve these issues by dividing the event table as well, leaving each context with the events that belong only to that same context. But there's still a problem when a module from one context triggers an event on another module from a different context. In these cases the event must be present on both contexts, otherwise the time would advance without restrictions. This would originate paradoxes, such as one module sending a packet to a different context and the departure and arrival dates not complying with the rule of causality.

RoutUM solves this issue by rewinding the simulation time up to the point where an unsynchronized event occurred. The missing event is then introduced to the event table of the target context, and the simulation resumes from that point. But the simulation can only rewind within a fixed time window. If an unsynchronized event lies beyond this window, then a causality violation error is thrown and the simulation accuracy is compromised.

The registration of events must pass through the parent module, since it is the only one with direct control of the tables of the synchronization modules.[22]

Event types

Several different event types are defined by RoutUM. These are:

1. **Synchronization points** - Used to synchronize the simulation clock.
2. **Simulation data transfer** - Used to signal the exchange of internal data (pertaining to the simulation) between nodes.
3. **Traffic data transfer** - Used to transfer various types of simulated traffic.
4. **Reports** - Used to generate and collect statistics at predefined points of the simulation.
5. **Control** - Signals global simulation control activities.

Communication protocols

By design, the RoutUM network simulator allows modules to be heterogenous, from the programming language on which they are implemented to the operating system where the module will run in. However, every module must comply to a single communication protocol, so they can communicate with each other despite the discrepancies in their implementations.

Addressing scheme

RoutUM uses an unique string to identify each module in the hierarchical structure of the simulator. Each string contains the simulation, context and module IDs. It doesn't identify any real-life resource (such as a computer or CPU), being purely functional.

Protocol primitives

As of writing this report, three primitives have been defined: [18]

- `send(protocol_version, pdu_type, payload_size, payload, dest_mod_id)`
- `receive()`
- `ack(dest_mod_id)`

As the name suggests, the `send` primitive is used to send data to the destination module's incoming buffer, and the `receive` primitive is used to retrieve data from this buffer. The `ack` primitive sends a confirmation of the received data.[18]

The Protocol Data Unit (PDU) contains the following fields:

1. **Version -** Protocol version (4 bits)
2. **Type -** Type of PDU (4 bits)
3. **Origin -** The source module's address (variable length)
4. **Destination -** The destination module's address (variable length)
5. **Sequence number -** The packet's sequence number (16 bits)
6. **Payload size -** The length of the payload, in bytes (32 bits)
7. **Payload -** The payload data (variable length)

There are two different types of PDU in the simulator. The first is used to encapsulate the data packets generated in the simulation, while the second one represents the information used to generate fluxes.[18]

Registration module

The registration module is used to establish the correspondence between functional and physical addresses on the system. This module is queried when the primitives `send` and `ack` are called. The resulting PDU is sent though a protocol from the IPC mechanism being used.[18]

Resource sharing and management protocols

In real networked devices, processes compete for shared resources (memory, CPU time, link access, etc). This sort of interaction has to be mirrored in network simulators for added accuracy. In RoutUM, access to shared resources is managed by the Resource Sharing Management Protocol. All resources in a simulation that requires access management must register in the Shared Resources Manager using this protocol.[18]

The resource sharing and management protocol defines an interface with the following methods:[18]

1. `register_resource(resource_id, resource_ptr, size, access_flags, priority)`
2. `read_resource(resource_id, resource_ptr, size)`
3. `write_resource(resource_id, resource_ptr, size)`
4. `unregister_resource(resource_id)`

2.4.3 Parallelization of the Simulation

RoutUM's *contexts* concept creates isolated zones of activity and allows them to execute simultaneously, therefore simplifying the parallelization of the simulation. In addition, the objects in the simulator translate to ordinary processes running on the operating system, communicating with each other through IPC mechanisms and using the protocols described in Section 2.4.2. These two elements of the simulator's architecture enable a lot of possible configurations for the distribution of tasks over several host machines.

Chapter 3

Research

In order to build an API that can successfully solve the practical goals of this dissertation, it is first necessary to identify the main challenges behind its development. This chapter addresses these challenges and the solutions that were obtained for them.

3.1 Objectives

The main topic of this dissertation originated during the planning stage of the RoutUM simulator. Due to its heavy dependency on IPC to carry the simulation, it would be convenient the have the ability to transparently identify and communicate with any object (each present on different processes) in the system, regardless of the communication mechanisms available. The relative locations of the processes and the need for maximization of performance also motivates the creation of a method to automatically choose the fastest IPC mechanism available.

In addition to the goals established in Chapter 1.3, some desirable characteristics could be identified for the API:

- It should be as light (small overhead) and fast as possible;
- It should be modular, allowing easy addition, removal, replacement and maintenance of sections of code for each functionality;
- Security mechanisms ought to be supported, if the communication medium or the user justifies so;
- Each process should be identified with a unique primitive (e.g. a number or a label);
- Set-up and configuration simplicity is desirable;
- Portability of the code and/or cross-platform support of the API is desirable;

- Support for several levels of openness of the context is desirable (e.g. only registered hosts can connect; any host can connect or any host from the local network can connect, etc);
- Some additional functions may be desirable to facilitate the implementation of other specific RoutUM requirements.

Due to time limitations and to avoid unnecessary complexity and overhead of the API, only the functionality that is strictly required will be researched and implemented. Additional functionality should be implemented on the application level as needed.

3.2 General Architecture

Figure 3.1 shows a rough representation of the elements on the system and the connections between them. A process can connect to another through remote (network) or local IPC mechanisms, depending on the relative location between the processes. The details of the communication are all hidden away, including protocols, security mechanisms, etc.

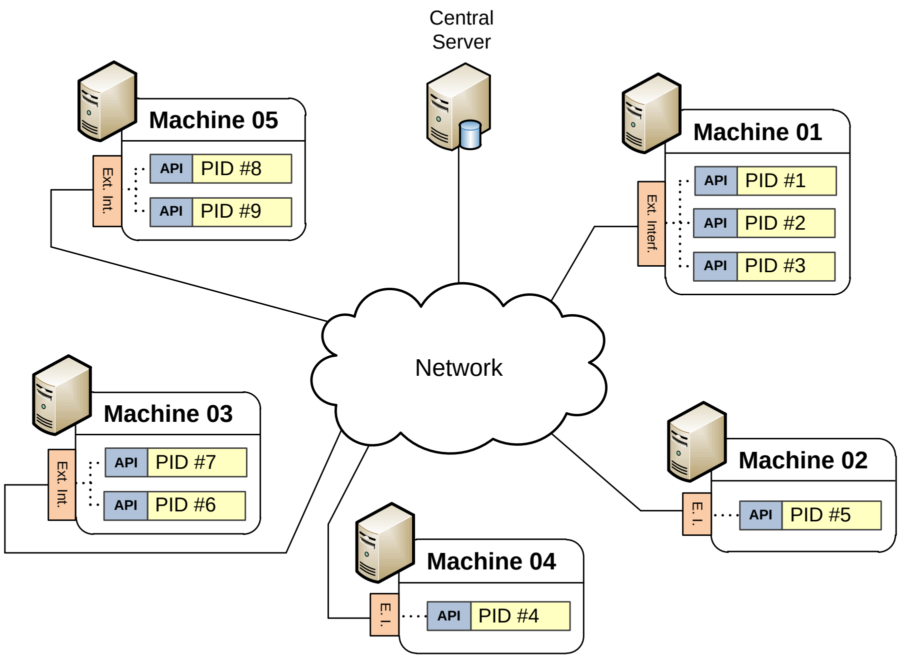


Figure 3.1: Example architecture (as seen from the API)

The central server is used to store a list of peers, their IPC interface information and a list of contexts with their settings. When a process wants to join the system, it connects to the server to obtain a list of contexts and peers it is allowed to connect to.

In the context of this dissertation, the terms *instances* and *peers* will be used interchangeably to refer to these API instances.

3.3 Programming Language

Ideally, the programming language used in this project should be able to achieve the following goals:

- Should allow the highest possible performance, independently of the underlying hardware configuration (this is a critical factor);
- Should include cross-platform and multi-architecture support (the mainstream OSs, UNIX-like and Windows, must be supported);
- Should include APIs for local and remote IPC mechanisms and cryptography functions (the access to the OS's functions should be direct, without the additional overhead introduced with wrappers, for instance);
- Should be a widely supported language (the API is going to be used with existing distributed applications).

With all these requirements in mind, the language chosen for development was `C`. Besides meeting all the goals written above, there are a number of compelling reasons for this decision, such as:

- It's a very mature and widely-used language (including in the simulation field), with all the needed APIs for this project. The same can be said about the compilers and libraries available.
- All the major operating systems are written natively in `C` and provide libraries in the same language. This means there is no need to access system functions though wrappers.
- If needed, it is possible to add a wrapper in order to support another language. Some of them are even built upon the C language. However, the opposite isn't true.
- It's a language which allows small, efficient, low overhead output (whether it's a program or a library). It can even be used with *assembly* to provide a nearly ideal level of code efficiency.
- Provides an high level of control over data structures, which is necessary to obtain an high degree of efficiency.

The main drawbacks of the `C` language are the relative difficulty of use and the absence of security-related features present on higher-level languages (such as Java's error handling mechanism). Plus the absence of hierarchical concepts found on higher-level languages introduces an increased level of complexity with management and addition of modules to the code.

Libraries and portability

The standard C libraries and POSIX APIs are to be used wherever possible, as they provide a uniform interface for all UNIX-based systems. For the Windows operating system, Microsoft provides the UNIX-subsystem add-on,[1] adding limited support for compilation and execution of POSIX applications. A more complete alternative, albeit not without a bigger performance degradation, is Cygwin.[2]

At least the IPC portions of the code are going to be OS-specific, which automatically introduces the need to use a portability strategy. To accomplish this, conditional compilation directives of the C preprocessor[3] can be used to separate code by the platform where it is being compiled in.

3.4 Inter-process Communication Mechanisms

As defined on chapter one, two IPC mechanisms will be implemented in the API, one for remote and one for local IPC. In this section, an attempt is made to establish which mechanisms allow the best performance and/or compromise between several other desirable characteristics.

3.4.1 Selection Criteria

All of the mechanisms described in Section 2.2 have their own advantages and disadvantages. When opting for the most suitable IPC mechanism, several factors have to be taken into consideration, namely:

- **Throughput -** This refers to the resulting data rate transmitted between processes, including bandwidth, processing time, error rate (detection and correction), transmission delay, jitter. There are some instances when, for example, a lower processing and transmission delay of one mechanism might end up giving an advantage over another one with a higher bandwidth. This is the case when two end-points exchange a series of small messages in a question-reply scenario.

[1] `http://technet.microsoft.com/en-us/library/bb496506.aspx`
[2] `http://www.cygwin.com/`
[3] `http://en.wikipedia.org/wiki/C_preprocessor#Conditional_compilation`

- **Scalability -** Some mechanisms perform better with an higher number of processes than others. This might also reflect on the programming complexity and the resources needed by the system.
- **Connection establishment -** Some mechanisms allow an easier identification and connection of processes than others. For instance, it is easier to find and identify a process to connect to through a file handle than through its process ID. Additionally, some mechanisms are full-duplex and others half-duplex, the latter requiring twice as many IPC descriptors for bidirectional communication.
- **Reliability -** Reliability can be offered in the form of minimum guaranteed overall throughput, delay and/or connection persistence.
- **Compatibility -** Some systems might be heterogeneous, using different operating systems, application, network protocols, etc. An IPC mechanism used in such system must be compatible across all of these characteristics.
- **Ease of implementation -** There are several factors that may increase programming, maintenance and debugging complexity. For instance, some IPC mechanisms do not contain error checking built-in, so it would be necessary to implement it.
- **Resource usage -** Some IPC mechanisms may require a lower CPU and/or memory usage for the same bandwidth than others. This has an impact on the resources left available to the rest of the system, potentially creating an indirect bottleneck of the throughput.

It is worth noting that although different operating systems might support the same set of standards and interface specifications, such as POSIX or System V Interface Definition (SVID), the low level mechanisms used to implement and support these specifications may be significantly different. As such, an application's performance may vary significantly between different operating systems and even between release versions. This issue is made more complex when different specifications of hardware and/or running software are taken into account. In short, there are too many variables for a single definitive conclusion.

3.4.2 Local IPC

Local IPC mechanisms are theoretically faster than remote ones, since the data transferred between processes never ends up leaving the local memory bus. Because of this, it is important to use them whenever possible to obtain the best performance. Due to the fact that the data never passes through an external connection (such as ethernet), there should be no need for certain features, such as error control or confidentiality.

Some IPC mechanisms won't be addressed here since they present characteristics which obviously put them at a disadvantage when compared to others. The first one are **file**-based communications, whose dependency on the file system implies that it is several orders of magnitude slower than purely memory-based mechanisms. Another one are anonymous **pipes** because they require a parent-child relationship, thus making it harder to develop CLI-based applications due to the standard stream redirection (the API should not never impose such restrictions on the application).

In the context of this project, **signals** and **semaphores** aren't useful when used alone, since they allow a very limited form of communication. Instead, they will be used as synchronization primitives, working side-to-side with other IPC mechanisms.

Evaluation of IPC mechanisms

This section presents the analysis of a few references and the results of tests developed specifically for this dissertation.

[9] presents a performance analysis of five IPC mechanisms over several UNIX-based systems. These mechanisms are POSIX pipes, POSIX FIFO, System V messages, System V shared memory (in conjunction with semaphores), and UNIX domain sockets. The tests were executed using a Producer/Consumer model as described in [24].

Five programs were written to analyze each of the selected mechanisms. For each test, 5000 messages were sent from one end to the other. The message size (in bytes) and the number of messages to transmit were passed as command-line arguments to each program. Each benchmark was executed on the following operating systems:

- Linux 2.2.5-15
- Linux 2.2.17
- Linux 2.4.0-test9
- RTLinux v2.3 (prepatched with Linux 2.2.14)
- FreeBSD 4.1
- FreeBSD 4.2

Each operating system was prepared for testing by performing an "out-of-the-box" default installation, modifying a few select IPC kernel parameters (to set the maximum amount of a shared memory segment to 8192 bytes on all OSs), rebuilding the kernel using default compiler/linker settings, and placing the system in multi-user mode. No attempts were made to fine-tune the operating systems to enhance performance.

The results were obtained by executing the benchmarks on a computer with the following configuration:

- Single processor system employing a Pentium III Processor operating at 450 MHz.
- 32 KB split L1 cache and 512 KB of pipelined L2 (unified) cache.
- 128 MB of 100 MHz SDRAM.
- 30 GB of hard disk space.
- Phoenix BIOS version 4S4EB2XO.10A.001-P03.
- Standard sets of input/output devices including keyboards, mouse, etc.

Figure 3.2 shows the throughput results for POSIX pipes, Figure 3.3 for POSIX FIFO queues, Figure 3.4 for System V messages, Figure 3.5 for System V shared memory (with semaphores) and Figure 3.6 for UNIX domain sockets. All figures were taken from [9].

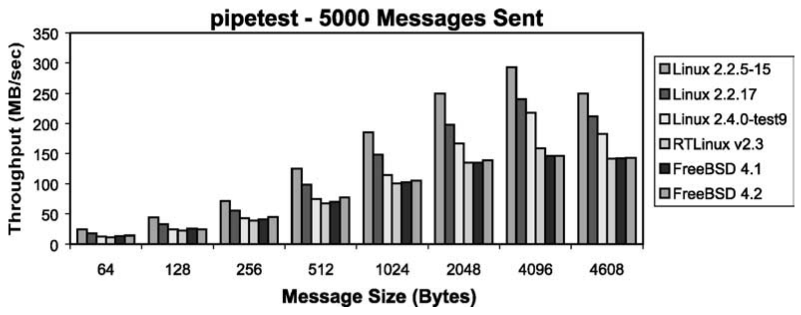


Figure 3.2: Throughput for POSIX pipe

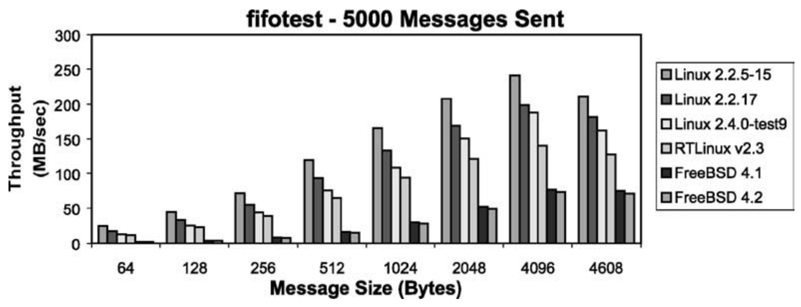


Figure 3.3: Throughput for POSIX FIFO queue

From these results, it is easy to conclude that Linux kernel 2.2.5-15 provided the best performance from all the benchmarks, with the exception of shared memory. Performance differences in all mechanisms are relatively consistent with message sizes and operating system. There is a direct proportionality between packet size and throughput, save for few (and not very significant) exceptions. Figure 3.7 (from [9]) shows the average throughput for all tests.

For each test, the time taken to create and destroy the IPC resources was also computed. Table 3.1 (from [9]) shows the results for 10.000 iterations.

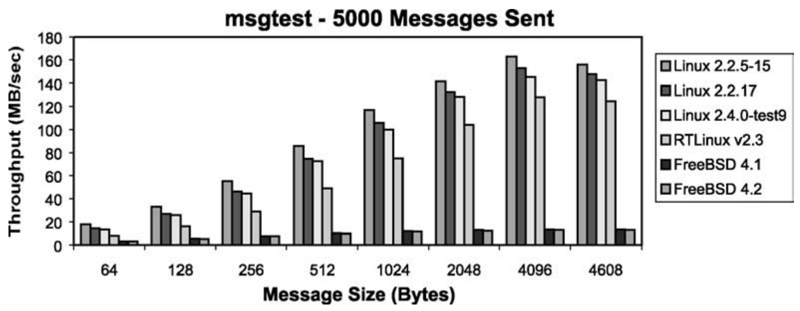


Figure 3.4: Throughput for System V messages

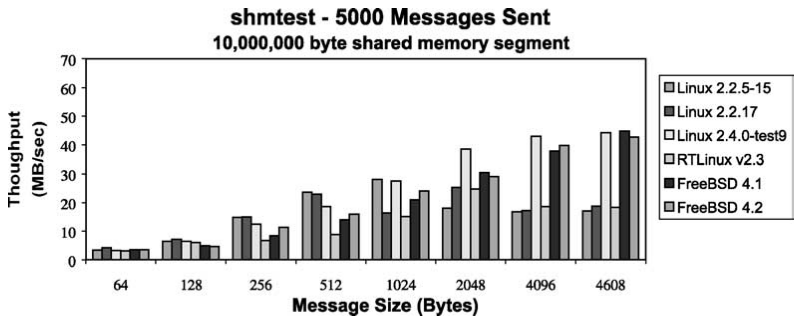


Figure 3.5: Throughput for System V shared memory (with semaphores)

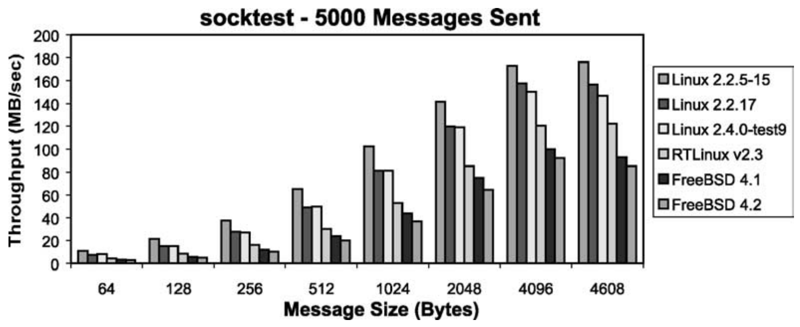


Figure 3.6: Throughput for UNIX domain sockets

It is possible to conclude that, on average, anonymous pipes consistently outperformed all the other mechanisms, followed by named pipes, System V messages and UNIX domain sockets.

Unfortunately, the results shown here refers to outdated versions of the Linux and FreeBSD's kernel. It's possible that significant internal structural changes have already been performed, which may have a significant impact on the validity of these results.

Another reference, [26], presents a comparative analysis between 3 different types of IPC on the Linux kernel. Three utilities were built to test

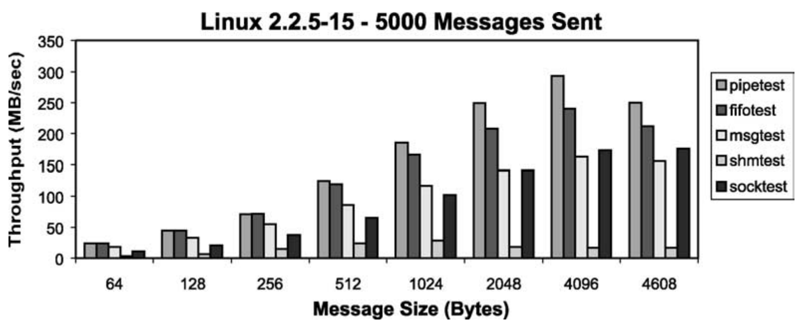


Figure 3.7: Average throughput by mechanism for Linux kernel 2.2.5-15

	Linux 2.2.5-15	Linux 2.2.17	Linux 2.4.0-test9	RTLinux v2.3	FreeBSD 4.1	FreeBSD 4.2
POSIX pipe	574	766	1321	1313	668	677
POSIX FIFO	4378	4614	6742	5908	86999	111170
SVID messages	238	309	296	609	357	341
SVID shared memory	6684	8373	9868	8303	1472	1471
Semaphores	235	270	267	418	330	332
UNIX domain sockets	6107	659	7740	7733	83439	1501836

Table 3.1: Time needed to create and destroy 10.000 IPC resources (in *ms*)

anonymous pipes, UNIX domain sockets and network domain TCP/IP sockets.

To determine the relative performance of each IPC mechanism, each was subjected to two benchmarks to find out the raw throughput of each mechanism under different circumstances. The first benchmark involved transferring data in packet sizes ranging from 1 MB to 100 MBs with an equivalent buffer size so only one system call was being made to the write function. The second benchmark involved sending a fixed amount of data, 100 MBs, with varying buffer sizes from 100 KBs to 100 MBs. There is a significant amount of overhead involved in making system calls due to the context switching. This benchmark was designed to get a better understanding of how the various combinations of memory operations and context switching affected performance. Both benchmarks were performed on many machines with different hardware configurations to minimize the effects of the differences in hardware.

For almost all of the data sizes transferred, Unix domain sockets outperformed the two other IPC mechanisms, as seen in Figure 3.8. Although Figure 3.8 (from [26]) shows the results from a specific machine, the transfer

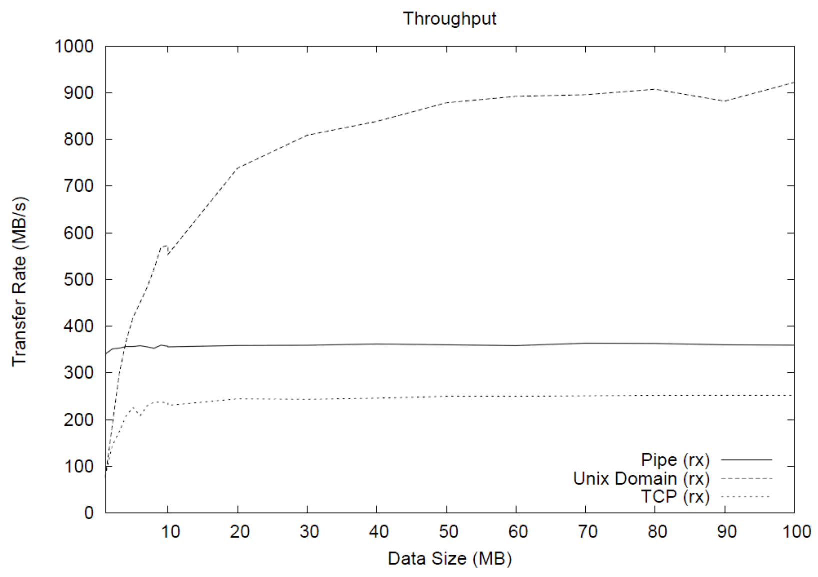


Figure 3.8: Benchmark 1 results

rates are consistent across all of them. On some machines, UNIX domain sockets reached transfer rates as high as 1500 MB/s.

For small data sizes, the throughput of UNIX domain sockets was below that of pipes. Investigation of the cause was inconclusive because the results were not consistent across all machines. On the Intel processor machines, pipes performed better than TCP/IP sockets. This was not true for the machines using the UltraSPARC processor.

Figure 3.9 (from [26]) shows the results of the second benchmark. While the performance of pipes and TCP sockets remained relatively stable, UNIX domain sockets showed a major decrease in performance as the buffer size increased.

In conclusion, UNIX domain sockets have proven to deliver the highest throughput when compared to the other mechanisms. While its dominance is still unclear for transfers of small amounts of data, it is otherwise the best mechanism to use within a single machine.

Transmission Throughput Test

An application was developed in the context of this dissertation, so as to provide more up-to-date-results. It was named `ttt` (short for Transmission Throughput Test) and it contains several modules programmed in C/C++ to test each of the following IPC mechanisms:

- TCP/IP network domain socket (loopback);
- User Datagram Protocol (UDP)/IP network domain socket (loopback);

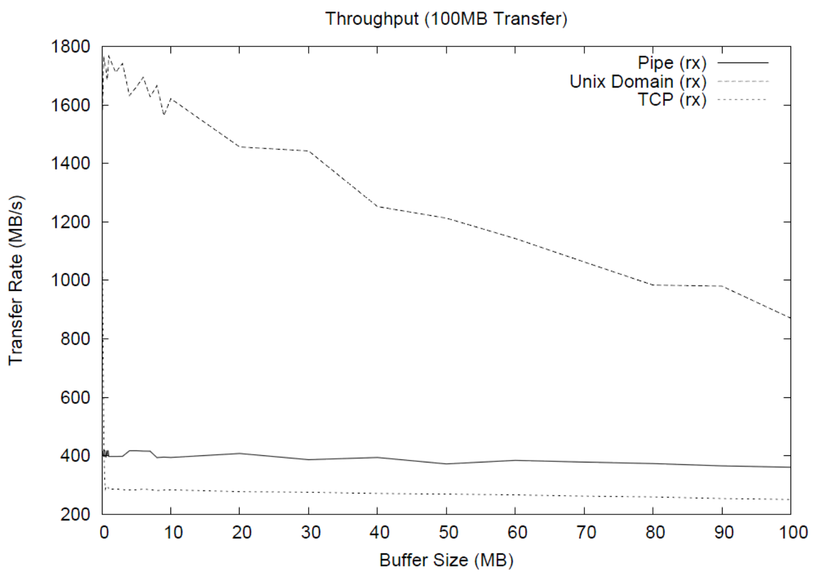


Figure 3.9: Benchmark 2 results

- Raw ethernet (level 2) network domain socket (loopback);
- UNIX domain socket;
- POSIX Shared memory (through memory mapping) with Semaphores;
- POSIX FIFO queue;
- POSIX Message queue.

The test machine is a notebook with the following configuration:

- CPU - Intel Core 2 Duo T9400 (2.53GHz, 64KB L1 cache, 6MB L2 cache)
- Ram - 4096MB DDR3 PC 1066 (533 MHz) 7-7-7-20
- Motherboard - LG Emerald (1067 MHz FSB)
- Storage - Fujitsu MHZ2320BH G2 (320GB, 5400rpm, 8MB buffer)
- Network adapter - Intel 82567LF Gigabit Network Connection (1500 Bytes of MTU)
- Operating system - Ubuntu Desktop v11.10 (Linux kernel v3.0.0-12, 32-bit)

The operating system was a standard Ubuntu Desktop installation, running in runlevel 5 and with no additional applications or daemons set up. Outside connections were also disabled during tests (using the command `ifconfig [interface] down`).

The tests consist on sending as much data as possible from one instance of the application to the other. A script was prepared so that each test ran for 10 minutes, first with data packets of 128 bytes, then 1024 and 16384 bytes. Five tests were done for each mechanism and all values were averaged. POSIX libraries and functions were used whenever possible. Figure 3.10 shows the message protocol used by the application. The function of each field is:

- **Flag -** Used to recover synchronization in lossy, stream-based IPC mechanisms (but used on all mechanisms);
- **Length -** The length of the entire packet, in bytes;
- **Sequence number -** Used to detect out-of-order packets;
- **Send time -** Used to compute delay times;
- **Checksum -** Used to detect errors in the header fields;
- **Padding data -** Used to set a message size bigger than the header itself. This data is formed by a fixed pattern, so it can also be used to detect errors.

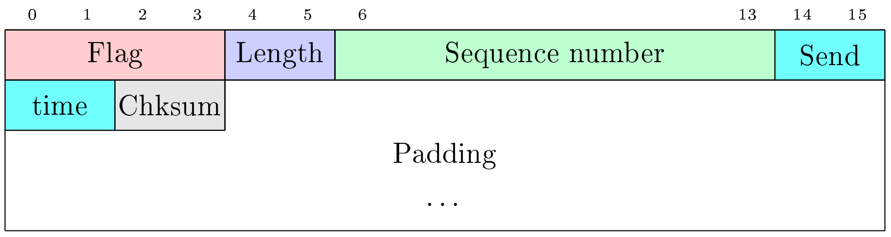


Figure 3.10: Transmission protocol header structure

The size of the padding data and a time limit for each test can be defined through command line arguments. The application was designed to be as efficient as possible and it was compiled with `-O3 -fno-strict-aliasing` options of GNU C Compiler (GCC).

The TCP/IP protocol was tested with three different configurations, were the first one is the standard TCP configuration and the second has the `TCP_NODELAY` option set. The latter disables the Nagle's algorithm, which attempts to increase bandwidth by concatenating several short messages in one packet. Enabling this option should reduce bandwidth due to the increase in overhead, but delay times should also be shorter (particularly for smaller packets). The `TCP_CORK` option used in the third test has the opposite effect. By only sending a TCP packet when the buffer is full it should be possible to maximize bandwidth (for small packet sizes), but with higher delay times.

The POSIX shared memory mechanism was tested with two different configurations. The original configuration reserves just enough memory to place a single message for each direction, and then POSIX semaphores are used to perform concurrency control between processes. But this method doesn't use buffering, putting it at a disadvantage when compared to the other IPC mechanisms. While the transmitting process is placing data on the memory, the receiving process has to wait for the operation to finish. The opposite is true when the receiving process is reading from that same portion of memory, thus not allowing an efficient use of CPU resources. An additional module was built to counteract this through the use of transmission buffers. It works by allocating N message "slots" for each direction, which are then filled sequentially and cyclically. Concurrency control is done by the same semaphores as in the original configuration, except they now count up to N.

Buffered Shared memory and Message queues require the user to manually specify the size of their buffers. To achieve optimum performance, several tests were made to compare the impact of buffer size (measured in number of messages) in both bandwidth and delay. Figures 3.11, 3.12 and 3.13 show the results for packets sizes of 128, 1024 and 16384 bytes, respectively.

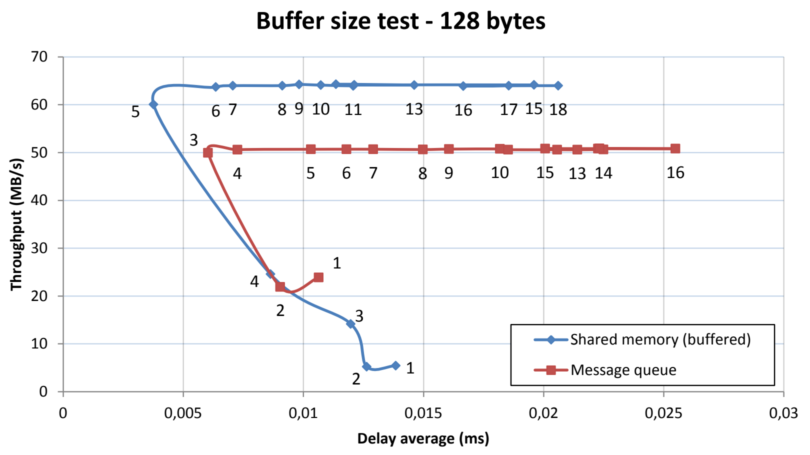


Figure 3.11: Buffer size test - 128 bytes

From these results, a conservative buffer size of 6 was chosen for shared memory and 4 for message queues. Most other mechanisms allow the modification of buffer sizes as well, but they were not tested due to time limitations and because doing so would involve the change of system-wide constants (thus affecting other applications).

Tables 3.2 and 3.3 shows the overall results. The CPU usage values shown in the Table 3.3 are approximate and they were read from the `top` command. The standard deviation and mean delay times were calculated from an implementation[4] of the algorithm described in [5] and [15]. The delay time refers to the time interval between the generation of the packet and the time it was processed in the receiving end.

Note that the word *packet* on the table refers to the internal packet struc-

[4]`http://www.johndcook.com/standard_deviation.html`

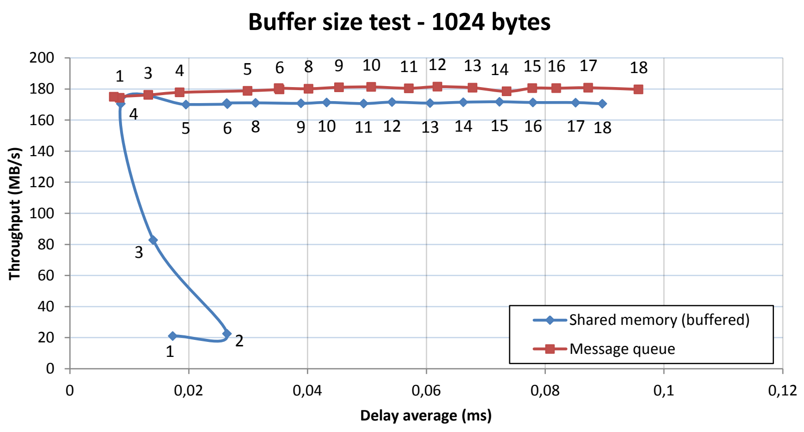


Figure 3.12: Buffer size test - 1024 bytes

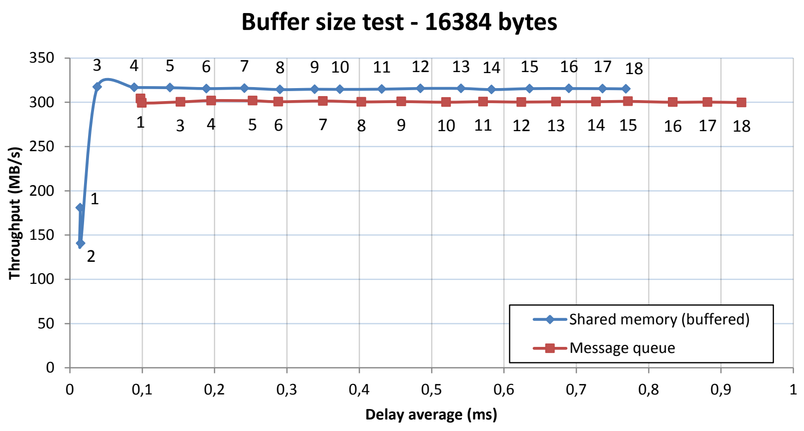


Figure 3.13: Buffer size test - 16384 bytes

ture shown on Figure 3.10, and not the packet structure being used by the protocol of the IPC mechanism.

The delay values were taken from how long a given packet took to reach its destination. These values were calculated during the throughput tests. Due to delays introduced by buffers on the IPC mechanisms, these may not be very representative of a question-reply scenario. Ideally, these values could be more useful if they were obtained as one each end of the line waits for one message before sending its own message and so on.

As expected, network domain sockets without error correction and flow control (UDP and level 2 raw sockets) were the only mechanisms that would require further development to adapt them for local IPC. The presence of out-of-order packets and packets with errors can probably be attributed to the high CPU usage. The frequent system calls from the sending process

	Packet size	Throughput (MB/s)	Delay: Mean time (ms)	Delay: Std. Deviation (ms)	Packet error rate	Out-of-Order packet rate
TCP/IP socket	128 B	48,63	0,64	25,31	0%	0%
	1024 B	167,74	4,00	62,98	0%	0%
	16384 B	269,95	4,03	63.27	0%	0%
TCP/IP socket (w/ `TCP_NODELAY`)	128 B	23,91	0.01	3,64	0%	0%
	1024 B	168,27	3,98	62,82	0%	0%
	16384 B	270,30	4,01	63,14	0%	0%
TCP/IP socket (w/ `TCP_CORK`)	128 B	48,39	3,25	56,73	0%	0%
	1024 B	168,08	3,93	62,48	0%	0%
	16384 B	270,87	4,00	63,03	0%	0%
UDP/IP socket	128 B	26,47	0,01	3,22	0,47%	0,01%
	1024 B	167,90	0,49	22,17	50,10%	24,78%
	16384 B	300,17	0,31	17,66	86,43%	13,54%
Raw L2 socket	128 B	47,70	0,97	31,03	1,98%	1,20%
	1024 B	176,68	0,51	22,59	61,63%	30,63%
	16384 B	307,75	0,31	17,52	88,80%	11,18%
UNIX socket	128 B	42,37	0,53	22,98	0%	0%
	1024 B	172,54	0,31	17,67	0%	0%
	16384 B	299,55	0,23	15,23	0%	0%
Shared memory	128 B	5,42	0,01	3,22	0%	0%
	1024 B	29,29	0,01	3,45	0%	0%
	16384 B	217,41	0,01	2,68	0%	0%
Shared memory (buffered)	128 B	63,81	0.01	2.66	0%	0%
	1024 B	169,66	0.03	5.13	0%	0%
	16384 B	317,37	0.24	15.44	0%	0%
FIFO queue	128 B	41,75	1,24	35,17	0%	0%
	1024 B	166,25	0,27	16,31	0%	0%
	16384 B	295,66	0,20	13,98	0%	0%
Message queue	128 B	50,51	0.01	3.21	0%	0%
	1024 B	178,49	0.02	4.90	0%	0%
	16384 B	301,65	0.25	15.84	0%	0%

Table 3.2: Internal IPC test results (1 of 2)

requires a lot of CPU time from the kernel, thus not leaving enough of it to the receiving process to extract packets from the UDP buffer in time. This causes the buffer to overflow, thus dropping packets and introducing errors.

Overall, buffered Shared memory stood out as the highest-performing mechanism from these results. It provided the highest overall throughput

	Packet size	Transmission CPU usage	Reception CPU usage
TCP/IP socket	128 B	40%	100%
	1024 B	8%	100%
	16384 B	2%	100%
TCP/IP socket (w/ `TCP_NODELAY`)	128 B	40%	100%
	1024 B	8%	100%
	16384 B	2%	100%
TCP/IP socket (w/ `TCP_CORK`)	128 B	40%	100%
	1024 B	8%	100%
	16384 B	2%	100%
UDP/IP socket	128 B	100%	80%
	1024 B	100%	90%
	16384 B	100%	95%
Raw L2 socket	128 B	100%	95%
	1024 B	100%	90%
	16384 B	100%	95%
UNIX socket	128 B	40%	100%
	1024 B	8%	100%
	16384 B	0%	100%
Shared memory	128 B	35%	40%
	1024 B	30%	50%
	16384 B	10%	75%
Shared memory (buffered)	128 B	95%	96%
	1024 B	54%	96%
	16384 B	14%	98%
FIFO queue	128 B	45%	100%
	1024 B	10%	100%
	16384 B	5%	100%
Message queue	128 B	96%	96%
	1024 B	56%	98%
	16384 B	18%	98%

Table 3.3: Internal IPC test results (2 of 2)

(except for packet sizes of 1024 bytes) with very low values of delay and standard deviation. It also produced relatively predictable results, due to the small standard deviation of the delay times.

Security concerns

Due to the intra-host nature of local IPC, data transfer between processes should be relatively safe against malicious users (when compared to remote IPC). Nevertheless, some security concerns can be identified, such as:

- Some mechanisms might be susceptible to eavesdropping and data corruption from processes running on the same machine (e.g. FIFO pipe and shared memory).
- When using shared memory, an application might ignore concurrency access rules and overwrite portions of the shared memory.
- Also when using shared memory, a malicious application might interfere with the semaphores used to control access to the data.
- A malicious application might interfere and connect to the original application as it initializes the IPC resource, not allowing it to reach its intended destination.

Some mechanisms, whose method of identification of resources passes through the filesystem, generally allow to set file permissions with the same settings as a normal file. But file system permissions are limited and unsuited for authentication and prevention of unauthorized use. A workaround could be used, consisting in running the applications as another, password-protected user. However, this is not a very desirable solution since it would force the change of the operating system's configuration and introduce complexity to the set-up process. Instead, internal API-level authentication could be applied.

In conclusion, the main security issues associated with local IPC are related to authentication, integrity and confidentiality. The level of security depends on the mechanisms used, the overall trust of the system on which it is being run on and the application's requirements.

Analysis and conclusion

Unfortunately, all the resources which were possible to obtain during the limited time available to the development of this section were either old, incomplete and/or inconclusive for the objectives proposed. Several other resources (not shown here) were obtained, but they were based purely on anecdotal evidence and were often contradictory. The internal application used for testing introduced even more uncertainty and confusion to the decision process. And finally, several important characteristics were not evaluated (e.g. scalability, minimum delay, programming libraries, etc), which means that the final decision will be taken with much less than desirable certainty.

A particular aspect of communication that was not taken into account is the broadcast of data from one process to the others. In this particular case, it should be easy to see that shared memory would probably provide much better scalability, since the data wouldn't have to be replicated for each process. Another noteworthy issue is that byte-stream IPC mechanisms require additional overhead to be able to distinguish discrete messages on the stream (due to the absence of message boundaries). This issue becomes more noticeable as the transmitted messages get smaller.

The only certainty is that choosing between the best performing IPC mechanism is a complex task, mainly due to the large amount of variables that are necessary to evaluate in order to reach an informed decision. On the upside, the development of the testing applications has shown that, with the proper modularization of the API, switching between IPC mechanisms should be relatively simple and quick to do. This might be necessary in the future if better options or new research results become available. Also, new developments on operating system design may render current IPC benchmarks obsolete, which means that this ability to change is indeed very important to the API.

In the end, it was decided that due to significant structural changes from the Linux kernel 2.2 to 3.0, the more trustworthy, future-proof results were likely to be the most recent ones. This means that **(buffered) shared memory** will be used for local IPC communication. There are also several additional advantages for using shared memory: they provide a very high level of scalability, since only two external handlers are needed by process (this also simplifies the process of finding and connecting to specific processes); it is *connectionless* (unlike sockets), thus avoiding the need for additional overhead: and finally the messages transmitted are discrete, avoiding the necessity of a boundary checking mechanism.

Updated - April 2012*: The original results and conclusion obtained from this section did not have in account the effects of buffering. Due to this, Message queues were initially chosen for local IPC, and the API's implementation was based on this mechanism as well. Nevertheless, the performance results from these mechanisms are not very far off from each other, and Message queues possess additional advantages to Shared memory, namely the presence of a message prioritization mechanism (which will be useful for RoutUM and possibly other applications), and their tight integration with the schedulers from real-time operating systems (which, if desired, should be able to produce more predictable results).*

3.4.3 Remote IPC

Opting for a remote IPC communication mechanism poses a significantly different challenge from the one discussed for local communication. This is because, unlike local IPC, the transmission medium isn't static and known. For this project, only ethernet-based (IP) computer networks will be approached, as they represent the very large majority of computer networks.

Only network-based socket IPC communications will be used, and the API should be able to work optimally on any type of IP-based network. This may be the Internet, a Wide Area Network (WAN), Local Area Network (LAN), mobile networks or anything in between.

The main challenge for remote IPC communication comes with the decision of the transport protocol to use. There are only two Transport Layer protocols which have virtually universal support in network systems (including routers with Network Address Translation (NAT)) - TCP and UDP. For an even higher throughput, these protocols could be set up with DiffServ/Quality of Service (QoS) parameters. There are, however, several disadvantages with both protocols, such as:

- The throughput advantage of UDP comes at the cost of absence of error-correction, flow and congestion control, and guaranteed in-order delivery of packets. These features are all required by the API.
- TCP only allows a connection-oriented server/client model, which introduces significant complexity and overhead to the system. This is because each process needs to be able to establish a connection and maintain its state with tenths, or even hundreds of peers;
- TCP may scale poorly to sudden data bursts due to the congestion avoidance mechanisms it uses.
- Due to its connection-oriented nature, TCP doesn't allow the transmission of broadcast data.
- Unlike UDP, TCP is stream-based, thus requiring a message delineation mechanism.

Neither of these two protocols are ideal for the type of connection and throughput requirements of the API. UDP would (theoretically) provide a much higher throughput, lower latency and reduced overhead, while failing to provide some fundamental features to the API. TCP has more overhead and higher latency, but the same mechanisms that may hold back transmission rates in TCP are the same ones which maximize performance on practical, high-latency, bandwidth-limited and/or error-prone networks. Features such as congestion control, error checking and in-order delivery all help in ensuring that all the data can be transmitted correctly over limited and unpredictable networks as fast as possible. In short, there are only two viable options available: using TCP or any other protocol built over UDP that can provide the features needed by the API. For the latter case, several protocols could be used, such as:

- Stream Control Transmission Protocol (SCTP)[5]
- Micro Transport Protocol (MTP or μTP)[6]
- Reliable User Datagram Protocol (RUDP)[7]

[5]`http://tools.ietf.org/html/rfc3286`

[6]`http://www.utorrent.com/help/documentation/utp`

[7]`http://tools.ietf.org/html/rfc1151`

- Transactional Transmission Control Protocol (T/TCP)[8]
- Multipurpose Transaction Protocol (MTP)[9]
- UDP-based Data Transfer Protocol (UDT)[10]

All of these protocols are layered over UDP. Different versions and/or configurations of the Transmission Control Protocol (TCP) protocol are also taken into consideration.

Due to the high complexity involved with the task of opting for a transmission protocol, it would make more sense to test them only *after* the API and the test application are up and running. References such as [17] also suggest that most protocols are optimized to the point where the differences between them are generally not very substantial and highly dependent of the network characteristics. There are many unknown variables in a practical scenario, and the limited time available for this dissertation would simply make it impossible to cover all scenarios and test all configurations for an overall balanced choice. This is because the performance of a protocol is highly dependent on the application's data transmission patterns, number of hosts in the system, the hardware, network layout, etc. In short, whatever test could be done in this limited time would turn out to be incomplete and inconclusive. Because of this, it was decided to use **TCP/IP** (in its default configuration) as a remote IPC mechanism, due to its maturity and the reasons described above.

Computer networks are inherently insecure, so **authentication** and **confidentiality** should be part of the API's implementation. **Data integrity** is also part of the security requirements, but it's already implemented by the TCP protocol.

3.5 Security

Security mechanisms should be entirely optional and used only when justified so, since they introduce overhead and complexity to the system. The methods employed should be proportional to the level of trust of the users involved in the system, the IPC mechanism, the transmission medium and the application's requirements.

At an early stage, only the two most essential security options will be available in the API: **authentication** and **confidentiality**. These should be able to be turned on/off on a communication context basis.

[8]`http://tools.ietf.org/html/rfc1644`
[9]`http://www.dataexpedition.com/MTP/`
[10]`http://udt.sourceforge.net`

3.5.1 Central Server Access

Since we need to use a central server to coordinate all processes in the system, it would make sense, from a security perspective, to demand both authentication and confidentiality on connections established from processes. To achieve this, a root server certificate will be created and contained within every instance of the API.

3.5.2 Inter-Process Access

It would be a viable option to implement a Public Key Infrastructure (PKI) to coordinate security for all processes in the system. If required, a symmetrical session key could be securely exchanged after authentication so as to provide confidentiality (and authentication) at a lower performance penalty.

Authentication is applied on a per-process basis and each instance should have its own certificate. This way, it is possible to reduce the likelihood that a malicious application will be able to connect to the system without being noticed.

3.5.3 Implementation

The OpenSSL[11] library should provide all the cryptographic functions needed in C language, as well as the required tools to implement a PKI. With some additional guidelines from the publicly available cipher recommendations from the National Institute of Standards and Technology (NIST),[12] it was decided to use the following algorithms by default:

- **Certificate algorithms -** These algorithms are used to create key pairs and to implement hash functions for digital signatures. From a system management perspective, it is relatively easier to revoke and emit new certificates to each API instance. For this reason, and to increase authentication speed between instances, a smaller key will be used here:

  - Digital signature - keys: RSA (1024 bit for API instances, 2048 bit for the central server);
  - Digital signature - hash algorithm: SHA-2 (256 bit);
  - Padding scheme: PKCS #1 v1.5 PSS.

- **Authentication and session establishment algorithms -** These are used for authentication of processes and session key negotiation. Following the same performance-security tradeoff as before, the key

[11] `http://www.openssl.org/`
[12] `http://www.nist.gov/manuscript-publication-search.cfm?pub_id=903633`

size used for authentication between API instances is smaller than authentication between them and the central server:

  - Key establishment: Diffie-Hellman (1024 bit for API instances, 2048 bit for the central server);
  - Protocol: Secure Sockets Layer (SSL).

- **Session algorithms -** Session keys are relatively short-lived, being only as valid as the connection itself. There is an important tradeoff to consider here: whether it is more important to guarantee confidentiality or authenticity. The latter can be achieved either through Message Authentication Codes (MACs) or through the session key (since only the two peers involved have knowledge of it). MACs are orders of magnitude slower than symmetric-key encryption, so the latter becomes a more attractive proposition when the API's priorities are taken into consideration. Assuming that authenticity is more important than confidentiality, a relatively small key size should be enough to guarantee it, while providing better performance than larger key sizes:

  - Symmetric key cipher: AES-128;
  - Hash function: SHA-2 (224 bit).

These algorithms and key sizes should provide an adequate compromise between performance and security. They are to be used only as default values. If necessary, they should be able to be changed through a configuration file, for instance.

Authenticity can be applied either on a per-connection or a per-message basis. The latter involves the use of digital signatures for each message. Since a symmetric session cipher is an order of magnitude slower than any digital signature checking algorithm, it would make more sense from a performance perspective to use it whenever per-packet authenticity is necessary, even if confidentiality isn't required.

The system administrator should have the options to enable or disable authentication and confidentiality on a per-context and per-IPC mechanism basis.

The use of IPsec was considered, but discarded due to its high overhead and incompatibility with several routers.

3.5.4 Further Development

The solution described in this section should be adequate for the majority of use cases. Nevertheless, further developments from a security perspective could include features such as data tampering and non-repudiation prevention mechanisms, support of Access Control Lists (ACLs) for application-specific features, individual settings for peers, etc.

3.6 Resource Identification

At least two types of Unique Identifier (UIDs) are required by the system:

- **API instance UID** - This is required to uniquely identify each and every instance of the API in the system.
- **Operating system instance UID** - This is required to enable an API instance to decide if it should use a local or remote IPC mechanism. Note that (ideally) it is important to uniquely distinguish between operating systems and not the physical machine, since the former could be running in a virtual machine.

The **API instance UID** will be automatically attributed when a peer is registered in the database. This UID will then be passed as an argument to the initialization function of the peer's API.

At the time of writing, there isn't a method that guarantees the automatic generation of an 100% unique and trustworthy **Operating system instance UID**. For instance, network MAC address can be spoofed, BIOS memory doesn't contain a big enough pool of different values and OSs do not possess unique identifiers. Nevertheless, one of the most reliable and common methods used to uniquely identify a machine is through the Medium Access Control (MAC) address of the Network Interface Controller (NIC), so this method ought to be used by the API as well. In theory, the MAC address should also allow the distinction between Operating Systems (OSs) running in virtual machines, since the virtualization layer typically emulates network interfaces as well (along with unique MAC addresses).

In the context of this dissertation, the API instance UID will simply be referred as “peer ID”, and the operating system instance UID as “host ID”.

3.7 Central Server

Figure 3.14 depicts the minimum required functionality that should be provided by the central server.

The central server is used to store and manage information about the peers and contexts in the system. It is constituted by a database and a front-end application, whose processes in the system will connect to. The system's configuration includes the following information:

- A list of communication contexts, including:
  - Common name;
  - List of peers;
  - Security settings (authentication and confidentiality).
- A list of peers, including:

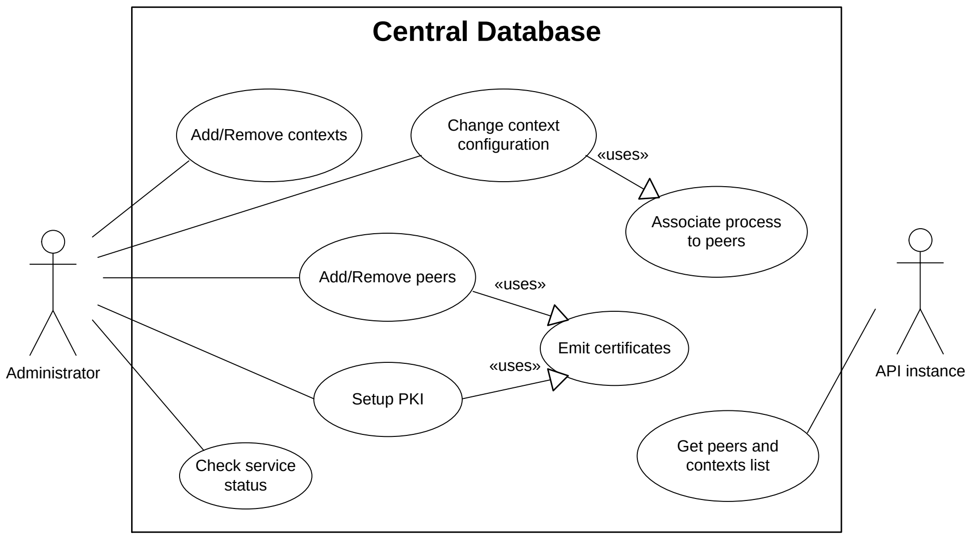


Figure 3.14: Use cases diagram for the central server

- Process UID;
- Host UID;
- Common name;
- List of communication interfaces and their respective configurations;
- The X.509 certificate;
- Whether of not it is allowed to connect (certificate revocation list).

An API instance can request a list of peers and their respective contexts at any time. This list contains only the peers it is allowed to connect to.

Chapter 4

Implementation

A software prototype was developed to fulfill the requirements of the referred API. The resulting project is (mostly) based on the knowledge and specifications discussed on the previous chapters, and it was named RoutUM's Distributed Computing (RDC) API.

4.1 Tools

The RDC API was compiled and tested under Ubuntu 11.10 (32-bit Linux kernel v3.0.0-15). The following tools were used in this project:

- GNU C Compiler (v4.6.1)
- GNU C Debugger GDB (v7.3)
- GNU Make (v3.81)
- Valgrind (tool for memory debugging, v.3.6.0)
- SQLite 3 development libraries (v3.7.2)
- OpenSSL C development libraries (v1.0.0e)
- Doxygen (documentation generator, v1.7.1)
- Geany (lightweight C IDE, v0.19.1)
- SQLite Database Browser (v1.3)

4.2 Architecture Overview

The API's project was split into the following sub-projects:

- **Database management API** - Allows the creation and manipulation of the central database;
- **Database management application** - Implements the database management API;
- **Server daemon** - The daemon process responsible for the authentication and handling of requests from peers;

- **Peer API** - The core API, used to communicate between peers in the system;
- **Peer API test application** - Implements the Peer API.

The relationships between these sub-projects are represented in Figure 4.1. This scheme contrasts slightly with the one established in Figure 3.1. Some changes were made to the architecture during its implementation, because they introduce several advantages to the system. These changes will be discussed later in this chapter.

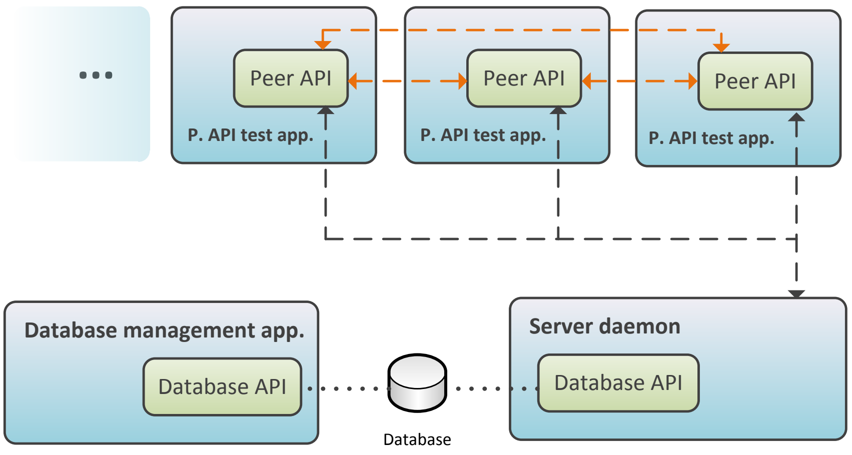


Figure 4.1: Implementation overview

The central server functions are now carried out by two distinct applications: a lightweight daemon and a database front-end application. A Structured Query Language (SQL) database was chosen to store all data on the central server. It contains the up-to-date information of every context and peer registered in the system. A peer can only connect to the system if it has an entry in the database.

The **Database management API** contains all the functions needed to establish a connection to a Database Management System (DBMS) (or similar software), create a new database, read and manipulate its contents. Having these functions in API form makes it possible to easily integrate them in virtually any application. In this particular project, a dedicated **database management application** was built to implement this API, allowing the manipulation of the database from a CLI. But this API could as well be implemented in any other application. As an example, it could be incorporated in the peer API test application, thus allowing a peer to add itself to the database, for instance.

The **server daemon** establishes an interface between the database and the peers. Since it works purely as a system service and does not require interactivity with a user, it was decided to develop it as a *deamon*. When a peer wishes to connect to the system, it must first authenticate itself with this daemon. If the authentication has been successfully performed, the daemon retrieves a list of peers (and contexts) from the database and sends it to the peer. The daemon itself is stateless and all events are reported to syslog.

Security

The security architecture proposed during research had to be dropped due to implementation issues. The main reason is related to the fact that the OpenSSL library supports only TCP/IP sockets and standard file descriptors[1] for PKI-based authentication. A manual implementation of the SSL protocol would be possible, but it would require far too much work and testing for it to become a viable proposition for this dissertation.

The solution adopted was to remove the PKI-based authentication and use key files instead. When a new peer is added to the system (through the database management application), a random keypair is generated. The public key from that pair is stored in the database and the private key in a file. This file should then be used by a peer API instance so it can authenticate itself.

Had the previous architecture been kept, it would be necessary to add another authentication method for the IPC mechanisms not supported by OpenSSL. This would introduce significant architectural and code fragmentation to the system. The new solution guarantees that the code and architecture are kept the same regardless of the IPC mechanism being used.

The cryptographic algorithms and key sizes that were used are similar to the ones specified in Section 3.5.3. In other words, they are:

- **Keypair algorithms**
  - Key generation and digital signature: RSA (1024 bit);
  - Digital signature - hash algorithm: SHA-2 (256 bit);
  - Padding scheme: PKCS #1 v1.5 PSS.
- **Session key algorithms**
  - Symmetric key cipher: AES-128 CBC;
  - Hash function: SHA-1 (224 bit).

Peer initialization

Figure 4.2 shows the sequence of events that occur when a peer joins the system.

One of the arguments of the initialization function (named `rdc_init()`) is a `struct` with the configuration of the peer API instance. It includes the

[1] `http://www.openssl.org/docs/crypto/bio.html`

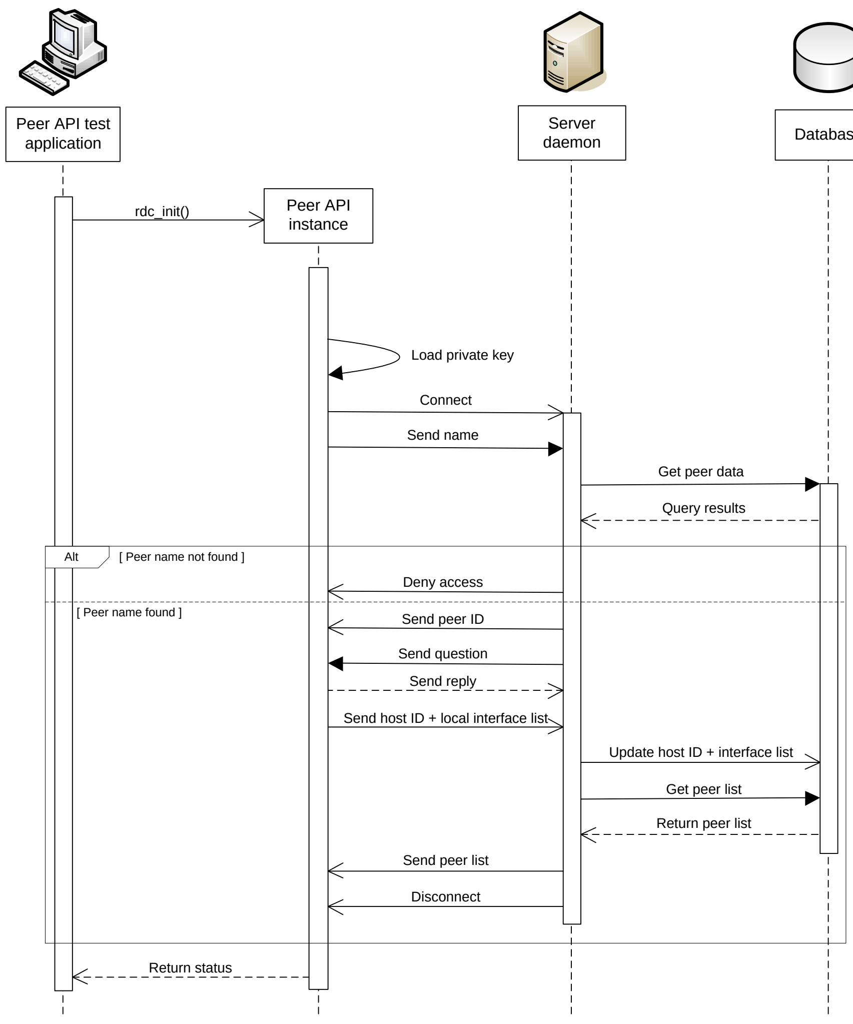


Figure 4.2: Peer initialization sequence

name of the peer, the configuration of the local IPC interfaces and the private key file. During initialization the peer is authenticated by the server and its IPC interface configuration and host ID are sent as well. This information is then stored in the database. Finally, a list with all the peers (and contexts) which this API instance is allowed to connect to is sent by the daemon.

The peers that are already connected to the system don't know which ones are active or not. The only way to discover it is to attempt to connect to them with the latest known interface configuration from that peer. On the other hand, when a new peer connects to the system, it tries to send an updated list of its own local IPC interfaces and host ID to all the other known peers (it is allowed to connect to) on the network. This way it is possible to ensure that every peer on the network has updated information about each other. Figure 4.3 shows the corresponding sequence diagram of this transaction.

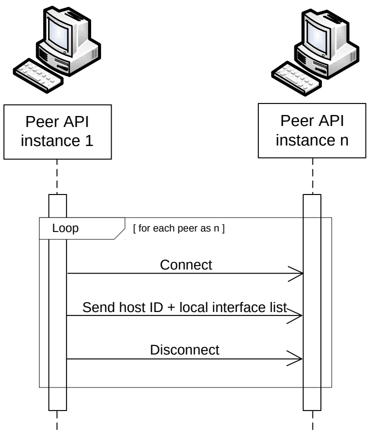


Figure 4.3: Peer-to-peer interface information update sequence

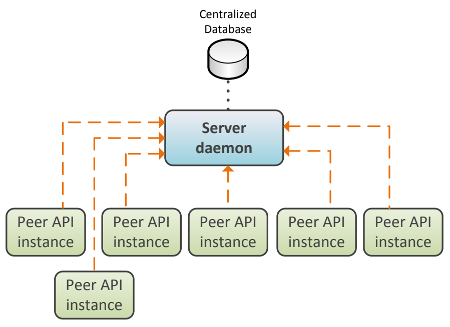
Redundancy

In any fully centralized system (Figure 4.4(a)), a single failure can stop the whole system from working properly. One of main motivations behind the architectural changes during the implementation phase was the introduction of an increased level of redundancy, scalability and fault tolerance into the system.

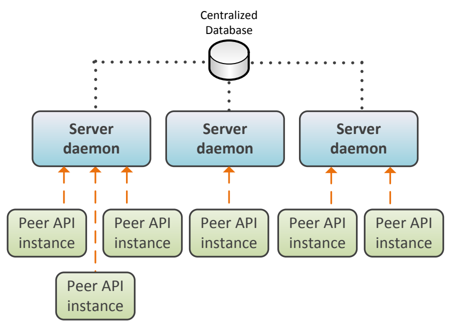
As the central server daemon is stateless, it is possible to use several instances of it in the same system. So two or more daemons can connect to the same central database (Figure 4.4(b)) or an instance of a database (Figure 4.4(c)) which is automatically replicated by the DBMS in charge. Figure 4.4 illustrates some scenarios enabled by this architectural change.

In a system with multiple server daemons, the peers can connect to them in two ways: either by using the IP address of a specific server, or through DNS domain names. DNS records can contain more than one IP address for each domain. This way, the user just needs to specify an address in the peer API's configuration, and it should randomly choose one address from the IP list given by the DNS.

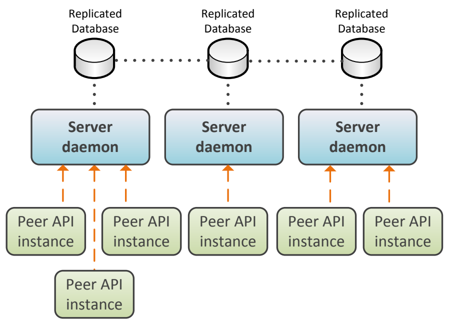
Since it is possible to split the work over multiple machines, another advantage of this decentralized approach is an increase in performance on large systems (i.e., an increase in scalability).



(a) Centralized database and server daemon



(b) Centralized database with server daemon redundancy



(c) Full database and server daemon redundancy

Figure 4.4: Architectural redundancy scenarios

4.3 Implementation Details

As previously stated, one of the main requirements in this project is code efficiency. In practice, most of the decisions taken during the architectural and implementation phases of the software were influenced by this requirement, including ones that imply a performance/code complexity tradeoff. Some of the steps taken to ensure high code efficiency include:

- Usage of efficient algorithms and data structures whenever possible;
- Usage of preprocessor directives to hide debug code;
- A low amount of function calls, threads, mutexes, data type casts, and memory allocations;
- Parallelization of the code, ensuring an optimal use of all available resources (namely, multi-core processing);
- On some instances, GCC's atomic locks[2] were used for specific variables (due to their small overhead when compared to mutexes);

[2]`http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html#Atomic-Builtins`

- CPU cache misses were taken into account when designing data structures.

Note that the C language doesn't contain classes or hereditary concepts usually found on higher-level languages. This makes for a fuzzier modularization of the code. Nevertheless, special care was taken to simplify module replacement.

4.3.1 Project Structure

The software project is made up of several folders and files, namely:

- `./` - Project root folder.
  - **makefile** - Configuration file for Make.
  - **doxyfile** - Settings file for Doxygen.
- `./include/` - Include files for the peer and database management API libraries
  - **db.h** - Database management API include file.
  - **rdc.h** - Peer API include file.
- `./lib/` - Peer and database management API libraries
  - **librdc.a** and **librdc.so.1.0.1** - Peer API include libraries.
  - **librdcdb.a** and **librdcdb.so.1.0.1** - Database management API libraries.
- `./src/rdc/api/` - Peer API root folder.
  - **rdc.c** and **rdc.h** - Main peer API interface and functions.
  - **ipc.c** and **ipc.h** - IPC interface implementation interface.
  - **evt_queue.c** and **evt_queue.h** - An implementation of a FIFO event queue.
  - **btree.c** and **btree.h** - A generic implementation of an AVL binary tree.
  - **api_dtypes.c** and **api_dtypes.h** - Data type definitions and local data management functions.
- `./src/rdc/api/ipc/` - IPC interface implementation for peer API.
  - **tcpip.c** and **tcpip.h** - TCP/IP IPC interface implementation.
  - **mqueue.c** and **mqueue.h** - Message queue IPC interface implementation.
- `./src/rdc/shared/` - Common files.
  - **crypto.c** and **crypto.h** - Cryptography related functions.

  - **dtypes.c** and **dtypes.h** - Global data types.

- `./src/rdc/srv/` - Server-side applications

  - **cf_parser.c** and **cf_parser.h** - Configuration file parser.
  - **const.h** - Server-side global constants.
  - **db.c** and **db.h** - Database management API.
  - **rdcd.c** - RDC server daemon.
  - **rdcm.c** - Database manager application.

This chapter contains several file diagrams[3] to describe the contents and dependencies of the source files. In these diagrams, the red color is used to represent APIs, blue for internal files used by the APIs, orange is used to represent executables and white to reference files from other directories. The content of all the project's headers can be found in Appendix A.

Figure 4.5 shows the file diagram for common files (`./src/rdc/shared/`) in the API.

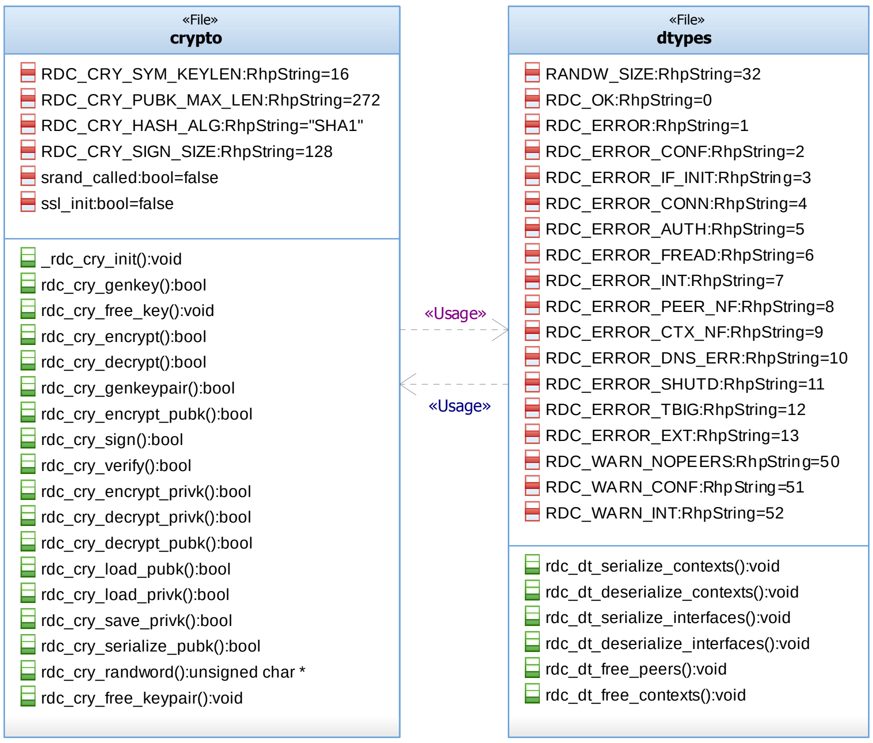

«File»
crypto
RDC_CRY_SYM_KEYLEN:RhpString=16
RDC_CRY_PUBK_MAX_LEN:RhpString=272
RDC_CRY_HASH_ALG:RhpString="SHA1"
RDC_CRY_SIGN_SIZE:RhpString=128
srand_called:bool=false
ssl_init:bool=false
_rdc_cry_init():void
rdc_cry_genkey():bool
rdc_cry_free_key():void
rdc_cry_encrypt():bool
rdc_cry_decrypt():bool
rdc_cry_genkeypair():bool
rdc_cry_encrypt_pubk():bool
rdc_cry_sign():bool
rdc_cry_verify():bool
rdc_cry_encrypt_privk():bool
rdc_cry_decrypt_privk():bool
rdc_cry_decrypt_pubk():bool
rdc_cry_load_pubk():bool
rdc_cry_load_privk():bool
rdc_cry_save_privk():bool
rdc_cry_serialize_pubk():bool
rdc_cry_randword():unsigned char *
rdc_cry_free_keypair():void
«Usage»
«Usage»
«File»
dtypes
RANDW_SIZE:RhpString=32
RDC_OK:RhpString=0
RDC_ERROR:RhpString=1
RDC_ERROR_CONF:RhpString=2
RDC_ERROR_IF_INIT:RhpString=3
RDC_ERROR_CONN:RhpString=4
RDC_ERROR_AUTH:RhpString=5
RDC_ERROR_FREAD:RhpString=6
RDC_ERROR_INT:RhpString=7
RDC_ERROR_PEER_NF:RhpString=8
RDC_ERROR_CTX_NF:RhpString=9
RDC_ERROR_DNS_ERR:RhpString=10
RDC_ERROR_SHUTD:RhpString=11
RDC_ERROR_TBIG:RhpString=12
RDC_ERROR_EXT:RhpString=13
RDC_WARN_NOPEERS:RhpString=50
RDC_WARN_CONF:RhpString=51
RDC_WARN_INT:RhpString=52
rdc_dt_serialize_contexts():void
rdc_dt_deserialize_contexts():void
rdc_dt_serialize_interfaces():void
rdc_dt_deserialize_interfaces():void
rdc_dt_free_peers():void
rdc_dt_free_contexts():void

Figure 4.5: File diagram for shared files of the RDC API

[3]A file diagram is a proposed equivalent of an Unified Modeling Language (UML) Class Diagram for functional languages. See `http://public.dhe.ibm.com/common/ssi/rep_wh/n/RAW14058USEN/RAW14058USEN.PDF` for more details.

4.3.2 Limitations

By examining and defining realistic limits on the maximum number of peers and contexts in the system, it may be possible to reduce the overhead of the messages exchanged between the peers (as their identifiers may be shorter) and even to save space on the central server's database. The chosen limits were picked from a multiples of 8 binary digits. The maximum allowed number of peers was set at 2^{24}, or 16.777.216. Before the development of the API, a target was defined in the order of tens of thousands of peers supported. An initial limit of 2^{16}, or 65.536 was chosen, but due to the way sequence numbers are stored in most databases (always increases when a new peer is added and never decreases), the next highest limit was picked just for safety. The maximum number of contexts on a system should be much lower than the number of peers, so a limit of 2^{16}, or 65.536 contexts, was set.

4.3.3 Database

The database for the API's prototype is based on SQLite. The main reason behind this decision is its small size. Proper DBMSs are also cumbersome to deploy and configure, and they don't bring any significant advantages during the prototype phase. On a real working scenario though, a DBMS can introduce several important advantages to the system, such as remote access, increased security and database replication.

Figure 4.6 represents the database structure used in this system. It is based on the requirements identified in Section 3.7

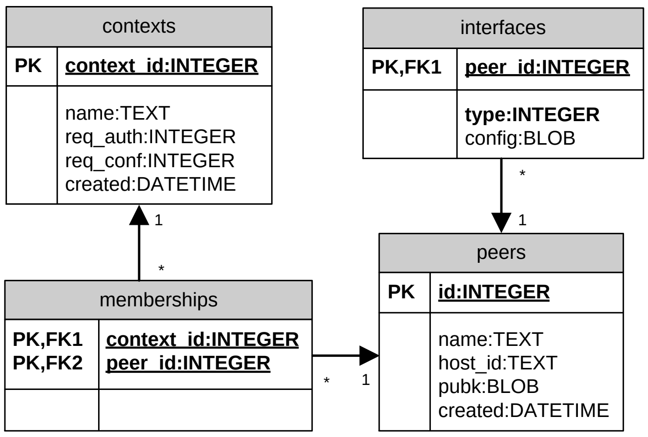


Figure 4.6: Database structure

The table **contexts** stores all the information related to simulation contexts. Each context has the option to individually enable or disable authentication and/or confidentiality as needed.

The table **peers** stores all information related to a peer instance. The public key of a peer is stored in Base64 format. The `host_id` is a value which uniquely identifies the host where the peer is located in. This value is used by the peers to select a local or remote IPC interface, depending on whether the host ID of the remote peer is the same as the local one or not, respectively.

Each row on the table **interfaces** stores the information of an IPC interface of a peer. This information is used by a peer to know which IPC interfaces are available on the other peers and how it can reach them.

The table **memberships** establishes the (many-to-many) relationships between contexts and peers.

All contexts and peers can be uniquely identified through an ID number *or* a unique name. The ID number is primarily used internally in the API for faster access and avoid other minor issues related with the use of strings. The name is primarily used as a convenient and human-readable way to identify a peer or context.

Note that an additional field might be necessary on the tables `peers` and `interfaces` due to the way some databases store binary blobs. Some DBMSs don't store the size of the blob, so an additional field is needed for it.

4.3.4 Transmission and Authentication Protocols

All messages exchanged between peers follow a common format, as shown in Figure 4.7:

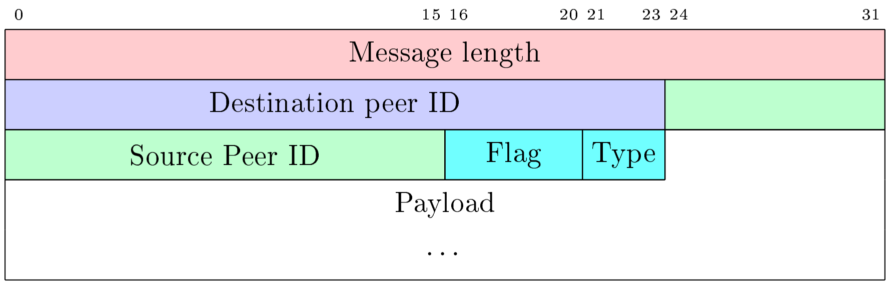


Figure 4.7: Peer-to-peer transmission protocol structure

The **flag** field was inserted so that the final sequence is a multiple of 8. This is a fixed sequence of bits (`10110`), and can be used to, for instance, check if the version of the RDC API is compatible between different peers.

The fields **message length**, **destination peer ID** and **Source peer ID** are optional. They are only used if the IPC mechanism justifies so. For instance, the message length field is present in TCP/IP connections, but not in a message queue IPC. This is because the first is stream-based with no message boundaries, while the latter isn't. Similarly, TCP/IP connections don't use the destination and source ID fields, because these never change during the lifetime of the connection. With message queues, the source ID may be different, so this field is present.

The field **type** and the payload structure can assume several values depending on the type of message it contains. These can be:

1. **Regular messages** - These are messages exchanged by the application that implements the RDC API. The payload structure can be seen in Figure 4.8. The Message Authentication Code (MAC) field is optional and only present if the message is being sent from or to a context that requires it;

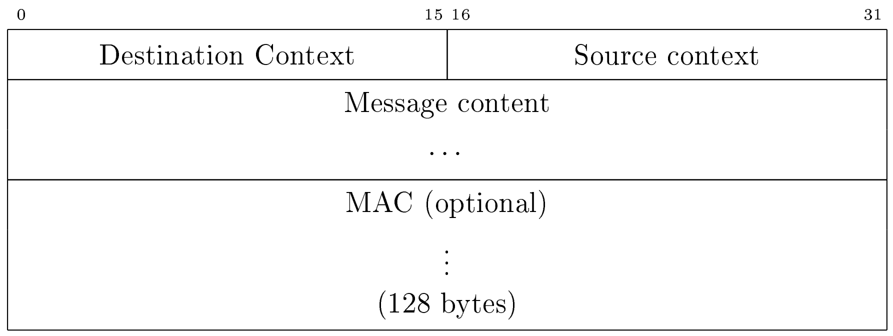


Figure 4.8: Payload structure of a regular/urgent message

2. **Urgent messages** - The payload structure is identical to regular messages, but they are given higher priority inside the peer API;

3. **Interface update notification** - These messages are sent by a recently connected peer to notify other peers about their interface and host ID information. Its payload structure can be seen on Figure 4.9;

4. **Authentication messages** - These messages are used to authenticate and establish a session key between peers. Authentication is a four step process and the payload for each message can be seen in Figure 4.10. These steps must be executed in order. Any failure to complete a step successfully results on a single-byte return message (containing the character "n");

As seen in Figure 4.10, a session key is exchanged between peers during the authentication process. This key is valid for the lifetime of the connection. If a message has a context with confidentiality enabled as a source or destination, the session key will be used to cipher the message.

Authentication between two peers is performed if they both belong to a context with confidentiality and/or authentication enabled. The authentication process is then initiated as soon as both peers have established a

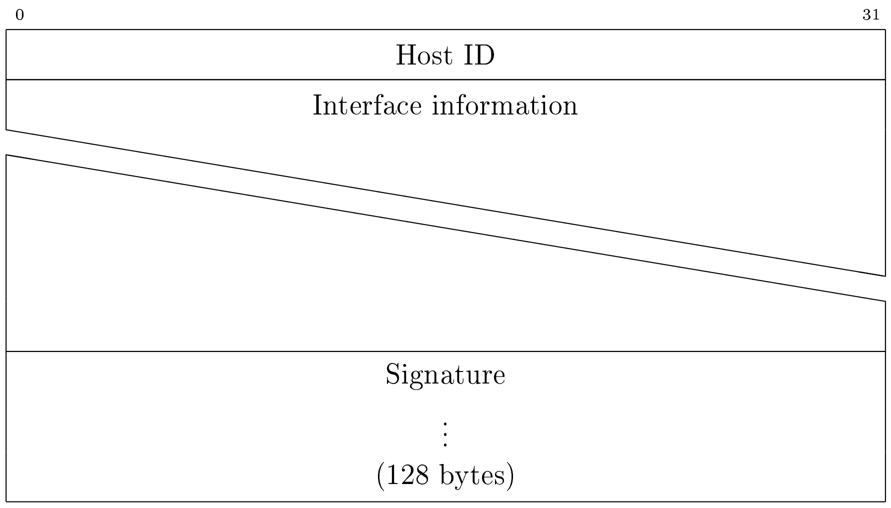


Figure 4.9: Payload structure of an interface update message

connection. If they both initiate authentication at the same time, the peer with the highest ID becomes the leader in the process.

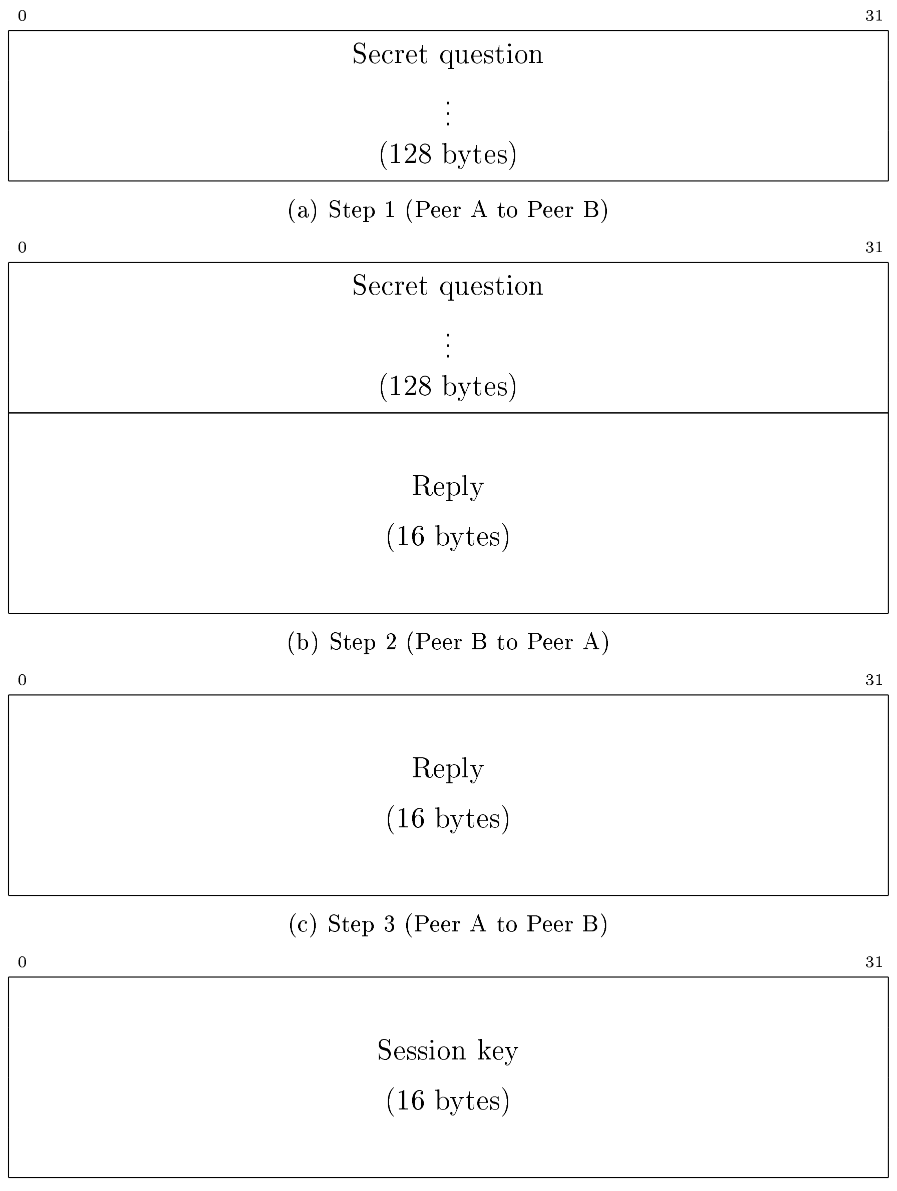
Note that the transmission protocol may be altered due to some very specific needs of an IPC interface, or as an improvement to the transmission characteristics. But these modifications ought to be implemented inside the files that contain the interface implementation itself. Some examples of these needs may include sequence numbering, error checking and/or correction and the incorporation of synchronization flags.

4.3.5 Database Management API

In this project, an SQLite implementation was used for the database API, due to reasons discussed in Section 4.2. Appendix A.1 shows the contents of the API's header file. It is possible to implement another DBMS, an LDAP directory, or a even a distributed database with minimal change of code (apart from the implementation itself). Besides the modification/replacement of the implementation file, one only has to change the data structures `rdc_db_t` and `rdc_db_cfg`.

Since SQLite does not store the size of binary blobs correctly, one additional field was added to the table `peers`, named `pubk_len`, and another to the table `interfaces`, named `config_len`. This is used to store the size of the binary blobs that contains the public key and the configuration data of the IPC interfaces of the peers, respectively.

The use of an LDAP directory was also considered due to its higher read performance, but ultimately dropped. The main reason for this is the high number of writes to the database and the absence of complex queries in an LDAP.



(a) Step 1 (Peer A to Peer B)

(b) Step 2 (Peer B to Peer A)

(c) Step 3 (Peer A to Peer B)

(d) Step 4 (Peer B to Peer A)

Figure 4.10: Payload structure of authentication messages

4.3.6 Database Management Application

The database management application (named `rdcm`) allows a user to access all the functions of the database management API through a traditional CLI interface. Figure 4.11 shows the file diagram for the database management application (`rdcm`) and server daemon (`rdcd`, which will be discussed in the next section). Configuration variables for the database API are read from a configuration file (by `rdcm`).

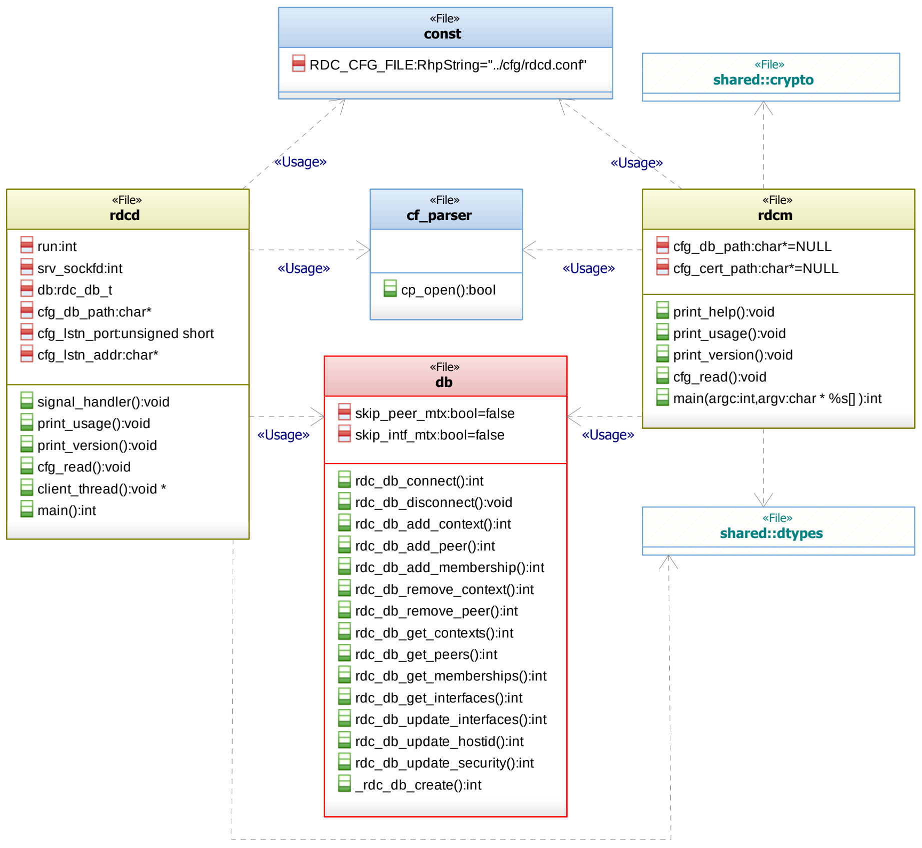


Figure 4.11: File diagram of the database management and server daemon files of the RDC API

The following functions are available in the application:

- Create a new database;
  `./rdcm -c`
- Insert a context;
  `./rdcm -A [context_name]`
- Remove an existing context;
  `./rdcm -R [context_name]`
- Edit context security parameters;
  `./rdcm -E [context_name] [use_confidentiality] [use_authentication]`
- Display a list of contexts;
  `./rdcm -L`
- Insert a peer;
  `./rdcm -a [peer_name]`
- Remove an existing peer;

`./rdcm -r [peer_name]`
- Display a list of peers;
  `./rdcm -l`
- Create a peer-context membership;
  `./rdcm -m [context_name] [peer_name]`
- Display a list of peer-context memberships;
  `./rdcm -s`

This application was developed to be script-friendly, so as to be used with automated deployment scripts. All parameter values are entered as arguments in the CLI. The application is non-interactive and never waits for user input.

4.3.7 Central Server's Daemon

The central server's daemon (named `rdcd`), is responsible for the authentication and exchange of RDC network information between peers. Figure 4.11 shows the file diagram for the server daemon.

As the name suggests, this application is not interactive. All events are reported to syslog, and during initialization (before the process is `fork()`'ed) they are also reported to `stdout`. The configuration is read from same configuration file used by `rdcm`.

The peers connect to the server through a regular TCP/IP connection. This daemon is multi-threaded and can serve multiple peers at the same time, following the procedure shown in Figure 4.2.

4.3.8 Peer API

The peer API is responsible to exchange the information between other peers. Its header is formed by eight functions, several definitions and data structures, as seen in Appendix A.2. Figure 4.12 shows its file diagram. The interface's functions are:

- `rdc_init()` - Initializes a new instance of the RDC API;
- `rdc_destroy()` - Finalizes an RDC instance and releases all resources associated with it;
- `rdc_sendmsg()` - Sends a message to a peer;
- `rdc_recv_evt()` - Retrieves an event from the event queue;
- `rdc_free_evt()` - Frees the resources associated with a retrieved event;
- `rdc_get_peers()` - Returns the information of all the peers in the system or only a specific peer;
- `rdc_get_contexts()` - Returns the information of all the contexts in the system or only a specific context;
- `rdc_update_db()` - Forces an update to the local RDC network database.

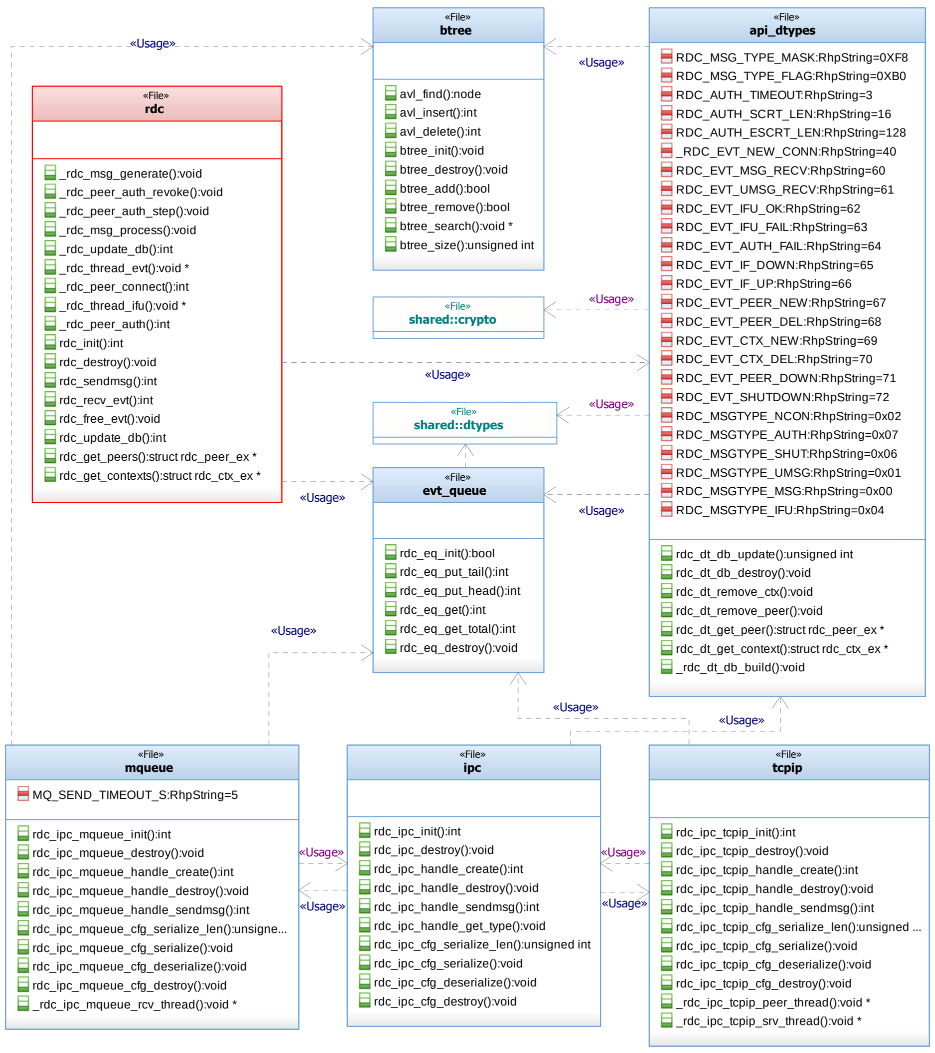

«Usage»
«File»
btree
avl_find():node
avl_insert():int
avl_delete():int
btree_init():void
btree_destroy():void
btree_add():bool
btree_remove():bool
btree_search():void *
btree_size():unsigned int
«File»
api_dtypes
RDC_MSG_TYPE_MASK:RhpString=0XF8
RDC_MSG_TYPE_FLAG:RhpString=0XB0
RDC_AUTH_TIMEOUT:RhpString=3
RDC_AUTH_SCRT_LEN:RhpString=16
RDC_AUTH_ESCRT_LEN:RhpString=128
_RDC_EVT_NEW_CONN:RhpString=40
RDC_EVT_MSG_RECV:RhpString=60
RDC_EVT_UMSG_RECV:RhpString=61
RDC_EVT_IFU_OK:RhpString=62
RDC_EVT_IFU_FAIL:RhpString=63
RDC_EVT_AUTH_FAIL:RhpString=64
RDC_EVT_IF_DOWN:RhpString=65
RDC_EVT_IF_UP:RhpString=66
RDC_EVT_PEER_NEW:RhpString=67
RDC_EVT_PEER_DEL:RhpString=68
RDC_EVT_CTX_NEW:RhpString=69
RDC_EVT_CTX_DEL:RhpString=70
RDC_EVT_PEER_DOWN:RhpString=71
RDC_EVT_SHUTDOWN:RhpString=72
RDC_MSGTYPE_NCON:RhpString=0x02
RDC_MSGTYPE_AUTH:RhpString=0x07
RDC_MSGTYPE_SHUT:RhpString=0x06
RDC_MSGTYPE_UMSG:RhpString=0x01
RDC_MSGTYPE_MSG:RhpString=0x00
RDC_MSGTYPE_IFU:RhpString=0x04
rdc_dt_db_update():unsigned int
rdc_dt_db_destroy():void
rdc_dt_remove_ctx():void
rdc_dt_remove_peer():void
rdc_dt_get_peer():struct rdc_peer_ex *
rdc_dt_get_context():struct rdc_ctx_ex *
_rdc_dt_db_build():void
«File»
rdc
_rdc_msg_generate():void
_rdc_peer_auth_revoke():void
_rdc_peer_auth_step():void
_rdc_msg_process():void
_rdc_update_db():int
_rdc_thread_evt():void *
_rdc_peer_connect():int
_rdc_thread_ifu():void *
_rdc_peer_auth():int
rdc_init():int
rdc_destroy():void
rdc_sendmsg():int
rdc_recv_evt():int
rdc_free_evt():void
rdc_update_db():int
rdc_get_peers():struct rdc_peer_ex *
rdc_get_contexts():struct rdc_ctx_ex *
«Usage»
«File»
shared::crypto
«Usage»
«Usage»
«File»
shared::dtypes
«Usage»
«File»
evt_queue
«Usage»
rdc_eq_init():bool
rdc_eq_put_tail():int
rdc_eq_put_head():int
rdc_eq_get():int
rdc_eq_get_total():int
rdc_eq_destroy():void
«Usage»
«Usage»
«Usage»
«File»
mqueue
MQ_SEND_TIMEOUT_S:RhpString=5
rdc_ipc_mqueue_init():int
rdc_ipc_mqueue_destroy():void
rdc_ipc_mqueue_handle_create():int
rdc_ipc_mqueue_handle_destroy():void
rdc_ipc_mqueue_handle_sendmsg():int
rdc_ipc_mqueue_cfg_serialize_len():unsigne...
rdc_ipc_mqueue_cfg_serialize():void
rdc_ipc_mqueue_cfg_deserialize():void
rdc_ipc_mqueue_cfg_destroy():void
_rdc_ipc_mqueue_rcv_thread():void *
«Usage»
«Usage»
«File»
ipc
rdc_ipc_init():int
rdc_ipc_destroy():void
rdc_ipc_handle_create():int
rdc_ipc_handle_destroy():void
rdc_ipc_handle_sendmsg():int
rdc_ipc_handle_get_type():void
rdc_ipc_cfg_serialize_len():unsigned int
rdc_ipc_cfg_serialize():void
rdc_ipc_cfg_deserialize():void
rdc_ipc_cfg_destroy():void
«Usage»
«Usage»
«File»
tcpip
rdc_ipc_tcpip_init():int
rdc_ipc_tcpip_destroy():void
rdc_ipc_tcpip_handle_create():int
rdc_ipc_tcpip_handle_destroy():void
rdc_ipc_tcpip_handle_sendmsg():int
rdc_ipc_tcpip_cfg_serialize_len():unsigned ...
rdc_ipc_tcpip_cfg_serialize():void
rdc_ipc_tcpip_cfg_deserialize():void
rdc_ipc_tcpip_cfg_destroy():void
_rdc_ipc_tcpip_peer_thread():void *
_rdc_ipc_tcpip_srv_thread():void *

Figure 4.12: File diagram for the peer API

Initialization

The function `rdc_init()` is used to initialize the peer API. It receives a pointer to the API's instance (`struct rdc_t`) and a pointer to the configuration structure (`rdc_cfg_t`). The initialization is a six step procedure.

In the **first step**, the host ID of the machine is retrieved and the configuration structure (passed as an argument to the function) is checked for completeness and errors. This is to avoid unnecessarily allocating and deallocating resources later in the function. The host ID is obtained programmatically (function `gethostid()`), and it can also be obtained through a CLI with the `hostid` command:

```
diogo@ubuntu:~$ hostid
a8c08f4a
```

In the **second step**, the private certificate is loaded from a file. Its path is contained in the configuration structure passed as a function argument.

In the **third step**, the peer connects to the central server (Figure 4.2) to retrieve the list of contexts it is part of and the peers it is allowed to connect to. The connection is based on a TCP/IP socket and the server's address can either be a specific IP, or automatically retrieved from a DNS name.

In the **fourth step**, the event handling thread (`_rdc_thread_evt()`) is initialized (with a default value of 50 elements). The role of this thread will be discussed later in this section.

In the **fifth step**, all local IPC interfaces are initialized. The initialization is deemed successfully if at least one of the interfaces could be initialized correctly. A warning is given if at least one, but not all interfaces have failed to initialize. The interfaces whose configuration struct is `null` are ignored.

In the **sixth and final step**, the interface information update thread (`_rdc_thread_ifu()`) is initialized. The task of this thread is to connect and send the local IPC information to all peers that initialized *a priori*. This information is sent internally through the same code path of a regular message, but with a different message type. A more detailed description will be presented later in this section.

A failure to complete any of these steps successfully causes the function to return an error code after releasing all resources allocated during the function call. More information on errors will be presented later in this section.

Local database

After the peer obtains the RDC network data from the server, it must store this data in a way that allows fast access to it. To achieve this, the local database of available peers is stored in two different types of data structures: a linear linked list and an AVL binary tree. The first type allows the fastest possible access for sequential algorithms. A binary tree (implemented in `btree.h` and `btree.c`) allows a very efficient random access to these peers. The relatively slow update of an AVL binary tree is not an obstacle, since it rarely (if ever) occurs after initialization.

When the application requests a specific peer, the API searches for it in the binary tree. But if it requests a full list of peers, only the first element of the linked list is retrieved (the `next` variable in a peer structure `struct rdc_peer_ex` points to the next element on the linked list).

Contexts are stored and retrieved in the same way, in a separate linked list and binary tree.

Event handling

One of the major design features of the peer API is the event-driven architecture. It consists of an event handling thread (function `_rdc_thread_evt()` in file `rdc.c`) and two event queues (implemented by `evt_queue.c and evt_queue.h`). The first queue is an incoming event queue. Any error, message or any other type of event triggered inside the API is placed in this queue. The event processing thread retrieves elements from this queue and processes them according to their type. Depending on the event type, this thread may need to notify the application of certain events. In these cases, the thread places these events on the second queue - the outgoing event queue. These can then be retrieved by the application at any time through one of the main peer API functions - `rdc_recv_evt()`. Figure 4.13 shows a rough schematic on how the event handling works inside the API.

The data structure `rdc_evt` in the file `evt_queue.h` is used to represent an event. Its contents are:

```
/** @brief Represents an event. */
struct rdc_evt {
    int   type; /// Event type.
    int   arg1; /// First argument.
    int   arg2; /// Second argument.
    int   arg3; /// Third argument.
    void* arg4; /// Fourth argument.
    int   len;  /// Length of arg4 (if applicable).
};
```

The values (event arguments) of the variables in the structure depend on the type of event itself. Table 4.1 shows the full list of event types in the peer API. Note that some events may only be defined for internal use of the API.

Peer-to-peer connection establishment and authentication

A peer will attempt to establish a connection with another on two occasions: when it needs to send its local IPC information update or when the application has requested for a message to be sent to it (through `rdc_sendmsg()`). In the current implementation, this connection is kept alive for the whole lifetime of the API or until it is lost.

There are two different types of interfaces defined in the API: message queues (identified internally as `IFTYPE_MQUEUE`) and TCP/IP sockets (`IFTYPE_TCPIP`). When a new connection is requested, it is first necessary to know which IPC interface to use. The function `rdc_ipc_handle_get_type()` (implemented in the files `ipc.c` and `ipc.h`) returns a list of interfaces by descending order of priority. The algorithm works as follows:

1. Start with an empty interface list;

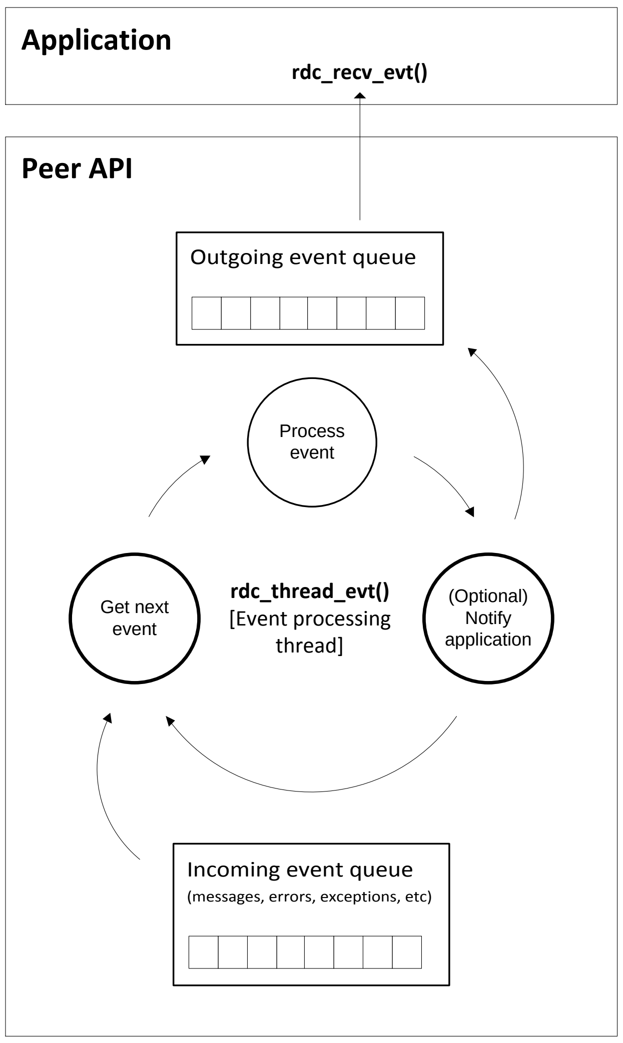


Figure 4.13: Event handling in peer API

2. If the target peer has the same host ID as the instance *and* a valid message queue IPC configuration, then `IFTYPE_MQUEUE` is added as the first element of the list;
3. If the target peer has a valid TCP/IP configuration, then `IFTYPE_TCPIP` is added as the next element on the list (or as first element if previously empty);

After this list is generated, the API attempts to establish a connection through the IPC interface whose type comes first on the list. If the connection attempt fails, it tries the second type on the list, third and so on until the connection is successfully established or the end of the list is reached. In the latter case, the connection attempt is deemed unsuccessful and the call-

Event type	Description	Internal only?
_RDC_EVT_NEW_CONN	New connection attempt received.	Yes
RDC_EVT_MSG_RECV	New message received.	No
RDC_EVT_UMSG_RECV	New urgent message received.	No
RDC_EVT_IFU_OK	Successfully notified a peer of local IPC interfaces.	No
RDC_EVT_IFU_FAIL	Could not notify a peer of a local IPC interfaces.	No
RDC_EVT_AUTH_FAIL	An error occurred while authenticating a peer.	No
RDC_EVT_IF_DOWN	An IPC interface was lost.	No
RDC_EVT_IF_UP	An IPC interface was recovered.	No
RDC_EVT_PEER_NEW	A new peer was added to the local database.	No
RDC_EVT_PEER_DEL	A peer was removed from the local database.	No
RDC_EVT_CTX_NEW	A new context was added to the local database.	No
RDC_EVT_CTX_DEL	A context was removed from the local database.	No
RDC_EVT_PEER_DOWN	A peer has disconnected.	No
RDC_EVT_SHUTDOWN	The API was successfully shut down.	No

Table 4.1: Event types in the peer API

ing function (usually `rdc_send_msg()`) returns the appropriate error code. If, on the other hand, the connection was successful, an interface handle is returned (whose type depends on the interface type itself) that can later be used to send messages to this peer.

After a connection is successfully established, the next step is to authenticate the peer. This step is only performed if both the local and remote peers belong to at least one context with authentication and/or confidentiality enabled, otherwise it's skipped. The authentication is a five-step process, where both peers authenticate each other and a session key is exchanged. If two peers happen to initiate the authentication process simultaneously, the peer with the highest peer ID becomes the leader in the process. In other words, the authentication request from the peer with the lowest peer ID is ignored (and the other one is given priority).

The authentication state of a peer is stored in the variable `auth_state` of a peer's structure (`struct rdc_peer_ex`). This is an integer with an initial value of five. As the authentication process progresses, this variable gets decremented until it reaches zero. In this state the peer is assumed to be authenticated. Figure 4.14 shows the state diagram for authentication. Figure 4.10 shows the content of these messages (payload only).

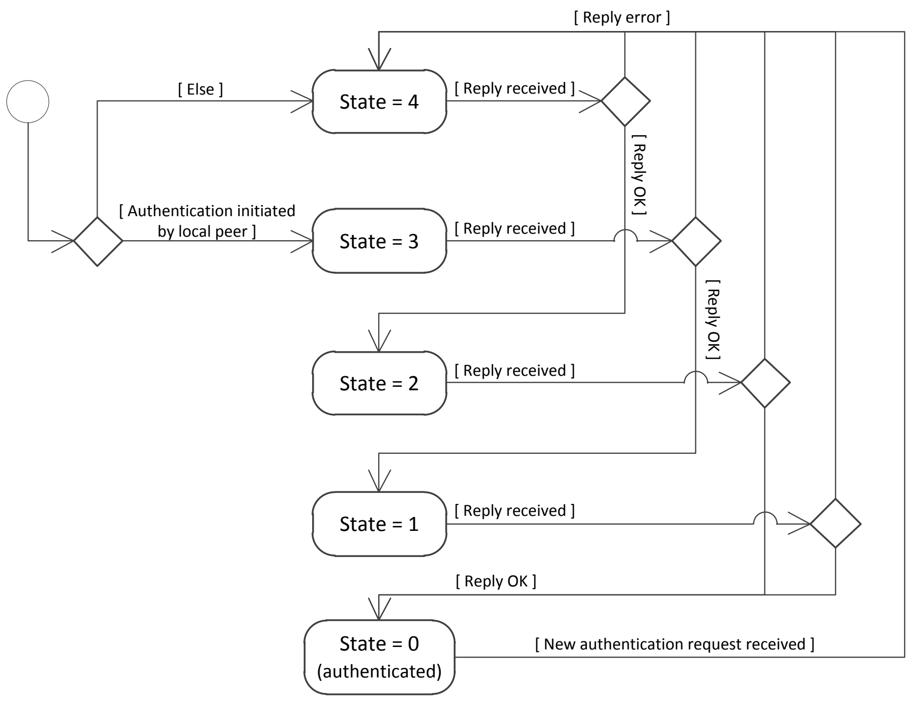


Figure 4.14: Peer-to-peer authentication state diagram

Peer-to-peer data transmission

After a pair of peers are connected and authenticated (if required), they are ready to exchange application-level messages. These are sent through the function `rdc_sendmsg()`. The flow chart in Figure 4.15 describes the algorithm used in this function.

If the source or target context require authentication (but not confidentiality), the message is sent with a Message Authentication Code (MAC) at the end. If it requires confidentiality (regardless of authentication), the message content is ciphered with the symmetric key that was agreed during the authentication process.

At the other end of the medium, the receiving peer reads the message (through whichever IPC mechanism is being used) and puts it in the incoming event queue. This event will be then processed by the event handling thread (`_rdc_thread_evt()`). If the message was received with no errors, the latter puts on the outgoing event queue to be retrieved by the application (through the function `rdc_recv_evt()`).

The flow chart in Figure 4.16 describes the algorithm used by the event processing thread for incoming messages.

The API keeps track of several statistics for each peer. These are stored in every peer structure (`struct rdc_peer_ex`) and can be retrieved at any

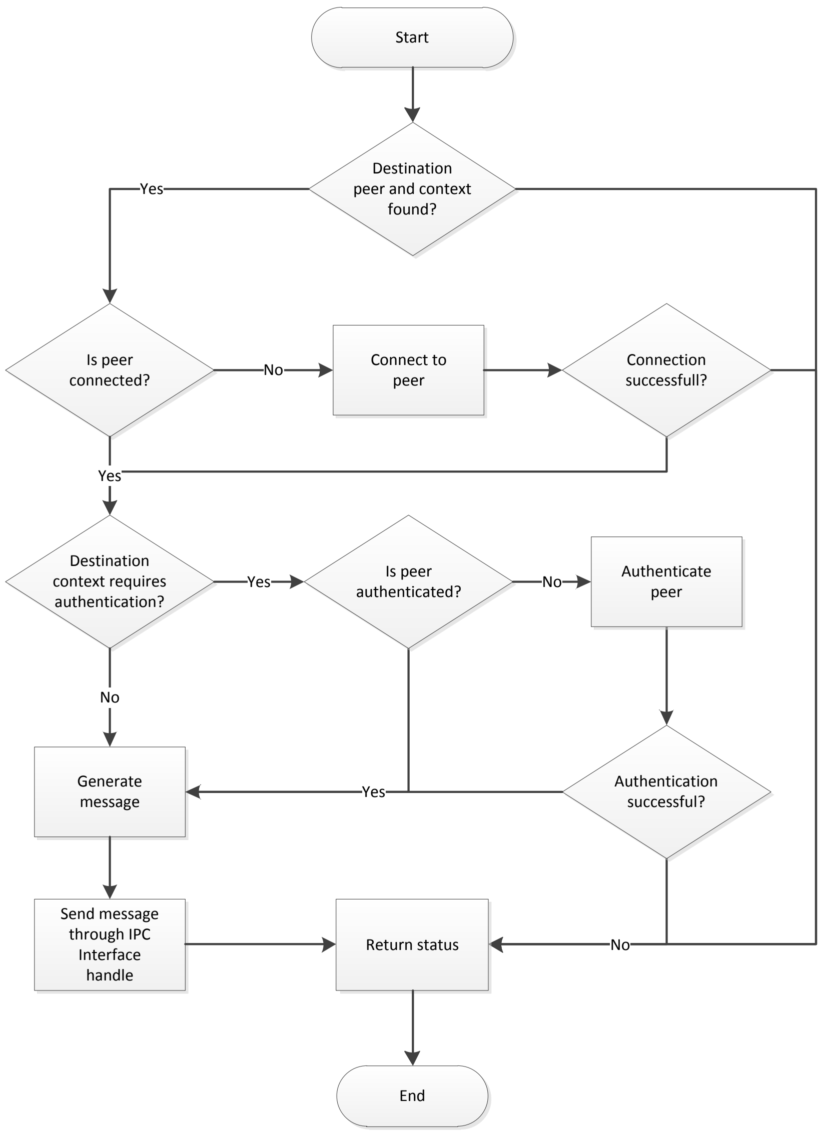


Figure 4.15: Peer to peer message transmission flowchart

time by the application. The variables are:

- `sent_msg` - Total sent messages;
- `sent_bytes` - Total sent bytes;
- `recv_msg` - Total received messages;
- `recv_bytes` - Total received bytes.

All events and regular-priority messages are always placed at the back of the outgoing event queue, effectively creating a FIFO queue. The only exception are urgent messages, which are placed at the beginning of it (or

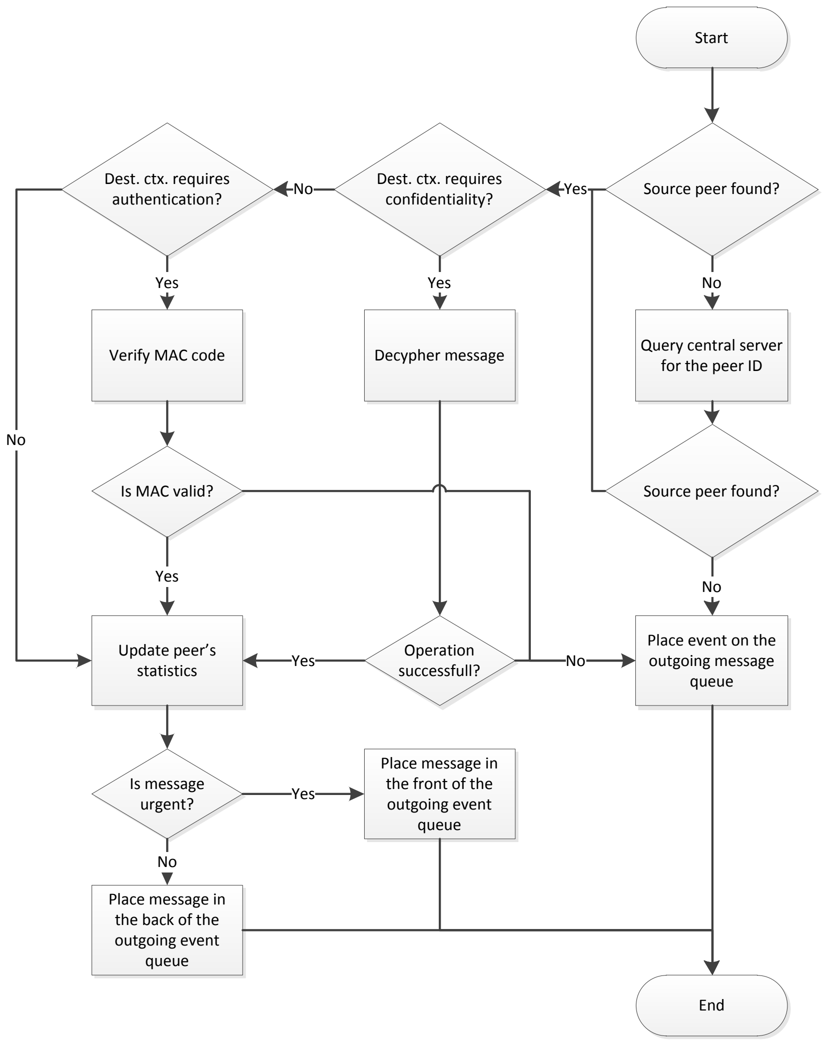

Start
Source peer found?
Yes
Dest. ctx. requires confidentiality?
No
Dest. ctx. requires authentication?
Yes
Yes
No
Verify MAC code
Decypher message
Query central server for the peer ID
No
Is MAC valid?
Source peer found?
Yes
No
Update peer's statistics
Yes
Operation successfull?
No
Place event on the outgoing message queue
Is message urgent?
Yes
Place message in the front of the outgoing event queue
No
Place message in the back of the outgoing event queue
End

Figure 4.16: Peer to peer message reception flowchart

at the back of the last urgent message), thus becoming the immediate next element to be retrieved from the queue.

Concurrency control

There are several threads in the API that require access to common resources simultaneously. Concurrency access control is enforced using four different types of mechanisms:

- **Pthread's mutexes** (`pthread_mutex_t`) - These are used when a critical section of the code should be accessed by only one thread at a time;

- **Pthread's read-write locks** (`pthread_rwlock_t`) - These are equivalent to mutexes, but they allow multiple threads to read a critical section of the code, but restricts access to a single thread for writing;
- **Binary semaphores** (`sem_t`) - These are functionally equivalent to Pthread's mutexes. They are used whenever the locking and unlocking is performed by different threads, for which a regular mutexes has undefined behavior;
- **GCC's atomic built-ins**[4] - These are a set of functions that allow a single atomic variable change. These functions have substantially less overhead than any of the other types mentioned above.

Each of these mechanisms were used whenever it was more adequate. For instance, Pthread's mutexes were used in the function `rdc_sendmsg()`, so as to ensure that no issues arise from several concurrent threads trying to send messages at the same time. Pthread's read-write locks are used for certain variables in the peer structure (`struct rdc_peer_ex`). These variables are read much more often than they are written, so it's much more efficient to use read-write locks instead of regular mutexes. Binary semaphores are only used in the event queue implementation (file `evt_queue.c`). The algorithms used require different threads to lock and unlock a given portion of the code, so an ordinary mutex could not be used. Finally, **GCC's atomic built-ins** are used in all variables related to statistics, such as the number of messages transmitted to and received by a peer.

The GCC's atomic built-ins have a lot less overhead that other types of concurrency handling mechanisms, mainly due to the fact that they don't trigger a system call. The downside of these primitives is that they aren't compatible with different compilers and/or architectures.[4]

Pthread's spin locks were also considered as a portable alternative to GCC's atomic built-ins, since they also have very little overhead. In the end they were left out, mainly due to the fact that they perform very poorly on machines with single-core processors.

Error codes and error handling

The application can be notified of errors in two ways: through the return codes of most API functions or through events.

Function return codes are defined in the file `dtypes.h`. The functions `rdc_init()`, `rdc_sendmsg()` and `rdc_update_db()` all return a subset of these codes. They are also used in the database API. Table 4.2 shows the description of these codes.

[4]`http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html#Atomic-Builtins`

Error type	Description
RDC_OK	No error.
RDC_ERROR	Unspecified error.
RDC_ERROR_CONF	Configuration error.
RDC_ERROR_IF_INIT	Could not initialize one or more IPC interfaces.
RDC_ERROR_CONN	Connection error.
RDC_ERROR_AUTH	Authentication error.
RDC_ERROR_FREAD	File access error.
RDC_ERROR_INT	Internal error.
RDC_ERROR_PEER_NF	Peer not found.
RDC_ERROR_CTX_NF	Context not found.
RDC_ERROR_DNS	Could not resolve DNS host name.
RDC_ERROR_SHUTD	API is shutting down.
RDC_ERROR_TBIG	Message is too big to send.
RDC_ERROR_EXT	Record already exists.
RDC_WARN_NOPEERS	No peers were found.
RDC_WARN_CONF	Non-critical configuration error.
RDC_WARN_INT	Non-critical internal error.

Table 4.2: Return codes of the RDC API

Local database updates

After the API has been initialized, the function `rdc_update_db()` can be called at any time to update the local database's contents. This function requests a list of peers and contexts to the server in the same way as it does during the initialization phase. Then, the new lists (of peers and contexts) are compared to the existing ones. The missing elements in the new lists are removed from the existing ones and the missing elements in the existing ones are added from the new lists. Every added and removed peer and context is reported to the application through events. Additionally, every peer is checked for differences in their configuration and updated if needed. The final result is an updated list of peers without the need to destroy and re-initialize the API's instance.

After an update is completed successfully, the API may reconnect to a peer if a new and better interface is available. Also, if the security parameters of an existing context are changed, or a new context with confidentiality or authentication enabled is added to a peer without security-enabled contexts, the authentication process for that peer is triggered.

Peer API shutdown

When the function `rdc_destroy()` is called, all interface handles and IPC interfaces are disconnected, along with some other additional resources. An event of the type `RDC_EVT_SHUTDOWN` is then created and put at the back of the outgoing queue. The API's resources are only fully released after this event is retrieved by the application. This way it is possible to predictably shut down the API with no loss of information.

4.3.9 Peer API Test Application

The peer API test application was created to evaluate functional characteristics of the peer API. To a lesser extent, this application was also used to evaluate performance bottlenecks during development.

There are actually two versions of the test application, and a module for `ttt` was developed as well. The first version of the application is used to perform functional tests on the RDC API. The second version was specifically developed to perform scalability tests. And finally, the `ttt`'s module is used to perform peer-to-peer throughput tests.

Functional tests

Due to the API's complexity, it became apparent from the beginning of the development process that it was crucial to implement effective debugging mechanisms, so as to accelerate the debugging process and to allow the conclusion of the project in a realistic time frame. One of these measures is the inclusion of the `assert()` macro in strategic places of the code. This allows the verification of simple conditions that must always be true if the code works correctly, and to cause a signal to be sent to the debugger otherwise. Debugging messages are also used extensively in the code (using `printf()` and `perror()`). They allow the analysis of the correct sequence of events and simple diagnostics (e.g. making sure a message in a context with security is correctly ciphered before being sent). All of these messages are surrounded by conditional preprocessor directives, so they can be "removed" from the code on request.

The API test application was used to test the stability and correct operation of the API. It suffered several modifications during development, so that any new functionality being worked on could be tested immediately. The final version tests all the functions in the API, namely:

- Connects to the server (tests correct initialization of the API);
- Displays a list of peers, interfaces and contexts retrieved (tests retrieval of data from the server, serialization/deserialization of said data and validity of local data structures);
- Uses several threads to send data bursts to all peers and contexts (tests stability and concurrency control);
- Tries to update the database while still sending and receiving data from peers (tests stability and concurrency control);
- Shuts down the API (tests correct shutdown and release of allocated resources).

The database manager application, the database API and the server daemon have a significant lower level of complexity than that of the peer's API, so the debugging measures shown above were not used since they weren't justified.

Performance tests

An additional module for the RDC API was developed for the `ttt` application used in Section 3.4.2. This way, it will be possible to obtain results and compare them to the primitive IPC mechanisms that were previously tested by the same application and the same algorithms. A minimal database was created for these tests, with only two peers and one context. Both IPC mechanisms will be tested, locally and remotely (if possible), with four different configurations for the context security (combinations of confidentiality and authentication parameters enabled and/or disabled).

A second version of the peer API test application was created to test the scalability of the system. It works by sending as much data as possible to every peer in the system, and then displaying the amount of all data transmitted to all peers. From these values it is possible to conclude just how well the peer API scales to large networks (having in mind the limitations of IPC mechanisms).

For all performance tests, the RDC API was compiled in release mode with `GCC`'s optimization flag `-O3` set.

4.3.10 Building and Executing the Project

A `Make` script (`makefile`) was used to build the software project. The following `make` targets were included:

- `make all` - Builds everything described below;
- `make rdcd` - Builds the central server daemon;
- `make rdcm` - Builds the database management application;
- `make rdc_db` - Builds the database API in the form of a software library (static library `librdc.a` and shared library `librdc.so.1.0.1`):
- `make rdc_api` - Builds the peer API in the form of a software library (static library `librdcdb.a` and shared library `librdcdb.so.1.0.1`);
- `make rdc_test` - Builds the peer API test application;
- `make tar` - Creates a `.tar` (compressed archive) file with all of the project's files.
- `make clean` - Removes all `.obj` (object code) and backup files.
- `make clear` - Same as `make clean`, but also removes the executables, libraries and tar file.

Debug and release versions of the output binaries can be specified by setting the global variable `DEBUG` as `1` (enabled) or `0` (disabled). For instance, to build the entire project with debug symbols, one could use the command `make all DEBUG=1`. For a release build, with no debug symbols and full code optimization, the command `make all DEBUG=0` should be used. When enabled, the `DEBUG` directive also has the following effects in the code:

- The server daemon does not get "daemonized". In other words, the launching process is not `fork()`'ed and the log messages are sent to the standard output (`stdout`), in addition to syslog;
- Debugging code is enabled in the peer API. This consists on several `printf()` functions in the code containing useful information about what is being executed and the status of the system.

If no `DEBUG` parameter is set, the debug release is selected by default.

After being executed, `make all` outputs the following executables:

- `rdc_test` - Peer API test application;
- `rdcm` - Database manager application;
- `rdcd` - Central server daemon.

Two different libraries are also created:

- `librdc.a` - Peer API library.
- `librdcdb.a` - RDC's database management API library.

4.4 Adding/Replacing modules

4.4.1 Database API

To change the DBMS being used by the database API, the following steps must be taken:

1. Modify `rdc_db_t` (in `db.h`) in order to contain all the necessary variables to identify an instance of the database;
2. Modify `struct rdc_db_cfg` (in `db.h`) to contain all the needed variables needed to configure the database;
3. Modify the file `db.c` so as to contain the implementation of all the functions defined in `db.h`.

Applications that uses the database API must change the configuration structure `struct rdc_db_t`, if necessary.

4.4.2 Peer API

The peer API supports adding and removing IPC interfaces. This is a more complex task due to the absence of hereditary concepts in the C language. The following procedure must be taken to add a new module:

1. Create files in path `./src/api/ipc` to implement the IPC interface (e.g. `ipcmodule.h` and `ipcmodule.c`);
2. Add an `#include "./ipc/ipcmodule.h"` directive in `ipc.c`;
3. Modify the content of all functions in `ipc.c` according to their task;
4. Perform the following modifications in `api_dtypes.h`:
   (a) Add an interface type entry in `enum rdc_if_type` (e.g. `IFTYPE_IPCMODULE`);
   (b) Add a configuration structure (e.g. `struct rdc_if_ipcmodule_cfg`);
   (c) Add a pointer to the configuration structure in `struct rdc_cfg`;
5. Add interface initialization code to the function `rdc_init()` contained in file `rdc.c`.

The `makefile` should also be updated for proper compilation and linking.

4.5 Deployment

The RDC API can be used in either C or C++ projects. To use the API in a project, one has to include `./include/rdc.h` in the source file and then link the application against `./lib/librdc.a` (for static linking) or `./lib/librdc.so.1.0.1` (for shared linking). The same applies to the database management API, but with the include file `./include/db.h` and library files `./lib/librdcdb.a` or `./lib/librdcdb.so.1.0.1`. The `rdcd` and `rdcm` applications are generic, and as such they are ready for deployment and can be used in a production environment with no modifications if necessary.

The entire source code of the project is commented using Doxygen's syntax. From the root folder of the project, the command `doxygen doxyfile` can be used to generate the full documentation of the API.

4.6 Conclusion

The implementation fulfills all the requirements identified in Section 1.3. Nevertheless, the time constrains in this dissertation means that the project presented here is just a prototype, and as such there's room left for improvement.

One of the biggest challenges faced during the planning phase was to decide how the RDC network information would be synchronized between all peers. Several designs were considered, but every one had its limitations. Ultimately, the adopted design is the one shown in this chapter, which stands for a good compromise between scalability and temporal accuracy of data. Some of the alternatives proposed (and how they compare to the current solution) include:

- Updates through permanent peer-server connections - Means an higher (centralized) load for the server, lower scalability due to limitations on

the number of socket connections, and the system may not function properly if the server goes down. On the other hand, it would be easier to know which peers had to be updated (if they needed to);
- Regular server polling - This would imply a delay for the update messages to propagate through the network, an higher workload and unnecessary usage of resources in the system;
- Server connects and sends a message to the peers when a new update arrives - This solution would introduce load problems if too many peers tried to connect at once. In terms of overall load it should be similar to the solution that was implemented.

Designing an architecture for the peer API that is both well structured and efficient revealed to be a particularly difficult task. A lot of effort was put in finding a good tradeoff between these two, but code legibility and maintainability still had to be sacrificed in favor of performance. Some design choices may appear to be less than ideal from an architectural point of view, but they are justified when evaluated from an efficiency point of view.

One of the most obvious ways the peer API could be improved is by instantiating additional event processing threads. This would allow a more efficient usage of resources to process the events, particularly in systems with Symmetric multiprocessing (SMP). The reason why this wasn't implemented is that it would require a mechanism to ensure the correct order (causality) of the messages. For instance, a new connection event should always take precedence to new messages of the same peer. But if two threads take turns in handling the incoming messages, the new message event could be processed before the correct parameters of the peer were set by the new connection event, and an error would occur. A mechanism would be needed to avoid this. On the downside, this feature would require more mutexes and threads, which would also increase overhead.

Another modification that could theoretically increase the throughput of the system would be sending all the regular messages directly to the outgoing queue as they are received by the implementation file of the IPC mechanism. Such implementation would increase parallelization of tasks (assuming that more than one IPC mechanism is being used) and would reduce unnecessary overhead by skipping the incoming event queue. On the other hand, this would reduce the modularization of the IPC interface's code and could introduce problems related with causality violation of events.

Another important improvement that could be made is the inclusion of a new field in the database to store the last time a peer connected and updated its data with the central server. This would allow other peers to know if they need to send interface information updates to it after they log in. If there were no updates to the interface information of peer A currently stored in the database since peer B connected to it, then peer A doesn't need to sent an interface information update message to it. This way, needless traffic could be avoided and faster initialization of the system could be accomplished.

Since the database API allows a peer to directly connect to the database, one of the arguments that could be made is that the dependency on central servers is a hindrance to the system as it becomes redundant. This system was designed while having in mind a relatively small number of peers (in the order of thousands), and it doesn't make much sense to require authentication if the peer itself can create its own key and add itself to whatever context it wants to. For this reason, it is recommended to never implement the database API only inside the peer. If really necessary, a workaround to allow each peer to connect directly to the database would be to simply include an instance of the server daemon on each peer.

One potential security issue in the current implementation has to do with the authentication process. Ideally, both peers should be involved in the generation of the session key, thus allowing equal responsibility in keeping the data safe. This feature was not included due to limitations of the OpenSSL library.

To conclude, the prototype fulfills all objectives proposed for this dissertation. Although there is still a good margin for improvement, it should provide a good starting point for future developments.

Chapter 5

Evaluation

The API was put to the test to evaluate its proper functionality and its capability to fulfill the goals it was designed for. This chapter presents the results obtained from these tests. As shown in Section 4.3.9, they were split between functional and performance assessments.

5.1 Resources

The lab tests were done over a couple of computers connected to each other. The complete hardware list is:

- **Computer A:**
  - CPU - Intel Core 2 Duo T9400 (2.53GHz, 64KB L1 cache, 6MB L2 cache)
  - Ram - 4096MB DDR3 PC 1066 (533 MHz) 7-7-7-20
  - Motherboard - LG Emerald (1067 MHz FSB)
  - Storage - Fujitsu MHZ2320BH G2 (320GB, 5400rpm, 8MB buffer)
  - Network adapter - Intel 82567LF Gigabit Network Connection (1500 Bytes of MTU)
  - Operating system - Ubuntu Desktop v11.10 (Linux kernel v3.0.0-12, 32-bit)
- **Computer B:**
  - CPU - Intel Core i5-2300 (2.8GHz, 4x32KB L1 cache, 4x256KB L2 cache, 6MB L3 cache)
  - Ram - 4096MB DDR3 (1333 MHz)
  - Motherboard - ASUS P8H61-M LX (Intel H61B3 chipset)
  - Storage - Samsung HD502HI (500GB, 5400rpm, 16MB buffer)
  - Network adapter - Realtek RTL8111E PCIe GBE Family Controller (1500 Bytes of MTU)

  - Operating system - Ubuntu Desktop v11.10 (Linux kernel v3.0.0-12, 32-bit)

- **Computers C1-C4:**
  - Computer C1:
    - CPU - AMD Semptron 3000+ (1800Mhz, 128KB L1 cache, 128KB L2 cache)
    - RAM - 512MB DDR400
    - Chipset - Nvidia NForce3
    - Network adapter - Realtek ethernet 10/100mbps RTL-8139/8139c/8139c+
    - Operating system - Lubuntu 11.10 (in Live CD mode) (Linux kernel v3.0.0-12, 32 bit)
  - Computer C2:
    - CPU - Pentium 4 @ 2.0Ghz (8KB L1 cache, 512KB L2 cache)
    - RAM - 512MB
    - Network adapter - Silicon Integrated Systems (SiS) 900 PCI Fast Ethernet 10/100Mbps
    - Operating system - Lubuntu 11.10 (in Live CD mode) (Linux kernel v3.0.0-12, 32 bit)
  - Computer C3:
    - CPU - Pentium 4 @ 2.4Ghz (8KB L1 cache, 512KB L2 cache)
    - RAM - 256MB
    - Chipset - Chipset: Via Technologies VT8753
    - Network adapter - Realtek ethernet 10/100mbps RTL-8139/8139c/8139c+
    - Operating system - Lubuntu 11.10 (in Live CD mode) (Linux kernel v3.0.0-12, 32 bit)
  - Computer C4:
    - CPU - Pentium 4 @ 2.0Ghz (8kb L1 + 512 KB L2 cache)
    - RAM - 512MB
    - Chipset - VIA Technologies P4X333/P4X400/PT800
    - Network adapter - VIA Technologies VT6102 10/100Mbps
    - Operating system - Lubuntu 11.10 (in Live CD mode) (Linux kernel v3.0.0-12, 32 bit)

- **Network A:**
  - Local network deployed on a Cisco Systems Catalyst 2900 series XL (10/100 Mbit)

The tool `lshw` was used to extract the hardware information from Computers C1-C4. This tool omitted details on some of the machines, so a complete hardware description of every machine could not be obtained.

All tests described in this chapter that required only one computer were executed in Computer A. This is the exact same machine and setup that was described in Section 3.4.2. All tests that required two machines were executed in Computer A and Computer B. They were connected to each other over an ethernet cable and their network interfaces operated at 1 Gbps (full-duplex). For all tests that required more than two computers, Computer A and Computers C1-C4 were used, connected to each other via Network A (as shown in Figure 5.1). The switch was configured with three local Virtual Local Area Networks (VLANs) of 6 elements each, but only one VLAN was used.

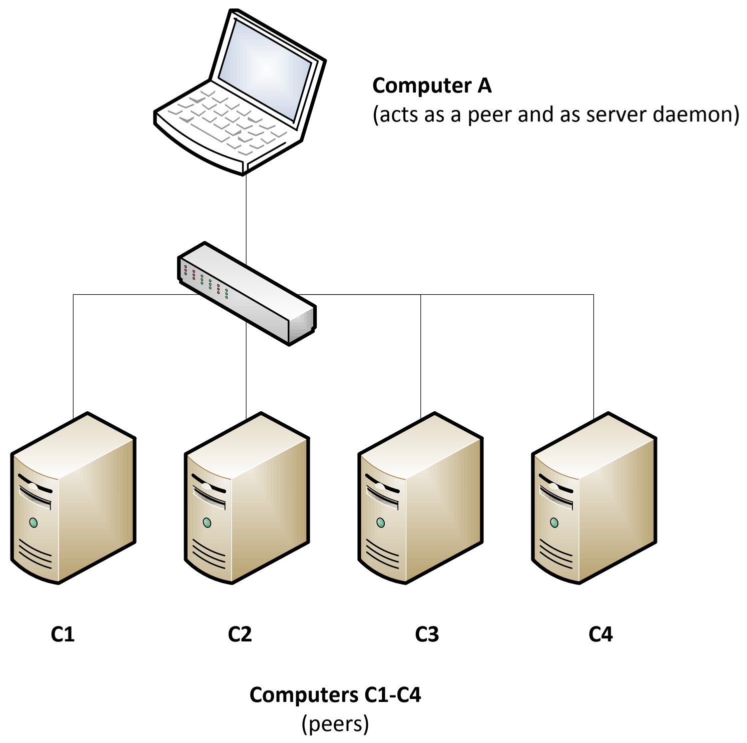


Figure 5.1: Network topology used for testing

Computer B had some problems related to the stability of its network drivers. To compensate for this issue, each test than involved this computer ran several times more than planned and the best values were kept. Computers C1-C4 and Network A showed problems related to synchronization and loose and/or defective cables. These issues interfered with the results fairly frequently, so the network was monitored as best as possible while the tests were running. Unfortunately, due to the sparse location of the network's elements, it was impossible to be absolutely sure that all connections were OK for the entire duration of each test.

5.2 Functional Tests

The RDC API was compiled in debug mode for the functional tests. The debug messages implemented on the code are representative of the code paths in the API. The format of these messages include the file and function where they are located in, a brief description of the operation being carried on and/or the value of some relevant variable(s). The test program used for functional tests was described in Section 4.3.9.

Due to the extensive length of some of the peer API's outputs, some screenshots were omitted and replaced by a verbatim output of the application.

5.2.1 API's Functions

Database management

A new configuration file was created in the predefined directory (set by the constant `RDC_CFG_FILE` in `./src/const.h`). Its contents were:

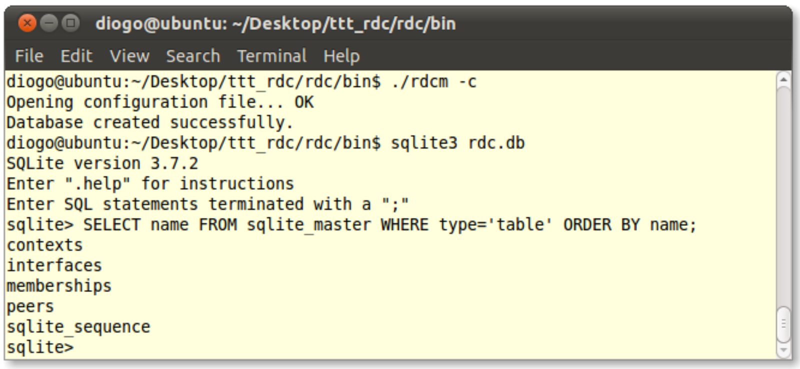
```
#
# Configuration file for RDC Daemon/RDC database manager
#

### Listen address and port:
lstn_addr    = 0.0.0.0
lstn_port    = 1024

### Locations:
db_path      = rdc.db
cert_path    = ../cert/
```

The next step was to create a new database and all its tables (as seen in Figure 4.6). This was done with the database management application (`rdcm`) by typing `./rdcm -c`. Figure 5.2 shows the output of the command and the tables that were added to the database file.

A new context and peer were added to the database and a membership between them was created. Figure 5.3 shows the sequence of commands executed and the contents of the database. Four more peers and three contexts were then added. All peers belong to context `ctx_1`, while peers `peer_1` and `peer_2` belong to `ctx_2`, and peers `peer_2` and `peer_3` belong to `ctx_3`.

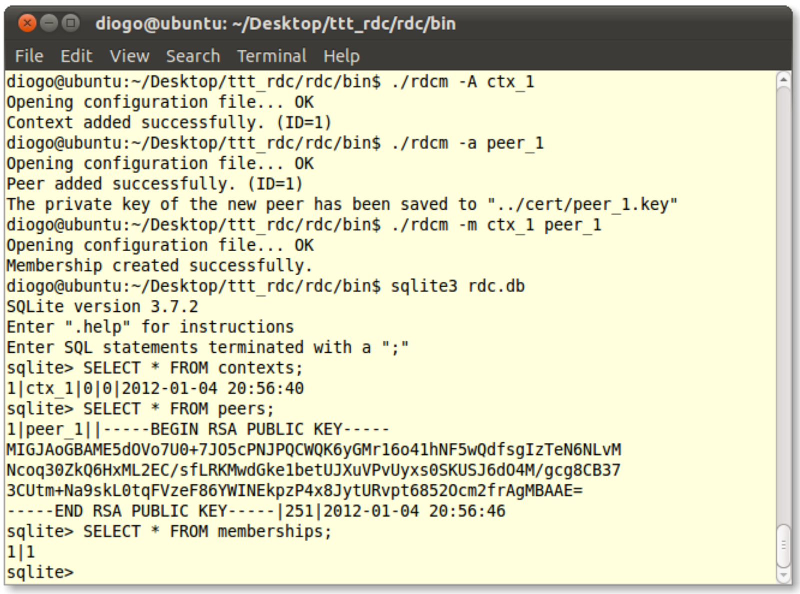
Peer API initialization and destruction

The server daemon (`rdcd`) and the peer API test application (`rdc_test`) were both initiated in debug mode. The latter was set up so as to initialize an RDC API instance, print the contents of the local database and destroy the same instance. The output from peer `peer_1` was the following:


diogo@ubuntu: ~/Desktop/ttt_rdc/rdc/bin
File Edit View Search Terminal Help

```
diogo@ubuntu:~/Desktop/ttt_rdc/rdc/bin$ ./rdcm -c
Opening configuration file... OK
Database created successfully.
diogo@ubuntu:~/Desktop/ttt_rdc/rdc/bin$ sqlite3 rdc.db
SQLite version 3.7.2
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> SELECT name FROM sqlite_master WHERE type='table' ORDER BY name;
contexts
interfaces
memberships
peers
sqlite_sequence
sqlite>
```

Figure 5.2: Creating a new database


diogo@ubuntu: ~/Desktop/ttt_rdc/rdc/bin
File Edit View Search Terminal Help

```
diogo@ubuntu:~/Desktop/ttt_rdc/rdc/bin$ ./rdcm -A ctx_1
Opening configuration file... OK
Context added successfully. (ID=1)
diogo@ubuntu:~/Desktop/ttt_rdc/rdc/bin$ ./rdcm -a peer_1
Opening configuration file... OK
Peer added successfully. (ID=1)
The private key of the new peer has been saved to "../cert/peer_1.key"
diogo@ubuntu:~/Desktop/ttt_rdc/rdc/bin$ ./rdcm -m ctx_1 peer_1
Opening configuration file... OK
Membership created successfully.
diogo@ubuntu:~/Desktop/ttt_rdc/rdc/bin$ sqlite3 rdc.db
SQLite version 3.7.2
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> SELECT * FROM contexts;
1|ctx_1|0|0|2012-01-04 20:56:40
sqlite> SELECT * FROM peers;
1|peer_1||-----BEGIN RSA PUBLIC KEY-----
MIGJAoGBAME5dOVo7U0+7JO5cPNJPQCWQK6yGMr16o41hNF5wQdfsgIzTeN6NLvM
Ncoq30ZkQ6HxML2EC/sfLRKMwdGke1betUJXuVPvUyxs0SKUSJ6dO4M/gcg8CB37
3CUtm+Na9skL0tqFVzeF86YWINEkpzP4x8JytURvpt6852Ocm2frAgMBAAE=
-----END RSA PUBLIC KEY-----|251|2012-01-04 20:56:46
sqlite> SELECT * FROM memberships;
1|1
sqlite>
```

Figure 5.3: Adding a context, a peer and a membership to the database

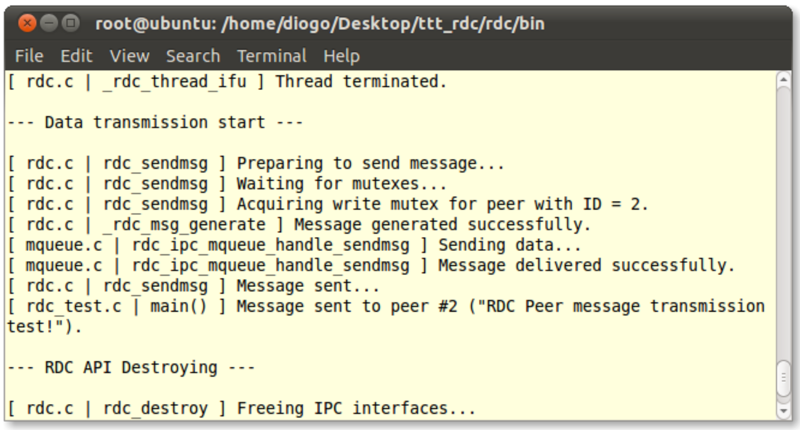
```
root@ubuntu:/home/diogo/Desktop/ttt_rdc/rdc/bin# ./rdc_test peer_1

--- RDC API Initializing ---

[ rdc.c | rdc_init ] Initializing...
[ rdc.c | rdc_init ] Host ID: a8c0944a.
[ rdc.c | rdc_init ] Verifying configuration...
[ rdc.c | rdc_init ] Loading private key...
[ rdc.c | rdc_init ] Obtaining database...
[ rdc.c | _rdc_update_db ] Connecting to central server...
[ rdc.c | _rdc_update_db ] Connection established.
[ rdc.c | _rdc_update_db ] Authentication: Sending name to server...
[ rdc.c | _rdc_update_db ] Authentication successful!
[ rdc.c | _rdc_update_db ] Sending peer list request...
[ rdc.c | _rdc_update_db ] Sending host ID...
[ rdc.c | _rdc_update_db ] Sending interface information to server...
[ rdc.c | _rdc_update_db ] Receiving peer information from server...
[ rdc.c | _rdc_update_db ] Interface information retrieved successfully.
[ rdc.c | _rdc_update_db ] Creating RDC network database...
[ api_dtypes.c | rdc_dt_db_update ] Updating database...
[ api_dtypes.c | rdc_dt_db_update ] Database is being initialized for the first
                                    time.
[ api_dtypes.c | rdc_dt_db_update ] Adding contexts and peers...
[ api_dtypes.c | rdc_dt_db_update ] All OK. 3 new peer(s) added.
[ rdc.c | _rdc_update_db ] Notifying application of new contexts...
[ rdc.c | _rdc_update_db ] Removing contexts from the database...
[ rdc.c | _rdc_update_db ] Notifying application of new peers...
[ rdc.c | _rdc_update_db ] Removing peers from the database...
[ rdc.c | _rdc_update_db ] All OK.
[ rdc.c | rdc_init ] Initializing incoming and outgoing message queues...
[ rdc.c | rdc_init ] Initializing event processing thread...
[ rdc.c | rdc_init ] Initializing IPC interfaces...
[ tcpip.c | rdc_ipc_tcpip_init ] Opening server socket...
[ tcpip.c | rdc_ipc_tcpip_init ] Binding socket...
[ tcpip.c | rdc_ipc_tcpip_init ] Socket bound to port 3572.
[ tcpip.c | rdc_ipc_tcpip_init ] Creating thread...
[ tcpip.c | rdc_ipc_tcpip_init ] All ok.
[ rdc.c | _rdc_thread_evt ] Initializing thread...
[ rdc.c | _rdc_thread_evt ] Waiting for next event...
[ mqueue.c | rdc_ipc_mqueue_init ] Opening message queue "/rdc_1"...
[ mqueue.c | rdc_ipc_mqueue_init ] Creating thread...
[ tcpip.c | _rdc_ipc_tcpip_srv_thread ] Waiting for new connection...
[ mqueue.c | rdc_ipc_mqueue_init ] All ok.
[ rdc.c | rdc_init ] Initializing interface information update thread...
[ rdc.c | rdc_init ] Initialization complete.
[ rdc.c | _rdc_thread_ifu ] Initializing thread...
[ mqueue.c | _rdc_ipc_mqueue_srv_thread ] Waiting for new message...
[ rdc.c | _rdc_thread_ifu ] Preparing to send interface information for peer #2...
[ rdc.c | _rdc_thread_ifu ] Creating new handle for peer with ID = 2. Waiting for
                            mutex lock...
[ rdc.c | _rdc_peer_connect ] Attempting to connect to peer with ID = 2.
[ rdc.c | _rdc_peer_connect ] Error: Could not create interface handle for peer
                              with ID = 2.
[ rdc.c | _rdc_thread_ifu ] Error: Could not create interface handle for peer
                            with ID = 2.
[ rdc.c | _rdc_thread_ifu ] Assuming peer ID #2 as offline.
[ rdc.c | _rdc_thread_ifu ] Preparing to send interface information for peer #3...
[ rdc.c | _rdc_thread_ifu ] Creating new handle for peer with ID = 3. Waiting for
                            mutex lock...
[ rdc.c | _rdc_peer_connect ] Attempting to connect to peer with ID = 3.
[ rdc.c | _rdc_peer_connect ] Error: Could not create interface handle for peer
                              with ID = 3.
[ rdc.c | _rdc_thread_ifu ] Error: Could not create interface handle for peer
                            with ID = 3.
[ rdc.c | _rdc_thread_ifu ] Assuming peer ID #3 as offline.
[ rdc.c | _rdc_thread_ifu ] Preparing to send interface information for peer #4...
[ rdc.c | _rdc_thread_ifu ] Creating new handle for peer with ID = 4. Waiting for
                            mutex lock...
[ rdc.c | _rdc_peer_connect ] Attempting to connect to peer with ID = 4.
[ rdc.c | _rdc_peer_connect ] Error: Could not create interface handle for peer
                              with ID = 4.
[ rdc.c | rdc_recv_evt ] Waiting for new event...
[ rdc.c | _rdc_thread_ifu ] Error: Could not create interface handle for peer
                            with ID = 4.
[ rdc.c | _rdc_thread_ifu ] Assuming peer ID #4 as offline.
[ rdc.c | _rdc_thread_ifu ] Thread terminated.
```

```
--- RDC database information ---

Contexts:
> ID=1, name=ctx_1
> ID=2, name=ctx_2

Peers:
> ID=2, name=peer_2
  Part of contexts #1 #2
> ID=3, name=peer_3
  Part of contexts #1
> ID=4, name=peer_4
  Part of contexts #1

--- RDC API Destroying ---

[ rdc.c | rdc_destroy ] Freeing IPC interfaces...
[ tcpip.c | rdc_ipc_tcpip_destroy ] Waiting for mutex...
[ tcpip.c | rdc_ipc_tcpip_destroy ] Shutting down interface...
[ tcpip.c | rdc_ipc_tcpip_destroy ] Waiting for server thread...
[ tcpip.c | _rdc_ipc_tcpip_srv_thread ] Termination requested. Exiting thread...
[ tcpip.c | _rdc_ipc_tcpip_srv_thread ] Thread terminated.
[ tcpip.c | rdc_ipc_tcpip_destroy ] Interface shut down.
[ mqueue.c | rdc_ipc_mqueue_destroy ] Waiting for mutex...
[ mqueue.c | rdc_ipc_mqueue_destroy ] Interface shut down.
[ rdc.c | rdc_destroy ] Waiting for threads...
[ rdc.c | _rdc_thread_evt ] Event retrieved (type: 72, arg1: 0, arg2: 0, arg3: 0,
                            arg4: (nil), len: 0, in queue: 0).
[ rdc.c | _rdc_thread_evt ] Thread terminated.
[ rdc.c | rdc_recv_evt ] Shutdown event retrieved. Releasing resources...
[ api_dtypes.c | rdc_dt_remove_peer ] Removing peer #2...
[ api_dtypes.c | rdc_dt_remove_peer ] Removing peer #3...
[ api_dtypes.c | rdc_dt_remove_peer ] Removing peer #4...
[ api_dtypes.c | rdc_dt_remove_peer ] Removing context #1...
[ api_dtypes.c | rdc_dt_remove_peer ] Removing context #2...
[ rdc.c | rdc_recv_evt ] All OK.
[ rdc.c | rdc_recv_evt ] New event retrieved (Type=72, Arg1=0, Arg2=0, Arg3=0,
                         Arg4=(nil), len=0).
[ rdc.c | rdc_destroy ] RDC instance resources freed sucessfully.
root@ubuntu:/home/diogo/Desktop/ttt_rdc/rdc/bin#
```

As shown in the output above, the API successfully retrieved the information of the peers it can connect to from the server, before initializing all local IPC interfaces. The list of peers and contexts shown on the output was obtained by the peer API test application (`rdc_test`) through the API's functions `rdc_get_peers()` and `rdc_get_contexts()`, respectively. In the last step of the initialization process, the API instance tries to connect to other peers on the system to send them the updated information of IPC interfaces. Since no other peer is active, all connection attempts have failed (as expected).

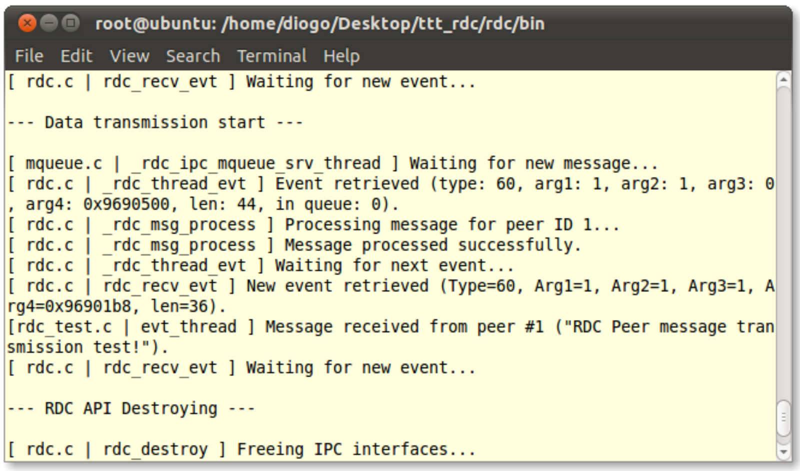
Authentication and data transmission

Two API instances were created (for `peer_1` and `peer_2`) to test data transmission between peers. The peer API test application was configured to connect to the system and send only one message from peer `peer_1` to `peer_2`. Figures 5.4 and 5.5 shows the output from `peer_1` and `peer_2`, respectively. Initialization and destruction code was omitted from the figures.


root@ubuntu: /home/diogo/Desktop/ttt_rdc/rdc/bin
File Edit View Search Terminal Help

```
[ rdc.c | _rdc_thread_ifu ] Thread terminated.

--- Data transmission start ---

[ rdc.c | rdc_sendmsg ] Preparing to send message...
[ rdc.c | rdc_sendmsg ] Waiting for mutexes...
[ rdc.c | rdc_sendmsg ] Acquiring write mutex for peer with ID = 2.
[ rdc.c | _rdc_msg_generate ] Message generated successfully.
[ mqueue.c | rdc_ipc_mqueue_handle_sendmsg ] Sending data...
[ mqueue.c | rdc_ipc_mqueue_handle_sendmsg ] Message delivered successfully.
[ rdc.c | rdc_sendmsg ] Message sent...
[ rdc_test.c | main() ] Message sent to peer #2 ("RDC Peer message transmission
test!").

--- RDC API Destroying ---

[ rdc.c | rdc_destroy ] Freeing IPC interfaces...
```

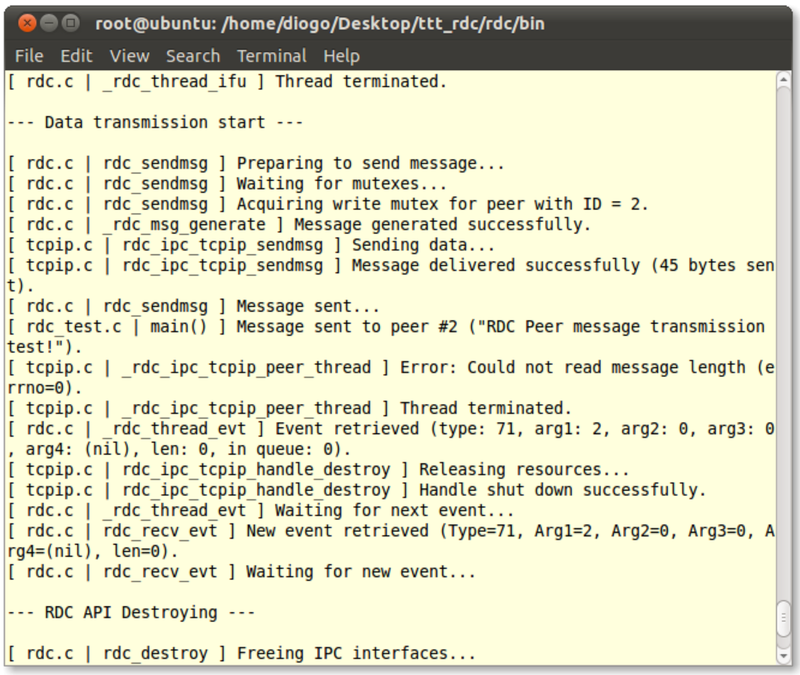
Figure 5.4: Message transmission from peer_1 to peer_2 (message queue interface)


root@ubuntu: /home/diogo/Desktop/ttt_rdc/rdc/bin
File Edit View Search Terminal Help

```
[ rdc.c | rdc_recv_evt ] Waiting for new event...

--- Data transmission start ---

[ mqueue.c | _rdc_ipc_mqueue_srv_thread ] Waiting for new message...
[ rdc.c | _rdc_thread_evt ] Event retrieved (type: 60, arg1: 1, arg2: 1, arg3: 0
, arg4: 0x9690500, len: 44, in queue: 0).
[ rdc.c | _rdc_msg_process ] Processing message for peer ID 1...
[ rdc.c | _rdc_msg_process ] Message processed successfully.
[ rdc.c | _rdc_thread_evt ] Waiting for next event...
[ rdc.c | rdc_recv_evt ] New event retrieved (Type=60, Arg1=1, Arg2=1, Arg3=1, A
rg4=0x96901b8, len=36).
[rdc_test.c | evt_thread ] Message received from peer #1 ("RDC Peer message tran
smission test!").
[ rdc.c | rdc_recv_evt ] Waiting for new event...

--- RDC API Destroying ---

[ rdc.c | rdc_destroy ] Freeing IPC interfaces...
```

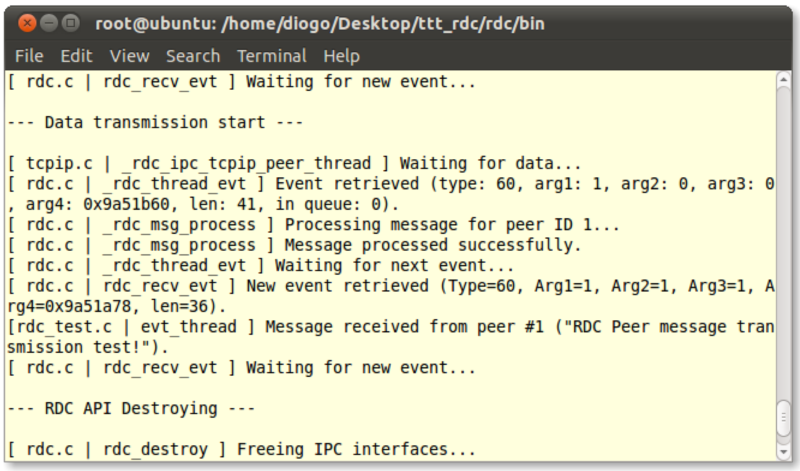
Figure 5.5: Message reception from peer_1 to peer_2 (message queue interface)

The message received by `peer_2` is first put on its incoming event queue and then retrieved by the event processing thread of the peer API. After the message is processed (where headers are stripped from the payload, peer statistics are updated, etc), it is placed on the outgoing message queue to be retrieved by the application through the function `rdc_recv_evt()`. Since both instances are located in the same machine, the message queue IPC interface was used. If the message queue configuration is disabled, the TCP/IP interface is used instead. Figures 5.6 and 5.7 shows the TCP/IP equivalent to figures 5.4 and 5.5, respectively. If both peers were located in different machines, their output would be identical to Figures 5.6 and 5.7.


root@ubuntu: /home/diogo/Desktop/ttt_rdc/rdc/bin
File Edit View Search Terminal Help

```
[ rdc.c | _rdc_thread_ifu ] Thread terminated.

--- Data transmission start ---

[ rdc.c | rdc_sendmsg ] Preparing to send message...
[ rdc.c | rdc_sendmsg ] Waiting for mutexes...
[ rdc.c | rdc_sendmsg ] Acquiring write mutex for peer with ID = 2.
[ rdc.c | _rdc_msg_generate ] Message generated successfully.
[ tcpip.c | rdc_ipc_tcpip_sendmsg ] Sending data...
[ tcpip.c | rdc_ipc_tcpip_sendmsg ] Message delivered successfully (45 bytes sen
t).
[ rdc.c | rdc_sendmsg ] Message sent...
[ rdc_test.c | main() ] Message sent to peer #2 ("RDC Peer message transmission
test!").
[ tcpip.c | _rdc_ipc_tcpip_peer_thread ] Error: Could not read message length (e
rrno=0).
[ tcpip.c | _rdc_ipc_tcpip_peer_thread ] Thread terminated.
[ rdc.c | _rdc_thread_evt ] Event retrieved (type: 71, arg1: 2, arg2: 0, arg3: 0
, arg4: (nil), len: 0, in queue: 0).
[ tcpip.c | rdc_ipc_tcpip_handle_destroy ] Releasing resources...
[ tcpip.c | rdc_ipc_tcpip_handle_destroy ] Handle shut down successfully.
[ rdc.c | _rdc_thread_evt ] Waiting for next event...
[ rdc.c | rdc_recv_evt ] New event retrieved (Type=71, Arg1=2, Arg2=0, Arg3=0, A
rg4=(nil), len=0).
[ rdc.c | rdc_recv_evt ] Waiting for new event...

--- RDC API Destroying ---

[ rdc.c | rdc_destroy ] Freeing IPC interfaces...
```

Figure 5.6: Message transmission from peer_1 to peer_2 (TCP/IP interface)


root@ubuntu: /home/diogo/Desktop/ttt_rdc/rdc/bin
File Edit View Search Terminal Help

```
[ rdc.c | rdc_recv_evt ] Waiting for new event...

--- Data transmission start ---

[ tcpip.c | _rdc_ipc_tcpip_peer_thread ] Waiting for data...
[ rdc.c | _rdc_thread_evt ] Event retrieved (type: 60, arg1: 1, arg2: 0, arg3: 0
, arg4: 0x9a51b60, len: 41, in queue: 0).
[ rdc.c | _rdc_msg_process ] Processing message for peer ID 1...
[ rdc.c | _rdc_msg_process ] Message processed successfully.
[ rdc.c | _rdc_thread_evt ] Waiting for next event...
[ rdc.c | rdc_recv_evt ] New event retrieved (Type=60, Arg1=1, Arg2=1, Arg3=1, A
rg4=0x9a51a78, len=36).
[rdc_test.c | evt_thread ] Message received from peer #1 ("RDC Peer message tran
smission test!").
[ rdc.c | rdc_recv_evt ] Waiting for new event...

--- RDC API Destroying ---

[ rdc.c | rdc_destroy ] Freeing IPC interfaces...
```

Figure 5.7: Message reception from peer_1 to peer_2 (TCP/IP interface)

When authentication is enabled, the peers perform the authentication process before application-level messages are allowed to be transmitted. Authentication for context `ctx_1` was enabled through the command `./rdcm -E ctx_1 true false`. Now when `peer_1` sent a message `peer_2`, its output was:

```
root@ubuntu:/home/diogo/Desktop/ttt_rdc/rdc/bin# ./rdc_test peer_1

--- RDC API Initializing ---

(...)

--- Data transmission start ---

[ rdc.c | rdc_sendmsg ] Preparing to send message...
[ rdc.c | rdc_sendmsg ] Waiting for mutexes...
[ rdc.c | rdc_sendmsg ] Waiting for authentication for peer with ID = 2.
[ rdc.c | _rdc_msg_generate ] Message generated successfully.
[ rdc.c | _rdc_peer_auth ] Sending authentication request for peer with ID = 2...
[ tcpip.c | rdc_ipc_tcpip_sendmsg ] Sending data...
[ tcpip.c | rdc_ipc_tcpip_sendmsg ] Message delivered successfully (133 bytes
                                    sent).
[ rdc.c | _rdc_peer_auth ] Authentication request sent. Waiting for transaction
                          finish...
[ tcpip.c | _rdc_ipc_tcpip_peer_thread ] Waiting for data...
[ rdc.c | _rdc_thread_evt ] Event retrieved (type: 60, arg1: 2, arg2: 0, arg3: 0,
                            arg4: 0x92a6188, len: 145, in queue: 0).
[ rdc.c | _rdc_msg_process ] Processing message for peer ID 2...
[ rdc.c | _rdc_peer_auth_step ] Waiting for write mutex...
[ rdc.c | _rdc_peer_auth_step ] Peer ID 2 has an authentication status of 3.
[ rdc.c | _rdc_msg_generate ] Message generated successfully.
[ tcpip.c | rdc_ipc_tcpip_sendmsg ] Sending data...
[ tcpip.c | rdc_ipc_tcpip_sendmsg ] Message delivered successfully (21 bytes
                                    sent).
[ rdc.c | _rdc_thread_evt ] Waiting for next event...
[ tcpip.c | _rdc_ipc_tcpip_peer_thread ] Waiting for data...
[ rdc.c | _rdc_thread_evt ] Event retrieved (type: 60, arg1: 2, arg2: 0, arg3: 0,
                            arg4: 0x92a62f8, len: 257, in queue: 0).
[ rdc.c | _rdc_msg_process ] Processing message for peer ID 2...
[ rdc.c | _rdc_peer_auth_step ] Waiting for write mutex...
[ rdc.c | _rdc_peer_auth_step ] Peer ID 2 has an authentication status of 1.
[ rdc.c | _rdc_peer_auth_step ] Peer 2 authenticated successfully.
[ rdc.c | _rdc_thread_evt ] Waiting for next event...
[ rdc.c | _rdc_peer_auth ] Authentication successfull for peer ID = "2".
[ rdc.c | rdc_sendmsg ] Acquiring write mutex for peer with ID = 2.
[ rdc.c | _rdc_msg_generate ] Signing message...
[ rdc.c | _rdc_msg_generate ] Message generated successfully.
[ tcpip.c | rdc_ipc_tcpip_sendmsg ] Sending data...
[ tcpip.c | rdc_ipc_tcpip_sendmsg ] Message delivered successfully (173 bytes
                                    sent).
[ rdc.c | rdc_sendmsg ] Message sent...
[ rdc_test.c | main() ] Message sent to peer #2 ("RDC Peer message transmission
                        test!").

--- RDC API Destroying ---

(...)

root@ubuntu:/home/diogo/Desktop/ttt_rdc/rdc/bin#
```

And the output from `peer_2` was:

```
root@ubuntu:/home/diogo/Desktop/ttt_rdc/rdc/bin# ./rdc_test peer_2

--- RDC API Initializing ---

(...)

--- Data transmission start ---

[ tcpip.c | _rdc_ipc_tcpip_peer_thread ] Waiting for data...
[ rdc.c | _rdc_thread_evt ] Event retrieved (type: 60, arg1: 1, arg2: 0, arg3: 0,
                            arg4: 0x9a2c8e0, len: 129, in queue: 0).
[ rdc.c | _rdc_msg_process ] Processing message for peer ID 1...
[ rdc.c | _rdc_peer_auth_step ] Waiting for write mutex...
[ rdc.c | _rdc_peer_auth_step ] Peer ID 1 has an authentication status of 4.
[ rdc.c | _rdc_msg_generate ] Message generated successfully.
[ tcpip.c | rdc_ipc_tcpip_sendmsg ] Sending data...
[ tcpip.c | rdc_ipc_tcpip_sendmsg ] Message delivered successfully (149 bytes
                                    sent).
[ rdc.c | _rdc_thread_evt ] Waiting for next event...
[ tcpip.c | _rdc_ipc_tcpip_peer_thread ] Waiting for data...
[ rdc.c | _rdc_thread_evt ] Event retrieved (type: 60, arg1: 1, arg2: 0, arg3: 0,
                            arg4: 0x9a2bb80, len: 17, in queue: 0).
[ rdc.c | _rdc_msg_process ] Processing message for peer ID 1...
[ rdc.c | _rdc_peer_auth_step ] Waiting for write mutex...
[ rdc.c | _rdc_peer_auth_step ] Peer ID 1 has an authentication status of 2.
[ rdc.c | _rdc_msg_generate ] Message generated successfully.
[ tcpip.c | rdc_ipc_tcpip_sendmsg ] Sending data...
[ tcpip.c | rdc_ipc_tcpip_sendmsg ] Message delivered successfully (261 bytes
                                    sent).
[ rdc.c | _rdc_peer_auth_step ] Peer 1 authenticated successfully.
[ rdc.c | _rdc_thread_evt ] Waiting for next event...
[ tcpip.c | _rdc_ipc_tcpip_peer_thread ] Waiting for data...
[ rdc.c | _rdc_thread_evt ] Event retrieved (type: 60, arg1: 1, arg2: 0, arg3: 0,
                            arg4: 0x9a2d720, len: 169, in queue: 0).
[ rdc.c | _rdc_msg_process ] Processing message for peer ID 1...
[ rdc.c | _rdc_msg_process ] Waiting for write mutex...
[ rdc.c | _rdc_msg_process ] Authenticating message...
[ rdc.c | _rdc_msg_process ] Message processed successfully.
[ rdc.c | _rdc_thread_evt ] Waiting for next event...
[ rdc.c | rdc_recv_evt ] New event retrieved (Type=60, Arg1=1, Arg2=1, Arg3=1,
                         Arg4=0x9a2bc00, len=36).
[rdc_test.c | evt_thread ] Message received from peer #1 ("RDC Peer message
                           transmission test!").
[ rdc.c | rdc_recv_evt ] Waiting for new event...

--- RDC API Destroying ---

(...)

root@ubuntu:/home/diogo/Desktop/ttt_rdc/rdc/bin#
```

When confidentiality was enabled instead (through the command `./rdcm -E ctx_1 false true`), `peer_1` gave the following output:

```
root@ubuntu:/home/diogo/Desktop/ttt_rdc/rdc/bin# ./rdc_test peer_1

--- RDC API Initializing ---
```

```
(...)

--- Data transmission start ---

[ rdc.c | rdc_sendmsg ] Preparing to send message...
[ rdc.c | rdc_sendmsg ] Waiting for mutexes...
[ rdc.c | rdc_sendmsg ] Waiting for authentication for peer with ID = 2.
[ rdc.c | _rdc_msg_generate ] Message generated successfully.
[ rdc.c | _rdc_peer_auth ] Sending authentication request for peer with ID = 2...
[ tcpip.c | rdc_ipc_tcpip_sendmsg ] Sending data...
[ tcpip.c | rdc_ipc_tcpip_sendmsg ] Message delivered successfully (133 bytes
                                    sent).
[ rdc.c | _rdc_peer_auth ] Authentication request sent. Waiting for transaction
                           finish...
[ tcpip.c | _rdc_ipc_tcpip_peer_thread ] Waiting for data...
[ rdc.c | _rdc_thread_evt ] Event retrieved (type: 60, arg1: 2, arg2: 0, arg3: 0,
                            arg4: 0x83952f8, len: 145, in queue: 0).
[ rdc.c | _rdc_msg_process ] Processing message for peer ID 2...
[ rdc.c | _rdc_peer_auth_step ] Waiting for write mutex...
[ rdc.c | _rdc_peer_auth_step ] Peer ID 2 has an authentication status of 3.
[ rdc.c | _rdc_msg_generate ] Message generated successfully.
[ tcpip.c | rdc_ipc_tcpip_sendmsg ] Sending data...
[ tcpip.c | rdc_ipc_tcpip_sendmsg ] Message delivered successfully (21 bytes
                                    sent).
[ rdc.c | _rdc_thread_evt ] Waiting for next event...
[ tcpip.c | _rdc_ipc_tcpip_peer_thread ] Waiting for data...
[ rdc.c | _rdc_thread_evt ] Event retrieved (type: 60, arg1: 2, arg2: 0, arg3: 0,
                            arg4: 0x83952f8, len: 257, in queue: 0).
[ rdc.c | _rdc_msg_process ] Processing message for peer ID 2...
[ rdc.c | _rdc_peer_auth_step ] Waiting for write mutex...
[ rdc.c | _rdc_peer_auth_step ] Peer ID 2 has an authentication status of 1.
[ rdc.c | _rdc_peer_auth_step ] Peer 2 authenticated successfully.
[ rdc.c | _rdc_peer_auth ] Authentication successfull for peer ID = "2".
[ rdc.c | rdc_sendmsg ] Acquiring write mutex for peer with ID = 2.
[ rdc.c | _rdc_msg_generate ] Encrypting message...
[ rdc.c | _rdc_msg_generate ] Message generated successfully.
[ tcpip.c | rdc_ipc_tcpip_sendmsg ] Sending data...
[ tcpip.c | rdc_ipc_tcpip_sendmsg ] Message delivered successfully (57 bytes
                                    sent).
[ rdc.c | rdc_sendmsg ] Message sent...
[ rdc_test.c | main() ] Message sent to peer #2 ("RDC Peer message transmission
                        test!").

--- RDC API Destroying ---

(...)

root@ubuntu:/home/diogo/Desktop/ttt_rdc/rdc/bin#
```

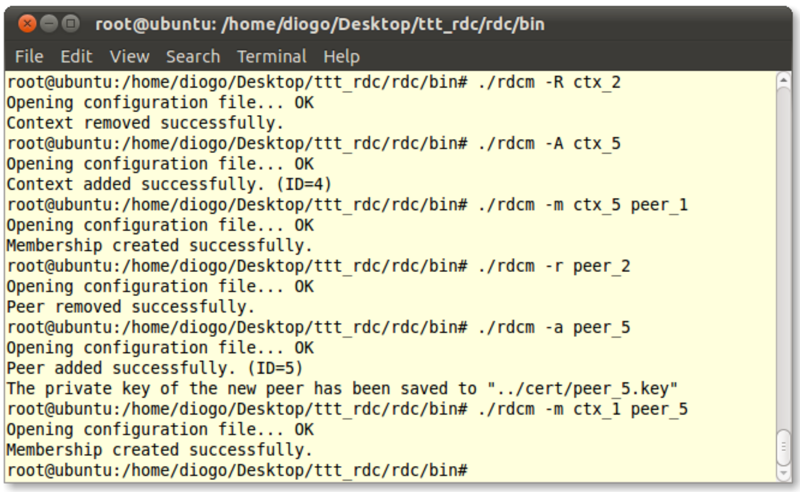
And the output from `peer_2` was:

```
root@ubuntu:/home/diogo/Desktop/ttt_rdc/rdc/bin# ./rdc_test peer_2

--- RDC API Initializing ---

(...)

--- Data transmission start ---

[ tcpip.c | _rdc_ipc_tcpip_peer_thread ] Waiting for data...
[ rdc.c | _rdc_thread_evt ] Event retrieved (type: 60, arg1: 1, arg2: 0, arg3: 0,
                            arg4: 0x9f228e0, len: 129, in queue: 0).
[ rdc.c | _rdc_msg_process ] Processing message for peer ID 1...
[ rdc.c | _rdc_peer_auth_step ] Waiting for write mutex...
[ rdc.c | _rdc_peer_auth_step ] Peer ID 1 has an authentication status of 4.
[ rdc.c | _rdc_msg_generate ] Message generated successfully.
[ tcpip.c | rdc_ipc_tcpip_sendmsg ] Sending data...
[ tcpip.c | rdc_ipc_tcpip_sendmsg ] Message delivered successfully (149 bytes
                                    sent).
[ rdc.c | _rdc_thread_evt ] Waiting for next event...
[ tcpip.c | _rdc_ipc_tcpip_peer_thread ] Waiting for data...
[ rdc.c | _rdc_thread_evt ] Event retrieved (type: 60, arg1: 1, arg2: 0, arg3: 0,
                            arg4: 0x9f21b80, len: 17, in queue: 0).
[ rdc.c | _rdc_msg_process ] Processing message for peer ID 1...
[ rdc.c | _rdc_peer_auth_step ] Waiting for write mutex...
[ rdc.c | _rdc_peer_auth_step ] Peer ID 1 has an authentication status of 2.
[ rdc.c | _rdc_msg_generate ] Message generated successfully.
[ tcpip.c | rdc_ipc_tcpip_sendmsg ] Sending data...
[ tcpip.c | rdc_ipc_tcpip_sendmsg ] Message delivered successfully (261 bytes
                                    sent).
[ rdc.c | _rdc_peer_auth_step ] Peer 1 authenticated successfully.
[ rdc.c | _rdc_thread_evt ] Waiting for next event...
[ tcpip.c | _rdc_ipc_tcpip_peer_thread ] Waiting for data...
[ rdc.c | _rdc_thread_evt ] Event retrieved (type: 60, arg1: 1, arg2: 0, arg3: 0,
                            arg4: 0x9f22430, len: 53, in queue: 0).
[ rdc.c | _rdc_msg_process ] Processing message for peer ID 1...
[ rdc.c | _rdc_msg_process ] Waiting for write mutex...
[ rdc.c | _rdc_msg_process ] Decyphering message...
[ rdc.c | _rdc_msg_process ] Message processed successfully.
[ rdc.c | _rdc_thread_evt ] Waiting for next event...
[ rdc.c | rdc_recv_evt ] New event retrieved (Type=60, Arg1=1, Arg2=1, Arg3=1,
                         Arg4=0x9f231d8, len=36).
[rdc_test.c | evt_thread ] Message received from peer #1 ("RDC Peer message
                           transmission test!").
[ rdc.c | rdc_recv_evt ] Waiting for new event...


--- RDC API Destroying ---

(...)

root@ubuntu:/home/diogo/Desktop/ttt_rdc/rdc/bin#
```

Database updates

Since the server daemon doesn't cache queries and keeps a persistent connection to the database, any modifications introduced through `rdcm` should be immediately reflected on the information transmitted by the daemon to the peers.

To test this functionality, the peer API test application was modified to initialize, idle and then update its local database through the function `rdc_update_db()` when triggered by an input. While the API was idling, one peer and one context were added to the database, while one existing peer and one context were removed from it (Figure 5.8).
The output from `peer_1` was:


root@ubuntu: /home/diogo/Desktop/ttt_rdc/rdc/bin
File Edit View Search Terminal Help

```
root@ubuntu:/home/diogo/Desktop/ttt_rdc/rdc/bin# ./rdcm -R ctx_2
Opening configuration file... OK
Context removed successfully.
root@ubuntu:/home/diogo/Desktop/ttt_rdc/rdc/bin# ./rdcm -A ctx_5
Opening configuration file... OK
Context added successfully. (ID=4)
root@ubuntu:/home/diogo/Desktop/ttt_rdc/rdc/bin# ./rdcm -m ctx_5 peer_1
Opening configuration file... OK
Membership created successfully.
root@ubuntu:/home/diogo/Desktop/ttt_rdc/rdc/bin# ./rdcm -r peer_2
Opening configuration file... OK
Peer removed successfully.
root@ubuntu:/home/diogo/Desktop/ttt_rdc/rdc/bin# ./rdcm -a peer_5
Opening configuration file... OK
Peer added successfully. (ID=5)
The private key of the new peer has been saved to "../cert/peer_5.key"
root@ubuntu:/home/diogo/Desktop/ttt_rdc/rdc/bin# ./rdcm -m ctx_1 peer_5
Opening configuration file... OK
Membership created successfully.
root@ubuntu:/home/diogo/Desktop/ttt_rdc/rdc/bin#
```

Figure 5.8: Database changes performed while peer_1 was waiting for input

```
root@ubuntu:/home/diogo/Desktop/ttt_rdc/rdc/bin# ./rdc_test peer_1

--- RDC API Initializing ---

(...)

Waiting for input...

--- Updating database ---

[ rdc.c | _rdc_update_db ] Connecting to central server...
[ rdc.c | _rdc_update_db ] Connection established.
[ rdc.c | _rdc_update_db ] Authentication: Sending name to server...
[ rdc.c | _rdc_update_db ] Authentication successful!
[ rdc.c | _rdc_update_db ] Sending peer list request...
[ rdc.c | _rdc_update_db ] Sending host ID...
[ rdc.c | _rdc_update_db ] Sending interface information to server...
[ rdc.c | _rdc_update_db ] Receiving peer information from server...
[ rdc.c | _rdc_update_db ] Interface information retrieved successfully.
[ rdc.c | _rdc_update_db ] Creating RDC network database...
[ api_dtypes.c | rdc_dt_db_update ] Updating database...
[ api_dtypes.c | rdc_dt_db_update ] Checking for deleted contexts...
[ api_dtypes.c | rdc_dt_db_update ] Removing context ID #2...
[ api_dtypes.c | rdc_dt_db_update ] Checking for new contexts...
[ api_dtypes.c | rdc_dt_db_update ] Adding context ID #4...
[ api_dtypes.c | rdc_dt_db_update ] Checking for deleted peers...
[ api_dtypes.c | rdc_dt_db_update ] Removing peer ID #2...
[ api_dtypes.c | rdc_dt_db_update ] Checking for new peers...
[ api_dtypes.c | rdc_dt_db_update ] Adding peer ID #5...
[ api_dtypes.c | rdc_dt_db_update ] Freeing resources...
[ api_dtypes.c | rdc_dt_remove_peer ] Removing context #1...
[ api_dtypes.c | rdc_dt_remove_peer ] Removing peer #3...
[ api_dtypes.c | rdc_dt_remove_peer ] Removing peer #4...
[ api_dtypes.c | rdc_dt_db_update ] All OK.
[ rdc.c | _rdc_update_db ] Notifying application of new contexts...
[ rdc.c | _rdc_update_db ] Removing removed contexts from the database...
[ rdc.c | _rdc_update_db ] Removing context #2...
[ rdc.c | _rdc_update_db ] Removing context information from peer #2...
[ api_dtypes.c | rdc_dt_remove_peer ] Removing context #2...
[ rdc.c | _rdc_update_db ] Notifying application of new peers...
[ rdc.c | _rdc_update_db ] Removing removed peers from the database...
[ rdc.c | _rdc_update_db ] Removing peer #2...
[ api_dtypes.c | rdc_dt_remove_peer ] Removing peer #2...
[ rdc.c | _rdc_update_db ] All OK.

--- RDC API Destroying ---

(...)

root@ubuntu:/home/diogo/Desktop/ttt_rdc/rdc/bin#
```

5.3 Stability

rdc_test was modified in accordance to the algorithm described in Section 4.3.9, so as to test the peer API's stability and the presence of memory errors (such as memory overflows, memory leaks and memory corruption bugs). The application valgrind was used to accomplish the latter.

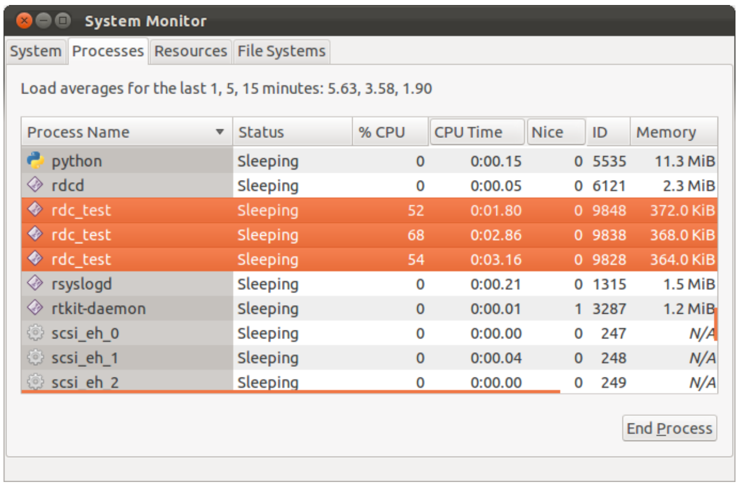
Running the application through valgrind allowed the successful detection and removal of memory errors during its implementation. There were, however, some leaks that could not be fixed as they were present inside two libraries used by the application - libssl and librt. The output from valgrind after running the stability test for one minute was:

```
==20519==
==20519== HEAP SUMMARY:
==20519==     in use at exit: 51,856 bytes in 2,230 blocks
==20519==   total heap usage: 6,763 allocs, 4,533 frees, 44,336,451 bytes
            allocated
==20519==
==20519== 144 bytes in 1 blocks are possibly lost in loss record 202 of 251
==20519==    at 0x4024F12: calloc (vg_replace_malloc.c:467)
==20519==    by 0x40117CB: _dl_allocate_tls (dl-tls.c:300)
==20519==    by 0x41C76A9: pthread_create@@GLIBC_2.1 (allocatestack.c:570)
==20519==    by 0x8060727: rdc_ipc_mqueue_init (mqueue.c:272)
==20519==    by 0x805A6D1: rdc_ipc_init (ipc.c:17)
==20519==    by 0x80599A7: rdc_init (rdc.c:2135)
==20519==    by 0x8055026: RdcModule::start() (RdcModule.cpp:100)
==20519==    by 0x804B4E9: main (ttt.cpp:197)
==20519==
==20519== 144 bytes in 1 blocks are possibly lost in loss record 203 of 251
==20519==    at 0x4024F12: calloc (vg_replace_malloc.c:467)
==20519==    by 0x40117CB: _dl_allocate_tls (dl-tls.c:300)
==20519==    by 0x41C76A9: pthread_create@@GLIBC_2.1 (allocatestack.c:570)
==20519==    by 0x8059A03: rdc_init (rdc.c:2150)
==20519==    by 0x8055026: RdcModule::start() (RdcModule.cpp:100)
==20519==    by 0x804B4E9: main (ttt.cpp:197)
==20519==
==20519== 244 bytes in 1 blocks are definitely lost in loss record 213 of 251
==20519==    at 0x4025BD3: malloc (vg_replace_malloc.c:236)
==20519==    by 0x43723FD: ??? (in /lib/libcrypto.so.0.9.8)
```

```
==20519==    by 0x4372A8B: CRYPTO_malloc (in /lib/libcrypto.so.0.9.8)
==20519==    by 0x43EA4E5: EVP_CipherInit_ex (in /lib/libcrypto.so.0.9.8)
==20519==    by 0x43E7570: EVP_EncryptInit_ex (in /lib/libcrypto.so.0.9.8)
==20519==    by 0x805AE1D: rdc_cry_encrypt (crypto.c:105)
==20519==    by 0x80558AD: _rdc_msg_generate (rdc.c:55)
==20519==    by 0x805A269: rdc_sendmsg (rdc.c:2427)
==20519==    by 0x80553F2: RdcModule_sendThread(void*) (RdcModule.cpp:190)
==20519==    by 0x41C6CC8: start_thread (pthread_create.c:304)
==20519==    by 0x42AC69D: clone (clone.S:130)
==20519==
==20519== 724 (88 direct, 636 indirect) bytes in 1 blocks are definitely lost in
          loss record 239 of 251
==20519==    at 0x4025BD3: malloc (vg_replace_malloc.c:236)
==20519==    by 0x43723FD: ??? (in /lib/libcrypto.so.0.9.8)
==20519==    by 0x4372A8B: CRYPTO_malloc (in /lib/libcrypto.so.0.9.8)
==20519==    by 0x43C6077: RSA_new_method (in /lib/libcrypto.so.0.9.8)
==20519==    by 0x43C62AD: RSA_new (in /lib/libcrypto.so.0.9.8)
==20519==    by 0x43C571C: ??? (in /lib/libcrypto.so.0.9.8)
==20519==    by 0x440249B: ??? (in /lib/libcrypto.so.0.9.8)
==20519==    by 0x4405707: ASN1_item_ex_d2i (in /lib/libcrypto.so.0.9.8)
==20519==    by 0x4405E94: ASN1_item_d2i (in /lib/libcrypto.so.0.9.8)
==20519==    by 0x43C5874: d2i_RSAPublicKey (in /lib/libcrypto.so.0.9.8)
==20519==    by 0x4416074: PEM_ASN1_read_bio (in /lib/libcrypto.so.0.9.8)
==20519==    by 0x4415655: PEM_read_bio_RSAPublicKey (in /lib/libcrypto.so.0.9.8)
==20519==
==20519== 2,376 (24 direct, 2,352 indirect) bytes in 1 blocks are definitely lost
          in loss record 248 of 251
==20519==    at 0x4025BD3: malloc (vg_replace_malloc.c:236)
==20519==    by 0x43723FD: ??? (in /lib/libcrypto.so.0.9.8)
==20519==    by 0x4372A8B: CRYPTO_malloc (in /lib/libcrypto.so.0.9.8)
==20519==    by 0x43EBB44: EVP_PKEY_new (in /lib/libcrypto.so.0.9.8)
==20519==    by 0x805B1C6: rdc_cry_sign (crypto.c:217)
==20519==    by 0x805592E: _rdc_msg_generate (rdc.c:70)
==20519==    by 0x8058C9F: _rdc_thread_ifu (rdc.c:1717)
==20519==    by 0x41C6CC8: start_thread (pthread_create.c:304)
==20519==    by 0x42AC69D: clone (clone.S:130)
==20519==
==20519== LEAK SUMMARY:
==20519==    definitely lost: 364 bytes in 4 blocks
==20519==    indirectly lost: 2,988 bytes in 42 blocks
==20519==      possibly lost: 288 bytes in 2 blocks
==20519==    still reachable: 48,216 bytes in 2,182 blocks
==20519==         suppressed: 0 bytes in 0 blocks
==20519== Reachable blocks (those to which a pointer was found) are not shown.
==20519== To see them, rerun with: --leak-check=full --show-reachable=yes
==20519==
==20519== For counts of detected and suppressed errors, rerun with: -v
==20519== ERROR SUMMARY: 6 errors from 6 contexts (suppressed: 33 from 10)
```

To test the actual stability of the system, three instances of the peer API test application were left running for a couple of minutes. The first used only message queue IPC, while the second and third peers used both message queues TCP/IP sockets. Figure 5.9 shows the memory usage a couple of seconds after the applications were launched. Figure 5.10 shows the memory usage from the same instances a couple of minutes later. No errors occurred during the tests.

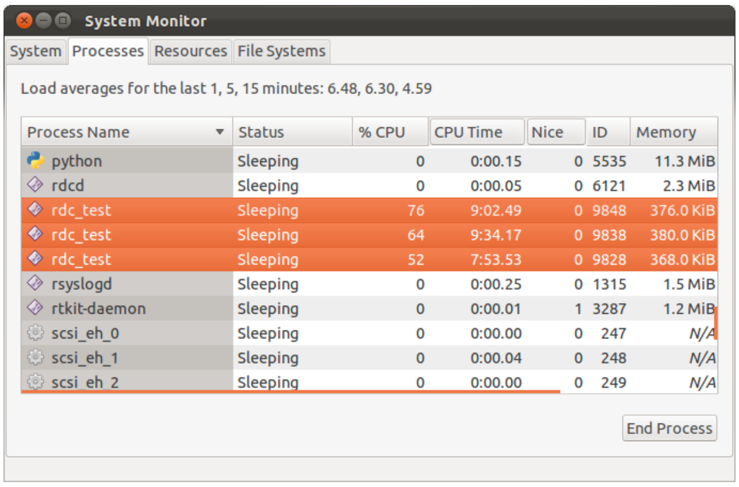
Further tests were made with an higher number of peers (which completed successfully) and with authentication and/or confidentiality enabled.



Load averages for the last 1, 5, 15 minutes: 5.63, 3.58, 1.90

Process Name	Status	% CPU	CPU Time	Nice	ID	Memory
python	Sleeping	0	0:00.15	0	5535	11.3 MiB
rdcd	Sleeping	0	0:00.05	0	6121	2.3 MiB
rdc_test	Sleeping	52	0:01.80	0	9848	372.0 KiB
rdc_test	Sleeping	68	0:02.86	0	9838	368.0 KiB
rdc_test	Sleeping	54	0:03.16	0	9828	364.0 KiB
rsyslogd	Sleeping	0	0:00.21	0	1315	1.5 MiB
rtkit-daemon	Sleeping	0	0:00.01	1	3287	1.2 MiB
scsi_eh_0	Sleeping	0	0:00.00	0	247	N/A
scsi_eh_1	Sleeping	0	0:00.04	0	248	N/A
scsi_eh_2	Sleeping	0	0:00.00	0	249	N/A

Figure 5.9: Memory usage at the start of the stability test



Load averages for the last 1, 5, 15 minutes: 6.48, 6.30, 4.59

Process Name	Status	% CPU	CPU Time	Nice	ID	Memory
python	Sleeping	0	0:00.15	0	5535	11.3 MiB
rdcd	Sleeping	0	0:00.05	0	6121	2.3 MiB
rdc_test	Sleeping	76	9:02.49	0	9848	376.0 KiB
rdc_test	Sleeping	64	9:34.17	0	9838	380.0 KiB
rdc_test	Sleeping	52	7:53.53	0	9828	368.0 KiB
rsyslogd	Sleeping	0	0:00.25	0	1315	1.5 MiB
rtkit-daemon	Sleeping	0	0:00.01	1	3287	1.2 MiB
scsi_eh_0	Sleeping	0	0:00.00	0	247	N/A
scsi_eh_1	Sleeping	0	0:00.04	0	248	N/A
scsi_eh_2	Sleeping	0	0:00.00	0	249	N/A

Figure 5.10: Memory usage after a couple of minutes into the stability test

Unfortunately, some issues with the OpenSSL library (`libssl`) prevented the successful completion of most tests that required the use of it (i.e., when authentication and/or confidentiality was used). Even when both of the OpenSSL's internal access control mechanisms were properly set up, the OpenSSL library kept halting at seemingly random `assert()`s inside the library itself. This issue never occurred when only two peers were connected, and the probability of error increased as the number of peers also increased. The bug disappeared once some additional mutexes were set up in the API in such a way as to only allow one thread to use `libssl`'s functions at a time. Closer inspection of the OpenSSL's source code revealed a very frequent usage of global variables. These facts suggest that the issue is probably due to presence of bug(s) related with the concurrent access control mechanisms built into OpenSSL. An older version of the library was also tested (v0.9.8o), but with similar results.

5.4 Peer-to-Peer Throughput Tests

An RDC API module was created for `ttt` (used in section 3.4.2), in order to assess the IPC performance of the API. Note that the messages exchanged between processes through the RDC API contain an higher overhead than the ones sent directly through `ttt`. Coupled with the processing overhead introduced by the API, a lower throughput was expected for the results.

The presence of error checking mechanisms in `ttt`'s message protocol also allowed the verification that no packets were dropped or contained errors, and that they were all delivered in the expected order.

In this section, the term *plain* will be used to describe that no authentication and confidentiality was set up in the communication context. *Auth* means that authentication was enabled (and confidentiality disabled), and *conf* means that confidentiality was enabled (and authentication disabled). Results for both authentication and confidentiality enabled presented identical values to the *conf* scenario, so they were not shown.

5.4.1 Local Test Results

Table 5.2 shows the results obtained and how they fare against the reference values. Only one of the two IPC mechanisms supported by the API was tested at a time, so that direct comparisons could be made. Note that the columns “Packet error rate” and “Out-of-order packet rate” were removed from the table because these values were always null.

For the most part, the results obtained were expected. The increase in overhead of the API has a negative impact on performance, and it is more noticeable with smaller message sizes since it coincides with an increase of processing time. However, it was somehow unexpected that the RDC API managed to provide an higher throughput with TCP/IP for packets sizes of 16384 bytes than the reference value for that same IPC mechanism. Looking at the CPU usage field, one can speculate that it could be related to the parallelization of the code in the receiving end of the RDC API. This allows the separation of the thread that is responsible to extract the message from the socket to the one that verifies and processes its contents. These threads can run in different processing cores, hence the CPU usage value above the 100% mark (meaning that more than one processing core is being used). In any case, an higher throughput is always a desirable characteristic.

Another uncertainty is why the RDC API managed to obtain a slightly higher throughput through the TCP/IP interface than through message queues. The most probable reason is related to the way message queues

	Packet size	Throughput (MB/s)	Delay: Mean time (ms)	Delay: Std. Deviation (ms)	Transmission CPU usage	Reception CPU usage
TCP/IP socket	128 B	48,63	0,64	25,31	40%	100%
	1024 B	167,74	4,00	62,98	8%	100%
	16384 B	269,95	4,03	63.27	2%	100%
RDC API (TCP/IP-plain)	128 B	20,10	4,60	67,35	42%	106%
	1024 B	108,30	4,29	65,21	40%	116%
	16384 B	299,19	4,31	65,35	40%	125%
RDC API (TCP/IP-conf)	128 B	13.82	4.44	66.14	52%	100%
	1024 B	44.29	4.21	64.44	70%	110%
	16384 B	72.04	4.36	65.53	84%	106%
RDC API (TCP/IP-auth)	128 B	0.03	2.57	50.51	100%	6%
	1024 B	0.23	2.57	50.35	100%	6%
	16384 B	3.63	4.53	67.06	100%	8%
Message queue	128 B	50,51	0.01	3.21	96%	96%
	1024 B	178,49	0.02	4.90	56%	98%
	16384 B	301,65	0.25	15.84	18%	98%
RDC API (MQueue-plain)	128 B	18.58	0.36	19.04	64%	100%
	1024 B	99.41	0.47	21.58	50%	106%
	16384 B	286.60	4.46	66.52	20%	116%
RDC API (MQueue-conf)	128 B	12.70	0.12	11.12	68%	108%
	1024 B	45.53	1.80	42.29	72%	108%
	16384 B	72.92	4.31	65.30	78%	102%
RDC API (MQueue-auth)	128 B	0.03	4.25	68.97	100%	4%
	1024 B	0.24	4.66	64.01	100%	6%
	16384 B	3.89	4.21	64.68	100%	7%

Table 5.1: Local IPC test results for RDC API

are processed. The TCP/IP implementation has one reception thread for each peer it is connected to, and it only does only two things: retrieve the message from the socket and put in on the incoming event queue. On the other hand, the message queue implementation has only one reception thread for *all* peers, and it has to do three things: retrieve the message from the message queue, check if the remote peer is present on the local database and connected to the local peer (and notify it if not), and put the message on the incoming event queue. This solution requires a low amount of memory and resources, but the increased load on what was already the most expensive thread of the API (the reception thread) may explain the lower throughput and the slightly lower overall CPU usage of message queues due to the reduction of work in the message processing thread. On the plus side, the delay values are relatively shorter, so message queues should provide a throughput advantage on a question-reply scenario.

As expected, performance suffered considerably once cryptographic algorithms were introduced. These operations are computationally expensive and this is particularly noticeable when asymmetric encryption ciphers are used, since these algorithms are several orders of magnitude slower that the ones used in symmetric encryption. For the *auth* scenario, the performance bottleneck is specifically located on the generation of the Message Authentication Codes (MACs) for the authentication of messages. Repeated tests in this particular scenario also showed a very high disparity of mean delay values, sometimes ranging from 2 ms to 10 ms for the same test. The results shown were obtained from the average of all tests.

5.4.2 Remote Test Results

The remote tests were executed in a similar setup as the one used for local IPC tests, except that data was sent from Computer A to Computer B. Reference values were also obtained using `ttt`'s TCP/IP module. Table 5.2 shows the results from these tests.

	Packet size	Throughput (MB/s)	Delay: Mean time (ms)	Delay: Std. Deviation (ms)	Transmission CPU usage	Reception CPU usage
TCP/IP socket	128 B	29,58	1,36	0,09	70%	100%
	1024 B	98,82	1,35	0,09	60%	100%
	16384 B	111,56	1,34	0,09	20%	68%
RDC API (TCP/IP-plain)	128 B	18,56	1,39	0,09	70%	165%
	1024 B	48,40	1,39	0,09	65%	150%
	16384 B	100,72	1,38	0,09	50%	103%
RDC API (TCP/IP-conf)	128 B	8,93	1,41	0,08	70%	155%
	1024 B	38,32	1,39	0,09	80%	145%
	16384 B	76,12	1,40	0,09	100%	130%
RDC API (TCP/IP-auth)	128 B	0.03	1,40	0,08	100%	10%
	1024 B	0.23	1,39	0,08	100%	12%
	16384 B	3.58	1,38	0,09	100%	18%

Table 5.2: Remote IPC test results for RDC API

The results shown in the table were expected. Due to an increase in overhead, the RDC API provides a lower throughput when compared to the reference values. And again, this is particularly noticeable for smaller data packets.

For the reference test with packet size of 16384 bytes, the CPU usage dropped well below 100% at both ends of the line. This suggests that the throughput performance was network-bound instead of CPU-bound for this particular test.

5.5 Scalability Tests

The second version of the test application (as described in Section 4.3.9) was used to test the scalability of the RDC API. Due to instability of OpenSSL's library, only tests for the *plain* scenario were executed.

5.5.1 Local Scalability Tests

A script was prepared to automate the testing process. Five tests were ran for each combination of IPC interface and number of peers, for 30 seconds each. All results were then averaged.

During the initialization of a message queue, an error may occur if the total amount of memory requested surpasses the maximum amount allowed by the operating system. Due to the large amount of peers involved in the stability tests, the default message queue limit had to be increased. This can be done in two different ways. The first method involves the usage of the `ulimit` tool:

```
diogo@ubuntu:~/Desktop$ ulimit -q [SIZE]
diogo@ubuntu:~/Desktop$
```

Where “SIZE” is the desired message queue length or “unlimited” for unlimited queue size. This sets the message queue limit for the current user, but it's only temporary and lasts only as long as the user is logged in. A permanent, system-wide solution involves appending the following lines to the file `/etc/security/limits.conf`:

```
hard msgqueue [SIZE]
soft msgqueue [SIZE]
```

The first method was used to temporarily remove this limit.

Figures 5.11, 5.12 and 5.13 shows the results obtained for message sizes of 128, 1024 and 16384 bytes, respectively. From these it is possible to see that message queues offer slightly more linear results than TCP/IP sockets, particularly for smaller package sizes. The differences in scalability between both IPC mechanisms are not very significant, except when working with small data packets. Unfortunately, message queues still provided a lower bandwidth than TCP/IP sockets for local IPC. This is probably due to the same reason described in section 5.4.1, but in these tests all peers are both sending *and* receiving data (and not just one at the same time), so the penalty in overhead is even more substantial. Message queues ended up providing a throughput advantage only for big packet sizes.

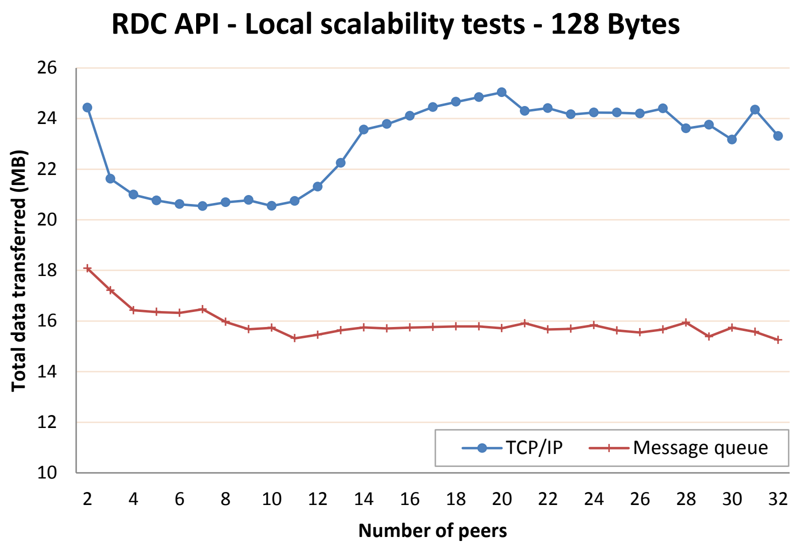


Figure 5.11: RDC API's local scalability test results (128 Bytes)

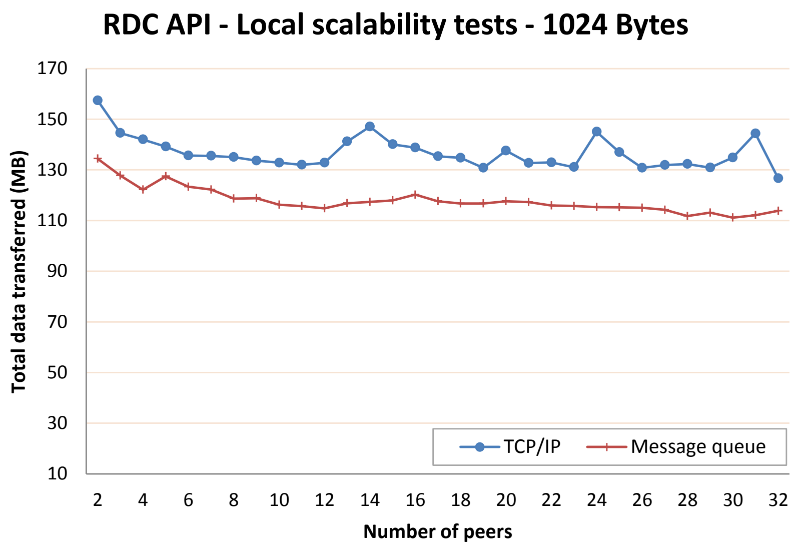


Figure 5.12: RDC API's local scalability test results (1024 Bytes)

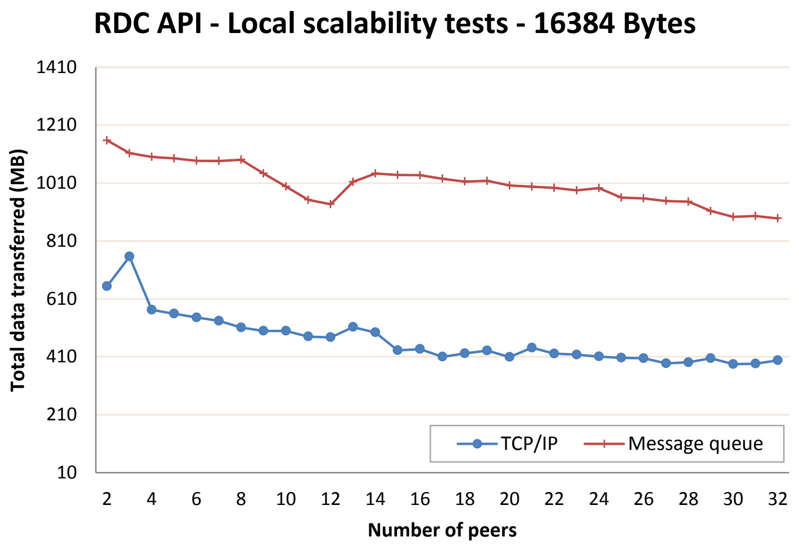


Figure 5.13: RDC API's local scalability test results (16384 Bytes)

5.5.2 Remote Scalability Tests

To evaluate the remote IPC scalability of the RDC API, three tests were executed for 60 seconds for each combination of total number of peers and data packet size. All values were then averaged. The tests began with just Computer A and Computer C1, and more computers from the group C were progressively added until a maximum of 5 elements was reached. Figure 5.14 shows the overall throughput of the system.

For the reasons described in Section 5.1, the resources available for this test were much less than ideal. It's not clear how the differences in the configuration of the computers impacts the results, and the values obtained also ended up with discrepancies between them, leaving little credibility to these values. A much better test would involve a group of more modern computers with identical configuration and a faster and more stable network. Unfortunately, such hardware was not available for the time reserved for the testing phase of this dissertation.

5.6 Multiple Simultaneous IPC Mechanisms

Additional tests were set up to test the multiple IPC interface capability of the RDC API. The version of `rdc_test` used to test the scalability of the API was used again to measure the differences of throughput when only one (TCP/IP) or both types of interfaces were used. Two instances of the API were executed for 60 seconds in each of the two different computers (A and B), for a total of four processes. Each test was repeated three times and all

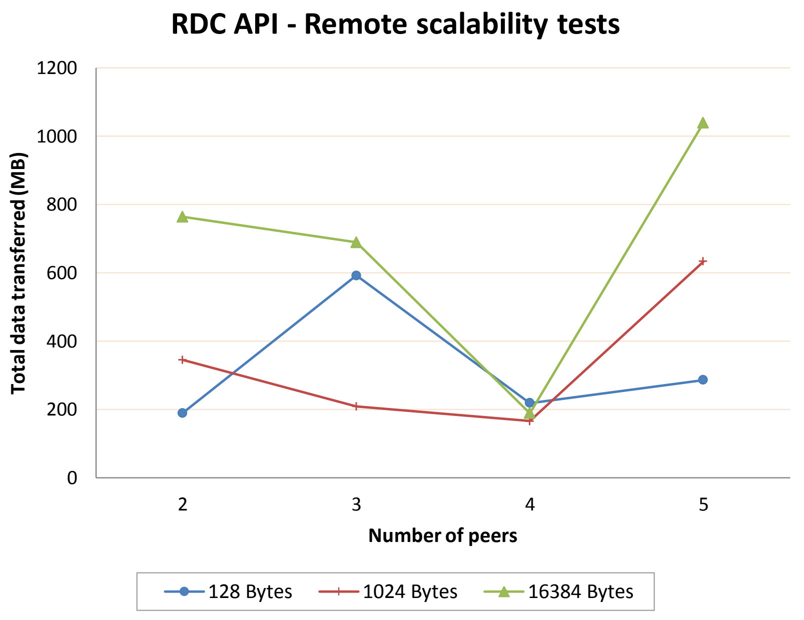


Figure 5.14: RDC API's remote scalability test results

values were averaged. Figure 5.15 shows the results.

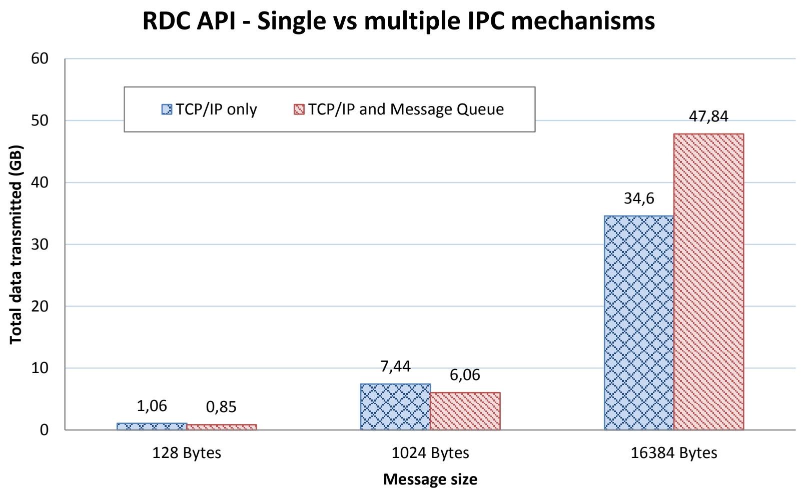


Figure 5.15: RDC API's single vs multiple IPC mechanisms

As expected from the previous tests, only large packet sizes give the API a throughput advantage when using message queues. Another reason may be related to the saturation of the ethernet link.

5.7 Conclusion

The API's prototype has proven to be fully functional and relatively stable, the latter being held back mostly due to concurrency bugs present on the OpenSSL's library. Performance tests have shown that the API would still require a bit more work to be able to extract the throughput potential of the multi-IPC capability. Nevertheless, the framework is built and functional, and an architectural change of the message queue's implementation code should be enough to boost the results. Other mechanisms (such as shared memory) could also be attempted for an higher throughput.

Unfortunately, the hardware available made for testing conditions that were much less than ideal. The results obtained for remote scalability tests are not trustworthy, and they ought to be repeated (if possible) in better conditions.

Some bugs in the API only started to show up during the functional/stability tests phase, where a large number of peers executed simultaneously. These bugs were particularly hard to diagnose and fix because it was necessary to replicate the correct sequence of events for them to show up, sometimes over multiple instances of the API. The debugging mechanisms embedded in the RDC API's code helped to track most bug's origins, but a closer an detailed examination with `gdb` was frequently required.

Further testing of the API could be done involving different configurations of the system, such as different kernel versions, architectures, hardware configurations (single-core vs multi-core CPUs), etc. Other tests could involve a comparison between the results from the RDC API and a similar third party API, which wasn't done due to the difficulty in finding similar APIs to the one developed. Further tests could also be done for the UDP-based alternative protocols described in Section 3.4.3, which were not made due to time constraints.

Chapter 6

Conclusion

This dissertation is the result of many thousands of hours of work, in which a prototype was planned, implemented and tested successfully. The potential for further development of this API is still significant, and this dissertation should provide a good starting point for future work on the subject.

All objectives identified in Sections 1.3 and 3.1 were met, and the majority of the desirable characteristics identified in Section 3.1 were met as well. No special tweaks for RoutUM were implemented in the API since no requests ended up being made from the RoutUM's development team. The only exception is the inclusion of support for urgent messages, which resulted in the creation of an high-priority code path in the API.

Despite all the work put into the software package, it is still not ready to be used in a production environment. Before it can reach this status, its reliability has to be improved (namely due to the problems identified in the OpenSSL library) and some minor functionality should be added to the system and issues corrected (e.g. the ability to split messages transmitted by message queues, when their size surpass what's allocated).

One of the most difficult and time-consuming tasks during the whole dissertation was finding a good architecture for the peer API. In one hand, it had to offer a good degree of readability and modularization (for further development) and at the same time it had to be as efficient as possible. These two priorities often translate into opposing choices, and a lot of thought had to be put into making the right decisions. The resulting compromise made up for a very complex base architecture, which went through various changes until the final version (presented here) was implemented. But until this version was developed, a lot of time was spent writing and rewriting less than ideal architectures several times over. And even after all the work, there are still some leads that a better architecture could have been achieved. In any case, the work presented here and the conclusions that were taken should provide a solid starting point for future developments.

The OpenSSL library forced me to do some unexpected changes to the system, and its instability prevented me from completing the tests as they were planned. Obviously, changes will have to be made to the API in order for it to become a viable proposition for a production environment. OpenSSL's notoriously poor documentation also meant that more work was needed to implement the cryptographic functions than expected.

As for the research phase, I couldn't find any good references about similar APIs as the one that was developed, which made up for some guesswork during the planning phase. The resources obtained about IPC performance tests were also poor and the internal tests were incomplete.

One of the most important lessons learned during development is the importance of effective debugging mechanisms when dealing with distributed systems and highly threaded code. The very large majority of the development time was spent with debugging, and the total time spent with the development largely surpassed what was planned from the beginning. Despite the debugging mechanisms that were implemented, some bugs took almost a week of work to find and fix.

Another important lesson was taken from the testing phase. As seen from the throughput results obtained by the API, synthetic benchmarks were not enough to reliably choose the highest performing IPC mechanism for an application. These raw benchmarks (throughput, delay, error rate, etc) are only as important as the type of IPC needed by the application in question. Other factors such as identification of incoming messages, CPU usage and parallelization of tasks, tracking of connection status, etc. can have a big impact on IPC performance as well. The issues related to the lower bandwidth of local IPC (for other than large data packets) showed me that I underestimated the importance of parallelization and CPU usage during the implementation phase. Despite this, I fully believe that architecture adjustments to the message queue implementation would allow it to better extract the throughput advantages of message queues, but not without adding complexity and resource usage of the system. This solution would involve the usage of a single message queue between each pair of peers in the system, thus allowing an increased level of parallelization of tasks and the creation of a faster reception thread. This was not implemented due to time limitations.

The theme of this dissertation demanded the software to be developed in API form. Despite this, I believe that there could be significant performance advantages if a more integrated approach was taken with RoutUM instead. In theory, this could allow higher code efficiency with less modularization, by unifying duplicate data and taking advantage of variable lookups that otherwise would have to be done twice. For instance, the data structures used to identify peers in the API could be coupled with RoutUM's own `structs`, thus reducing the need to lookup data from the same peer twice (one from the application and one from the API) when sending application-level messages. Another example is related to RoutUM's PDU structure, which could be integrated with that of RDC API, allowing a slight reduction of overhead by removing redundant fields.

Despite all of this, I consider the knowledge taken from this dissertation to be very valuable, as it encompasses a multitude of areas of knowledge, including networks, distributed systems, programming, optimization, etc. The development of the API gave a deeper understanding of the problems that are generally involved in the development of distributed systems. I also acquired a lot of secondary but useful knowledge during the development of the software, from the time that I spent debugging with `gdb`, working out issues with linkers (e.g. when I attempted to mix C with C++), figuring out what existing tools and systems I could use to solve various problems, etc.

6.1 Future Work

The resulting software package from this dissertation is merely a prototype and as such it can be further developed in several areas. Besides improvements related to efficiency and stability of the code, some additional and useful features could be added to it. Some suggestions may include:

- **Message delivery confirmation** - Some IPC mechanisms do not posses delivery confirmation, and even when they do it's not possible to be sure if a message was delivered successfully or not. A message may also not be delivered due to other factors, such as corruption, interception, etc. This feature is already included in RoutUM, but an API level implementation would make it accessible for any applications that implements it.

- **Usage of multiple simultaneous IPC interfaces between peers** - This feature would introduce advantages related to throughput and fault tolerance but could significantly increase code complexity. Causality between all received and transmitted messages would have to be kept between IPC interfaces with different transmission characteristics (bandwidth, delay, etc). Smart algorithms would have to be developed for when small "question-reply" data packets are transmitted, or this feature could potentially become an hindrance as the throughput advantage would be nullified by an increase in delay.

- **An improved error report system** - With a more detailed and explicit error report system, the application would have more information about issues with the API and thus be more prepared to take measures to correct said issues. A simple example would be when the initialization of an interface fails during `rdc_init()`. In the current implementation, the application has no way of knowing which and why an interface has failed to initialize.

- **Adding a timed event queue mechanism** - This feature would be the equivalent of an event queue (as implemented by `evt_queue.h` and `evt_queue.c`, but its events would only trigger after a specified amount of time. As long as the trigger time wasn't reached, the programmer would have the option to remove that event from the queue. This feature could be useful for a number things such as knowing if a peer is still reachable after a long period of inactivity, or creating automatic and regular database updates.

- **Make security mechanisms dependent on the type of interface** - Some IPC mechanisms may require a lower level of security than others, in which case using authentication and/or confidentiality would simply be a waste of resources. Additionally, different ciphers could be chosen by context, type of interface, etc.

- **Local shared memory pool** - It may be desirable to include the ability to have a portion of shared memory between several peers in the same system. If a peer had a big portion of memory that had to be transmitted to other peers, this would drastically reduce the amount of memory that had to be transmitted, while also allowing random access to said data.

- **Server-side authentication** - This could avoid potential issues related to server forgery.

Bibliography

[1] *Alternate IPC Mechanisms: A Comparison of Their Use Within an ORB Framework*. Objective Interface Systems, 2002.

[2] G.S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin-Cummings publishers, Redwood City, CA., 1989.

[3] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison Wesley, 1999.

[4] Abhijit Belapurkar et al. *Distributed Systems Security: Issues, Processes and Solutions*. John Wiley & Sons, Ltd, 2009.

[5] Tony F. Chan, Gene H. Golub, and Randall J. LeVeque. *Algorithms for Computing the Sample Variance: Analysis and Recommendations*. Tech. rep. The American Statistician, 1983.

[6] routUM: Computer Communications Group. *A Modular Distributed Computer Networks Simulator*. Tech. rep. Department of Informatics, School of Engineering, University of Minho, 2007.

[7] L. Ferreira et al. *Introduction to Grid Computing with Globus*. IBM Redbooks, 2003.

[8] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[9] Patricia K. Immich, Ravi S. Bhagavatula, and Dr. Ravi Pendse. "*Performance Analysis of Five Interprocess Communication Mechanisms Across UNIX Operating Systems*". In: The Journal of Systems and Software 68 (2003), pp. 27–43.

[10] R. G. Ingalls. "*Introduction to Simulation*". In: WSC'02: Proceedings of the 34th Conference on Winter Simulation, 2002.

[11] Teerawat Issariyakul and Ekram Hossain. *Introduction to Network Simulator NS2*. Springer, 2010. ISBN: 1441944125.

[12] Leonard Kleinrock. *Queueing Systems. Volume 1: Theory*. Wiley-Interscience, 1975. ISBN: 0471491101.

[13] Ajay K. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms and Systems*. Cambridge University Press, 2008.

[14] *Distributed Cooperative Apache Web Server*. WWW10 - Tenth International World Wide Web Conference. 2001.

[15] Robert F. Ling. *Comparison of Several Algorithms for Computing Sample Means and Variances*. Tech. rep. The American Statistician, 1974.

[16] Manolescu, D. Beckman, and B. B. Livshits. *Volta: Developing Distributed Applications by Recompiling*. Microsoft Live Labs., Redmond, WA, 2008.

[17] Masayoshi Nabeshima and Kouji Yata. "*Performance Evaluation and Comparison of Transport Protocols for Fast Long-Distance Networks*". In: *IEICE Transactions on Communications, Vol. E89-B* (2006).

[18] Ana Nunes and Sara Fernandes. *Communication and Resource Sharing Management Protocols for the ROUTUM Network Simulator*. Tech. rep. Department of Informatics, School of Engineering, University of Minho, 2007.

[19] Jianli Pan. *A Survey of Network Simulation Tools: Current Status and Future Developments*. Tech. rep. Washington University in St. Louis, 2008.

[20] Papadimitriou and Christos H. *Computational Complexity*. Addison-Wesley, 1994.

[21] *Parallel Simulation Made Easy with OMNeT++*. Delft, The Netherlands: Proceedings of European Simulation Symposium, 2003.

[22] Pedro Silva. *RoutUM - Sincronização Temporal*. Tech. rep. Department of Informatics, School of Engineering, University of Minho, 2007.

[23] Pedro Sousa. *RoutUM - Core*. Tech. rep. Department of Informatics, School of Engineering, University of Minho, 2006/2007.

[24] W. Richard Stevens. *UNIX Network Programming: Networking APIs: Sockets and XTI; Volume 1*. Prentice Hall PTR, 1998. ISBN: 013490012X.

[25] Klaus Wehrle, Mesut Günes, and James Gross. *Modelling and Tools for Network Simulation*. Springer, 2010. ISBN: 3642123309.

[26] Kwame Wright, Kartik Gopalan, and Hui Kang. "*Performance Analysis of Various Mechanisms for Inter-Process Communication*". In: (2007).

Appendix A

Code Headers

A.1 Database API

db.h

```
#include <stdio.h>
#include <pthread.h>
#include "../shared/dtypes.h"

#ifndef _RDC_DB_H
#define _RDC_DB_H

    typedef struct {
        void*           db_handle;
        pthread_mutex_t db_mutex;
    } rdc_db_t;

  struct rdc_db_cfg {
    char* path;
  };

    int  rdc_db_connect(rdc_db_t* db, struct
        rdc_db_cfg* cfg, bool create);
    void rdc_db_disconnect(rdc_db_t* db);

    int  rdc_db_add_context(rdc_db_t* db, int* id,
        const char* name);
    int  rdc_db_remove_context(rdc_db_t* db, const
        char* name);
    int  rdc_db_get_contexts(rdc_db_t* db, struct
        rdc_ctx** context, int context_id, int peer_id,
         bool exclusive);
    int  rdc_db_update_security(rdc_db_t* db, const
        char* context_name, bool use_conf, bool
        use_auth);

    int  rdc_db_add_peer(rdc_db_t* db, int* id, const
        char* name, rdc_cry_keypair_t** key);
    int  rdc_db_remove_peer(rdc_db_t* db, const char*
        name);
    int  rdc_db_get_peers(rdc_db_t* db, struct
        rdc_peer** peer, int context_id, int peer_id,
        char* peer_name);
    int  rdc_db_get_interfaces(rdc_db_t* db, struct
        rdc_if** intf, int peer_id);
    int  rdc_db_update_interfaces(rdc_db_t* db, const
        char* peer_name, struct rdc_if* interface);
    int  rdc_db_update_hostid(rdc_db_t* db, const char
        * peer_name, long host_id);

    int  rdc_db_add_membership(rdc_db_t* db, const
        char* context_name, const char* peer_name);
    int  rdc_db_get_memberships(rdc_db_t* db, struct
        rdc_ctx** context, int context_id);

#endif
```

A.2 Peer API

rdc.h

```
#include <netdb.h>
#include <pthread.h>
#include <stdbool.h>

#include "api_dtypes.h"

#ifndef _RDC_H
#define _RDC_H

  int  rdc_init(rdc_t* rdc, struct rdc_cfg* cfg);
  void rdc_destroy(rdc_t* rdc);
  int  rdc_sendmsg(rdc_t* rdc, int peer_id, int
      src_cid, int dst_cid, char* msg, unsigned int
      msg_len, bool urgent);
  int  rdc_recv_evt(rdc_t* rdc, struct rdc_evt* msg);
  void rdc_free_evt(struct rdc_evt* evt);

  struct rdc_peer_ex* rdc_get_peers(rdc_t* rdc, int
      peer_id);
  struct rdc_ctx_ex*  rdc_get_contexts(rdc_t* rdc, int
       ctx_id);

  int rdc_update_db(rdc_t* rdc, unsigned int* added,
      unsigned int* removed, unsigned int* total);

#endif /* _RDC_H */
```

api_dtypes.h

```
#include <time.h>

#include "../shared/crypto.h"
#include "../shared/dtypes.h"
#include "btree.h"
#include "evt_queue.h"

#ifndef _RDC_API_DTYPES_H
#define _RDC_API_DTYPES_H

  #define RDC_MSG_TYPE_MASK   0XF8
  #define RDC_MSG_TYPE_FLAG   0XB0
  #define RDC_AUTH_TIMEOUT    5
  #define RDC_AUTH_SCRT_LEN   16
  #define RDC_AUTH_ESCRT_LEN  128

  union int4
  {
    unsigned int in;
    char         cn[4];
  };

  #define _RDC_EVT_NEW_CONN   40
  #define RDC_EVT_MSG_RECV    60
  #define RDC_EVT_UMSG_RECV   61
  #define RDC_EVT_IFU_OK      62
  #define RDC_EVT_IFU_FAIL    63
  #define RDC_EVT_AUTH_FAIL   64
  #define RDC_EVT_IF_DOWN     65
  #define RDC_EVT_IF_UP       66
  #define RDC_EVT_PEER_NEW    67
  #define RDC_EVT_PEER_DEL    68
  #define RDC_EVT_CTX_NEW     69
  #define RDC_EVT_CTX_DEL     70
  #define RDC_EVT_PEER_DOWN   71
  #define RDC_EVT_SHUTDOWN    72

  #define RDC_MSGTYPE_NCON 0x02
  #define RDC_MSGTYPE_AUTH 0x07
  #define RDC_MSGTYPE_SHUT 0x06
  #define RDC_MSGTYPE_UMSG 0x01
  #define RDC_MSGTYPE_MSG  0x00
  #define RDC_MSGTYPE_IFU  0x04

  enum rdc_if_type {
    IFTYPE_TCPIP,
    IFTYPE_MQUEUE,
    _IFTYPE_TOTAL
  };
  typedef struct {
    int   type;
    void* data;
  bool   req_msg_len;
  bool   req_dst_id;
  bool   req_src_id;
} rdc_ipc_t;

struct rdc_if_ex {
  int                  type;
  void*                cfg;
  struct rdc_if_ex*    next;
};
struct rdc_peer_ex {
  int                  id;
  char*                name;
  long                 host_id;
  int                  if_type;
  void*                if_handle;
  rdc_cry_keypair_t*   pubk;
  rdc_cry_key_t*       skey;
  unsigned long long   sent_bytes;
  unsigned long long   sent_msg;
  unsigned long long   recv_bytes;
  unsigned long long   recv_msg;
  pthread_rwlock_t     wmutex;
  pthread_mutex_t      smutex;
  pthread_mutex_t      auth_mutex;
  unsigned char*       auth_reply;
  char                 auth_state;
  bool                 if_updated;
  struct rdc_ctx_ex**  ctx;
  struct rdc_if_ex*    first_if;
  struct rdc_peer_ex*  next;
};
struct rdc_ctx_ex {
  int                  id;
  char*                name;
  bool                 req_auth;
  bool                 req_conf;
  struct rdc_ctx_ex*   next;
};

struct rdc_if_tcpip_cfg {
  unsigned short lstn_port;
  char*          lstn_addr;
};
struct rdc_if_mqueue_cfg {
  unsigned int max_msg;
  unsigned int msg_len;
};
struct rdc_cfg {
  char*                        name;
  char*                        privk_path;
  char*                        srv_addr;
  unsigned short               srv_port;
  bool                         use_dns;
```

```
    struct rdc_if_tcpip_cfg*  tcpip;
    struct rdc_if_mqueue_cfg* mqueue;
  };

  typedef struct {
    int                  id;
    long                 host_id;
    struct rdc_cfg*      cfg;
    rdc_cry_keypair_t*   pk;
    struct btree         peer_db;
    struct btree         ctx_db;
    struct rdc_peer_ex*  first_peer;
    struct rdc_ctx_ex*   first_ctx;
    void*                ser_ifs;
    unsigned int         ser_ifs_len;
    bool                 run;
    rdc_eq_t*            in_queue;
    rdc_eq_t*            out_queue;
    pthread_t            evt_thread;
    pthread_t            ifu_thread;
    pthread_rwlock_t     db_lock;
    rdc_ipc_t*           ifs[_IFTYPE_TOTAL];
  } rdc_t;

  unsigned int rdc_dt_db_update(rdc_t* rdc, void* buf,
       int peer_id,
                                struct rdc_peer_ex**
                                    peers_remove,
                                    struct rdc_ctx_ex**
                                     ctxs_remove,
                                struct rdc_peer_ex**
                                    peers_add, struct
                                    rdc_ctx_ex**
                                    ctxs_add);
  void rdc_dt_db_destroy(rdc_t* rdc);
  void rdc_dt_remove_ctx(struct rdc_ctx_ex** ctx);
  void rdc_dt_remove_peer(struct rdc_peer_ex** peer);
  struct rdc_peer_ex* rdc_dt_get_peer(rdc_t* rdc, int
      id);
  struct rdc_ctx_ex* rdc_dt_get_context(rdc_t* rdc,
      int id);

#endif
```

btree.h

```
#include <stdbool.h>

#ifndef _BTREE_H
#define _BTREE_H

  struct value_s {
    int    val;
    void*  data;
  };

  typedef struct value_s* value_t;

  typedef struct node_s {
     value_t          value;
     struct node_s *next[2];
     int              longer:2;
  } *node;

  struct btree {
    node          tree;
    unsigned int  total;
  };

  void btree_init(struct btree* tree);
  void btree_destroy(struct btree* tree);
  bool btree_add(struct btree* tree, int val, void*
      data);
  bool btree_remove(struct btree* tree, int val);
  void* btree_search(struct btree* tree, int val);
  unsigned int btree_size(struct btree* tree);

#endif
```

evt_queue.h

```
#pragma once

#include <semaphore.h>

#include "../shared/dtypes.h"

#ifndef _RDC_EVT_QUEUE_H
#define _RDC_EVT_QUEUE_H

  struct rdc_evt {
    int    type;
    int    arg1;
    int    arg2;
    int    arg3;
    void*  arg4;
    int    len;
  };

  typedef struct {
    struct rdc_evt*  buffer;
    unsigned int     size;
    unsigned int     rpos;
    unsigned int     wpos;
    unsigned int     total;
    sem_t            rlock;
    sem_t            wlock;
    pthread_mutex_t  wmutex;
    pthread_mutex_t  rmutex;
    bool             closed;
  } rdc_eq_t;

  bool rdc_eq_init(rdc_eq_t* queue, unsigned int size)
      ;
  int rdc_eq_put_tail(rdc_eq_t* queue, int type, int
      arg1, int arg2, int arg3, void* arg4, int len);
  int rdc_eq_put_head(rdc_eq_t* queue, int type, int
      arg1, int arg2, int arg3, void* arg4, int len);
  int rdc_eq_get(rdc_eq_t* queue, int* type, int* arg1
      , int* arg2, int* arg3, void** arg4, int* len);
  int rdc_eq_get_total(rdc_eq_t* queue);
  void rdc_eq_destroy(rdc_eq_t* queue);

#endif
```

ipc.h

```
#include <netinet/in.h>

#include "api_dtypes.h"

#ifndef _RDC_IPC_H
#define _RDC_IPC_H


  int rdc_ipc_init(rdc_t* rdc, rdc_ipc_t** intf, void*
       cfg, rdc_eq_t* evt_q, enum rdc_if_type iftype);
  void rdc_ipc_destroy(rdc_t* rdc, rdc_ipc_t** intf);

  int rdc_ipc_handle_create(rdc_t* rdc, rdc_ipc_t*
      intf, struct rdc_peer_ex* peer, void** out_handle
      );
  void rdc_ipc_handle_destroy(rdc_t* rdc, rdc_ipc_t*
      intf, void** handle);
  int rdc_ipc_handle_sendmsg(rdc_t* rdc, rdc_ipc_t*
      intf, void* handle, void* msg, unsigned int
      msg_len);
  void rdc_ipc_handle_get_type(rdc_t* rdc, struct
      rdc_peer_ex* peer, int* types[], unsigned int*
      types_len);

  unsigned int rdc_ipc_cfg_serialize_len(int iftype,
      void* cfg_struct);
  void rdc_ipc_cfg_serialize(int iftype, void*
      cfg_struct, void** buf, unsigned int* buf_len);
  void rdc_ipc_cfg_deserialize(struct rdc_if_ex* intf,
       void* buf);
  void rdc_ipc_cfg_destroy(int iftype, void* cfg);

#endif
```

mqueue.h

```
#include "../ipc.h"

#ifndef _RDC_IPC_MQUEUE_H
#define _RDC_IPC_MQUEUE_H

  #define MQ_SEND_TIMEOUT_S 5

  int  rdc_ipc_mqueue_init(rdc_t* rdc, rdc_ipc_t**
       intf, struct rdc_if_mqueue_cfg* cfg, rdc_eq_t*
       evt_q);
  void rdc_ipc_mqueue_destroy(rdc_t* rdc, rdc_ipc_t**
       intf);

  int  rdc_ipc_mqueue_handle_create(rdc_t* rdc,
       rdc_ipc_t* intf, struct rdc_peer_ex* peer, void**
        out_handle);
  void rdc_ipc_mqueue_handle_destroy(rdc_t* rdc,
       rdc_ipc_t* intf, void** handle);
  int  rdc_ipc_mqueue_handle_sendmsg(rdc_t* rdc,
       rdc_ipc_t* intf, void* handle, void* msg,
       unsigned int msg_len);

  unsigned int rdc_ipc_mqueue_cfg_serialize_len(struct
        rdc_if_mqueue_cfg* cfg_struct);
  void rdc_ipc_mqueue_cfg_serialize(struct
       rdc_if_mqueue_cfg* cfg_struct, void** buf,
       unsigned int* buf_len);
  void rdc_ipc_mqueue_cfg_deserialize(struct rdc_if_ex
       * intf, void* buf);
  void rdc_ipc_mqueue_cfg_destroy(void* cfg);

#endif
```

tcpip.h

```
#include "../ipc.h"

#ifndef _RDC_IPC_TCPIP_H
#define _RDC_IPC_TCPIP_H

  int  rdc_ipc_tcpip_init(rdc_t* rdc, rdc_ipc_t** intf
       , struct rdc_if_tcpip_cfg* cfg, rdc_eq_t* evt_q);
  void rdc_ipc_tcpip_destroy(rdc_t* rdc, rdc_ipc_t**
      intf);

  int  rdc_ipc_tcpip_handle_create(rdc_t* rdc,
      rdc_ipc_t* intf, struct rdc_peer_ex* peer, void**
       out_handle);
  void rdc_ipc_tcpip_handle_destroy(rdc_t* rdc,
      rdc_ipc_t* intf, void** handle);
  int  rdc_ipc_tcpip_handle_sendmsg(rdc_t* rdc,
      rdc_ipc_t* intf, void* handle, void* msg,
      unsigned int msg_len);

  unsigned int rdc_ipc_tcpip_cfg_serialize_len(struct
      rdc_if_tcpip_cfg* cfg_struct);
  void rdc_ipc_tcpip_cfg_serialize(struct
      rdc_if_tcpip_cfg* cfg_struct, void** buf,
      unsigned int* buf_len);
  void rdc_ipc_tcpip_cfg_deserialize(struct rdc_if_ex*
       intf, void* buf);
  void rdc_ipc_tcpip_cfg_destroy(void* cfg);

#endif
```

A.3 Shared

crypto.h

```
#pragma once

#include <openssl/rsa.h>
#include <stdbool.h>

#ifndef _RDC_CRYPTO_H
#define _RDC_CRYPTO_H

  #define RDC_CRY_SYM_KEYLEN    16
  #define RDC_CRY_PUBK_MAX_LEN 272
  #define RDC_CRY_HASH_ALG      "SHA1"
  #define RDC_CRY_SIGN_SIZE     128

  typedef struct {
    RSA*       kp;
    EVP_PKEY* skey;
  } rdc_cry_keypair_t;
  typedef struct {
    unsigned char key[RDC_CRY_SYM_KEYLEN];
    unsigned char iv[RDC_CRY_SYM_KEYLEN];
    EVP_CIPHER_CTX* e_ctx;
    EVP_CIPHER_CTX* d_ctx;
  } rdc_cry_key_t;

  bool rdc_cry_genkey(rdc_cry_key_t** outkey);
  void rdc_cry_free_key(rdc_cry_key_t** key);
  bool rdc_cry_encrypt(rdc_cry_key_t* key, void* in,
      unsigned int in_len, void** out, unsigned int*
      out_len);
  bool rdc_cry_decrypt(rdc_cry_key_t* key, void* in,
      unsigned int in_len, void** out, unsigned int*
      out_len);

  bool rdc_cry_genkeypair(rdc_cry_keypair_t** key);
  void rdc_cry_free_keypair(rdc_cry_keypair_t** key);
  bool rdc_cry_encrypt_pubk(rdc_cry_keypair_t* key,
      void* in, unsigned int in_len, void** out,
      unsigned int* out_len);
  bool rdc_cry_encrypt_privk(rdc_cry_keypair_t* key,
      void* in, unsigned int in_len, void** out,
      unsigned int* out_len);
  bool rdc_cry_decrypt_privk(rdc_cry_keypair_t* key,
      void* in, unsigned int in_len, void** out,
      unsigned int* out_len);
  bool rdc_cry_decrypt_pubk(rdc_cry_keypair_t* key,
      void* in, unsigned int in_len, void** out,
      unsigned int* out_len);
  bool rdc_cry_sign(rdc_cry_keypair_t* key, void* in,
      unsigned int in_len, void** sign, unsigned int*
      sign_len);
  bool rdc_cry_verify(rdc_cry_keypair_t* key, void* in
      , unsigned int in_len, void* sign, unsigned int
      sign_len);
  bool rdc_cry_load_pubk(rdc_cry_keypair_t** out, void
      * buf, unsigned int buf_len);
  bool rdc_cry_load_privk(rdc_cry_keypair_t** out,
      char* path);
  bool rdc_cry_save_privk(rdc_cry_keypair_t* key, char
      * path);
  bool rdc_cry_serialize_pubk(rdc_cry_keypair_t* key,
      void** out, unsigned int* out_len);

  unsigned char* rdc_cry_randword(int length);

#endif
```

dtypes.h

```
#include <stdbool.h>

#include "crypto.h"

#ifndef _RDC_DTYPES_H
#define _RDC_DTYPES_H

  #define RANDW_SIZE 32
```

```

#define RDC_OK              0
#define RDC_ERROR           1
#define RDC_ERROR_CONF      2
#define RDC_ERROR_IF_INIT 3
#define RDC_ERROR_CONN      4
#define RDC_ERROR_AUTH      5
#define RDC_ERROR_FREAD     6
#define RDC_ERROR_INT       7
#define RDC_ERROR_PEER_NF 8
#define RDC_ERROR_CTX_NF    9
#define RDC_ERROR_DNS_ERR 10
#define RDC_ERROR_SHUTD     11
#define RDC_ERROR_TBIG      12
#define RDC_ERROR_EXT       13
#define RDC_WARN_NOPEERS    50
#define RDC_WARN_CONF       51
#define RDC_WARN_INT        52

struct rdc_if {
  int            type;
  void*          cfg;
  unsigned int   cfg_len;
  struct rdc_if* next;
};
struct rdc_peer {
  int                 id;
  char*               name;
  long                host_id;
  char*               created;
  rdc_cry_keypair_t* pubk;
  char*               pubk_ser;
  unsigned int        pubk_len;
  struct rdc_if*      first_if;
  struct rdc_peer*    next;
};
struct rdc_ctx {
  int               id;
  char*             name;
  bool              req_auth;
  bool              req_conf;
  char*             created;
  struct rdc_peer* first_peer;
  struct rdc_ctx*   next;
};

void rdc_dt_serialize_contexts(struct rdc_ctx* in,
    void** out, unsigned int* out_len, int
    excl_peer_id);
void rdc_dt_deserialize_contexts(void* in, struct
    rdc_ctx** out);
void rdc_dt_serialize_interfaces(struct rdc_if* in,
    void** out, unsigned int* out_len);
    void rdc_dt_deserialize_interfaces(void* in, struct
        rdc_if** out);
    void rdc_dt_free_contexts(struct rdc_ctx** context);
    void rdc_dt_free_peers(struct rdc_peer** peer);

#endif
```

A.4 Server

cf_parser.h

```
#include <stdbool.h>

#ifndef _CF_PARSER_H
#define _CF_PARSER_H

    bool cp_open(const char* path, void(*func)(char*,
        char*, void*), void* arg);

#endif
```

const.h

```
#ifndef _RDC_CONST_H
#define _RDC_CONST_H

    #define RDC_CFG_FILE "../cfg/rdcd.conf"

#endif /* _RDC_CONST_H */
```