PARIPARI: CONNECTIVITY OPTIMIZATION

RELATORE: Ch.mo Prof. Enoch Peserico Stecchini Negri De Salvi
CORRELATORE: Ing. Paolo Bertasi
LAUREANDO: Francesco Peruch

Corso di laurea in Ingegneria Informatica

A.A. 2010-2011

UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

PARIPARI: CONNECTIVITY OPTIMIZATION

RELATORE: Prof. Enoch Peserico Stecchini Negri De Salvi

CORRELATORE: Ing. Paolo Bertasi

LAUREANDO: Francesco Peruch

A.A. 2010-2011

To Valentina

Contents

Abstract

PariPari is a multi-functional and extensible P2P platform, currently developed at the Department of Information Engineering of the University of Padova.

Its plug-in architecture, accessible through a set of clear and simple APIs, considerably eases the addition of new features: the variety and complexity of its modules – from file sharing to VoIP, from Web hosting to distributed storage – are expected to increase, eventually integrated by third party applications. This development requires *scalability*, *transparency*, *efficiency*, *security* and *anonymity* qualities.

PariConnectivity is the plug-in that provides Network Communication APIs and aims to ensure these qualities regarding the network I/O.

In its first implementation, it was a substantial reworking of the already existing Java libraries: though simple, this solution soon revealed its inadequacy. This thesis illustrates how PariConnectivity has been *reengineered* and *optimized*. Through the development of a new set of APIs, built on the Java NIO libraries, we have created a new network abstraction layer, providing easiness-to-use and efficiency thanks to the introduction of advanced features as, for instance, the asynchronous I/O. On this basis, we have introduced and enhanced some crucial services like bandwidth limitation, NAT Traversal, Multicast and Anonymity.

Sommario

PariPari è una piattaforma P2P multifunzionale ed estensibile, attualmente in sviluppo presso il Dipartimento di Ingegneria dell'Informazione dell'Università di Padova.

La sua struttura a plug-in, accessibile tramite un insieme di semplici API, è studiata per facilitare l'aggiunta di nuove funzionalità: la varietà e la complessità dei moduli offerti – che già ora spaziano dal semplice file sharing al VoIP e al file hosting – sono destinate a crescere, anche con l'integrazione di applicazioni prodotte da terze parti. Uno sviluppo di questo genere richiede che vengano garantiti alcuni aspetti fondamentali quali scalabilità,trasparenza, efficienza, sicurezza e anonimato.

PariConnectivity è il plug-in che fornisce le API per le comunicazioni di rete, e , per esse, punta ad assicurare queste caratteristiche.

Nella sua prima implementazione, questo plug-in costituiva sostanzialmente un riadattamento delle librerie Java già esistenti: sebbene semplice, questa soluzione si è presto rilevata inadeguata. Questa tesi descrive come PariConnectivity è stato completamente riprogettato e ottimizzato: lo sviluppo di un nuovo insieme di API, fondate sulle librerie Java NIO, ha fornito un nuovo livello di astrazione per l'accesso alla rete in PariPari, arricchito da un sistema efficiente di I/O asincrono. Su questa base è stato poi possibile realizzare nuovi servizi quali limitazione di banda, NAT Traversal, Multicast e Anonimato.

Chapter 1

Introduction

PariPari is a P2P platform currently developed by more than 60 students at the Department of Information Engineering of the University of Padova. Whereas in its decentralized, serverless architecture it resembles most traditional P2P applications like eMule, Skype or Azureus, PariPari aims at broadening P2P's horizons, pursuing an ambitious goal: it does not focus on a particular service but, instead, provides a multifunctional and extensible platform on which multiple services can run simultaneously and cooperatively.

The number and heterogeneity of provided services – already spacing from file sharing to VoIP, from Web hosting to distributed storage – are still expected to increase, even with the participation of third party developers. In order to achieve this, PariPari exhibits a highly modular architecture, where every service is provided by a plug-in (Chapter 2).

In order to adequately orchestrate this large number of plug-ins, particular attention must be paid to the management of network resources. Requirements to be fulfilled are manifold: some plug-ins need to establish dozens of connections in a few seconds, others may require to simultaneously manage many TCP connections, others ask for some network parameters (for instance, minimum bandwidth or maximum latency) to be guaranteed. In addition, since networking is a basic operation performed by all plug-ins, these features should be easily accessible through a set of advanced, but still clear, APIs.

The necessity of an adequate network abstraction layer naturally arises – and *PariConnectivity* is the plug-in devoted to provide it.

In its first implementation, Connectivity was a substantial reworking of the

already existing Java libraries; though simple, this solution soon revealed its inadequacy. This thesis illustrates how PariConnectivity has been entirely reengineered and optimized.

First of all, we aimed at providing a new set of APIs. Efficiency requirements suggested us the adoption of the new Java NIO libraries: indeed, with respect to the traditional Java libraries, NIO classes achieve a more scalable I/O, better leveraging the I/O capabilities of the underlying operating system (Chapter 3).

Though high-performance, NIO libraries are not so easy to deal with. Therefore, we decided to introduce an intermediate layer between them and the APIs accessed by external plug-ins: this layer translates NIO's structures into more manageable objects, and is provided by a Connectivity's module, the *Connectivity Core* (Chapter 4).

As a further improvement, an asynchronous I/O mechanism was designed: our implementation provides a degree of scalability comparable to that provided by the NIO APIs, exhibiting a more *developer-frienldy* interface (Chapter 5).

On this basis, we introduced and enhanced some crucial services.

PariConnectivity now aims at offering an advanced QoS, relying on a set of bandwidth limitation functionalities that let us better fit the different bandwidth requirements exhibited by different plug-ins (Chapter 6).

Thanks to the NAT Traversal mechanism, enhanced by Tunneling services, now communications can be established even among hosts lying behind NAT and firewall devices; given the configuration adopted by most local networks nowadays, this represents a crucial service (Chapter 7).

Some plug-ins (for instance, those providing distributed hosting) may require multicast or anonymity-granting facilities. These services are provided by two submodules of Connectivity – Multicast (Chapter 8) and Anonymity (Chapter 9), respectively. Both these submodules, already provided by the old Connectivity module, incurred a revision and a profound renewal.

PariConnectivity is developed by students. Leading this special kind of workforce, as well as establishing fruitful relationships with the teams developing other plug-ins, entail the adoption of a well-defined organizational structure and a set of appropriate programming techniques (Chapter 10).

While a hard work has been done, further developments are still possible: they are briefly depicted at the end of this thesis (Chapter 11).

Chapter 2

PariPari

PariPari is a serverless P2P multi-functional application currently under development at the Department of Information Engineering of the University of Padova. It differs from traditional P2P applications like eMule [2], Skype [6] or Azureus [1] in that it provides a *multifunctional* and *extensible* platform: a large number of heterogeneous services, ranging from the traditionally P2P ones – e.g., file sharing and VoIP – to more server-based ones – like Web and mail hosting, IRC chat and DNS hosting – are accessible through a collection of simple and uniform APIs, allowing a large body of students (presently, more than 60 people) to cooperate developing and – in the future – third party developers to write their own applications. One of the key features of PariPari is its highly modular architecture: in this chapter, we briefly review its plug-in organization, orchestrated by the Core (Section 2.1), and its network layout (Section 2.2), trying to better contextualize the role of *PariConnectivity* in the system as a whole (Section 2.3).

2.1 Plug-in architecture

PariPari hosts are based on a plug-in architecture. Users can decide which plug-ins to load, and can load them at runtime; plug-ins can not communicate directly with each other or with the outer world: every request must pass through *PariCore*, the authentic kernel of this system. It takes care to route messages among the running plug-ins, granting PariPari with security checking all messages (Subsection 2.1.1) and exploiting *PariCredits* information (Subsection 2.1.2). Given their crucial role, PariCore and PariCredits modules are described in more details

in the following Subsections.

2.1.1 PariCore

The modular architecture of PariPari relies on a specific plug-in – PariCore **T.A.L.P.A.** (**T**he **A**cronym for **L**ightweight **P**lug-in **A**rchitecture). This Core version was born in January 2009 and was designed to be the kernel of PariPari. Its main functions are managing plug-ins, routing their messages and protecting users and good plug-ins from malicious ones: we face these aspects in more detail in the following.

Managing plug-ins

One of the main features of the Core is granting a high level of *standardization* among plug-ins. All the services plug-ins provide are specified by a set of abstract classes – the *APIs*: every API implements the super interface `paripari.API.API` (which contains, in particular, some methods needed by the *Credit System*) and must be extended by the specific class implementing the service. This distinction between *definition* and *implementation* makes it easier, in particular, for different plug-ins to provide the same service: interferences among such plug-ins (arising, for instance, when using identical class names) are avoided by instantiating a different *classloader* for each plug-in. In every plug-in jar, the file (`descriptor.xml`) must be present: it lists plug-in dependencies as well as provided services.

Routing messages

Communications take place through appropriate calls to the Core. Each plug-in holds two data structures – *Mound Putter* and *Private Mound*. The former is used for the *outgoing* messages: every message a plug-in sends (a request or a response to a previous received message as well) is first dispatched to the Core – which forwards them to other plug-ins when necessary; the latter stores all the requests coming from other plug-ins: in order to readily catch these messages, every plug-in generally holds a `Listener` thread monitoring Private Mound. Figure 2.1 illustrates this mechanism.

Granting security

PariCore manages security-related aspects by means of the *PariPariSecurity-Manager*, which replaces the standard Web Start's Security Manager. Only a few plug-ins – the so called *inner circle* plug-ins – are trusted by default and are allowed to perform potentially dangerous operations, such as disk and network I/O; such inner plug-ins are developed by the PariPari Team and should not be replaced even in the future. Currently, this group counts three modules: *PariStorage* for local file access, *PariConnectivity* for network I/O and *PariDHT* for PariPari network access.

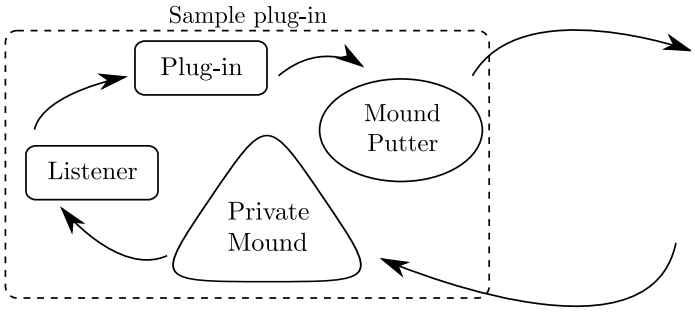

Figure 2.1: How plug-ins deal with messages.

2.1.2 Credit System

A dedicated module handles the *Credit System* of PariPari. Roughly speaking, we can divide the whole Credit System into two layers:

Inter-peer Credits layer, regulating communications among hosts

Intra-peer Credits layer, regulating communications among plug-ins lying in the same host

The Inter-peer Credits mechanism shares some purposes with other usual P2P credit systems: namely, it aims at encouraging participation (for instance, by means of sharing files or offering disk space) and discouraging parasitic behaviors. However, PariPari Inter-peer credits pursue a further goal: this layer – still in development – is intended to create a scalable, transitive and cohesive *barter based* economy among peers [31].

The Intra-peer Credits layer has been almost entirely implemented. This layer, exploiting the fact that all plug-ins share the same goals and hold the same authority, relies on a simple mechanism managed by the Core: the exchange of *tokens*. More precisely, the Core periodically assigns a number of *tokens* to every running plug-in (this amount varies according to the guidelines established by the user – it can be a predefined or an adaptive one); when a plug-in needs some kind of resource from another plug-in, it must *pay* a certain amount of *tokens*. In this way, plug-ins requiring the same resource compete in an auction system, which actually determines the price of the resource itself. Analyzing such prices, plug-ins can choose a strategy that minimizes the expense of *tokens*: for example, a compression of data can be worthwhile if CPU cycles are much cheaper compared to bandwidth. Once the price p of a resource is established, the plug-in *buys* it for a number t of time units paying $p \cdot t$ *tokens*. Note how, in order to obtain a proper charging, plug-ins offering a resource must precisely quantify how much a time unit lasts and notify the Credit module about every change occurred in resource's availability.

2.2 Network layout

The PariPari Network layout is based on a Distributed Hash Table (**DHT**), *PariDHT*, providing a high degree of scalability, decentralization and fault tolerance. Recent years have seen DHTs adopted in several applications with a vast public (e.g. the popular eMule [2] and Azureus [1] filesharing clients and the JXTA system [5])

The classic mechanisms a DHT relies on are well known. Broadly speaking, each node is assigned a number in a d-bit address space – the node*ID*; each resource is represented by a key-value (k, v) pair: the key usually corresponds to a keyword associated with the resource itself and is mapped into a key*ID*, belonging to the same d-bit address space of nodeIDs, by means of a well defined hash function v (that is, keyID$= v(k)$). A specific notion of *distance* is defined among nodeIDs and keyIDs; in these regards, PariDHT adopts a value of d equal to 256, and the same *XOR metric* already employed in Kademlia [28].

Resources are stored and retrieved by nodes thanks to the shared address space: in particular, two fundamental primitives (*store* and *search*) are provided.

When a node *stores* a resource r (corresponding to a key k and a value v), r is assigned to the node n_k belonging to the network whose ID is the closest to $v(k)$ according to the XOR metric. When another node *searches* for that resource, it contacts n_k. As we can easily see, both primitives rely on a mechanism for looking for a node across the network. Thanks to the particular structure owned by PariDHT, a node in the network can contact any other node in $O(\log N)$ hops, where N is the number of active nodes. In order to achieve this, each node n keeps contacts with a small number m of nodes in the other half of the network (with respect to n's nodeID), m nodes in the other quarter, k in the other eighth and so on (see Figure 2.2). Although it is theoretically enough to keep $m = 1$, PariDHT adopts $m = 20$ in order to improve the robustness of the network. The structure storing these contacts is usually called *routing table*. Given this structure, both store and search primitives are achieved within $O(\log N)$ hops. Figure 2.3 exemplifies the search procedure: node u wants to get the key k, so it contacts u_1, its known contact laying in the same half of the network as k; u_1 returns the address of u_2, that is the closest to the key among its contact, and so on, in a iteratively fashion, until the node owning the required resource (the node whose ID is the closest to $v(k)$) is reached.

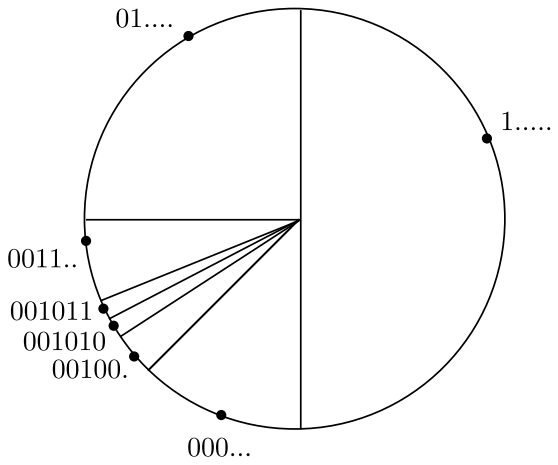

Figure 2.2: A feasible routing table configuration
for the node with *nodeID*= 001010, in a DHT with $d = 6$ and $m = 1$.

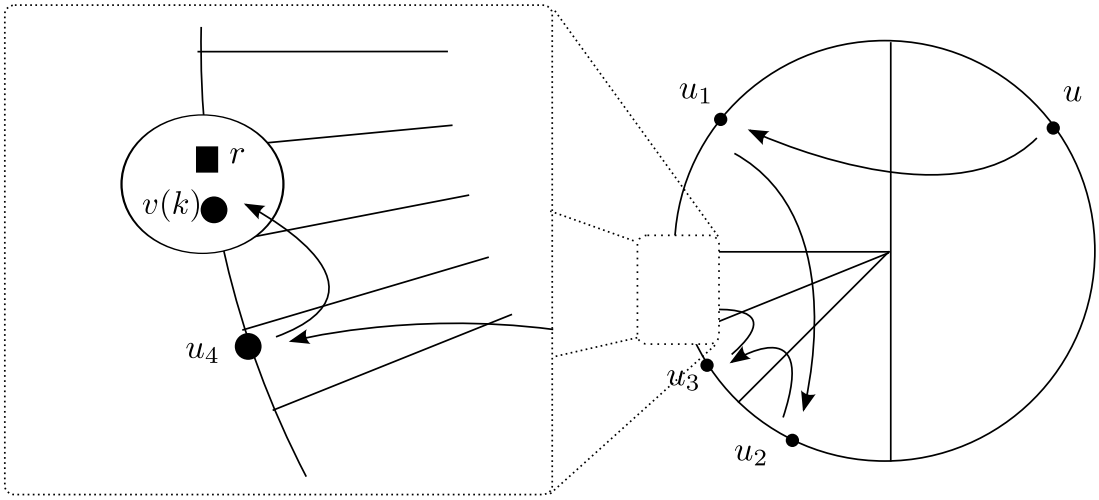

Figure 2.3: An example of lookup.

2.3 PariConnectivity

As mentioned above, *PariConnectivity* is the *inner circle* plug-in designed to provide *network services* to the others. It implements a set of APIs for carrying out efficient communications directly over TCP [34] and UDP [33], or over more advanced and customized protocols; with the exception of PariCore, all plug-ins can perform network I/O *only through* PariConnectivity: it is the *only* plug-in allowed to access network I/O structures by the PariPariSecurityManager.

In particular, PariConnectivity APIs provide other plug-ins with a set of *objects*, through which all the communications are performed: these objects can be thought as abstractions of different kinds of *sockets* (e.g., TCP sockets, UDP sockets, SSLTCP sockets and so on). For the sake of clarity, in the following we will refer to them by a single, comprehensive term: *Network Access Points* (NAPs).

The main goal PariConnectivity addresses is designing these APIs in such a way that the development of network components is made simple and efficient for the other plug-ins: in other words, PariConnectivity aims at supplying an *efficient* and *easy-to-use* network service.

As we will see, a first step in this direction has been the introduction of the *Java NIO* APIs (see Chapter 3).

In addition, our plug-in has been enhanced by a set of more advanced features. Although they will be widely analyzed in the following Chapters, we think it is

convenient to offer a succinct, yet exhaustive overview here.

Asynchronous I/O

Plug-ins generating a heavy network traffic highly benefit from the PariConnectivity's system of asynchronous calls. It is functional and, furthermore, extremely easy to use. This system exhibits a degree of expressiveness comparable to that owned by the Java NIO Asynchronous I/O, while avoiding the cumbersome approach adopted by the latter. Expensive structures like those implemented in `Selector` or `SelectionKey` classes or features like registration mechanisms and registration processes (see Chapter 5) are appropriately wrapped: for a plug-in aiming at performing network I/O it is enough to submit a set of operations to PariConnectivity, which will take care of them and notify the plug-in once they have been fulfilled.

Bandwidth limitation

The amount of bandwidth available to each peer is a limited and precious resource, so it should be adequately handled and distributed among the different running plug-ins. In order to achieve this goal, PariPari imposes a strict rule: that is, each plug-in must exactly quantify the bandwidth it will ask for, providing an adequate counterweight in terms of credits. Thanks to this well defined charging system, exceeding amounts of bandwidth can be reutilized in order to improve the provided QoS.

Communication over PariPari Network

Usually, a service can be identified across a network by means of one or more (IP, port) pairs. But sometimes – as it actually happens in PariPari – even a node without public ports (usually, behind NAT) may wish to offer a service, publishing its availability on the DHT network. The APIs provided by PariConnectivity let PariPari peers communicate even in such cases: provided NAPs make use of the (PariDHT ID, virtual port) pair in order to identify a service on a peer, and transparently try to establish a connection.

Multicast

When the same information has to be simultaneously sent to many peers, a multicast service may be useful. A subplug-in of PariConnectivity, *Multicast*, is meant to exactly offer this service (see Chapter 8).

Anonymity

NAPs providing anonymity services are supplied by a separate subplug-in of PariConnectivity as well. The main purpose of the *Anonymity* module (widely described in Chapter 9) is to ensure the *untraceability* of peers performing communications: that is, a message cannot be associated with its sender nor with its receiver and, furthermore, it can carry an *encrypted* payload.

Chapter 3

Java NIO

Many applications rely on the well known `java.net` package [3] in order to perform network I/O. Although could be a good choice in most contexts, since the abstraction layer it provides allows to easily implement a wide range of I/O services, this package suffers from two main drawbacks: it does not scale well when handling thousand of simultaneous connections and does not support some common I/O functionalities provided by most operating systems nowadays.

Starting from J2SE SDK v.1.4, Sun Microsystems provides the *Java NIO* (New IO) APIs in the `java.nio` package [4] and its subpackages; Java NIO straightens out the problems we have mentioned above, providing *high-speed*, *scalable* and *block-oriented* network I/O. This chapter briefly describes the main features of the `java.nio` package, analyzing in particular its most important structures: buffers (Section 3.1), channels (Section 3.2) and selectors (Section 3.3). The interested reader can find an exhaustive explanation in [23].

3.1 Buffers

A buffer represents a *container* for data: buffers are useful especially when large amounts of bytes should be accessed, transferred or processed.

They are as easy to use as arrays, allowing to access stored data in a totally random fashion. Besides, they implement some additional interesting features: buffers support efficient methods for sequentially filling and draining stored elements and for easing bulk data transfers among buffers and I/O channels (see Section 3.2). Indeed, the new NIO's *Buffer* classes are the linkage between regular

classes and the channel abstractions themselves.

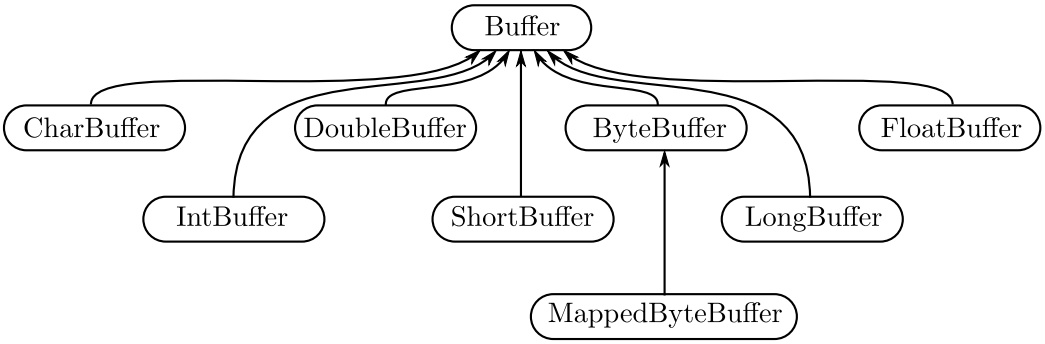

Figure 3.1: Buffer family tree.

As we can see from the *Buffer* class-specialization hierarchy shown in Figure 3.1, there exists one *Buffer* class for each primitive data type, with the exception of the boolean type. Note how, although buffers act upon the data types they store, they have a strong bias toward bytes: many operations dealing with channels can be done only by means of buffers of bytes (that is, by means of instances of the `ByteBuffer` class). This kind of buffer provides an interesting features: the capability of performing *direct I/O*, as exposed in Subsection 3.1.1.

3.1.1 Direct buffers

The most significant way in which byte buffers are distinguished from other buffer types is that they can be the sources and/or targets of I/O performed by *channels*.

In particular, NIO APIs introduce the notion of *direct buffers*. Direct buffers are intended for interaction with channels and native I/O routines; they make a best effort to store byte elements in a memory that a channel can use for direct, or *raw*, access by using native code to tell the operating system to drain or fill the memory area directly.

Direcdt byte buffers are usually the best choice for I/O operations. By design, they support the most efficient I/O mechanism available to the JVM. The memory used by direct buffers is allocated by calling through to native, operating-system specific code, *bypassing* the standard JVM heap.

DatagramChannel	$\Longleftrightarrow$	DatagramSocket
SocketChannel	$\Longleftrightarrow$	Socket
ServerSocketChannel	$\Longleftrightarrow$	ServerSocket

Table 3.1: Correspondence between channels and `java.net` classes.

3.2 Channels

The most important new abstraction provided by NIO is the concept of a channel. A *Channel* object models a communication connection; in general, a channel can be thought of as the pathway between a buffer and an I/O service.

They are not an extension or enhancement, but a new, first-class Java I/O paradigm: channels greatly reduce the abstraction represented by sockets, allowing a strictly interaction and a direct connection with the native I/O services provided by the OS layer. Moreover, they provide a standardized environment, in which one can substantially adopt the same mechanisms when exchanging data over a network, dealing with a file or communicating with another thread.

The Channel family tree is quite articulated; a simplified, but still sufficient for our purposes, version is shown in Figure 3.2. In the following, after depicting some channel basics, we will focus on a specific channel interface (`InterruptibleChannel`) and on some abstract classes (`DatagramChannel`, used when dealing with UDP datagrams, `SocketChannel` and `ServerSocketChannel`, used when dealing with TCP connections, (`SelectableChannel`)). Before starting our explanation, it may be worthwhile to underline how each channel corresponds to an old `java.net`'s socket class (see Table 3.1).

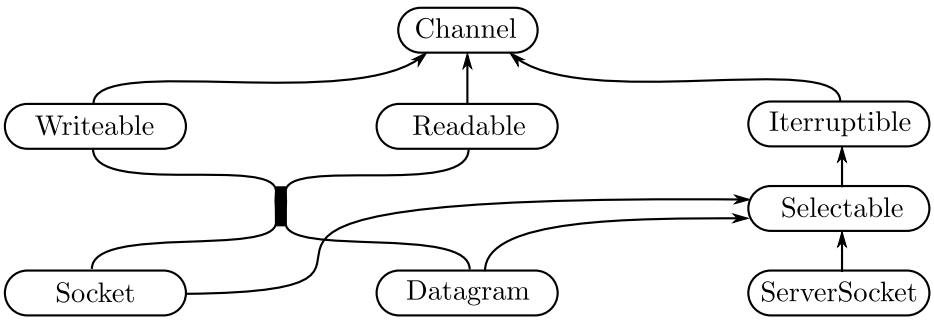

Figure 3.2: Simplified Channel family tree

3.2.1 Channels basics

Channel objects can be directly created using factory methods; the `Channel` superclass provides the basic methods to be invoked for opening and closing a channel or testing whether or not it is open.
In particular, the static methods employed for *opening* network channels are:

- `SocketChannel SocketChannel.open()`
- `ServerSocketChannel ServerSocketChannel.open()`
- `DatagramChannel DatagramChannel.open()`

Closing and checking whether a channel is open can be done, respectively, invoking:

- `void close() throws IOException`
- `boolean isOpen()`

3.2.2 InterruptibleChannel interface

An `InterruptibleChannel` can be interrupted at any time, even if a thread is doing an I/O on that channel.

In particular, if a thread A is blocked on an `InterruptibleChannel` and A is interrupted (by calling the thread's `interrupt()` method), the channel will be closed and a `ClosedByInterruptException` will be thrown. The same happens if the interrupt status of A is set, and A attempts to access the channel. Finally, if a channel is closed when a thread A is blocked on it, this causes a `AsynchronousCloseException` to be thrown.

3.2.3 Main abstract classes

In this subsection, we briefly describe the four most important abstract classes provided by NIO: although, as we will see in the following Chapters, these classes are not directly accessed by external plug-ins, a deeper comprehension of the mechanisms underlying them is essential in order to embrace how PariConnectivity works.

SocketChannel A `SocketChannel` object is a channel connected to a TCP socket: it provides data transfer capabilities over TCP and methods for establishing a connection with a remote server. Both functionalities are provided both in blocking and non-blocking mode.

One can connect to a remote TCP socket simply invoking:

- `boolean connect(SocketAddress remote)throws IOException`

If the `SocketChannel` is blocking, the thread calling that method waits until the connection is established or an I/O error eventually occurs. In non-blocking mode, the above method returns *true* only if the connection can be established immediately; if *false* is returned, the connection can be finalized once the `connectable` state of the channel is set, by means of the method:

- `boolean finishConnect() throws IOException`

Data transfers can be performed only if the `SocketChannel` is already connected. Read and write operations are carried out to and from `ByteBuffer` objects: in both cases, these operations return the number of transferred bytes. The most widely employed methods for reading and writing are:

- `int read(ByteBuffer dst) throws IOException`
- `int write(ByteBuffer src) throws IOException`

In blocking mode, the read method waits until at least one byte is available, while the write method waits until all data remaining in the buffer are sent; in non-blocking mode, the call immediately returns. In this latter case, if either no data are available in the socket input buffer or the socket output buffer is full, read and write methods – respectively – return a value equal to 0.

ServerSocketChannel `ServerSocketChannel` is a channel-based TCP connection listener: it waits for incoming connections on a given port. Since methods for specifying this listen port are not provided, one ought to fetch the related `ServerSocket` instance and use it in order to bind to a port. The method employed for listening for incoming connections is:

- `SocketChannel accept() throws IOException`

The object returned by `accept` is exactly the `SocketChannel` connected with the remote host requesting the connection. If the channel is in blocking mode, the method follows the same behavior as that provided by a `ServerSocket` instance (that is, the caller waits until a connection request comes); in non-blocking mode, the method immediately returns `null` if no incoming connections are currently pending.

DatagramChannel `DatagramChannel` class provides data transfer capabilities over UDP. Since UDP is a connectionless packet-oriented protocol, a single object is enough to manage both incoming and outgoing packets: a further class providing listener functionalities is not requested. As the `ServerSocket` channel described above, the `DatagramChannel` does not allow the user to specify the local listen port: this should be done using the related `DatagramSocket` instance.

Data transfer is packet-oriented, and payloads are managed through `ByteBuffer` objects. Methods belonging to this class and worth mentioning are:

- `SocketAddress receive(ByteBuffer dst)`
- `int send(ByteBuffer src, SocketAddress target)`

The first method receives packets coming from any host and returns information about the source's address. If the packet contains more data than the `ByteBuffer dst` capacity, any excess is simply discarded.

The second method sends data to the remote destination specified by the `target` argument and returns the number of sent bytes. In blocking mode, the caller waits until all data are sent; in non-blocking mode, all data are packed in a single datagram if possible; if not, nothing is sent and 0 is returned.

SelectableChannel All the channels used in PariConnectivity extend the `SelectableChannel` superclass – that is, they can be multiplexed via a `Selector` (see Section 3.3). One of the main features provided by selectable channels is the distinction between *synchronous* and *asynchronous* I/O: indeed, a selectable channel can be either in *blocking* mode or in *non-blocking* mode.

In blocking mode, every I/O operation invoked upon the channel will block the caller until it completes (as it is usually the case with *java.net* objects); in non-blocking mode, an I/O operation will never block, even transferring fewer

bytes than were requested or possibly no bytes at all. One of the advantages of asynchronous I/O is that it allows to do I/O from many inputs and outputs at the same time; one can listen for I/O events on an arbitrary number of channels, without necessity of polling and without extra threads.

Blocking mode can be set using the appropriate method provided by the `SelectableChannel` class:

- `void configureBlocking (boolean block) throws IOException;`

3.3 Selectors

A selector is a multiplexor of `SelectableChannel` objects: it represents a crucial object when dealing with asynchronous I/O.

Selectors provide the ability to do readiness selection, offering a mechanism by which one can determine the status of one or more channels. A selector is where one registers interest in various I/O events and is the object that takes care of notifying when those events occur. Using selectors, a large number of active I/O channels can be monitored and serviced by a single thread easily and efficiently.

A selectable channel can be registered with one or more selectors, with respect to some well defined operations; querying a selector, one can know which channels – registered with it – are ready for a specific operation.

A selector is instantiated invoking the factory method:

- `Selector open()`

and is closed with:

- `void close()`

3.3.1 Registration

As we have briefly mentioned above, selectable channels can be registered with a selector monitoring their readiness: this relationship is encapsulated in a `SelectionKey` object. Generally, a selector manages many channels and, specularly, a single channel can be registered with many selectors: therefore, there exists a $M - N$ relationship among selectors and channels, as shown in Figure 3.3.

Figure 3.3: Relationship among channels, selectors and selectionKeys in an E-R diagram.

Operation	Int Value	Supported Channel		
		SocketChannel	ServerSocketChannel	DatagramChannel
ACCEPT	16		X	
CONNECT	8	X		
WRITE	4	X		X
READ	1	X		X

Table 3.2: The registration process: selectable operations, their encoding and their supported channels.

One can register a channel with a selector invoking the `register` method, whose signature is:

- `SelectionKey register(Selector sel, int ops)`

This method is declared in the `SelectableChannel` class: when it is called on a specific channel object, that channel is registered with the selector `sel` for the operations specified by the integer `ops`. This number is calculated as a bitwise exclusive OR among the primitive operations the selector must test the readiness of the channel on. In general, selectable operations depend on the specific channel employed: Table 3.2 shows the operations allowed in correspondence with different types of channel.

A `SelectableChannel` registration with a selector is represented by a `SelectionKey` object (returned by the `register` method itself): when a selector must notify of an incoming event, it does this by supplying the `SelectionKey` object that corresponds to that event. These `SelectionKey` objects are also used to deregister the channel.

3.3.2 Selection process

As previously discussed, a selector maintains a set of registered channels, and each of these registrations is encapsulated in a `SelectionKey` object. The core

of the selector class is the *selection process*.

The selection process monitors the state of all the channels currently registered with the selector itself. It updates the readiness of such channels with respect to the operations stored in the `SelectionKey` object whenever the following method, provided by the `Selector`, class is called:

- `int select()`

It returns the number of ready channels that were not in the previous selection and updates the selector's selected keys set with the selection keys of the ready channel. A caller of this method blocks until at least one channel registered on that selector is ready.

There exist two further methods letting us perform the selection process in a slightly different manner:

- `int select(long timeout)`
- `int selectNow()`

The former lets us set a timeout, while the latter represents a non-blocking call.

Chapter 4

PariConnectivity: the Core

The main module of PariConnectivity is *Connectivity Core*. It receives all the network I/O requests coming from the other plug-ins and takes care of satisfying them. Not only external plug-ins, but also other modules of PariConnectivity as well (for instance, Anonymity – see Chapter 9 – and Multicast – see Chapter 8) must pass through Connectivity Core when performing network oeprations. This represents a extremely modular approach, that will let us eventually modify and make more efficient the internal mechanisms of the Core without affecting the generic interfaces it exhibits to the outer world.

Connectivity Core provides its users with a particular *abstraction layer*: each plug-in aiming at transferring data through a network interface will use an *object* and will call a set of methods on the object itself. In particular, every object corresponds to a specific kind of connectivity: this concept will be faced in details in Section 4.1. Every method tells Connectivity which specific operation should be performed on that object, as explained in Section 4.2.

4.1 A new abstraction layer

In order to perform any I/O operation, a plug-in must request the *object* providing it by means of the structures supplied by the PariCore. Note how this strict rule is necessary mainly for two reasons, that is, for exploiting the features offered by the T.A.L.P.A. architecture (e.g., the high degree of modularity and the automatically performed dependency resolution) and for introducing an adequate charging system through the Credits module.

Different kinds of connectivity, each characterized by different settings, are available, depending on the functionalities requested by plug-ins. We can introduce a quite clear classification:

mode: plug-ins can handle transfer of data according to two distincts *modes*: *synchronous* and *asynchronous*. In the former case, the calls made by plug-in's threads in order to send or receive data are blocking; in the latter, these calls are non-blocking: a system of queues is introduced letting threads submit data and be notified whenever necessary.

network: communications can take place between two PariPari's peers or between any pair of hosts. When both the nodes involved in the communication belong to the PariPari network, tunneling and NAT Traversal services are provided.

protocol: the transmission model of either TCP or UDP can be chosen, accordingly to the actual requirements of the requesting plug-in.

When requesting such objects, the maximum incoming and outgoing bandwidth must be specified: indeed, this parameter is essential for the Credits module to calculate the corresponding charging.

In addition to these objects, plug-ins should configure a set of communication-related parameters through an API provided by PariConnectivity: the `PPConnQueryAPI`. This API is necessary since we have to charge global aspects, not only the ones related to a single communication; each plug-in must ask for a `PPConnQueryAPI` object, specifying the maximum and minimum amount of bandwidth the plug-in will use.

4.2 Enhancing I/O performance

The APIs providing connectivity functionalities essentially implement methods for reading and writing data; besides, when the TCP protocol is used, methods for opening and closing connections are provided too.

When a plug-in invokes a method on an API's instance, Connectivity translates this request into a sequence of basic I/O operations and starts performing them. Let us consider the first operation: the channel involved in it is registered with a selector, and a thread periodically executes a selection process on

that selector. Once the readiness of the channel is notified, the channel for the second operation (if different from the first) is registered. All the channels concerning operations waiting for completion are registered with the *same* selector, and monitored by the *same* thread T_c. In such a way, the number of threads we have to instantiate is dramatically reduced, without affecting performance: every elementary I/O operation simply corresponds to a non-blocking call made on a single channel, that is, a simple movement of a small amount of data from an area of memory to another, without any further delay due to I/O-related mechanisms. The response time achieved by a basic I/O operation performed in this way is directly comparable with that provided by a multithreaded approach.

Furthermore, these basic operations do not suffer from any starvation problem: every time the selection process is executed, all the channels being ready at that instant are processed, before carrying out the subsequent call to the `select` method.

The way this "breaking down" into basic operations is accomplished depends on the type of object and on the specific operation required on it. For simplicity, here we will analyze the simplest case: the synchronous communication toward a node not belonging to PariPari. Examining this mechanism will help the reader appreciate the enhancement introduced by the asynchronous I/O, as exposed in Chapter 5.

4.3 Synchronous communications

Let us consider how a synchronous communication toward nodes not belonging to PariPari is achieved. In this case, each object X is put into a *one-to-one* correspondence with a channel C_X.

We can describe how data are *sent* in a very simple manner: when the plug-in P, owner of X, instantiates a thread T_P and, through it, wants to send an amount of data, T_P is blocked and C_X is registered with the selector in order to carry out the requested write operations. Once the Connectivity's thread T_c has processed all the data, C_X is deregistered and T_P unblocked.

Data are received in a similar manner. In this case, C_X is registered soon after its instantiation and put into correspondence with a *private* `Direct ByteBuffer` object, owned by X. Whenever P wants to read data from X, it carries out

the corresponding call by means of a thread T_P: if the buffer is empty, T_P is blocked, waiting for eventually incoming data; otherwise, all the bufferized bytes are returned. Note how, for the buffer Br_X, there exists a well defined size limit, set by the plug-in during the instantiation phase. When Br_X fills up, C_X is deregistered from the selector: from now on, the OS takes charge of handling all the incoming packets. This implies that such packets could be bufferized or dropped, depending on the specific OS settings; if we are dealing with TCP connections, the OS will take care also of managing the *Advertise Window*. For an exemplification of the whole mechanism, see Figure 4.1.

If the communication takes place over TCP, all the connection-related aspects should be taken into account. When a plug-in aims at opening a connection, this call is translated into a blocking one, and the thread undertaking the operation has to wait until either the connection is established or a timeout expires (if a specific timeout is specified). Note that Connectivity performs this operation in an asynchronous manner: for this purpose, the same thread handling the I/O on channel is adopted.

The *closing* phase is managed by a thread provided by Connectivity too. However, in this case the NIO APIs do not offer any asynchronous method, so a `PariPariThread` object, appositely instantiated, carries out the operation and notifies the calling plug-in about its outcome. The overhead of running a different thread at every different connection closure is negligible, since the `PariPariThread` is mapped by the Core into a *thread pool*. The time needed in order to perform a closure operation is very short: as a consequence, the threads running at the same time are generally not so much; anyway, even if high congestion conditions should occur, the policies adopted by the Core, such as ad-hoc queueing mechanisms, prevent the number of such threads becoming excessively high.

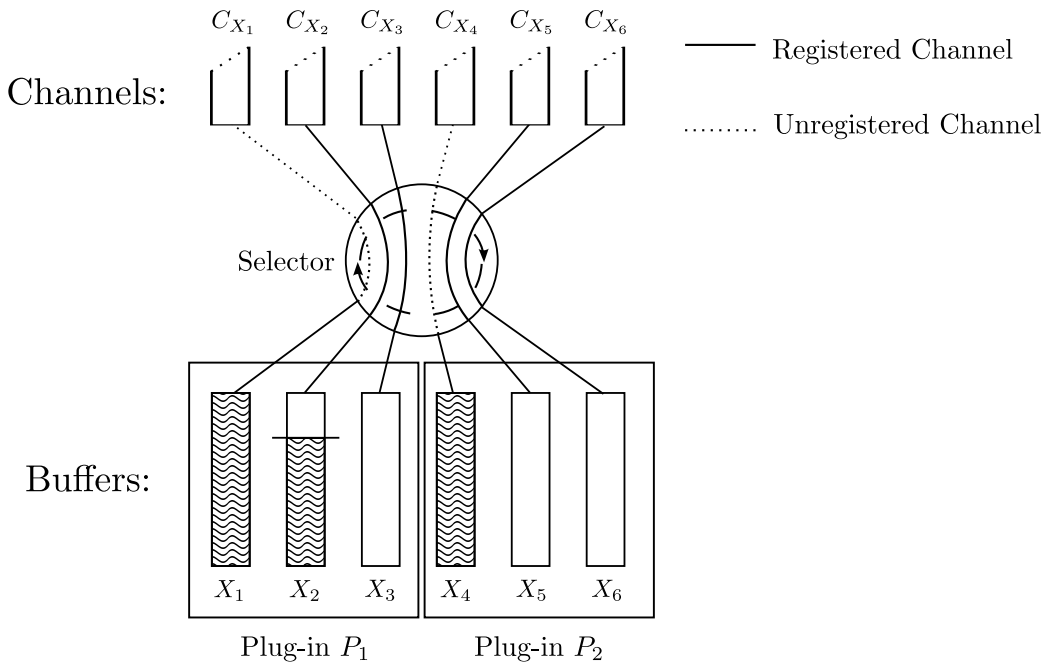

Figure 4.1: How plug-ins receive incoming data.

Chapter 5

Asynchronous I/O

More and more often, modern applications aim at enhancing their performance by using *asynchronous I/O*: in this way, I/O and computation can be overlapped. Anyway, asynchronous I/O suffers from a main drawback, namely the overhead it may introduce. Indeed, whereas employing many threads running at the same time can represent a good choice in some cases, in others it turns out to be a limiting one: when the number of threads becomes too high, different and more efficient choices should be made.

In this Chapter, we first provide an overview about features and advantages of asynchronous I/O in Section 5.1; then, we illustrate its implementation in PariConnectivity in details, exemplifying the interaction with other plug-ins in Section 5.2.

5.1 Some preliminary notions

Asynchronous (or non blocking) I/O is a kind of input/output processing, that enhances the overlapping between I/O and computation.

In general, when I/O requests are expected to take a large amount of time (as it is the case when dealing with network communications), asynchronous I/O is thought of to be a good choice in order to increase processing efficiency [12]. However, a further aspect must be considered: when an application requests many (and fast) I/O operations, asynchronous I/O often introduces a noticeably higher overhead than synchronous I/O, even making the latter more suitable than the former.

Asynchronous I/O mechanisms and implementation details may widely vary, depending on platforms and contexts of use. Generally, the most employed solutions are:

Polling: often, this is not considered as asynchronous I/O and is called *synchronous non-blocking* I/O. This means that, instead of waiting for I/O completion (like in traditional synchronous I/O), the operation is done only if it can be immediately satisfied; otherwise, an error code is returned. This solution is extremely easy to be implemented using the *selectable* channels, but implies that an I/O request may not be immediately satisfied. Therefore, an application may be forced to do numerous calls while waiting for completion, facing the difficulty to estimate the proper interval of time between two subsequent calls.

Using threads: in a multithreaded environment, each I/O flow can be handled by a separate thread. This was the only viable solution in Java before the introduction of NIO APIs, using a traditional blocking synchronous I/O for each thread. This approach may preclude large-scale implementations running a very large number of threads, and is particularly disadvantageous in PariPari, where each plug-in can only use the well bounded number of threads provided by the Core.

Select loops: this technique performs a non-blocking I/O with *blocking notifications*. According to this model, non-blocking I/O is set, and blocking `select` calls are used to simultaneously determine the readiness of many I/O operations. This call is usually made in a loop, alternating select calls and I/O executions (returned whenever ready). This is the solution chosen by NIO architects and is used for writing the PariConnectivity Core: in this case, the select loops just described exactly correspond to the selection process we have mentioned in Chapter 3.

Completion queues: I/O requests are issued in an asynchronous manner, but notifications of completion are provided via a synchronizing queue mechanism according to the order in which they have been completed. This solution is the one that best suits the needs of PariPari plug-ins and – indeed – is the one implemented by PariConnectivity.

5.2 Asynchronous I/O in PariConnectivity

While PariConnectivity's synchronous APIs maintain high fidelity and full compatibility with Java APIs, the asynchronous APIs provided by our plug-in adopt a quite different approach. In particular, we have chosen to enhance the easiness of use, partially wasting flexibility aspects. For instance, the selection process – which can cause some kinds of troubles when channels and selectors interact, especially when used in conjunction with multithreading – has been substituted by a *queuing mechanism*. All the transferred data are put into and picked up from such queues.

The PariConnectivity's asynchronous APIs are available over both TCP and UDP; besides, they can be useful for carrying out communications over the Pari-Pari network thanks to the Tunneling and NAT Traversal services. However, UDP and NAT Traversal services are packet-oriented, so a single object, handled by a single thread, is enough in order to carry out more than one communication: the actual advantages granted by the asynchronous APIs reveal themselves mostly in TCP connections, where each requires the instantiation of a different thread.

A generic plug-in P has to instantiate a queue Q implementing the `BlockingQueue` interface; such an interface is used by Connectivity in order to notify the plug-in itself about the accomplishment of the requested operations. Each call to asynchronous API's methods (like `send()`, `receive()`, `connect()` etc.) forces the calling plug-in to specify a notification queue Q and an object O (implementing the `AsynchronousNotification` interface), through which it will be notified. O often provides a `process()` method, whose code contains all the operations that need performing after the notification, as well as further I/O requests.

This mechanism is shown in Figure 5.1.

5.3 Interaction with other plug-ins

In this subsection, we try to exemplify the structure owned by a plug-in P_1 sending and receiving files over TCP, in asynchronous mode, toward different peers.

Each file F_i is segmented into n_i chunks (say $F_i^1..F_i^{n_i}$) of constant size. A `Listener` thread monitors the state of the notification queue Q. Two classes (named R, W), implementing the `AsynchronousNotification` interface, serve

as notifier for receiving and sending events, respectively. R holds a reference to the file F_i, to the chunk being currently downloaded F_i^j and to the number of bytes already received for F_i^j. In this case, we can imagine that the `process()` method will perform the following operations: adding the received bytes to the currently processed chunk, writing the chunk on disk once it has been wholly received, determining the next chunk to be processed and asking for the subsequent I/O operation. W holds a reference to the file F_i, storing the index j of the chunk being currently uploaded; here, a call to `process()` determines the next chunk j to be sent and requests the transfer of F_i^j, by using the instance itself as notification object.

Therefore, the code in P_1 managing the beginning of data transfers provides an adequate instance of R (or W) and performs the first asynchronous I/O call in order to start the transfer process. The corresponding `Listener` threads waits on queue Q, calling the `process()` on every instance inserted into such a queue by Connectivity, until the file is totally transferred.

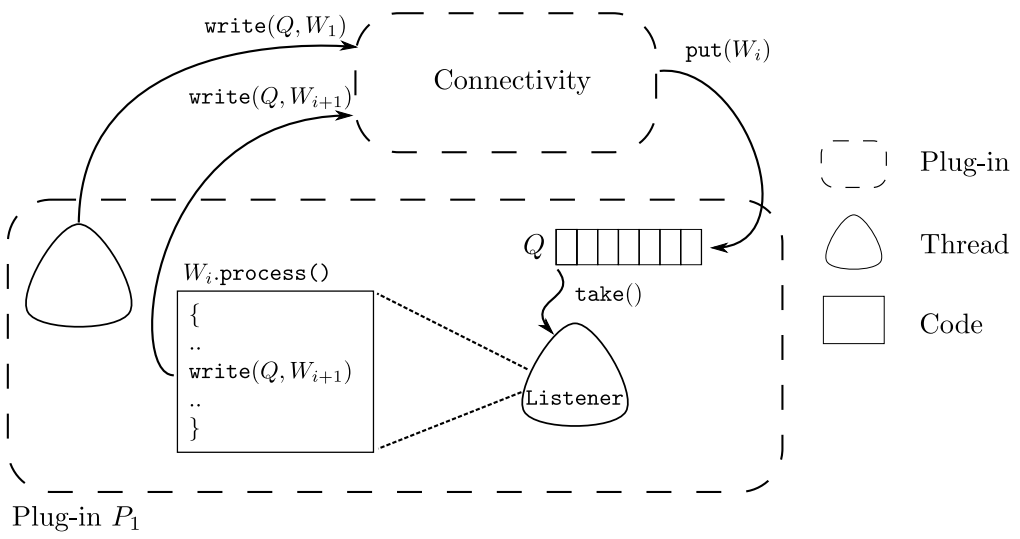

Figure 5.1: PariConnectivity's asynchronous I/O.

Chapter 6

Bandwidth limitation

The amount of bandwidth available to each peer is a limited and precious resource, so it should be adequately handled and distributed among the different running plug-ins. In order to achieve this goal, PariPari imposes a strict rule: that is, each plug-in must exactly quantify the bandwidth it will ask for, providing an adequate counterweight in terms of *credits*. Thanks to this well defined charging system, exceeding amounts of bandwidth can be reutilized in order to improve the provided QoS.

The declared maximum amount of bandwidth remains a constant upper bound that should not be violated; for such a purpose, PariPari adopts a bandwidth limitation strategy, represented by the Token Bucket algorithm [21]. An overview of its most known version is presented in Section 6.1; Section 6.2 illustrates how PariConnectivity manages all the aspects concerning bandwidth management; finally, Section 6.3 depicts some guidelines for the development of a new QoS.

6.1 The Token Bucket algorithm

The Token Bucket algorithm is used to control the amount of data transferred across a network, allowing for bursts of data to be sent. The control mechanism it relays on establishes when traffic can be transmitted on the basis of the presence of *tokens* in a *bucket*. Roughly speaking, a token represents a certain amount of data (generally a byte); a bucket is a container for a limited amount of tokens. When a packet is compared with a token bucket, we have two possible outcomes: if the amount of tokens (say, bytes) in the bucket is equal or greater than the size

of the packet, the packet is said to have *conformed* to the token bucket definition; on the contrary, if there are less tokens in the bucket than bytes in the packet, we say that the packet itself has *exceeded* the token bucket definition [20].

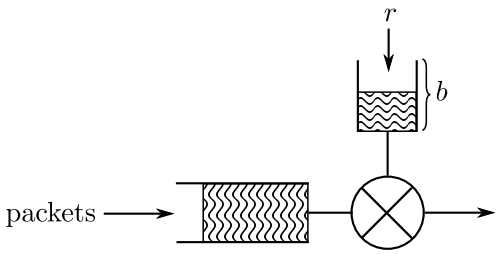

Figure 6.1: A token bucket.

The actual algorithm can be summarized as follows:

- at every interval of time t, r tokens are added to the bucket;
- a bucket can hold at most b tokens; if a bucket is full, no tokens can be added;
- when a packet, with a size of n bytes, needs transferring, and the packet has *conformed* to the bucket, n tokens are removed from the bucket itself;
- if the packet has *exceeded* the bucket, no tokens are removed.

As needed, different actions can be taken for the packets having conformed or exceeded the bucket. In most implementations, conformed packets are immediately transferred, while exceeding ones are dropped. Adopting these policies, the Token bucket algorithm grants a transferring rate equal to r/t. Furthermore, as mentioned above, bursts of data are allowed. If we indicate the burst's transmission rate by M, M can be sustained for maximum amount of time T_{max} calculated as:

$$T_{max} = \frac{b}{M} + \frac{r \cdot T_{max}}{M} \Rightarrow T_{max}(1 - \frac{r}{M}) = \frac{b}{M}$$

$$T_{max} = \frac{b}{M - r}$$

6.2 Implementing the bandwidth limitation

Accordingly to the current implementation, each plug-in can *buy* an amount of bandwidth, paying it in credits. As for the other resources, the cost of each bandwidth unit is determined considering the balance between supply and demand; in order to evaluate the former, the available bandwidth is periodically estimated or, alternatively, specified by users by means of a configuration file in Connectivity. The actual usage cannot exceed the declared (and bought) amount: the bandwidth limitation is accomplished by means of the Token Bucket algorithm.

More in detail, each plug-in is provided with a pair of token buckets (the one concerning the outgoing communications, the other concerning the incoming ones): both are associated with a data rate r^I/t and r^O/t, respectively, on the basis of the demand for bandwidth made by the plug-in. The length of the interval of time t between two subsequent insertions of tokens into the bucket can be specified; if no such value is provided, a default value equal to 0.1 s is assumed. Smaller values for t are, generally, useful for plug-ins requesting traffic trends to hold steady and suffering from high jitter fluctuations (for instance, the VoIP plug-in). Note how keeping low the value of t entails harder computation efforts for Connectivity; as a consequence, on equal terms of requested bandwidth, smaller values of t translate into higher prices.

In addition to the plug-in's bandwidth, input and output bandwidth (r_i^I and r_i^O) for each different NAP O_i can be specified. This demand is not subjected to charging (since the plug-in pays its demand as a whole); however, two token buckets B_i^I and B_i^O are instantiated anyway, in order to manage r_i^I and r_i^O. In a sense, this NAP-specfic bandwidth restriction represents a service available to the single plug-in, helping it in correctly managing the amount of bandwidth it bought.

Transmitting n bytes over a NAP is allowed only if there are at least n tokens in the plug-in's token bucket B^O and in the object's token bucket B_i^O as well; if so, they are taken away from *both* tokens. If this condition is not satisfied, the request eventually translates into a blocking call, that can be satisfied only when the requested amount of tokens is reached.

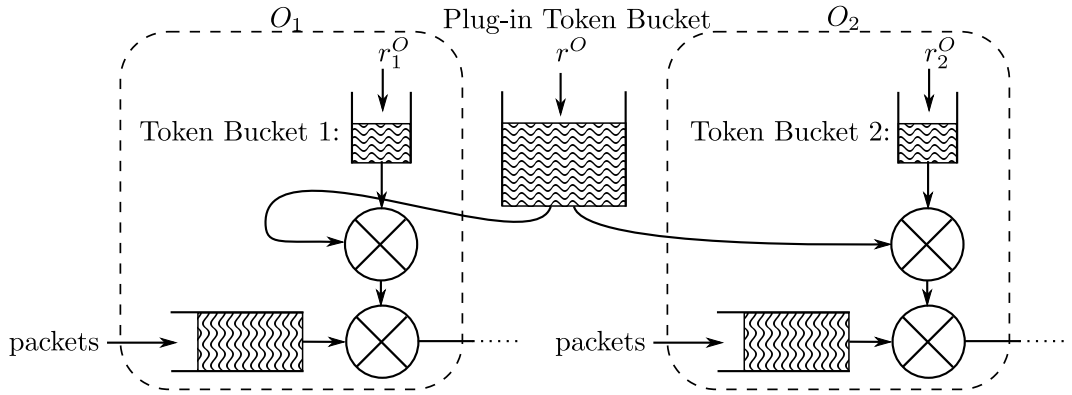

Figure 6.2: Implementation of the Token bucket algorithm in PariConnectivity.

6.3 Improving the QoS

Setting a well established upper bound on the maximum available bandwidth for each plug-in, as described in the previous Section, represents only a first step. More should be done in order to provide a QoS handling all the network resources in a truly efficient and rational manner.

Different applications can exhibit different needs with respect to the requested network parameters. Let us consider a few common examples:

1. VoIP;
2. video streaming;
3. throughput-focused TCP applications;
4. on-line games.

1. and 2. need a minimum bandwidth to be guaranteed, as well as the possibility of occasional bursts to take place in case of codec VBR [1]; 1. and 4. strongly benefit from reductions in latency, while "greedy" applications like 2. and 3. may need limitations in the bandwidth they subtract to others.

Such improvements are currently in a planning stage. For the sake of completeness, we can briefly synthesize the main guidelines that should be followed:

- an overall limitation of bandwidth – for the PariPari software as a whole – is guaranteed;

[1] Variable Bit Rate.

- performance are periodically estimated not only through the usual traffic analysis, but also by means of an active monitoring of the network;
- plug-ins can buy a minimum granted amount bandwidth $\frac{b}{t}$ (that is, the plug-in is given the assurance of transferring b bytes in time t). Therefore, demanding for a bandwidth $\frac{b}{t}$ or $\frac{\alpha \cdot b}{\alpha \cdot t}$ makes difference. These demands can take place at the plug-in level and at the NAP level as well, but, when summed up, must not exceed the actual availability or the restrictions possibly set at a higher level;
- a fraction of bandwidth exceeding the granted one or not currently used can be bought. Summing up all the fractions sold to the different plug-ins must return a result equal to 1: in this way, once the plug-ins asking for a minimum amount of bandwidth have received their portion, the remaining quantity of bandwidth can be used by plug-ins proportionally to the further bought fraction.

Chapter 7

NAT Traversal

NAT Traversal is the module of PariConnectivity providing the NAPs used for carrying out communications across the PariPari network. It allows peers – even behind NAT – to be easily reached through UDP using common NAT Traversal techniques. Besides, thanks to the introduction of *tunneling*, the number of used ports can be noticeably reduced.

A detailed contextualization of NAT (Network Address Translation), among with the problems it involves, is provided in Section 7.1; services and techniques commonly adopted when performing NAT Traversal are discussed in Section 7.2; the PariConnectivity's NAT Traversal module and its interaction with external plug-ins is described in Section 7.3; finally, Section 7.4 illustrates our tunneling techniques.

7.1 Communicating behind NAT

Network Address Translation *(NAT)* [38] is a process performed by a routing device that remaps a given address space into another, modifying the network address information contained by packet headers. Usually NAT is used in order to perform *IP masquerading*: this is a technique that hides a whole IP address space behind a single IP address (usually a public one). In this context, a (IP, port) pair belonging to a hidden address space is associated with a public port. This procedure is more properly named *PAT* (*Port Address Translation*).

In order to achieve this, a routing device maintains a set of stateful translation tables, mapping hidden (address, port) pair to a public port and then rewriting

packet headers so they appear originated by the router. Along the reverse path, responses are mapped back using the states stored in the translation tables themselves. As described, this method enables communication only if it is originated from the hidden address space. If a peer behind a NAT wants to be reachable, a static entry can be stored into the translation tables. This operation is named *port forwarding* and is made by properly configuring NAT devices. Note how this often can not be performed by users. In fact, in business environments it is generally denied to users to deal with NAT configurations.

7.2 NAT Traversal: services and techniques

Usually, NAT Traversal refers to a set of techniques that let users circumvent many of the NAT problems described above. In particular, NAT Traversal provides methods to establish connections traversing NAT gateways, where the conversation can be originated outside the masquerading network. In this Section, we describe how NAT traversal mechanisms are commonly implemented and illustrate the services that should be provided to a masqueraded host.

7.2.1 IP Discovery

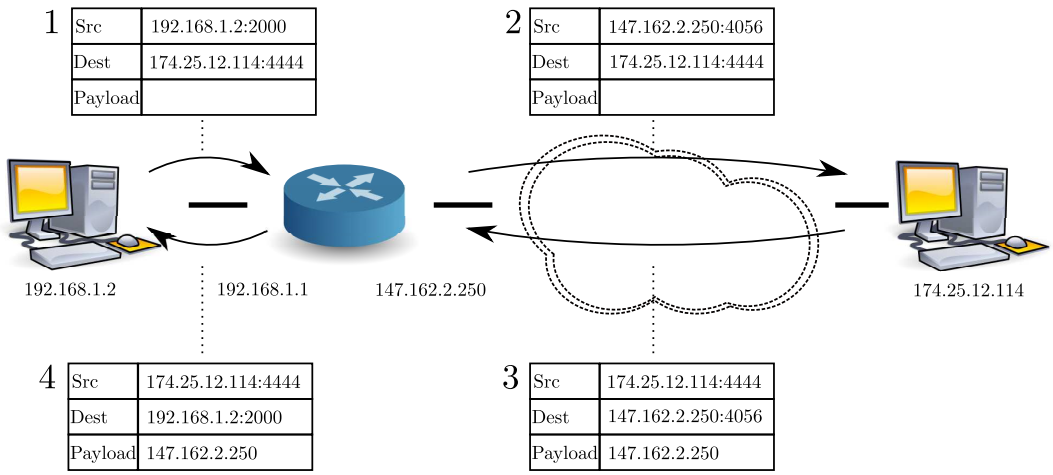

Figure 7.1: IP discovery.

Each node needing to be contacted from other ones should know its public IP address, which, of course, is different from the local IP address if a NAT gateway is present. This service can be simply provided by another peer, accepting an

UDP packet on a public (IP, port) pair and replying to the sender with a packet containing its IP as payload (see Figure 7.1). The knowledge of its own IP lets a peer establish whether it is behind a NAT or not. If a NAT gateway is present, but *port forwarding* is enabled on any port, this information is enough to make the peer contactable from others.

7.2.2 The STUN protocol

STUN (**S**ession **T**raversal **U**tilities for **N**AT) is a proposed standard protocol [36] that can be used by a peer to determine the public IP address and port number that NAT has allocated to it. STUN also allows to maintain NAT bindings, thanks to its keep-alive functions. In PariPari, STUN is adopted only on UDP, although further versions are available also on TCP and TLS-over-TCP. The STUN protocol requires the presence of a server, that is usually located in Internet.

A peer can learn its NAT status by sending a datagram to a STUN server. The server answers with a datagram to the sender from the port where it has received the request and tells another server to send the same packet. According to the received packets, the peer can understand its reachability state:

- if the peer receives **no responses**, probably a firewall blocks all the connections crossing the used port and the client should not use this port at all. Additional attempts can be possibly done using different STUN servers and ports.

- if the peer receives **both datagrams**, either a port forwarding can be enabled or the NAT device has created a temporary new entry in the traslation tables that can be used by all the remote hosts. If necessary, one can periodically send a packet to the STUN server in order to maintain the entry up to date.

- if the peer receives **only the first datagram**, the client can not be directly contacted from other peers on the port used for the STUN server without introducing some tricks. It should be determined if, when data addressing another destination are sent from the same local port, NAT maps such packets into a different public port or not. In the first case, in order to establish a communication with another peer behind a NAT device, a TURN

server must be used (see); in the second case, the UDP hole punching method can help (see 7.2.3).

7.2.3 UDP Hole Punching

UDP Hole Punching is a technique adopted for establishing connectivity over UDP between two hosts communicating across one or more NATs. It is useful when a NAT device maps different connections from the same internal (IP, port) pair into the same external port. As required by NAT, a peer must send a preventive UDP packet to another peer, in order to receive incoming packets from it. This case can be notified to it by a public server, with which a connection is already established. If the other peer is not behind a NAT or virtual server policies is enabled, this is made by a simply callback operation. On the other hand, if both are behind a NAT, this operation is possible only if at least a NAT device maps the connections as stated before; otherwise, both peers can not know on which public port they should contact each other.

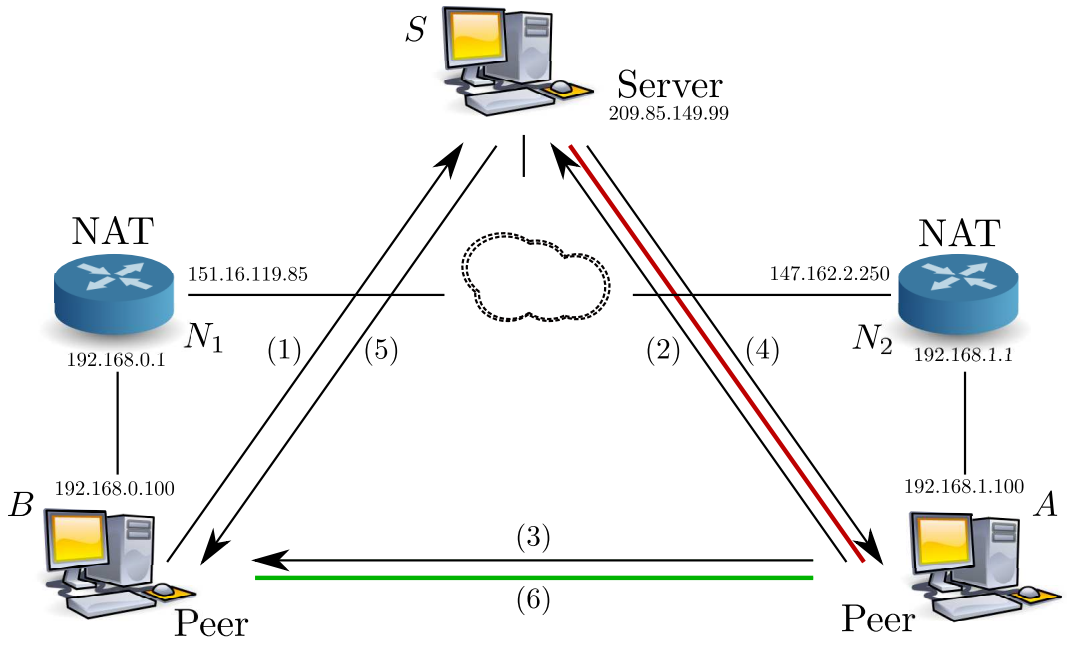

Figure 7.2: UDP Hole Punching mechanism.

More schematically, let A and B be two peers behind NATs N_1 and N_2 respectively; assume they know their public port assigned by NAT, using the STUN protocol; A is connected with a server S. If B wants to contact A, the following should be performed (see also Figure 7.2):

1. B tells S their public port, signaling that it wants to contact A;
2. S informs A about this;
3. A sends a packet to B's public port, creating an UDP translation state on N_1 with B's public address;
4. A confirms to S that the packet directed to B has been sent;
5. S tells to B that now A is reachable;
6. B begins the communication creating the correct translation also on N_2.

7.2.4 The TURN protocol

TURN (**T**raversal **U**sing **R**elay **N**AT [27]) is a protocol that allows a peer behind a NAT or firewall to receive incoming data over TCP or UDP connections. It is used when a direct communication path can not be found, so an intermediate host is used as a relay for the packets. This relay is named a *TURN server*, typically hosted in Internet (see Figure 7.3).

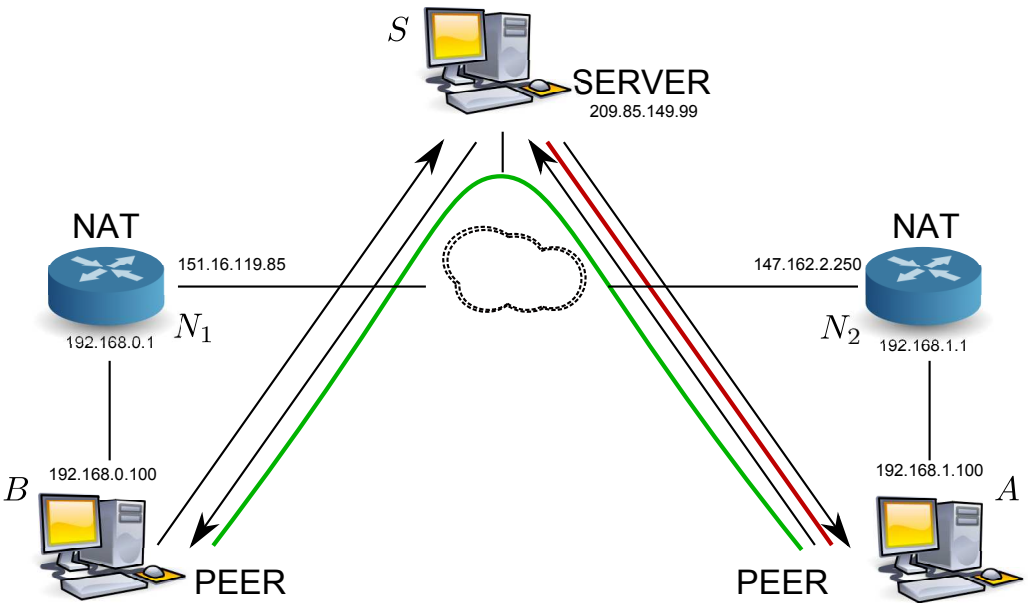

Figure 7.3: A TURN Server servicing two clients.

This NAT Traversal method exhibits poor performance, since all the traffic between two peers has to pass through the TURN server: therefore, it is used

only when the other methods fail. This is particularly true in PariPari, where TURN servers are not dedicated servers: only a very small number of clients can deal with the same TURN server.

7.3 Interaction with other plug-ins

PariConnectivity implements NAT Traversal techniques, letting peers communicate even behind NAT. Since not every node can be identified through a (IP, port) pair, PariConnectivity adopts the nodeID provided by PariDHT as node identifier: a strict communication between PariConnectivity and PariDHT modules is therefore necessary.

The parameters required by the NAT Traversal NAP are: a `PPId` object, containing PariDHT `id` and other information discussed below, a *Virtual Port* and a *Tunnel Type*, used to tunnel multiple connections (see Section 7.4).

As already said, services in PariPari are not provided by dedicated servers. Each peer joining the network is encouraged to provide some services – in order to earn credits. Therefore, a peer looking for a service can not rely on a well-known server: it should find another peer providing that service at that time. Information about this can be efficiently retrieved from the PariDHT network. Broadly speaking, if a peer A wants to provide a service s *stores* on the DHT the pair K_s, P_A, where K_s is a *key* identifying the service s and P_A is a pointer to the peer A. Another peer B needing that service calls the `search(`K_s`)` procedure, obtaining P_A and the pointers to other peers providing the same service. Pointers are directly managed by plug-ins performing search, so – in theory – they can be represented by (IP, port) pairs, DHT `id`s or whatever else. However, a plug-in aiming at using NAT Traversal facilities must use an apposite `PPId` object, provided by PariDHT. It contains all the information necessary for PariConnectivity to contact a determined peer, like:

- PariDHT `id`;
- public address (IP, UDP tunneling port, TCP tunneling port);
- STUN server address (IP, port);
- TURN server address (IP, port);

- flags identifying working NAT Traversal techniques.

All these fields (with the exception of the first) should be filled by PariConnectivity, using the information acquired thanks to the STUN protocol. This information is then sent to the PariDHT module, which complete the structure with the PariDHT `id`. The process of *storing* and *retrieving* information about a service, in which two peers A, B are involved, can be summarized as follows (see Figure 7.4):

1. the PariConnectivity module running on peer A creates the `PPId` object, using the information gathered with a STUN server, and sends it to PariDHT, which completes it with the `id` field;

2. a plug-in P requests the complete `PPId` object to PariDHT;

3. P can `store` a service identified by a key K_S with the *value* `PPId`;

4. The same plug-in on the peer B can look for the service requesting `retrieve(`K_s`)` to PariDHT

5. PariDHT returns the `PPId` (and eventually other values stored with the same *key*);

6. plug-in P running on peer B can now use `PPId` in order to connect with A.

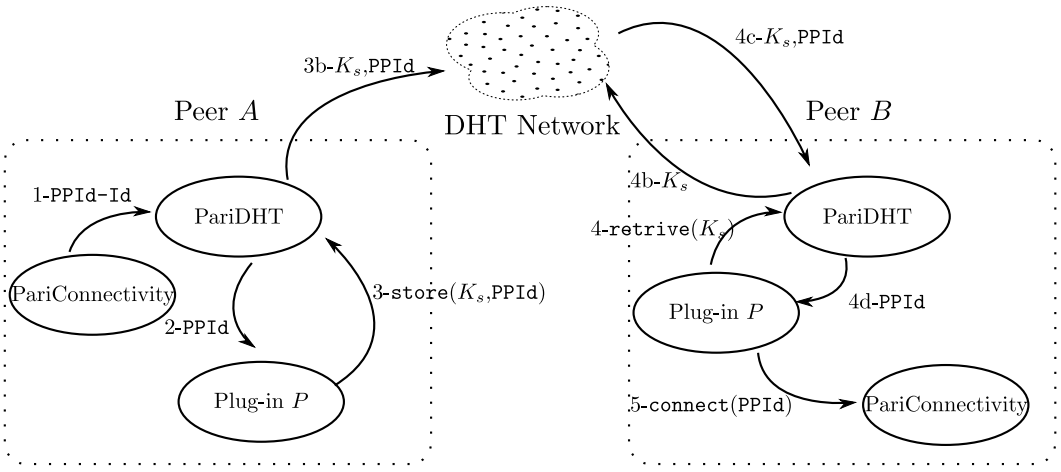

Figure 7.4: Store and retrieve of a *service* over PariDHT using a `PPId`

Start-up process

What we have just described is the process commonly used in PariPari for offering and finding services. PariConnectivity use it too, in order to provide *IP discovery*, *STUN* services and *TURN* services. Peers providing them must be directly reachable thorugh a (IP, port) pair; therefore, it can be used as pointer in the place of a `PPId` object. The main difference between these three NAT Traversal services and other services (that is, services provided by other plug-ins) is that a peer has to look for them at an early stage and its NAT status can be unknown.

NAT status detection is important for a peer which wants to join the PariPari network. In fact, if another peer requires an UDP Hole Punching or a traffic relay for communicating with it, contacting it for exchanging only few data can be inefficient. In this case, it is not recommended to allow the peer to own some PariDHT resources or take part in other DHT operations. On the other hand, these peers can provide services such as the effort of providing NAT Traversal techniques is repayed. Therefore, PariDHT lets nodes behind a NAT with no port forwarding enabled perform store and search, but without assigning them any resource or using them as inner hops. This status is determined by PariConnectivity using NAT Traversal facilities and, then, communicated to the PariDHT plug-in.

More precisely, this procedure consists of the following steps:

- PariConnectivity starts up, providing simple communications over TCP and UDP;
- PariDHT starts up, providing only *store* and *retrieve* features; the peer is supposed to be behind NAT;
- PariConnectivity gets a list of available NAT Traversal servers, using the PariDHT *search* feature;
- PariConnectivity identifies the NAT state of the peer using the STUN server, generates the `PPId` object and communicates them to PariDHT.

7.4 Tunneling

As seen before, NAPs providing NAT Traversal facilities require some parameters about *Virtual Port* and *Tunnel Type* to be set. This is necessary in order to mux multiple communications over a single port. The necessity of this feature derives from the high number of ports required in many contexts: for instance, an IRC servers and a Web Distributed servers require three ports each, while a DBMS Distributed server requires three ports for every hosted database. This service is provided by a PariConnectivity module – simply named *Tunneling* – and, currently, can be used only indirectly by other plug-ins, through the use of NAT Traversal facilities.

A plug-in can require two different types of tunneling:

Best effort tunneling, in which flows of information are mixed without any further assurance, except the correctness of multiplexing and demultiplexing operations. Here, the overhead introduced by the tunneling process is minimum.

Tunneling with guarantees, in which ordered data transfer and retransmission of lost packets are guaranteed. These controls cause a certain overhead and a subsequent waste of bandwidth; so, they should not be used only when strictly necessary.

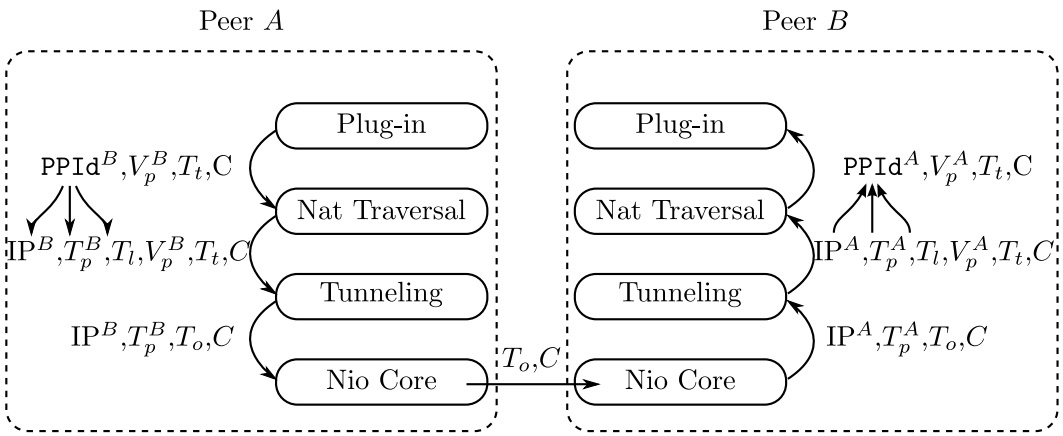

Figure 7.5: The Tunneling mechanism.

Currently, tunneling is provided only over UDP; the TCP version is under development. The choice between TCP or UDP will be made by the NAT Traversal

module itself, depending on the possibilities offered by the network and identified using the STUN server. If both TCP and UDP communication are available, the NAT Traversal module requests the *besteffort Tunneling* over UDP and the *Tunneling with guarantees* over TCP.

In order to send tunneled data, the NAT Traversal module must know IP address, tunneling port, virtual port and tunneling type of the remote service. These last two parameters are specified by the plug-in requesting the communication. IP address and tunneling port can be directly determined using the `PPId` object provided, if the tunneling port of the destination is open; otherwise, they can be determined after using the proper NAT Traversal techniques.

When receiving data, Tunneling returns to NAT Traversal the same four parameters, with regard to the remote peer. From the (IP, port) pair returned, NAT Traversal creates a valid *PPId* object and returns it to the correct plug-in: if the packet comes from an unknown peer, there is no need of any NAT traversal mechanism for the reply, so the *PPId* contains only IP and port values; if the packet comes from a peer which has already established a communication, NAT Traversal must know some further information in order to contact it; this information is included in the *PPId* object.

Chapter 8

Multicast

The *Multicast* module resides in a separate package and makes use of PariConnectity's facilities like any other plug-in with no special privileges. This allows to keep multicast services separated from basic I/O operations, providing the ability of changing the one without affecting the other. Multicast provides developers with a communication service among groups of users with *one-to-many* or *many-to-many* connections. An overview of the Multicast module is given in Section 8.1; two different Multicast plug-in architectures are described in Sections 8.2 and 8.3: the former is currently implemented in PariPari, while the latter is still in development; finally, Section 8.4 illustrates how a peer covering management roles can be chosen.

8.1 Preliminary notions

In computer networks, there exist four types of addresses [11]:

Unicast: the most familiar one. Here, an address identifies a single host;

Broadcast: the broadcast address identifies all the hosts belonging to the network;

Multicast: the multicast address identifies a "host group", that is, a set of zero or more hosts identified by a single IP destination address [17] [15];

Anycast: a packet targeting an anycast address is delivered to at least one, and preferably only one host, in a subset of hosts identified by the anycast address [30]. It is included in version 6 of the Internet Protocol [22]

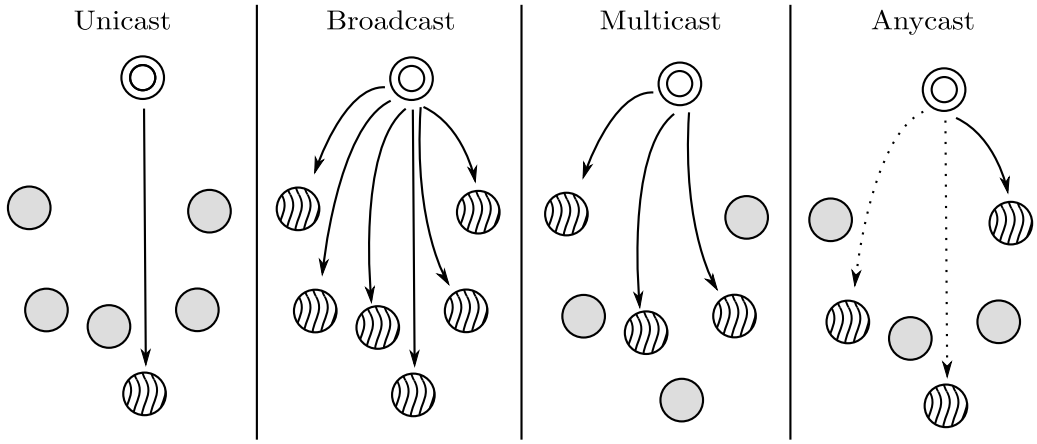

Figure 8.1: Types of addresses in computer networks.

Multicast connections are typically required by applications involving groups of users. Examples include telephone conferences via VoIP, chats between three or more users and multimedia streaming services like IPTV. Communications in these applications can be *many-to-many*, where all members of the group are both sources and destinations of information, or *one-to-many*, where the data stream originates from a single source.

The problem of routing multicast traffic is more complex than routing a packet to a single destination. A tree in the network must be built, in order to efficiently route information. Nodes belonging to the network must replicate the packages to reach the recipients only when necessary, optimizing bandwidth consumption. Another feature of multicast is the creation of a group and the recognition of its members.

Although the multicast is already present as a feature of the IPv4 protocol [17], its use is severely limited outside local area networks, because it has not been widely adopted by most ISPs. Therefore, multicast communications on the PariPari network must be maintained by the application layer implementing multicast forwarding functionalities only at end-point hosts. This led to the development of the multicast module, which provides facilities to other plug-ins for sending one-to-many or many-to-many information. In this way, independence between plug-ins using Multicast and mechanisms employed in the transmission is maintained. It was also considered useful to separate Multicast from basic I/O features: indeed, this led to the decision of implementing it as a separate plug-in, interacting with PariConnectivity through the facilities offered by the Core.

In order to provide such a service, an overlay network connecting the members belonging to the same multicast group is necessary. Two possible architectures were designed to enable many-to-many communications (one-to-many ones can be seen as a sub-case). The first is the currently implemented one and consists of a single node, elected as the *leader* of the group, dealing with all communication-related aspects (see 8.2). The second exhibits a higher scalability, creating a particular hierarchical structure that allows a more uniform distribution of the workload (see 8.3).

8.2 Centralized architecture

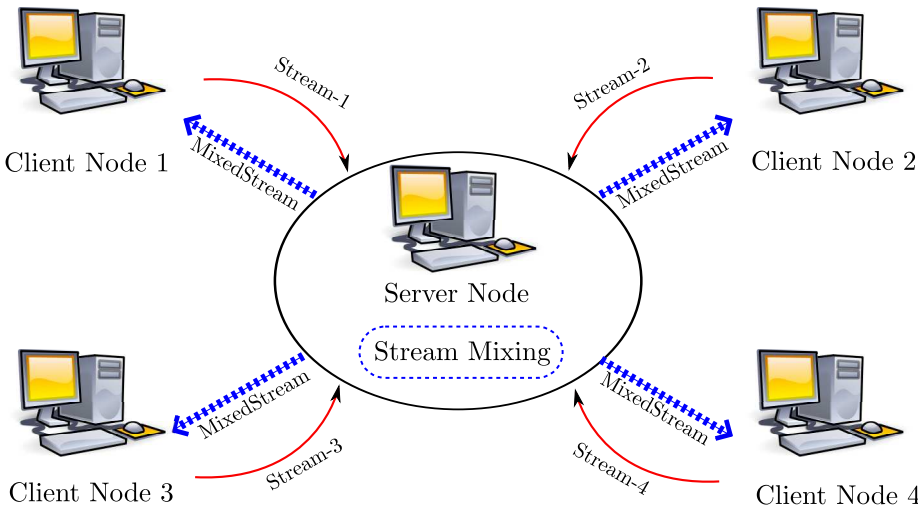

Figure 8.2: The SimpleConference architecture.

The currently implemented Multicast architecture is centralized and is named `SimpleConference`. In this organization, a peer is elected as a "server" and handles the multicast connection. All the other peers belonging to the group act as "clients". The server node holds a direct connection with clients, receiving their individual data streams and sending back the multicast traffic.

Generally, the information to be distributed using multicast is a combination of data streams generated by individual members of the group. For example, in an audio conference, audio stream from all peers should be mixed, resulting in a single stream. This merging operation is highly dependent on the application using the multicast services. In the `SimpleConference` architecture, the merging is performed by the server. In that node, the Multicast module acquires streams

from all the clients and returns them as a mixed stream, leaving its computation to be performed by the application using the multicast services.

Note how implementing this architecture is relatively simple, because the handling of the communication is centralized in the server node. In addition, the `SimpleConference` maintains low latency in the distribution of content, a feature highly desirable in many contexts, like audio or video conferences. Unfortunately, there are two flaws that limit its use. First, the **robustness** of the network is affected by its centralized nature. The whole system strictly depends on the server; if the server node fails, the entire communication is interrupted regardless of the status of other nodes. Second, the system guarantees poor **scalability**. In fact, when considering the computational load needed for mixing information flows and the bandwidth needed for transmitting them toward the other nodes, the server node turns out the bottleneck of the network. Generally, bandwidth and computational resources requirements increase linearly as the number of nodes in the group. These issues make this architecture feasible only for groups where nodes are not too many. A more scalable architecture, currently developed, is described in the next Section.

8.3 Distributed architecture

The architecture described in this Section is named `AdvancedConference`. It is currently being implemented and organized as a distributed architecture, where the load is splitted among several nodes. It is designed to improve the scalability and reliability of the previously seen `SimpleConference` architecture.

The basic structure is taken from the NICE[1] application-layer multicast protocol [8]. A hierarchically-connected control topology is created and the data delivery path is implicitly defined, according to the hierarchical organization: this is crucial in order to increase the scalability of the network. Nodes are organized in a layered fashion, and are sequentially numbered beginning from the bottom. The first layer L_0 contains all the peers, which are then partitioned into a set of clusters. Only one peer in each cluster is included in the second layer L_1 – the so called *cluster-leader*. Generalizing, layer L_i is partitioned into clusters, each of which has a leader that is a layer L_{i+1} member. An example considering

[1] An acronym standing for "NICE is the Internet Cooperative Environment"

three layers is reported in Figure 8.3.

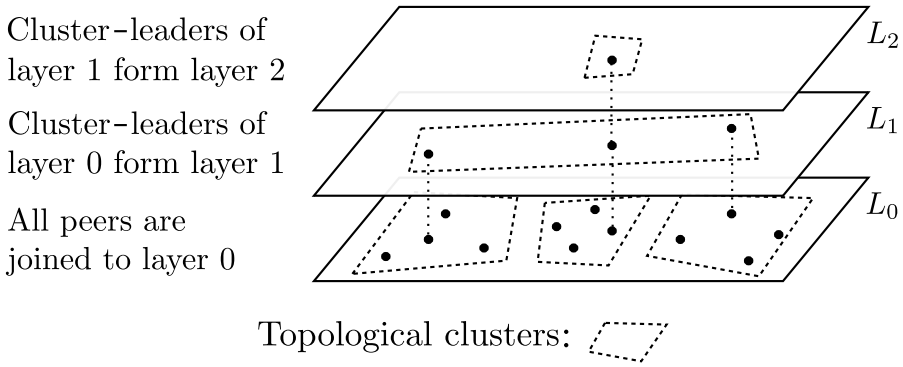

Figure 8.3: An example of Distributed Architecture topology.

Data delivering is organized as follows:

- all the peers send data to all the members of their layer L_0 cluster;
- a cluster-leader mixes all the data from this cluster and sends them to the higher-layers clusters whose it is a member;
- a member of a cluster mixes data from its cluster and sends them to lower-layers clusters whose it is a cluster-leader.

Figure 8.4 shows an example of this protocol. L_0's peers are partitioned into four clusters, each counting four peers. B_i and C_0 are leaders of their cluster in L_0. These peers together form a cluster in L_1, which is leaded by C_0. (a) and (b) subfigures show the paths followed by messages using this routing protocol. In the first case, date come from a peer that resides only in a L_0 cluster. In the second case, they come from B_0, which is a cluster-leader and should route data to the peers belonging to the upper layers.

As described above, a cluster must exhibit a fully connected topology. Moreover, each non-leader peer must mix streams coming from all the members of its cluster. This is not a problem as long as the maximum cluster size counts only a few nodes, but for higher sizes, each cluster can be structured as a `SimpleConference` group (see 8.2). In this case, the cluster-leader must act as a server, mixing and redistributing all the data coming from the members of the same cluster as well as those coming from the upper layers.

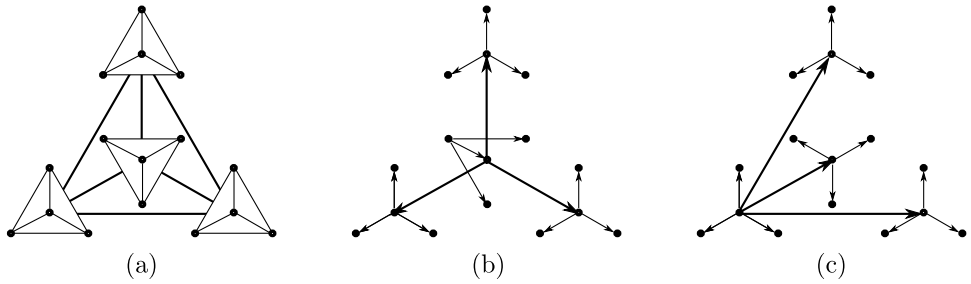

Figure 8.4: Data delivery according to the `AdvancedConference` architecture.

In the `MultiConference` structure, if the cluster size is equal to C and there are n peers, the number of layers is at most $N_L(n) \in \Theta(\log_C n)$. The maximum number of hops between two peers is $2(N_L(n) - 1) + 2 = 2N_L(n) \in \Theta(\log_C n)$, so the number of hops is logarithmic in n. Regarding bandwidth, peers in L_0 requires a bandwidth B that is the sum of their output stream and the mixed input stream. If a peer is also residing in a greater layer L_i (with $i > 0$), requested bandwidth is $(C \cdot B)i + B$.

It is should be clear, now, how the `AdvancedConference` architecture achieves a better scalability than the `SimpleConference`. Latencies suffered by the `AdvancedConference` may appear higher, since they are logarithmic in the number of nodes instead of constant, but, for groups smaller than the size of a cluster, latency is constant.

Another advantage of the `AdvancedConference` is its **robustness**. In fact, the failure of a leader can isolate communications to and from its cluster, but the rest of the network remains unaffected.

In conclusion, the `AdvancedConference` distributed architecture presents a more complex topology, but exhibits a better scalability when compared with the `SimpleConference`; furthermore, it is more suitable for medium and large groups. On the other hand, the `SimpleConference` remains preferable to multicast communications within-a-few, mainly because it is easy to manage . The boundary between these two strategies can be fixed or based on bandwidth and computational capabilities of a server node in the `SimpleConference`. In fact, if the workload of the server exceeds its capabilities, reliance to `AdvancedConference` architecture allows to split the load across multiple peers.

8.4 Selecting servers and cluster-leaders

In the two architectures just seen, the choice of the nodes acting as server or cluster-leader is a crucial aspect. A non-optimal choice may translate into a performance degradation. The three main criteria to be adopted are: latency, bandwidth and reliability. The relative importance of these parameters depends on the specific application which is using multicast services. However, it is necessary to estimate these values for each peer in order to select the more appropriate one.

Latency

Latency is an essential parameter for most many-to-many communications, including instant messaging, VoIP, videoconferencing, etc.. Let us denote by δ_{ij} the RTT[2] between peers i and j, both belonging to the multicast group's subset G. In order to determine the subset of peers $\tilde{i}$ with the best overall latency, we consider:

- $\tilde{i} = \left\{ i \in G : \max \delta_{ij} \leq \max \delta_{kj} \ \forall k \in G, j \in G \right\}$
  i.e. peers with minimum maximum latency;
- $\tilde{i} = \left\{ i \in G : \sum_{j \in G} \frac{\delta_{ij}}{|G|} \leq \sum_{j \in G} \frac{\delta_{kj}}{|G|} \ \forall k \in G, j \in G \right\}$
  i.e. peers with minimum average latency;

Other formulas or policies may be used, depending on RTT values and other requirements.

Most of the criteria designed for determining the best peer in terms of latency require knowledge of the RTT between all the pairs of peers. This can be trivially found doing $O(|G|^2)$ measurements, considering all the pairs of peers in the interested set $|G|$. An alternative and more scalable solution is to use an algorithm for RTT estimation [14] [29] [16]. In these algorithms, a multi-dimensional coordinate space is used: peers are placed through the measurement of RTT with a small number of other peers. The RTT of any pair of peers can now be evaluated by applying a function to their coordinates.

[2]Round Trip Time

Bandwidth

This parameter is particularly significant when each peer must receive a lot of data or there are several peers joining the multicast group.

Denote by R the required bandwidth of a given application between a single pair of peers. A node with bandwidth B can interact with no more than $\lfloor \frac{B}{R} \rfloor$ peers simultaneously. Choosing a server with a higher band in the centralized architecture allows to increase the maximum number of peers that can join the group. Similarly, in the distributed architecture, having cluster-leaders with a higher bandwidth lets increase the size of the clusters and, therefore, decrease the number of layers and the overall latency.

The measurement of the available bandwidth of a peer is not a definitively solved problem yet. The bandwidth purchased by the Multicast plug-in (see Section 6.2 for further details) is considered as the actual available bandwidth. This can be a good approximation in an absolute sense, but it could be very different from the real bandwidth available for other members of the group. A partial solution may be monitoring the internal traffic and exchanging additional packages, allowing an estimate of the available bandwidth between each pair of nodes.

Reliability

In order to guarantee the stability of the overlay network, servers and nodes must exhibit a high degree of reliability.

Since we are dealing with a P2P network, an important parameter is represented by the *Remaining Stay Times* (RST). Several studies [10] [37] highlight that peers with a higher age tend to stay longer in the group, so the age of a node could be used as a good predictor of the RST. Given this result, in the Multicast plug-in, reliability of a node is estimated on the basis of its uptime, where more stable nodes are those that have been connected to the network for a longer time.

To improve reliability, it is also necessary an adequate failure detection and recovery system. The main symptom of a failure is generally the interruption of data flow: when this condition occurs, in the current implementation of Multicast, it is enough to replace the peer suspected to be broken with another which is known to work. In order to replace a failed node and recover the network status, new peers should obtain the topological information stored by the failed node:

for this purpose, one should replicate the information held by a leader through a certain amount of nodes belonging to the group.

Chapter 9

Anonymity

The main purpose of the *Anonymity* package is to ensure the *untraceability* of PariPari users. Similarly to Multicast (Chapter 8), Anonymity resides in a package external to the Connectivity plug-in itself, ensuring modularity with respect to other I/O services.

Anonymity provides, in particular, facilities for carrying out data transmissions not revealing the IP addresses owned by the communication endpoints. In general, hiding the content of a message just encrypting it could not be a sufficient protection: one could track that message, discovering in such a way *who* is talking with *whom*. This tracking is identified by a specific name: *traffic analysis*. Anonymity plug-in exactly aims at preventing such analysis, by means of *onion routing* techniques: Section 9.1 details these mechanisms.

Since the plug-in can be divided into two distinct "subunits", *Sender Anonymity* and *Receiver Anonymity*, we explain in more detail each subunit separately – in Sections 9.2 and 9.3, respectively.

9.1 Onion routing: an overview

In the last decade, Anonymity techniques have noticeably evolved: as a few examples, we can cite Crowds [7], Freenet [13] and OneSwarm [24]. In our plug-in, we have adopted the Onion Routing model [35], whose most famous implementation is perhaps Tor [19].

Onion routing provides bidirectional, application-independent and nearly real-time communications; in particular, it relies on a group of nodes – the *onion*

routers (OR) – providing an anonymity service: it is exactly through these nodes that encrypted communications take place. The last hop along this "chain" is the *exit relay* node: it performs a clear transmission of messages toward the server the anonymous connection should be established with.

Therefore, a client aiming at connecting to a server must perform the following steps:

- retrieving (by contacting a *Directory Server*) a list of onion routers usable to redirect the traffic;
- identifying a sequence $p_1...p_n$ of such nodes: this will constitute the "communication chain" toward the server;
- encrypting the data to be sent and its own address with p_n's public key, obtaining the payload c_n;
- calculating the generic payload c_i, encrypting the payload c_{i+1} together with p_{i+1}'s address with p_i's public key;
- sending a packet with c_1 as payload to p_1.

Thanks to this structure, each node along the chain can decrypt the packet just received, obtaining a new payload and the address of the next hop. Note how, in such a way, it knows only its predecessor and its successor along the chain; no other nodes are revealed.

Figure 9.1 schematizes the communication flow we have just explained.

Actually, in PariPari we do not deal with entities like clients and servers: we deal with peers. This means that a peer providing a service is not necessarily known by every other node belonging to the network: it simply publishes that service through the DHT 2.2; in the same manner, a node looking for a service does not have any server at its disposal: it performs a search across the network, in order to find out an active node currently offering that service. As we easily see, a difficulty arises at this point: both these publications and researches must be anonymous, as well as the establishment of communication between the involved peers. The following Sections explain how Anonymity addresses these problems.

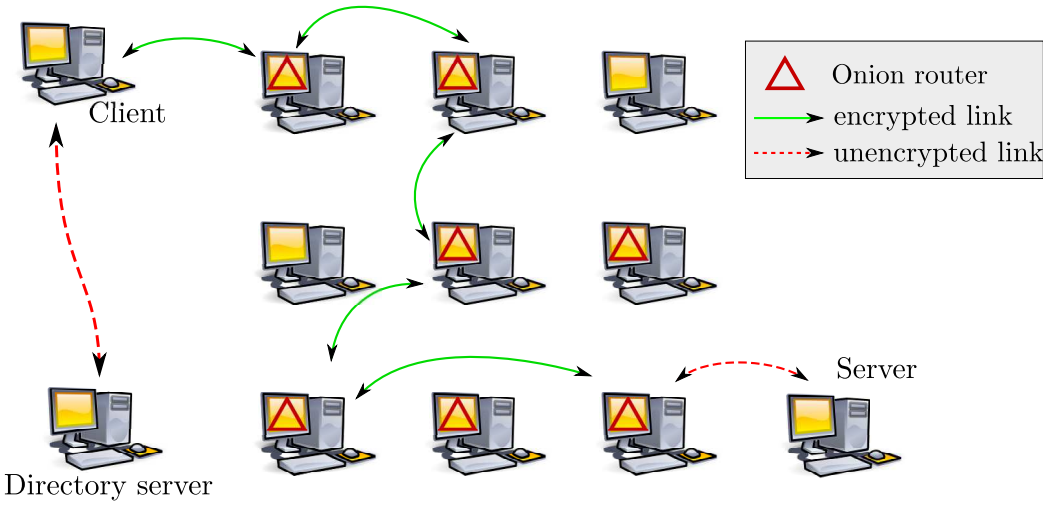

Figure 9.1: Communicating by means of onion routers.

9.2 Sender Anonymity

The Sender Anonymity service allows a node to publish a resource across PariDHT in an totally anonymous manner. In particular, this means that the node the `store(`K, V operation is called on does not know which node is actually calling that store; furthermore, even if a pointer is stored, the actual identity of the node being referenced remains hidden. This is accomplished thanks to onion routing techniques similar to those described in Section 9.1. Let us represent the node p_0 aiming at (anonymously) publishing a resource as the head of the chain. The procedure it should follow can be easily summarized:

1. p_0 retrieves a list of nodes providing the anonymity service;
2. among those nodes, a succession $p_1..p_n$ of nodes is selected: p_0 must acquire their public keys $K^+_{p_1}..K^+_{p_n}$;
3. p_0 obtains the payload c_n for the peer p_n, constituted by a random generated number Id_{n-1} together with the data required by the store operation, encrypted with $K^+_{p_n}$;
4. s obtains the payload of the generic packet p_i, containing the (IP_{p_i+1}, c_{i+1}) pair: it is encrypted with $K^+_{p_i}$ and the randomly generated number Id_{i-1}, representing the previous node;
5. p_0 sends a packet with payload c_1 to p_1, as shown in Figure 9.3.

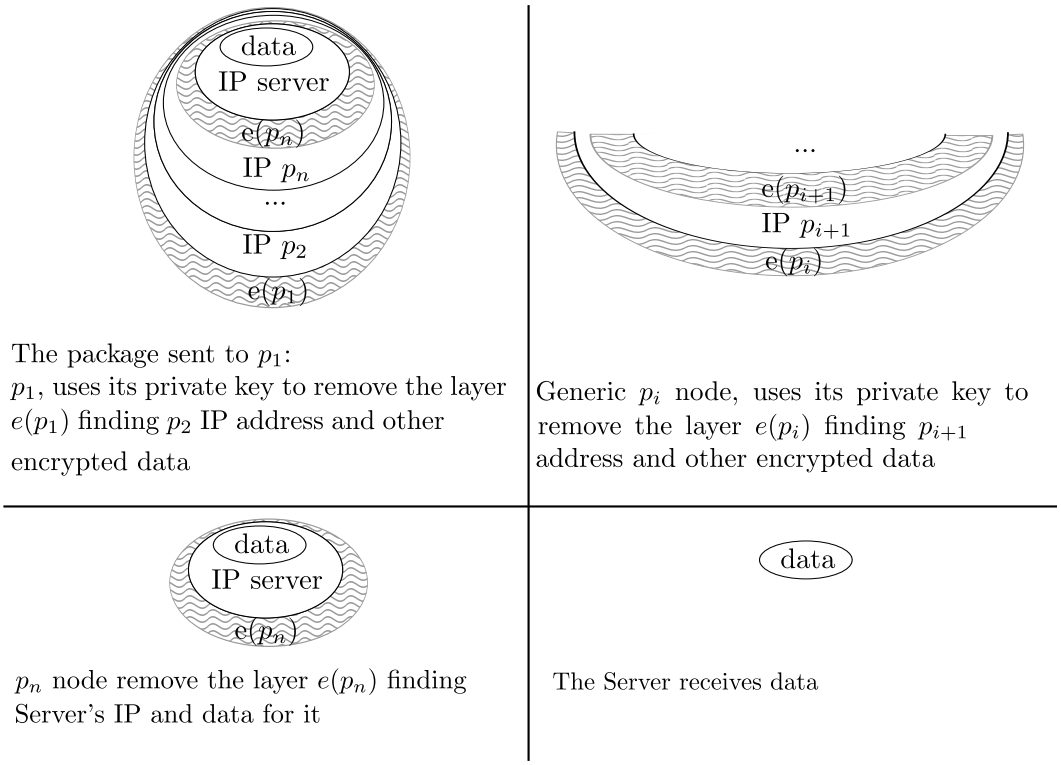

Figure 9.2: Onion routing encapsulation.

Note how, according to this structure, each intermediate node p_i receives a packet containing Id_{i-1}, p_{i+1}'s IP address and the payload c_{i+1} to be forwarded. From the last one, the value of Id_i can be extracted, since it is not encrypted. Before forwarding the message, p_i adds the quadruple constituted by $(Id_{i-1}, IP_{i-1}, Id_i, IP_{i+1})$ to a routing table it constantly keeps up-to-date: by using this table, p_i can forward subsequent messages coming from and going to pair of adjacent nodes along the chain.

The node laying at the head of the chain, p_n, performs the `store` operation on PariDHT, accordingly to the data brought by the received packet's payload. A flag indicates whether V represents a value or a reference; in this case, the referred node is p_n: that is, every request addressing the resource is forwarded to p_n, which has to forward the request along the chain in turn.

$$c_1 = Id_0, (IP_{p_2} \overbrace{Id_1, (IP_{p_3}, \overbrace{Id_2, (IP_{p_4}, \ldots \overbrace{\ldots Id_{n-1}, (\texttt{ store } (K,V))^{K^+_{p_n}}}^{c_3})^{K^+_{p_3}}}^{c_2})^{K^+_{p_2}}}^{})^{K^+_{p_1}}$$

Figure 9.3: Sender Anonymity: payload of first sent packet.

Id_{i-1}	IP_{i-1}	Id_i	IP_{i+1}
56434	147.162.2.250	62662	174.25.12.114
54735	147.162.2.250	12394	174.25.12.114
34234	147.162.2.250	22344	89.5.14.24
24546	95.43.56.123	23424	90.15.62.45

Table 9.1: An example of routing table.

9.3 Receiver Anonymity

As we have already mentioned, the Receiver Anonymity module addresses the problem of making the retrieval of a generic resource $r = (K, V)$ anonymous across PariDHT. If V represents a pointer to a peer, it is possible to establish an anonymous communication with that peer by using the same chain of peers crossed during the `search` operation. The mechanism, here, is quite similar to that seen for the Sender Anonymity module: the creation of the chain follows the usual onion routing technique's rules, with the head of the chain acting on the place of the node requesting the anonymity and taking care of forwarding messages.

A node p_0, aiming at performing an anonymous `search` on PariDHT, will follow the same steps we have already seen in Section 9.2: in this case, the payload c_n of the packet sent to the head of the chain contains the parameters according to those the retrieval is accomplished. Packets transmitted among the intermediate nodes p_i exhibit the same structure as in the Sender Anonymity process; similarly, the operations such as the forwarding of payloads from one hop to another and the insertion of the $(Id_{i-1}, IP_{i-1}, Id_i, IP_{i+1})$ quadruple into

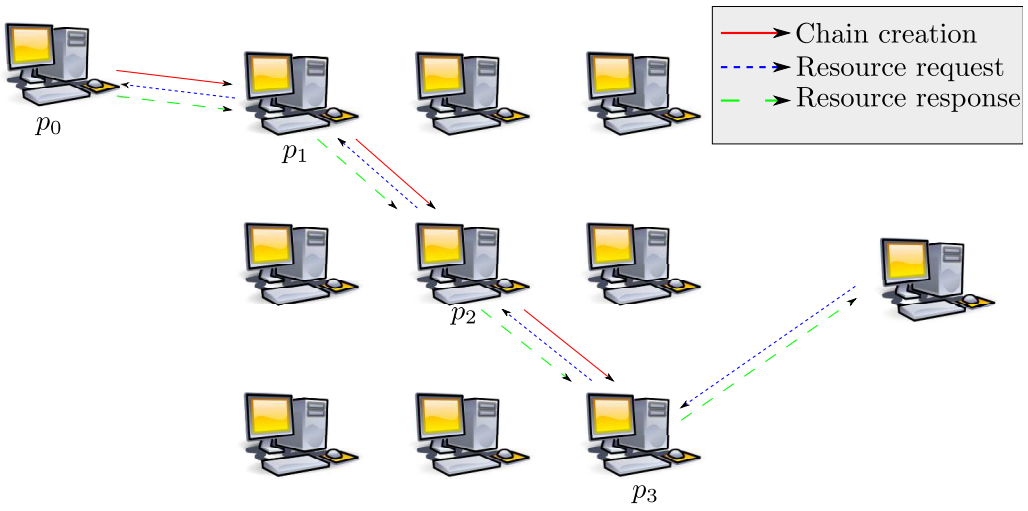

Figure 9.4: Sender Anonymity: an example of communication chain.

the routing table are performed in the same manner as well.

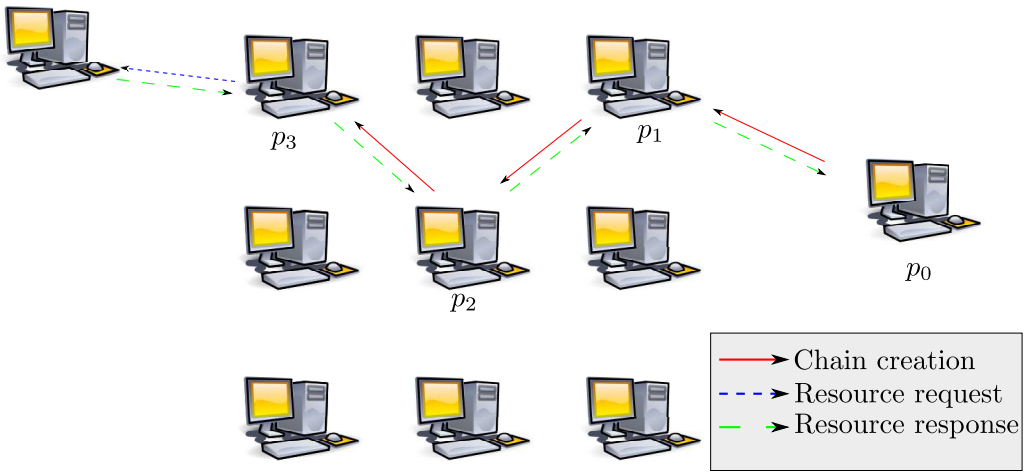

Figure 9.5: Receiver Anonymity: an example of communication chain.

9.4 A completely anonymous communication

As discussed above, Anonymity lets peers both provide and retrieve a resource or a service in an anonymous manner. Using both services gives rise to a completely anonymous communication, in which peers offering or requesting resources remain unknown. In fact, if a server p_0^s holds an anonymity chain $p_1^s..p_n^s$, and a corresponding client, similarly, an anonymity chain $p_1^r..p_n^r$, we have that:

- peers $p_1^s..p_{n-1}^s$ and $p_1^r..p_{n-1}^r$ know only their adjacent peers, without knowing their own position along the chain;
- peers p_n^s and p_n^r know that they are, respectively, the head of the chain and the destination address of the packets passing through the chain; but this node only represents the head of the other chain, not the actual destination;
- peers p_0^s and p_0^r known all the members belonging to their chain, but only the head of the other chain. All these peers are simply anonymizer nodes.

Chapter 10

Team management

PariPari is currently being developed by 60+ students, mainly as their thesis work or as a course project. With the exception of the two project leaders, the team is entirely composed by students, whose time availability, programming skills and personal initiative may widely vary.

Given these characteristics, the need of a robust hierarchical organization, as that described in Section 10.1, naturally arises; furthermore, some programming techniques maximizing the productivity in this context are introduced, as discussed in Section 10.2. An overview of the employed workforce is provided in Section 10.3.

10.1 Organizational structure

Each PariPari's plug-in is developed by a team of students. The modular organization of the software leads to a semi-isolation of each plug-in from the rest of PariPari: many times, communications between teams simply reduce to reading documentation of the other plug-in's APIs. Some sets of plug-ins share many components, so they are grouped into confederations. An example of such a confederation is represented by PariConnectivity, PariAnonymity and PariMulticast.

The entire workforce is organized according to a hierarchical structure, as shown in Figure 10.1.

At the top, there are the *project leaders*, who define the main lines of the project and the deadlines for the pre-releases. They are constantly kept updated by the plug-in leaders about the overall situation, through frequent meetings

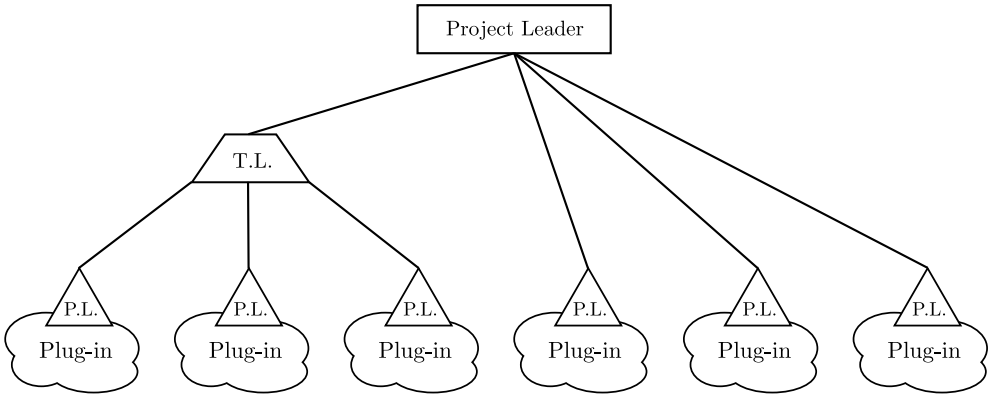

Figure 10.1: Hierarchical organization in PariPari.

and weekly reports. They are the only ones that can decide about rewarding or punishing students. Currently, the project leaders are the Ph.D. student Michele Bonazza and the assistant professor Paolo Bertasi.

Team leaders determine the priority of tasks for each plug-in in the confederation, taking into account the already established pre-release deadlines. They have a clear view of all the confederation's plug-ins interaction and know well the structure of PariPari.

Plug-in leaders organize the job assignments within a group, generally consisting of 2-10 students. If a plug-in is not part of any confederation, its plug-in leader also acts as team leader.

10.2 Programming techniques

Since the workforce of PariPari consists entirely of students, it is subject to a very high turnover; typically, students spend about nine months working on the project. This fact, combined with the large number of programmers, make necessary to define precise rules and guidelines in the development process. In particular, these choices have been made:

- the OOP[1] paradigm is adopted: it enables faster code development, easier maintenance and simpler testing process;
- eXtreme Programming software development methodology is followed, in

[1] Object Oriented Programming

order to improve software quality and responsiveness to requirements changings [9];

- a Test-driven development (TDD) process; it is related to the test-first programming concepts of the eXtreme Programming and drastically reduces the number of bugs introduced during refactoring or code changing;
- plug-ins are mutually isolated and communicate only through public APIs, allowing a more effective testing, less necessary knowledge to newcomers to begin the development and less influence of bugs across plug-ins;
- many development-related aspects are standardized; just for citing a few, we mention: style and format for the source code, tools used for writing code, compiling it and reporting bugs. This allows a simpler code-merge and an easier interaction among programmers.

10.3 Workforce management

Most of the PariPari developers are students, who do this work as their thesis project. This year, a group of students attending the course of Software Engineering has joined the PariPari time, for the duration of an entire semester. These two categories of workforce present completely different characteristics: we will analyze them separately, specifying, for each of them, strengths, faults and the most appropriate allocations in the project.

10.3.1 Working on PariConnectivity for a thesis

These students are generally recruited during large, sporadic recruitment events, involving dozens of people. During these events, an intensive advertising about PariPari's benefits is made. The importance of joining the project as a unique opportunity to learn the tools and techniques used in the industry is emphasized. The possibilities offered by the software and the future prospects of being able to take over many contemporary P2P solutions, represent usually a good argument in order to motivate students to join. The advantage of this type of recruitment is that it is able to attract motivated students who believe in the project, at the cost of not having any control over their other qualities.

Indeed, it is difficult – almost impossible – to determine the actual skills owned by a student during recruitment. As a consequence, the assignment of a student to a determined plug-in is dependent only on the preferences suggested by the student itself. In some cases, students whose commitment is low or erratic, or whose prior knowledge is totally insufficient for the project, are recruited; the time spent by senior-members to manage and train them is greater than the contribution provided to the project by these students.

These members suffer from a main disadvantage: they do not have predictable periods in which they can focus on code development. Each of them writes code for about nine months, but the commitment is distributed in a completely heterogeneous way, generally depending on the exams' dates and on the personal organization of the time devoted to study. This makes it difficult to schedule task and to organize milestones in order to synchronize jobs, especially when managing plug-ins that provide APIs to others, for which the deadline are much more stringent. This time fragmentation also brings difficulties in the newcomers training. In fact, although many student will join the project during the same recruitment session, they are available to begin code development in very different periods.

An advantage of this workforce are the relationships that are established among students during collaboration. The team meetings are generally informal, and are organized through non-academic communication channels. The climate is necessarily collaborative, since the attitude of direct competition among students makes no sense in this context, and all the developers share an interest in the overall project. The relationships with the leaders are generally good too, since they do not make an accurate assessment of the work, but more properly evaluate the correspondence between work and requests and provide help and tips to achieve them, when necessary.

Another advantage is the interest and initiative shown in the project. Students decide to join PariPari because they are interested in what the project can offer. This leads some students to offer more time and resources to the project than those that would be required for a normal thesis work. This is often a crucial aspect, that distinguishes future team leaders.

In conclusion, these students, after doing an early assignment that lets us train them and test their skills, are extremely useful in new or young plug-ins.

They should work on challenging tasks, that maintain high their level of interest; they are also good candidates to become future team/plug-in leaders.

10.3.2 Working on PariConnectivity as a Software Engineering student

This year, students attending the course of Software Engineering had to carry out a practical code development activity, that contributed in assessing their final evaluation.

Students were involved in four different projects (PariPari, Psort, ELaw and BioScanner). Due to a "numerus clausus" policy, only 60 students could be admitted (40 in PariPari). Over 150 students applied: this high request allowed a careful selection. Since practical aspects of development heavily influenced the final evaluation, it was decided to include some team leaders in the selection procedure. It was composed of a simple logic test, an individual interview with one or more team leaders and a coding homework of moderate complexity (with a three days long deadline). This allowed to admit students with good basic skills and to assign them to the plug-ins that best suited their abilities.

The relationships among the students are poor. Students with previous friendships are rarely put in the same group when using this selection procedure. Interactions with other members take place only during lessons, reducing the possibility of creating new friendships. Also, relationships established with the team leaders are very different with respect to those established by students working on their thesis. All the team leaders had to periodically report to the course's professor about the quality of the work being done: this clearly requires to establish a less confidential relationship between the student and the team leader.

Another problem is the interest in the project. These students are interested in learning the basics of Software Engineering, many of them are not specifically interested in PariPari. Their main incentive is the final evaluation: this means that they generally have a high productivity during the course, but are rarely interested in keep working in the project.

Many important benefits derive from the fact that all these students should develop PariPari during a well-defined course schedule, finishing the work at home if necessary. This allows to greatly reduce the time spent for training, since it is done to all the students simultaneously. Additionally, these students focus on

this course like on any other exam, without giving always low priority to code development like other PariPari members. This leads to an easier synchronization across different tasks and to a much faster development process.

Summarizing, the pros and cons of this workforce are basically opposed to those of other members. These students can be used to develop parts that require rapid development, which have high dependencies with other parts of the code or that regard less interesting aspects. Instead, they are less useful for those plug-ins that still are at an early stage of development.

Chapter 11

Conclusions and future work

In this thesis, we have illustrated the reengineering and optimization of PariConnectivity has been achieved. The code handling all connection-related aspects (that is, the Connectivity Core) has been completely rewritten, using the Java NIO API; this allowed to provide more advanced features to other plug-ins like an asynchronous I/O, a centralized QoS System and NAT Traversal techniques, with tunneled flows. Furthermore, plug-ins like Multicast and Anonymity were improved and adapted to the new API.

Currently, the code exhibits a good stability even when dealing with I/O intensive plug-ins, such those providing file sharing and distributed storage services. Note how these are the areas where the introduction of the new features has more relevance. Gradually, the plug-ins less interested in the new features (that is, less involved in intensive network I/O) are migrating to the new version too, mainly for compatibility reasons.

However, as we have seen, further developments must take place. In the next Sections, we briefly overview them.

11.1 Intensive testing

An advantage of Connectivity is that it's used by almost all other plug-ins, so finding bugs is easier, even if the pressure when they must be fixed is higher. Test-writing and bug-fixing will therefore always represent a very important part of the work to be done in PariConnectivity.

11.2 TLS/SSL support

Transport Layer Security (TLS) [18] and its predecessor, Secure Sockets Layer (SSL), are cryptographic protocols that provide communications security over the Internet. They use symmetric cryptography in order to guarantee privacy and a keyed message authentication code in order to guarantee message reliability. Providing a socket that natively supports these protocols currently represents a very high-priority task.

11.3 Serialization support

Many plug-ins require to transfer objects instead of primitive types over the network. Currently, this involves a proper management done by the interested plug-ins, since this feature is not natively supported by the APIs provided by PariConnectivity. Moreover, the facilities provided by Java are generally used to serialize objects, which in many cases is a solution with low performance and interoperability with external applications. Therefore, we need to provide a set of APIs that natively allow an easy exchange of objects over the network, using different types of serialization.

11.4 Direct file transfer

Java NIO channels are used not only to perform network I/O, but also to deal with files; besides, efficient methods for exchanging data between two channels are provided. From this fact, the idea of implementing a set APIs for the direct transfer of data between network and local storage is born. This can be particularly useful for file sharing and distributed storage plug-ins, that currently have to waste RAM for temporarily storing the data exchanged between PariConnectivity and the module itself.

Bibliography

[1] Azureus. `http://azureus.sourceforge.net/`.

[2] emule. `http://www.emule-project.net`.

[3] Javadoc page for java.net package. `http://download.oracle.com/javase/6/docs/api/java/net/package-summary.html`.

[4] Javadoc page for java.nio package. `http://download.oracle.com/javase/6/docs/api/java/nio/package-summary.html`.

[5] Jxta. `https://jxta.dev.java.net/`.

[6] Skype. `http://www.skype.com`.

[7] Crowds: anonymity for Web transactions. *ACM Transactions on Information and System Security*, Volume 1:pages 66–92, 1998.

[8] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *Proceedings of the ACM SIGCOMM*, volume 32, pages 205–217, 2002.

[9] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[10] M. Bishop, S. Rao, and K. Sripanidkulchai. Considering Priority in Overlay Multicast Protocols Under Heterogeneous Environments. In *Proceedings of IEEE INFOCOM*, pages 1–13, 2006.

[11] Russell Bradford. *The Art of Computer Networking*. Pearson/Prentice Hall, 2007.

[12] David Buettner, Julian Kunkel, and Thomas Ludwig. Using Non-blocking I/O operations in high performance computing to reduce execution times. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 134–142, 2009.

[13] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies*, volume 2009 of *Lecture Notes in Computer Science*, pages 46–66. Springer Berlin / Heidelberg, 2001.

[14] Manuel Costa, Miguel Castro, Antony Rowstron, and Peter Key. PIC: Practical Internet Coordinates for Distance Estimation. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 178–187. IEEE Computer Society, 2004.

[15] M. Cotton, L. Vegoda, and D. Meyer. IANA Guidelines for IPv4 Multicast Address Assignments. RFC 5771 (Best Current Practice), 2010.

[16] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. In *Proceedings of the ACM SIGCOMM*, 2004.

[17] S.E. Deering. Host extensions for IP multicasting. RFC 1112 (Standard), 1989. Updated by RFC 2236.

[18] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), 2008. Updated by RFCs 5746, 5878.

[19] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: the second-generation onion router. In *Proceedings of the 13th conference on USENIX Security Symposium (SSYM'04)*, volume 13, 2004.

[20] John William Evans and Clarence Filsfils. *Deploying IP and MPLS QoS for Multiservice Networks: Theory & Practice*. Morgan Kaufmann Publishers Inc., 2007.

[21] Paul Ferguson and Geoff Huston. *Quality of service: delivering QoS on the Internet and in corporate networks.* John Wiley & Sons Inc., 1998.

[22] R. Hinden and S. Deering. Internet Protocol Version 6 (IPv6) Addressing Architecture. RFC 3513 (Proposed Standard), 2003. Made obsolete by RFC 4291.

[23] Ron Hitchens. *Java NIO.* O'Reilly, 2002.

[24] Tomas Isdal, Michael Piatek, Arvind Krishnamurthy, and Thomas Anderson. Privacy-preserving P2P data sharing with OneSwarm. In *Proceedings of the ACM SIGCOMM*, pages 111–122, 2010.

[25] Benedek Kovács. Mathematical remarks on token bucket. In *Proceedings of the 17th international conference on Software, Telecommunications and Computer Networks (SoftCOM'09)*, pages 151–155, 2009.

[26] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach (4th Edition).* Addison Wesley, 2007.

[27] R. Mahy, P. Matthews, and J. Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766 (Proposed Standard), 2010.

[28] Petar Maymounkov and David Mazi'eres. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of the 1st International Workshop on Peer-to Peer Systems (IPTPS02)*, 2002.

[29] T. S. E. Ng and Hui Zhang. Predicting internet network distance with coordinates-based approaches. In *Proceedings of IEEE INFOCOM*, pages 170–179, 2002.

[30] C. Partridge, T. Mendez, and W. Milliken. Host Anycasting Service. RFC 1546 (Informational), 1993.

[31] E. Peserico. P2P economies. In *Proceedings of the ACM SIGCOMM*, 2006.

[32] Larry L. Peterson and Bruce S. Davie. *Computer Networks, Fourth Edition: A Systems Approach.* Morgan Kaufmann Publishers Inc., 2007.

[33] J. Postel. User Datagram Protocol. RFC 768 (Standard), 1980.

[34] J. Postel. Transmission Control Protocol. RFC 793 (Standard), 1981. Updated by RFCs 1122, 3168, 6093.

[35] Michael G. Reed, Paul F. Syverson, and David M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, volume 16:pages 482–494, 1998.

[36] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). RFC 5389 (Proposed Standard), 2008.

[37] Kunwadee Sripanidkulchai, Bruce Maggs, and Hui Zhang. An analysis of live streaming workloads on the internet. In *Proceedings of ACM SIGCOMM conference on Internet measurement (IMC '04)*, pages 41–54, 2004.

[38] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022 (Informational), 2001.

[39] P. Srisuresh, B. Ford, and D. Kegel. State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs). RFC 5128 (Informational), 2008.

[40] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM*, pages 149–160, 2001.

[41] Andrew Tanenbaum. *Computer Networks.* Prentice Hall Professional Technical Reference, 4th edition, 2002.

[42] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition).* Prentice-Hall, Inc., 2006.

[43] Puqi Perry Tang and Tsung-Yuan Charles Tai. Network traffic characterization using token bucket model. In *Proceedings of IEEE INFOCOM*, pages 51–62, 1999.

List of Figures