Università
degli Studi
di Padova

DEPARTMENT OF
INFORMATION
ENGINEERING
UNIVERSITY OF PADOVA

Facoltà di Ingegneria
Dipartimento di Ingegneria dell'Informazione

Scuola di Dottorato di Ricerca in Ingegneria dell' Informazione – XXII Ciclo
Indirizzo: *Scienza e Tecnologia dell'Informazione e della Comunicazione*

PariPari: Design and Implementation of a Resilient Multi-Purpose Peer-to-Peer Network

Direttore della Scuola
Ch.mo Prof. Matteo Bertocco

Supervisore
Ch.mo Prof. Enoch Peserico

Dottorando
Paolo Bertasi

Abstract

Recent years have seen a considerable and constant growth in peer-to-peer (P2P) traffic over Internet. Internet Service Providers (ISPs) and software houses have begun to change their attitude towards P2P applications, no longer seen as bandwidth-eager enemies, but rather as interesting profit opportunities.

PariPari is a P2P platform under development at Department of Information Engineering Engineering of the University of Padova. It differs from traditional P2P applications like eMule, Skype or Azureus in that it provides a multifunctional, extensible platform on which multiple services - from filesharing to VoIP to mail/web/IRC services — can run simultaneously and cooperatively. PariPari offers a collection of APIs allowing third party developers to write their own applications; but unlike similar P2P development frameworks such as JXTA, PariPari already works "out of the box" for the end user offering a large number of applications.

The number and the heterogeneity of services offered by PariPari together with the possibility of extending this platform with future, not currently defined, applications offer a number of challenges: effective use of Java, coordination of multiple services, design of a powerful but easy to use GUI, efficient and robust algorithms for clock synchronization and search etc.

Effective group management was the key to successful development of PariPari. Over the past three years more than one hundred students have cooperated developing PariPari. To coordinate such a large number of people we have adopted software engineering techniques such as eXtreme Programming and Test Driven Development. However, these paradigms had to be adapted to a group of non-professional, although motivated, developers. This coordination process was difficult, but extremely rewarding, and taught us a number of lessons about software engineering that might be useful in other software projects involving large numbers of relatively inexperienced, part-time developers with high turnover.

Sommario

Negli ultimi anni il traffico dovuto al peer-to-peer (P2P) è aumentato costantemente. Gli Internet Service Providers (ISPs) e le software house hanno iniziato a considerare le applicazioni P2P non come programmi avidi di banda ma come interessanti opportunità.

PariPari è una piattaforma P2P in sviluppo al Dipartimento di Ingegneria dell'Informazione. È molto diversa da altre ben note applicazioni P2P come eMule, Skype o Azureus. dato che fornisce una piattaforma multifunzionale e estensibile sulla quale diversi servizi — dal filesharing al VoIP all'email — possono funzionare simultaneamente. Inoltre, PariPari fornisce un insieme di API utili agli sviluppatori terzi per scrivere le loro applicazioni, ma diversamente dagli altri framework P2P come JXTA, PariPari offre già da subito un gran numero di applicazioni fruibili dall'utente finale.

PariPari offre, quindi, un gran numero di servizi eterogenei e la possibilità di estendere la piattaforma, in futuro, con applicazioni non ancora definite. Per produrre questi due risultati, la progettazione di PariPari ha dovuto affrontare diverse interessanti sfide tra cui un uso efficiente di Java, la possibilità di coordinare diversi servizi e la studio di nuovi algoritmi per la sincronizzazione e la ricerca.

La chiave del successo dello sviluppo di PariPari è sicuramente la gestione del gruppo. Negli ultimi tre anni, più di un centinaio di studenti hanno lavorato allo sviluppo di PariPari. Per coordinare tanti contributi abbiamo adottato tecniche tipiche dell'ingegneria del software come l'eXtreme Programming e il Test Driven Development. Questi paradigmi, tuttavia, hanno subito pesanti modifiche per essere adattati al nostro gruppo di sviluppatori dalle peculiari caratteristiche: gli studenti, sebbene motivati, non hanno nè il rendimento nè la preparazione di un professionista. La gestione è stata complessa ma estremamente appagante ed ha prodotto molti interessanti spunti che possono essere studiati ed applicati ad altri progetti che coinvolgono molti sviluppatori non professionisti con alto turn-over.

Ringraziamenti

Scrivere la pagina dei ringraziamenti suscita sempre emozioni contrastanti. La felicità per la possibilità di ringraziare le persone cui devo molto si scontra con la paura di dimenticarne qualcuna o di non riuscire ad esprimere bene i miei pensieri.

In primis, "devo", ringraziare Daniela, mia moglie, per avermi sempre aiutato a risolvere i problemi che ho incontrato aiutandomi a squadrarli da una prospettiva diversa e, soprattutto, per aver sopportato tutti i week-end che ho passato a lavorare e le notti che ho trascorso in laboratorio con Marco ed Enoch. Grazie, anche, perché continua a pungolarmi proponendomi continui stimoli per la mia crescita.

Ora che sono padre della piccola Lucia mi rendo ancor più conto di quanto i miei genitori mi abbiano dato nei miei primi 30 anni di vita e di quanto i loro sforzi e i loro insegnamenti continuino a sorreggere la mia coscienza e il sistema di valori che mi hanno trasmesso. Ringrazio loro certo di non poter riuscire ad esprimere a parole quello che penso e tutta la mia riconoscenza.

Non avrei nemmeno iniziato questo dottorato se non mi fossi imbattuto nel gruppo di calcolo ad alte prestazioni. Il periodo che ho passato come lavoratore a progetto mi ha dato la possibilità di percepire con quali persone avrei potuto collaborare e mi ha spinto ad intraprendere il dottorato (anche senza la tranquillità di una borsa di studio). In particolar modo ringrazio:

- Andrea e Geppino per tutto quello che mi hanno insegnato durante i lavori su Aeolus;
- Mauro per avermi seguito nel mio ingresso in ACG e per avermi risposto: "Welcome to the scientific world!' quando gli ho fatto notare che eravamo sempre in ritardo sulle deadline.
- i dottorandi della vecchia guardia (Alberto B., Francesco V., Fabio) per aver condiviso ore e ore di duro lavoro e l'organizzazione delle nostre pantagrueliche grigliate (Advanced Cooking Group). In particolare voglio ringraziare coloro i quali mi hanno aiutato nella stesura della tesi: Francesco S. e Marco;
- i nuovi dottorandi (Michele e Alberto P.) per aver riportato ad alti livelli la serietà del gruppo tuttavia lasciandosi influenzare dal buon umore respirato in laboratorio.
- ed infine Enoch Peserico Stecchini Negri De Salvi per aver creduto in me anche e soprattutto nei momenti in cui sarebbe stato più facile lasciar perdere.

Un ringraziamento va anche a Francesca Musiani dell' Ecole des Mines per i suoi preziosi consigli sulla stesura della parte ”sociologica“ di questa tesi.

Desidero inoltre ringraziare tutti i PariParisti presenti e passati, e in particolar modo i team/plug-in leader, per aver creduto in questo progetto e avermi aiutato in queste fasi finali convulse della scrittura della tesi. Sono loro l'anima di questo progetto e la forza di questo progetto. In particolare, ringrazio Michele B. per avere corretto la bozza di questa tesi.

Infine, voglio ringraziare Sandra Calore e Marco Filippi per avermi aiutato sempre, tempestivamente ed in modo ineccepibile quando ne ho avuto bisogno.

Indice

Chapter 1

Introduction

PariPari is a serverless P2P multi-functional application designed to offer all traditional P2P services (file-sharing, distributed storage, VoIP,. . .) along with a number of traditionally server-based ones (email hosting, IRC chat, web hosting, NTP,...). It is currently developed by more than 50 students from the University of Padova, but aims at reaching a larger community of developers as soon as it is officially released. In fact, all the services offered by each plug-in are described by simple APIs. Using these APIs a third party developer can interact with existing plug-ins and develop new applications, taking advantage of an already established P2P network, potentially with a large user base. PariPari is written in Java, so that it runs on all Operating Systems that have a Java Virtual Machine without the need to recompile the code. Moreover, students at University of Padova are very familiar with Java which is essentially the only language they are taught at some depth - any other language choice would have forced us to spend considerable energies re-training our developer base, with uncertain results. Unfortunately, Java precludes access to a number of low level functionalities of the machine and of the operating system; this has forced us in a number of cases to adopt rather roundabout solutions.

The remainder of this thesis is split into two parts. The first part deals with the design and implementation of PariPari. In particular, after a description of the application structure we examine every plug-in. For each we analyze the state of the art, the challenges offered by "P2Pization", and how we faced them.

The second part of the thesis discusses human resource management and programming methodologies. We review the characteristics of our developer base (inexperienced, part time, unpaid, with high turnover), and the challenges it poses; and how we dealt with those challenges with careful personnel management and with a modified version of Extreme Programming.

Figure 1.1: PariPari logo.

Part I

PariPari: Network Project and Development

Chapter 2

Structure

PariPari is designed as a multifunctional, extensible platform for P2P applications. By allowing all such applications to run with a "common base" on the user's machine, PariPari can avoid conflicts, exploit synergies, and allocate resources to those applications that need them most.

To simplify the development of PariPari applications - both by future third party developers and by the large student body currently working on PariPari - it was paramount to structure PariPari in a highly modular fashion. Each service (IRC, distributed storage etc.) is provided by a module - we call these "plug-ins" - that can be loaded on demand by PariCore (in some sense PariPari's kernel). PariCore also assigns resources to plug-ins, and manages communication between them through a simple, extensible set of APIs.

All plug-ins can be divided roughly into two classes. The first comprises those plug-ins offering basic services — often invisible to the end user — supporting the PariPari infrastructure e.g., connectivity, local and remote storage etc. These form in some sense the "operating system" of PariPari, and are accorded a number of privileges by PariCore. The second class of plug-ins comprises those plug-ins offering end user services e.g. filesharing, IRC etc. These are often useful only to a limited fraction of users, and so can be loaded on demand by PariCore, which accords them only limited privileges. We expect the vast majority of third party plug-ins to fall into this second category.

PariPari is designed to be launched as a Java Web Start application. In this way, anybody can use it, without installation, just clicking on a web page. Another strong point is that JWS keeps PariPari software constantly updated to the latest version avoiding the user all the maintenace operations.

The following chapters will each describe one of the plug-ins currently in PariPari or under development, beginning with the "Operating System layer" plug-ins and continuing with the “application layer” ones.

- Section 3.1 describes PariCore.
- Section 3.2 describes PariDHT. This plug-in manages the PariPari Distributed HashTable. Plug-ins, using PariDHT can join and search the network and offer their resource and services to the plug-ins operating on other peers.
- Section 3.3 describes PariStorage. This plug-in manages the accesses to the storage systems. PariStorage can use all the PariPari network as a large distributed storage system.
- Section 3.4 describes PariConnectivity. This plug-in manages all the network related traffic providing some advanced features such as multicast, anonymity and NAT traversal.
- Section 4.1 describes PariSync. This plug-in, mainly developed to satisfy PariStorage needs, provides the peers with clock synchronization.
- Section 4.3 describes PariMessaging. This plug-in provides PariPari with an infrastructure to deal with all the most popular chat and instant messaging systems. Actually it implements Microsoft Messenger and IRC chat systems.
- Subsection 4.4.1 describes Mulo. This plug-in provides the peer with a eMule compatible client.
- Subsection 4.4.2 describes Torrent. This plug-in provides the peer with a Bittorrent compatible client.
- Section 4.5 describes PariWeb. This plug-in offers a (distributed) web hosting service.
- Section 5.4 describes PariCredits. This plug-in is embedded in PariCore; it provides the functionalities needed to allow PariCore to arbitrate plug-ins requests.
- Subsection 5.1.1 describes PariDNS. This plug-in offers a distributed DNS service.
- Subsection 5.1.2 describes PariLogin. This plug-in manages a decentralized login system to allow users to be recognized by the network.
- Section 5.2 describes PariDBMS. This plug-in offers a distributed DBMS service.

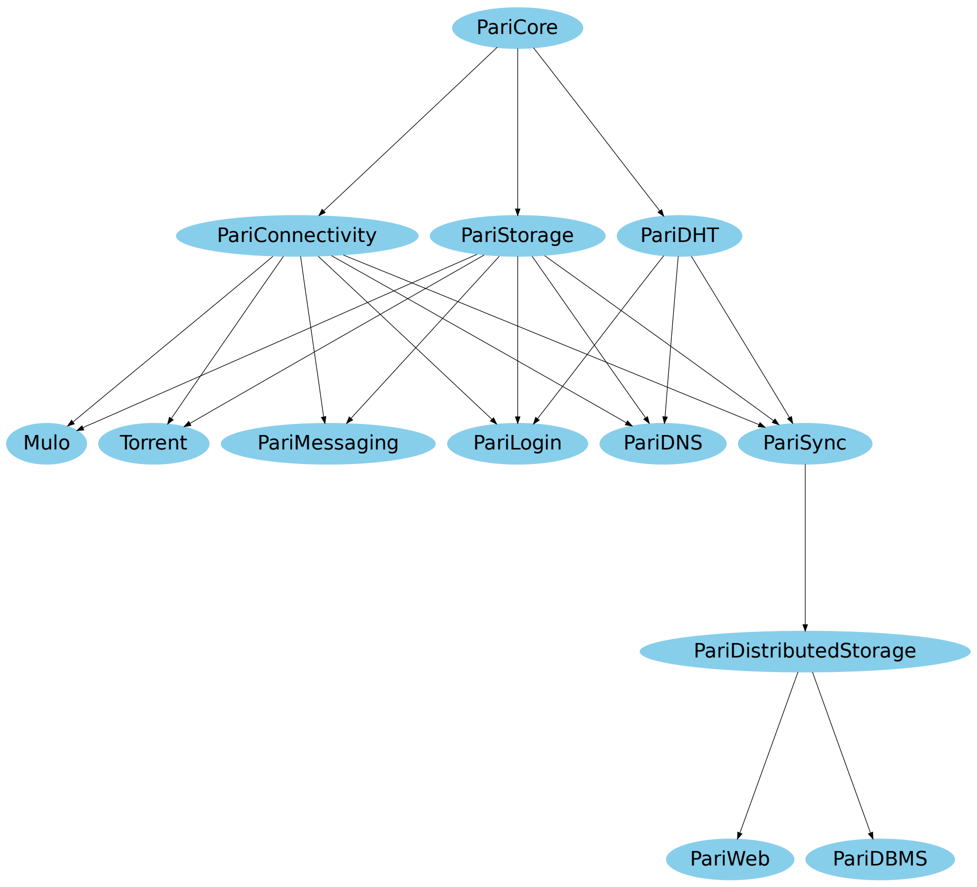


Figure 2.1: Transitive reduction of the plug-in dependencies graph of PariPari.

Chapter 3

Operating System Layer Plug-ins

3.1 PariCore

PariCore is the kernel of the system. The goal of PariCore is to load, start and stop at runtime the other plug-ins and to make them work correctly. To accomplish these tasks PariCore uses a dependency solving algorithm. Hence, whenever a plug-in is required to start, PariCore loads and starts all requested plug-ins. In order to keep the PariPari framework safe and to let developers write easily their own plug-ins, we do not let plug-ins communicate with each other. PariCore routes messages among the running plug-ins. This way, each plug-in cannot be accessed directly by any other plug-in: only the PariCore can, preventing malicious or badly written code from being executed by good plug-ins. Moreover, PariCore grants PariPari with security by checking all messages, and by exploiting PariCredits information.

To summarize, PariCore provides the following services:

- It loads the requested plug-ins solving any dependencies.
- It inhibits plug-ins from performing unwanted operations.
- It delivers messages and resources among plug-ins.
- It executes operations requested by PariCredits.
- It performs basic network tasks (i.e., checking for updates and verifying jar signatures).

To provide these functions we developed a new plug-in architecture called **T.A.L.P.A.** and described below.

3.1.1 T.A.L.P.A. PariCore

T.A.L.P.A. - **T**he **A**cronym for **L**ightweight **P**lug-in **A**rchitecture was designed to be the kernel of PariPari. It manages plug-ins, routes their messages and protects users and good plug-ins from malicious ones.

To handle plug-ins offering the same service running at the same time **T.A.L.P.A.** instantiates a separate classloader for each plug-in. Having separate namespaces easily permits a peaceful coexistence of plug-ins sharing some package names, as class names are of the form `class_name@classloader_name`. However, in this way plug-ins must know how to use the objects they request to others, and if two different implementations of the same service have different fields or methods there's no possibility that the same code could run unchanged with both.

To deal with this problem we decided to separate the definition of a service from its specific implementations.

All services available in PariPari are defined by a set of abstract classes named `API`. A super interface that all APIs must implement (`paripari.API.API`) contains some methods needed to let the Credit System manage exchanges. Two plug-ins offering the same service must provide an object extending the same reference API.

There are several advantages in doing so:

- All the documentation about classes, methods, constructors and variables can be written only once.
- Developers are not required to know anything about a specific implementation, reading the javadoc is all it takes to know how to use a specific service.
- Code can be used independently of the particular implementation of the service is running.
- Having a single standard for each service helps in writing better and more complete plug-ins, as the development of a feature in an implementation of the service strongly encourages other implementations to develop it as well.
- Having different implementations for the same service improves efficiency. When plug-ins ask for some service (identified by its reference API) the actual implementation chosen by PariCore varies on a single-case basis, depending on which plug-in offers the best performance (the cheapest in terms of Credits of those meeting the buyer requirements).

Moreover, having one classloader for each plug-in also helps improve security. Plug-ins cannot access any of the classes contained in other plug-ins jars, as they're loaded in different classloaders. Separate namespaces mean that reflection does not work at all if not performed on the same jar in which the running code is in. This is a strong security improvement, because every plug-in could before access any variable, class or method (whether public, protected or even private!) in any jar. Also, our Security Manager can easily understand where method calls come from by just looking at the classloader that loaded the running class.

PariPariSecurityManager

To increase user safety we wrote our own Security Manager that replaces Web Start's. It is more restrictive, since the only plug-ins that we trust by default are PariDHT, PariStorage and PariConnectivity. These are modules written by the PariPari Team and should not be replaced even in the future. It is of crucial importance that these four modules (including the Core) are closely inspected by PariPari developers and not by external teams, as being able to manipulate some code in one of them could allow the transformation of PariPari into a malaware platform.

The Java Security Architecture is usually based on some **policy files** containing lists of permissions to be granted to various code sources. These files must be manually placed in some directories that in Unix systems are often accessible only by super-users. This is not a viable solution for a web application.

An alternative to using policy files is that of replacing in the security chain the default Java Security Manager with a custom one. The default Security Manager reads policy files to know whether to permit certain security-related method calls. Our Security Manager uses the keystore couple ID-public key to know whether the calls come from one of the above-quoted plug-ins or not. We decided not to let users change this behavior, anything they want, since understanding the complex and subtle interplay of permissions and security levels is probably beyond the average third party application developer, and we felt that the limited reduction in flexibility was well worth the substantial increase in safety.

Moreover PariCore, for security reasons and to allow the strict control of the resources, inhibits the use of standard `java.io` threads. Nevertheless, it provides PariPariThread. These objects work like standard threads but are under direct control of PariCore and Credits.

Future Work: Memory Management

One of the initial objectives of the PariCore was memory management. Studying the problem in depth, though, revealed that Java does not offer enough to accomplish this task. Assigning a determined amount of memory to a specific thread, for instance, is something that simply cannot be done directly, using Java. Using indirect memory analysis seems to be a feasible approach to the problem. A substantial help could come from the next version of Java (Java 7) that could include the Resource Consumption Management API required and specified by JSR 264 [jsr]. A set of API to manage resource (memory, CPU, disk) consumption would ease the development of that section of PariCore.

3.2 PariDHT

PariDHT implements the fundamental data structure to search for resources (files, services etc.) in the PariPari network. As the name suggests, this data structure is a Distributed Hash Table.

After a surge of interest in the academic community (e.g., Chord [SMK+01], CAN [RFS+01], Pastry [RD01], Kademlia [MM02]) recent years have seen DHTs adopted in several applications with a vast public (e.g., the popular eMule [emu] and Azureus [azu] filesharing clients and the JXTA system [jxt]). While DHTs can be implemented in many different ways, almost all the mainstream ones (including all the ones we cited above) function according to a basic scheme that we summarize below.

Roughly speaking, each node in a DHT is assigned a random address in a b-bit ID space (b is chosen sufficiently large, typically 160 or more, to avoid collisions). Some form of distance (pseudo-)metric is defined on this address space (e.g., the XOR metric used in Kademlia [MM02]), so that one can partition the space, for any given node, in the 2^{b-1} addresses in the "other half" of the network, the 2^{b-2} addresses in the same half but in the other quarter, the 2^{b-3} addresses in the same quarter and the other eighth, and so on . Each node then keeps contacts with a small number k of nodes in the other half of the network, k nodes in the other quarter, k in the other eighth and so on (see Figure 3.1). Theoretically $k = 1$ would suffice, but in practice some redundancy is introduced to provide robustness and typically $5 \leq k \leq 20$ is used. Of course less than k nodes might be present in some of the smallest regions, in which case all the nodes in any such region are kept as contacts.

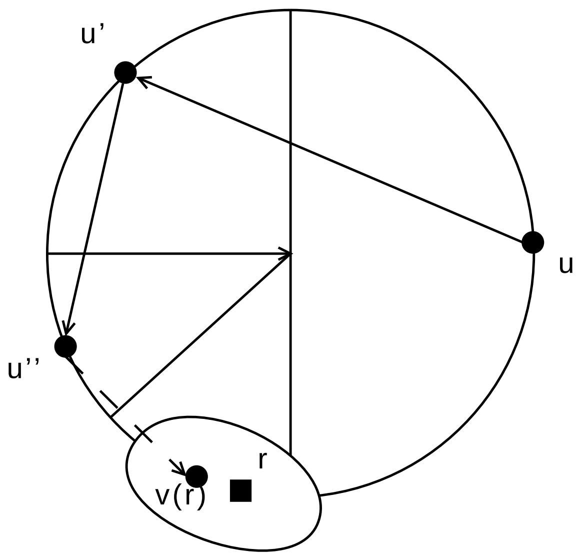


Figure 3.1: The search structure in a typical DHT.

Each resource r (e.g., a file) is also mapped into the same address space using a pseudorandom hash of the keyword(s) that will be used to locate it; information about how to reach it (e.g., the IP of the machine from which it can be accessed) is stored in the node $v(r)$ closest to it in the address space. To locate $v(r)$ — whether to retrieve the information on how to access r, or to store it in the first place — a node u will forward the query to the node u', among its contacts, closest to r in the address space. In the worst case, u' will be in the other half of the network — but it will certainly be in the same half as r. It can then forward the query to another node u'' that will certainly be in the same quarter as r — and so on, until $v(r)$ is reached in a number of steps logarithmic in the size of the network with high probability.

3.2.1 Protocol: Kademlia

PariDHT currently implements the Kademlia data structure [MM02] Kademlia adopts as a distance function between nodes and/or resources the XOR between their IDs. It is not difficult to prove that XOR is a metric (i.e. satisfies the properties of non-negativity, symmetry, identity of indescernibles and triangle inequality). In addition, XOR is extremely easy to compute and exhibits several other "nice" properties (see [MM02] for a more comprehensive treatment). Note that, in general, IDs are completely unrelated to geographical position. Thus it could happen that a node in Germany could be the closest to a resource in Australia. We are currently investigating the possibility of "locality preserving" IDs (see [IMRV97]).

Every node in a Kademlia network stores information to route messages. Routing tables, called k-buckets, consist of a list for each bit of the node id. (e.g., if a node ID consists of 128 bits, a node will keep 128 k-buckets.) A bucket has many entries. Every entry in a bucket holds the necessary data to locate another node. The data in each bucket entry is typically the IP address, port, and node id of another node. Every bucket corresponds to a specific distance from the node. Nodes that can go in the nth bucket must have a differing nth bit from the node's id; the first n-1 bits of the candidate id must match those of the node's id. This means that it is very easy to fill the first bucket as $\frac{1}{2}$ of the nodes in the network are far away candidates. The next bucket can use only $\frac{1}{4}$ of the nodes in the network (one bit closer than the first), etc.

As nodes are encountered on the network, they are added to the lists. This includes store and retrieval operations and even when helping other nodes in finding a key. Every encountered node will be considered for inclusion in the lists. Therefore the knowledge that a node has of the network is very dynamic. This keeps the network constantly updated and adds resilience to failures or attacks. Moreover k-

buckets stores information not only on just a sole node but on k nodes at the right distance. Typically $k = 20$.

It is known that nodes that have been connected for a long time in a network will probably remain connected for a long time in the future [SGG02]. Because of this statistical distribution, Kademlia selects long connected nodes to remain stored in the k-buckets. This increases the number of known valid nodes at some time in the future and provides for a more stable network.

When a k-bucket is full and a new node is discovered for that k-bucket, the least recently seen node in the k-bucket is PINGed. If the node is found to be still alive, the new node is place in a secondary list; a replacement cache. The replacement cache is used only if a node in the k-bucket stops responding. In other words: new nodes are used only when older nodes disappear.

Kademlia uses four kind of messages.

`Ping:` like ICMP ping: used to verify that a node is still alive.

`Store:` to store a (key, value) pair in a node.

`Find Node:` The recipient of the request will return the k nodes in his own buckets that are the closest ones to the requested key.

`Find Value:` as `Find Node`, but if the recipient of the request has the requested key in its store, it will return the corresponding value.

Each message includes a random value from the initiator. This ensures that when the response is received it corresponds to the request previously sent.

Using these messages, it is simple to understand the lookup procedure. Node lookups can proceed asynchronously. The quantity of simultaneous lookups is denoted by $\alpha = 3$. A node initiates a `Find Node` request by querying the k nodes in its own k-buckets that are the closest ones to the desired key. When these recipient nodes receive the request, they will look in their k-buckets and return the k closest nodes to the desired key that they know. The requester will update a results list with the results (node ID's) it receives, keeping the k best ones (the k nodes that are closest to the searched key) that respond to queries. Then the requester will select these k best results and issue the request to them, and iterate this process again and again. Because every node has a better knowledge of his own surroundings than any other node has, the received results will be other nodes that are every time closer and closer to the searched key. The iterations continue until no nodes are returned that are closer than the best previous results. When the iterations stop, the best k

nodes in the results list are the ones in the whole network that are the closest to the desired key.

Information is located by mapping it to a key. A hash is typically used for the map. The storer nodes will have information due to a previous `Store` message. Locating a value follows the same procedure as locating the closest nodes to a key, except the search terminates when a node has the requested value in its store and returns this value.

A node that would like to join the net must first go through a bootstrap process. In this phase, the node needs to know the IP address and port of another node (obtained from the user, or from a stored list) that is already participating in the Kademlia network. If the bootstrapping node has not yet participated in the network, it computes a random ID number that is supposed not to be already assigned to any other node. It uses this ID until leaving the network.

The joining node inserts the bootstrap node into one of its k-buckets. The new node then does a `Find Node` of its own ID against the only other node it knows. The "self-lookup" will populate other nodes' k-buckets with the new node id, and will populate the new node's k-buckets with the nodes in the path between it and the bootstrap node. After this, the new node refreshes all k-buckets further away than the k-bucket where the bootstrap node falls in. This refresh is just a lookup of a random key that is within that k-bucket range.

3.2.2 Implementation Hacks

We have added a number of small hacks in the implementation of the Kademlia data structure. In this section we detail two. The first allows for richer metadata to characterize queries and/or resources. The second cuts almost in half the latency required to locate a resource in the (not infrequent) case of highly reliable nodes in the "main search path" leading to the resource.

Options field. Each node on the Kademlia network is identified by a unique ID and by the piece of information needed to reach the node itself (its IP and PORT). Hence we can define a structure called triplet containing these three fields that is 52-byte long. However we chose to keep the length of triplets as a multiple of 16 bytes. This choice leaves us 12 bytes of spare space that can be used as an "options" field to provide additional functionalities. For example, this field can be successfully used to store a small public key.

Sometimes plug-ins need more information to be carried by the datagram. However a more consistent increase in the datagram size could lead to a network per-

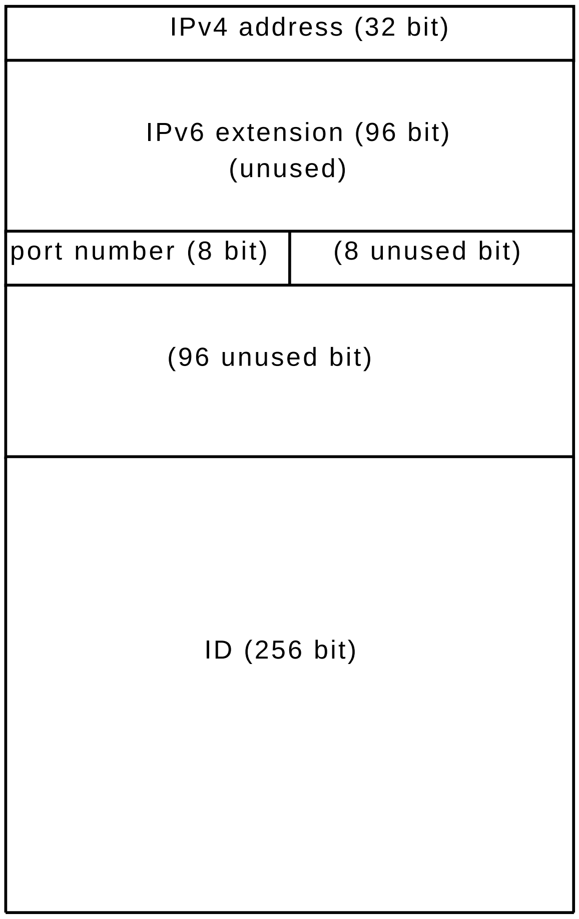


Figure 3.2: The PariDHT datagram (64 bytes).

formance worsening. So we established a new protocol to associate a note with a node stored in the DHT. The note is not actually stored by the node that holds the triplet. The note lies in the originating node (i.e., the node linked by the node that holds the triplet). This way the requesting node can reach the originating to read the content of the node.

Favorite Child. To cut down the latency during lookup methods we are developing a simple hack to the original algorithm. We called this method the “favorite child lookup”. Roughly speaking, the vanilla Kademlia algorithm make the requesting node to iterate its request on each step during the look-up procedure waiting every time for the answer of the inquired node. Differently, the “favorite child lookup” eliminate these waiting charging the inquired node to continue the look-up procedure. Every time a node starts a lookup operation it chooses a “favorite child” among its neighbors. This node, instead of answering to the requester following the standard procedure, directly forwards the request to one of its neighbors, which in turn will forward the request to another chosen neighbor and so on. The directly contacted nodes are labeled as “favorite children”. It is clear that, in case of successful requests, the latency is almost halved.

3.2.3 Future work: Multi attribute range query

One of the main directions of future work for PariDHT is an efficient implementation of (multi attribute) range queries. A range query is an operation that retrieves all keys whose value is between an upper and lower boundary. Searching for peers providing services like storage or computation power can be done efficiently only allowing range queries. As far as we know the best solution for our purpose is described in [PLGS04]. This solutions exploits P-trees (a distributed version of B+-trees [Com79]) to efficiently evaluate range and equality queries.

Multi attribute queries combine results from two or more search queries in a way that is similar to an SQL join in a database. To efficiently provide this feature it is necessary to perform the join while doing the search. Our proposed solution is based on [WS06]. This system introduces DHR-Trees (based on Hilbert R-trees [GG97]) and exploits P-tree range queries to perform Multi attribute range query in $O(\log_d(N))b$ (with N number of Nodes on d-dimension space).

3.2.4 Future work: Load balancing

An implementation of a DHT usually solves the load balancing problem by fairly spreading the load among all participants, that is, uniformly mapping nodes and keys into the same (shared) identifier space using a consistent hash function. However, it has been shown [GDB05, SGG02] that real queries are not uniformly distributed among all addresses, but they tend to follow a Zipf-like distribution. Hence, an uniform mapping leads to unfair load assignment: some nodes, often called hot spots, receive more than the fair load, while other nodes are idle most of the time. To cope with this phenomenon, a DHT should be enhanced with load balancing algorithms that taking into account query distribution. In particular, we can distinguish two kind of problem. The traffic problem and the popularity problem. Roughly speaking, the traffic problem occurs because some nodes are flooded of queries originated by other node look-up operations. The popularity problem is instead due to the overpopulation of resources sharing the same ID. Nodes that are close to that ID should keep a large number of links.

Traffic problem Several strategies for reducing traffic problems in a structured network have been proposed. For example, Kaashoek and Stoica [KS03] introduced the concept of Virtual Server (VS): when a node joins the network, it assigns to itself several addresses, each corresponding to a virtual position in the identifier space; only one address is used at a time, the selection of such address depending on the load observed. Recently several enhancements have been proposed: a finer estimation of the number of virtual positions is proposed by Ledlie and Seltzer [CHXY08], while heterogeneity is treated by Chen et al. [GSBK04]. Even if traffic is balanced using VS, the problem of item popularity is not solved.

Popularity problem To deal with item popularity, several caching algorithm have been proposed. The idea is to replicate items on nodes other than the original owners and spread the load on all copies (original and replicated) as fairly as possible. The problem is to find the optimal number of copies and the best place for them so that the entire replication scheme is effective. Gopalakrishnan et al [GSBK04] proposed to create a replica on each node of the path from requester to original owner, while Bianchi et al. [BSFK06] observed that, as replicas move far from original position, their effectiveness decreases and therefore they proposed to place replica only at the last hop. Rao et al. [RCFB07] replicate on nodes of a k-ary tree rooted at the original resource owner, where the children are the k neighbors of a node, the replication scheme is similar to a breadth-first filling and the popular items are supposed to be replicated in more tree levels than non-popular ones. This solution seems the best fit for our system, but it needs some minor changes.

3.3 PariStorage

One of the fundamental components of PariPari is a system for reliable, distributed storage of data across the network. In a nutshell, this requires a) a system for safe, efficient local storage on each node of the network and b) a scheme to add redundant data so as to cope with failing nodes.

3.3.1 Local Storage

LocalStorage the resource manager in charge of accessing (and limiting access to) local permanent storage devices (hard drives, USB pens etc.)

This means that every plug-in wanting to create a file or access an existing file needs to forward a request to Local Storage, through PariCore, which verifies the request and check if it is satisfiable or not. If the request is satisfiable, the asking plug-in will receive in reply a FileAPI object that offers all needed methods to execute operations of creating, reading, writing, deleting for a file, keeping consequently updated the plug-in quota. Local Storage can prevent from plug-in unauthorized disk accesses.

At boot time, Local Storage looks for installed plug-ins and for each of them it retrieves name, available disk space, used disk space, maximum number of ownable files and number of owned files. It does so by scanning plug-in directories and saving a list of those files it finds.

Plugins informations are saved:

- in a file located in Local Storage directory. The file is not directly readable by other plug-ins and is read at Local Storage boot time to load informations about active plug-ins quota. It is updated after every operation that modifies plug-in quota.

- in proper files located in local directories of any plug-in, readable and writable both by plug-in and user (and therefore are not a reliable source for Local Storage goals). They are updated when any open stream on a file is closed (and not at every operation, to increase plug-in performance), so we always get a perfect data consistency.

Local Storage, furthermore, allows access to any file on the host disks but only after explicit user authorization (to prevent unwanted access).

3.3.2 Distributed Storage

A common method used to achieve high reliability and availability of data over a network is replication. This method consists in distributing replicated copies of a file over several nodes on the network. This way it is sufficient that at least one node is alive to recover the file. Another way is to use an *erasure code*. Erasure codes divide an object into m fragments and recode it into n fragments, where $n > m$. We call $r = \frac{m}{n} < 1$ the *rate* of encoding. This means that a rate r code increases the storage cost by a factor of $\frac{1}{r}$. The key property of erasure codes is that the original object can be reconstructed from *any* m fragments. For example, using $r = \frac{1}{8}$ we divide the file into $m = 8$ fragments and encode the original m fragments into $n = \frac{m}{r} = 64$ fragments. In this example the storage cost is increased by a factor of $\frac{1}{r} = 8$. We can observe that the set of erasure codes is a superset of replication-based systems. For example a system that creates eight replicas for each file can be described by an $m = 1$ $n = 8$ erasure code.

Under some circumstances, with fixed storage overhead erasure codes provide higher availability.

Availability: replication vs erasure codes

We now compare erasure codes and replication for what concerns the availability of a file. The file availability is the probability that, in a fixed moment, it is possible to recover the file. We characterize the availability of a peer by a parameter μ, known as peer availability. Peer availability is the probability that a peer is online and it is able to give a previously stored block in a fixed moment. We are assuming that all peers have the same peer availabilities. We call $S = \frac{1}{r}$ the storage overhead where r is the rate of encoding. We notice that for a replication-based system S represents the number of copies.

Replication

In a replication-based system the file is replicated S times and stored by S peers. In order to retrieve the file it is sufficient that at least one peer is available. So the file availability is

$$A_r(S) = \sum_{i=1}^{S} \binom{S}{i} \mu^i (1-\mu)^{(S-i)} \tag{3.1}$$

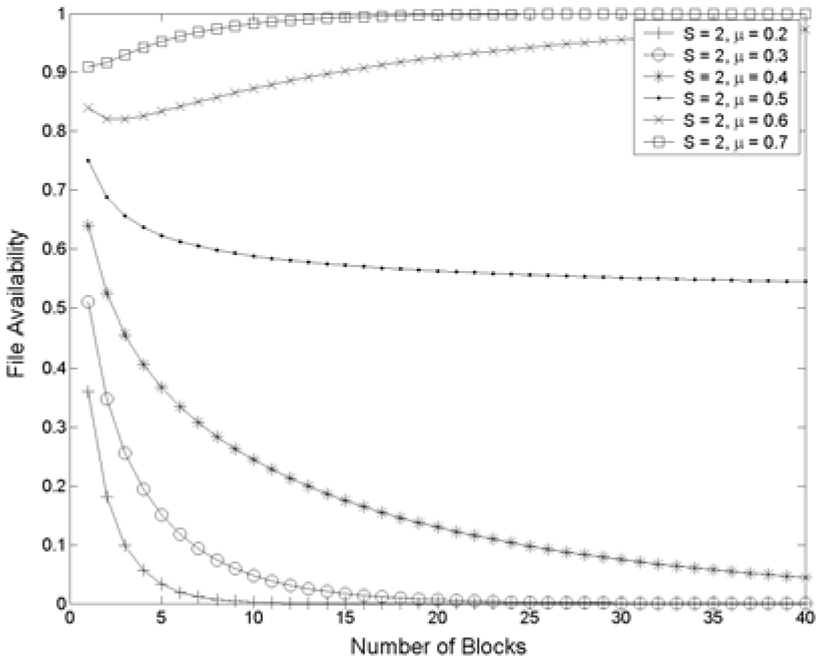


Figure 3.3: Effect of changing μ on the file availability.

Erasure codes

In an erasure code system we store on the network $n = S \cdot m$ blocks. We assume that the number of peers is large compared to n. With this assumption each block is allocated to one peer and therefore each block is independent of each other. In order to retrieve the file we need to retrieve at least m blocks so the availability of the file is

$$A_e(S \cdot m) = \sum_{i=m}^{Sm} \binom{Sm}{i} \mu^i (1-\mu)^{(Sm-i)} \tag{3.2}$$

We notice that if $m = 1$ then $A_r = A_e$ because replication is a particular case of erasure codes, so we analyze A_e and obtain information about A_r setting $m = 1$

Comparison

File availability depends on 3 parameters: μ, S and m. Since we want to compare replication ($m = 1$) and erasure coding ($m > 1$) we fix μ and S and we plot the availability against m.

Figure 3.3 is a plot of file availability against the number of blocks m in the system, with different peer availabilities and fixed $S = 2$ using equation 3.2. From the result we see that when the peer availability is low, replication is better than erasure code.

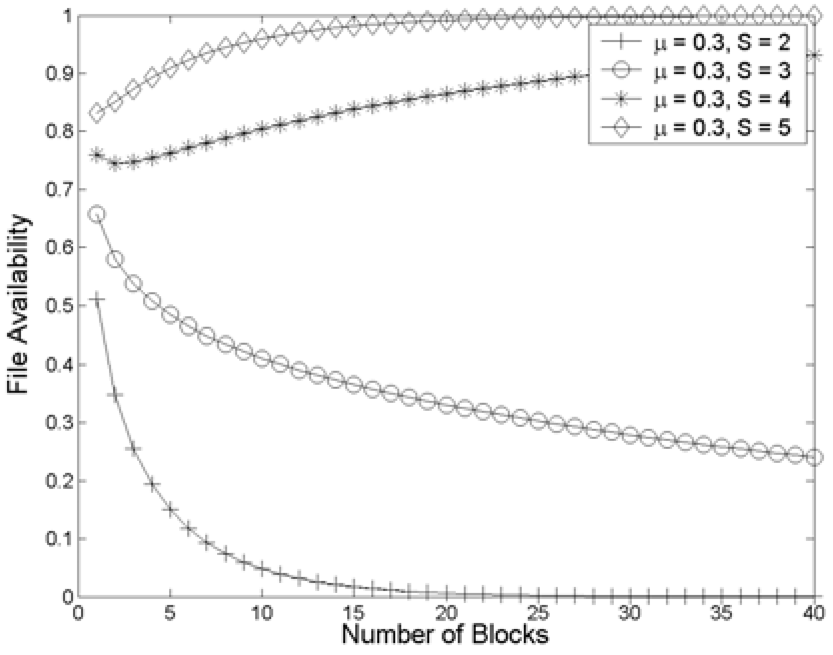


Figure 3.4: Effect of changing S on the file availability.

Figure 3.4 is a plot of file availability for fixed $\mu = 0.3$ and different storage overheads S. We notice that for high storage overhead (4-5) we have erasure code beating replication while for low overheads (2-3) it is better to use replication.

A brief overview on erasure codes

There are two main classes of erasure codes: *optimal* erasure codes and *near optimal* erasure codes. The first class of codes has the property that for each m subsets of n blocks it is possible to reconstruct the whole file. If the code is systematic we call the m blocks of the original message *source blocks* and the $m - n = \beta m$ blocks *check blocks*. An optimal erasure code can recover the original message from a random loss of at most a β fraction of blocks.

Near optimal erasure codes need, instead, $m + \epsilon$ blocks in order to reconstruct the whole file, so they can recover from a random lost of at most $(1 - \epsilon)\beta$ fraction of blocks. This suboptimality is usually rewarded by faster decoding and encoding algorithms. In term of distributed storage we have a trade off between storage overhead, that is minimized by optimal codes, and encoding and decoding speed. We will now describe some optimal and near optimal erasure codes, then we will focus on *Tornado code*, a very fast near optimal erasure code.

Code	Encoding complexity	Decoding complexity
Reed-Solomon	$O(n \log n)$	$O(n^2)$
Tornado	$O(n \log \frac{1}{\epsilon})$	$O(n \log \frac{1}{\epsilon})$

Figure 3.5: Complexity comparison

Reed-Solomon code Reed-Solomon is an optimal erasure code. The key idea behind a Reed-Solomon code is that data encoded is first visualized as a polynomial. The code relies on a theorem from algebra that states that any m distinct points uniquely determine a polynomial of degree, at most, $m-1$. The sender builds a degree $m-1$ polynomial, over a finite field, that represents the k data points. For example if we want to send the m-ple

$$A = (a_0, a_1, \cdots, a_{m-1}) \text{ with } a_i \in \mathbb{F}$$

we can build the $m-1$ degree polynomial

$$P_{m-1}(z) = a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + \cdots + a_1 z + a_0$$

The polynomial is then encoded by its evaluation at n points (with $n > m$), and these n values are actually sent. During transmission, some of these values may become corrupted. However, as long as almost m blocks are received correctly, the receiver can deduce what the original polynomial was, and hence decode the original data. Two advantages of Reed-Solomon are storage efficiency and deterministic behavior while, the con is that encoding and decoding algorithms are slow (Table 3.5) compared to other near-optimal erasure codes we will show below.

LDPC codes LDPC codes are a large class of near optimal erasure codes provided for the first time by R. Gallager in 1960 that introduced the idea of using a bipartite graph to approach the erasure code problem. This type of graph contains two disjoint sets of nodes with edges between nodes of different sets, but no edges between nodes within the same set. The key idea is to divide a file in some blocks (called source blocks), to create a graph with a set of nodes corresponding to the source blocks and a set of nodes corresponding to the check blocks and to connect some of the source blocks to the check blocks. The encoding process consists of setting a check block equal to the exclusive or (XOR) of the source blocks to which it is connected. The classical approach is based on a regular graph in which all nodes of each type have the same number of edges. In this code, the number of edges is proportional to the number of nodes. Since the resulting edge count is small compared to the number of edges in a fully connected bipartite graph, these are called low density parity check (LDPC) codes.

In 1997, Luby et. al [LMS+97] proved that LDPC codes have improved properties if the number of edges connected to each node varies from node to node (so using not regular graphs). For such irregular codes, the challenge is to characterize the degree of irregularity that yields the best coding performance. This is accomplished in two stages. First, a probability distribution is specified for the degree of a check block. A distribution may also be specified for the degree of a source block. Then a procedure is developed for implementing an effective code based on the distribution. In contrast to Reed-Solomon codes, these codes provide probabilistic erasure correction through an iterative decoding algorithm and suboptimal performance on storage overhead; the advantage is that this class of codes provides a significant reduction in computational cost for encoding and decoding (see Table 3.5).

Tornado code Tornado is a LDPC code based on irregular bipartite graphs. This means that the degrees of source and check blocks are not fixed but are arranged following a fixed distribution.

Tornado codes work via XOR. The XOR of a number of binary variables is called their "parity" and this is often used in error detection and correction. Tornado codes use it for error correction. They use another checksum (like CRC-32 or MD5) for error detection.

The Tornado code algorithm starts with the sender breaking an input file or message into equal sized blocks of bytes. Let's call these blocks $A[1]$ through $A[N]$. The sender records the index of each block and computes a checksums for the block and its index. (These will be used to determine if a block has been damaged during transmission and therefore needs to be recovered.) The sender also calculates some parity blocks, $B[1]$ through $B[K]$. Each of these parity blocks holds the parity for a subset of the input blocks $A[1]$ through $A[N]$. The size and composition of these subsets is key to the speed and success of this algorithm. For each parity block, the sender records the indices of the input blocks and a checksum for the parity block and its input indices.

The sender now sends the input and parity blocks (with their indices and checksums) to the receiver. During this transmission, some of the blocks may be corrupted.

The receiver uses the checksums to identify bad blocks and discards them. The receiver is now left with a subset of the input blocks and some parity blocks. As long as the receiver has received N + C blocks (where C is some constant), it is highly probable that the receiver can recover the file. Now, each parity block is associated with a subset of input blocks and for most parity blocks there may be multiple input blocks missing from its subset. However, given the size of the random subsets, it is highly likely that there exists one parity block that is missing only one of its input blocks. Using the XOR operation described above, that missing input block can be recovered. Once it is recovered, a parity block that was previously missing two input blocks may now be missing just one and that one can now be recovered. This process continues - input blocks being recovered and more parity blocks being available to recover missing blocks — until the entire input file or message is recovered.

The true power of the algorithm stems from the uneven size of the subsets. On average, the sizes are low — making it very fast to create them and fast to recover the file. However, occasionally they are large - covering most of the input blocks - so that any missing block can be recovered.

Raptor code Raptor codes (RAPid TORnado) are one of the first known classes of fountain codes with linear time encoding and decoding. They were invented by Amin Shokrollahi in 2000/2001 and were first published in 2004 as an extended abstract [Sho06].

Raptor codes encode a given message consisting of a number of symbols, k, into a potentially limitless sequence of encoding symbols such that knowledge of any k or more encoding symbols allows the message to be recovered with some non-zero probability. The probability that the message can be recovered increases with the number of symbols received above k becoming very close to 1, once the number of received symbols is only very slightly larger than k. A symbol can be any size, from a single bit to hundreds or thousands of bytes.

Raptor codes may be systematic or non-systematic. In the systematic case, the symbols of the original message are included within the set of encoding symbols. An example of a systematic raptor code is the code defined by the 3rd Generation Partnership Project for use in mobile cellular wireless broadcast and multicast and also used by DVB-H standards for IP datacast to handheld devices (see external links). Online codes are an example of a non-systematic raptor code.

Raptor codes are formed by the concatenation of two codes.

A fixed rate erasure code, usually with a fairly high rate, is applied as a "pre-code" or "outer code". This pre-code may itself be a concatenation of multiple codes, for example in the code standardized by 3GPP a high density parity check code derived from the binary Gray sequence is concatenated with a simple regular low density parity check code. Another possibility would be a concatenation of a Hamming code with a low density parity check code.

The inner code takes the result of the pre-coding operation and generates a sequence of encoding symbols. The inner code is a form of LT code. Each encoding symbol is the XOR of a randomly chosen set of symbols from the pre-code output. The number of symbols which are XOR'ed together to form an output symbol is chosen randomly for each output symbol according to a specific probability distribution.

This distribution, as well as the mechanism for generating random numbers for sampling this distribution and for choosing the symbols to be XOR'ed, must be known to both sender and receiver. In one approach, each symbol is accompanied with an identifier which can be used as a seed to a pseudo-random number generator to generate this information, with the same process being followed by both sender and receiver.

In the case of non-systematic raptor codes, the source data to be encoded is used as the input to the pre-coding stage.

In the case of systematic raptor codes, the input to the pre-coding stage is obtained by first applying the inverse of the encoding operation that generates the first k output symbols to the source data. Thus, applying the normal encoding operation to the resulting symbols causes the original source symbols to be regenerated as the first k output symbols of the code. It is necessary to ensure that the random processes which generate the first k output symbols generate an operation which is invertible.

Erasure Code in PariPari

In PariPari we chose to adopt the LDPC approach because of its ease of implementation[1]. We chose the 1-level Systematic LDPC code.

We are designing and developing an Erasure Code that also allows the incremental modification of a split and distributed file. With "standard" erasure codes the file is split in blocks and spread among peers until a user wants to retrieve the file itself. In this case the blocks are gathered to reassemble the file. Rather than a simple "save and retrieve" feature we are designing a distributed filesystem with very basic functionalities. More specifically, we want our storage system to support:

store: split a file in blocks and spread it among the peers

retrieve: retrieve the file blocks and reconstruct the original file

delete: delete the blocks of a file (once distributed)

[1]Changing the algorithm once the framework is ready should be quite fast.

regeneration: check the availability of the blocks in the network and regenerate some blocks to avoid file destruction

incremental modification: add a log of modifications to the file blocks to embody changes (once the file is retrieved of regenerate)

restricted access: allow all operations only to the file owner. The owner can choose to let everybody read/write the file.

3.4 PariConnectivity

PariConnectivity provides connectivity to every other plug-in in PariPari. PariConnectivity can provide UDP, TCP (and HTTPS) sockets, limiting the resource consumption of individual plug-ins so as to avoid conflicts, and providing bandwidth and latency QoS guarantees. PariConnectivity also provides a number of (currently experimental) advanced features, including Anonimization (Section 3.4.2), NAT (Section 3.4.3), Multicast transmissions (Section 3.4.4) and Transmission Tunneling (Section 3.4.5).

3.4.1 Point-to-Point

PariConnectivity provides APIs to allow plug-ins to communicate with other peers and other host on Internet. These communications are regulated by a defined ruleset that affects the number of opened socket per plug-in and the available bandwith (per socket and per plug-in). Bandwidth limitation is implemented using the Token Bucket algorithm [Tan96, 402–404] and allows dynamic band variation (i.e., changes bandwidth limit also after socket creation).

To be more precise, PariConnectivity offers:

TCP Socket. This allows plug-ins to send and receive data using a TCP connection according to a defined ruleset.

UDP Socket. This allows plug-ins to send and receive data using a UDP datagram according to a defined ruleset.

HTTPS Socket. This allows plug-ins to send and receive data using HTTPS protocol. It is very useful to establish in a simple way an authentication session with well known server (e.g., Messenger 4.3.1).

3.4.2 Anonymity

The purpose of this package is to ensure the untraceability of any PariPari user. In particular we want to guarantee that any communication cannot be associated with the IP of the originating user nor with the IP of the receiver. Also, all communications are encrypted using AES[2], which gives strong security guarantees paired with excellent performance (a simple PC can easily support 50-100Mbit/s).

[2]Advanced Encryption Standard is an encryption standard adopted by the U.S. government. The standard comprises three block ciphers, AES-128, AES-192 and AES-256, adopted from a larger collection originally published as Rijndael [DDD+98].

Anonimizer chains

Intuitively to let the user A to hide its IP address while sending data, we can mix it up in a set of other peers. PariPari adopts a widely used technique known as Onion Routing [GRS99].

Onion routing is a technique for anonymous communication over a computer network. Messages are repeatedly encrypted and then sent through several network peers. Each peer removes a layer of encryption to uncover routing instructions, and sends the message to the next node where this is repeated. This prevents these intermediary nodes from knowing the origin, destination, and contents of the message. To be more precise the message is recursively encrypted, once for each peer that re-routes it (with the key of that peer).

In such way each peer can decrypt only its layer (in the right order) to know who is the next hop. Any peer, then, only knows about its predecessor and its successor but does not know anything about the sender nor the receiver.

We worked to adapt this well known paradigm to PariPari to preserve the anonymity guarantees.

The same scheme is symmetrically adaptable to the receiver of a communication. In fact, any user willing to keep herself anonymous, can prepare a chain of peers to forward any message for her. Any peer of the chain, except the first, has to be not contacted directly. But no peers knows if it is contacted by the original receiver or by a peer already in the chain. After this set up round any peer is ready to receive and forward the message along the chain, not knowing anything about the sender (or its chain) nor the receiver.

Then, if at least a peer in the chain downstream is honest and follows the protocol, it is impossible to learn the identity of the receiver. Similarly, a single honest peer in the chain upstream hides the identity of the sender.

3.4.3 NAT traversal

This package allows peers behind NAT o firewall to be reachable from the PariPari network.

Network Address Translation (NAT) is the process of modifying network address information in datagram packet headers while in transit across a traffic routing device for the purpose of remapping a given address space into another. Most often today, NAT is used in conjunction with network masquerading (or IP masquerading) which is a technique that hides an entire address space, usually consisting of private network addresses [RMK+96], behind a single IP address in another, often public address space. This mechanism is implemented in a routing device that uses stateful translation tables to map the “hidden” addresses into a single address and then rewrites the outgoing Internet Protocol (IP) packets on exit so that they appear to originate from the router. In the reverse communications path, responses are mapped back to the originating IP address using the rules (“state”) stored in the translation tables. The translation table rules established in this fashion are flushed after a short period without new traffic refreshing their state. As described, the method enables communication through the router only when the conversation originates in the masqueraded network, since this establishes the translation tables.

To allow a peer hidden by a NAT to be reached we implement a NAT traversal mechanism. NAT traversal is a general term for techniques that establish and maintain TCP/IP network and/or UDP connections traversing NAT gateways. Next we describe some ways to implement a NAT traversal mechanism - and the minimum requirements to do so.

IP Discover Each node willing to publish its resource(s) must know its own IP address. If the peer is behind a NAT it cannot directly know the public IP address that appears to have. In order to discover it we have implemented a simple protocol to allow peers to ask their neighbors its public IP. For performance reasons we use UDP to implement this service.

STUN The STUN (Session Traversal Utilities for NAT) protocol allows applications operating through a NAT to discover the presence of a network address translator and to obtain the mapped (public) IP address (NAT address) and port number that the NAT has allocated for the application’s User Datagram Protocol (UDP) connections to remote hosts. The protocol requires assistance from a third-party network server (STUN server) located on the opposing (public) side of the NAT, usually the public Internet.

A peer can learn its reachability by pinging a STUN server. The server, running on a PariPari node, answers with two UDP datagrams: one originating from the same destination port of the ping and one from a different port. The peer, depending on the number of received answers, understands its reachability state.

No answer: the ping does not arrive or the incoming port is closed. Hence, the peer is completely frozen out (at least for the port tested).

One answer originating form the ping destination port: the peer is behind a simple NAT that correctly maps the incoming request matching the outgoing traffic.

Two answers: the peer is behind a NAT performing a virtual Server policy[3].

In the second case to allow further incoming traffic the STUN server has to continuously send datagrams to the target port. So, the peer, can safely its public IP and the target port.

UDP Hole Punching UDP hole punching is a method for establishing bidirectional UDP connections between Internet hosts in private networks using NAT. It is a simple algorithm that uses information gathered with STUN.

Let A and B be the two peers, each behind a NAT; N1 and N2 are the two NAT devices; S is a public STUN server with a well-known globally reachable IP address.

1. A and B each begin a UDP conversation with S; the NAT devices N1 and N2 create UDP translation states and assign temporary external port numbers.

2. S relays these port numbers back to A and B.

3. A and B contact each others' NAT devices directly on the translated ports; the NAT devices use the previously created translation states and send the packets to A and B.

TURN Traversal Using Relay NAT (TURN) is a protocol that allows for an element behind a NAT or firewall to receive incoming data over TCP or UDP connections. It uses a simple relay to route traffic from and to two peers behind NATs. Every peer logs in to a well-known globally reachable TURN server and then they exchange data sending to and receiving from the server. Although simple and effective, this method exhibits poor performance: all the traffic has to pass through the TURN server — possibly leading to congestion.

3.4.4 Multicast

Multicast is not yet implemented. The main challenge in the development of this module is to make its interface to other plug-ins as application-independent as possible.

Depending on the type of communication we defined:

Simple Conference. The classic audio/video conference with few participants (e.g., [sky])

[3]In a virtual server, every traffic incoming on the tested port is forwarded to the peer.

Broadcast. A communication characterized by a sender and many receivers (e.g., a streaming application)

Multicast. A communication characterized by many participants who want to send information to — and receive it from — other participants.

Unfortunately, although the IPv4 protocol supports multicast, it is often impossible to use it in the real world due to the limitations introduced by the use of proxies, NATs, firewalls.

Simple Conference We are planning to implement this feature using a *Skype* like approach. The initiator of the conference acts as a server and routes the traffic from and to the other participants. This server should mix the audio/video streams in a smart way to avoid the echo problem.

Broadcast After a literature survey we decided to adopt the scheme shown in [WXL07]. This approach provides a tree-like structure to deploy the streams to each requesting peer. The tree inner nodes, that are PariPari peers, route the streams towards the leaves which are the final receivers. It is very important to note that using a simple tree structure can lead to large failures when an inner node fails. To avoid this kind of troubles the tree is fortified with a mesh. The tree depth should be adapted to the number of requesting nodes, to the number of inner nodes

and to the latency of the peers in the network.

Multicast We need a scalable application-layer multicast protocol designed keeping in mind low-bandwidth peers. The scheme in [BBK02] is based upon a hierarchical clustering of the peers and can support a number of different data delivery trees with desirable properties. To be more precise, peers are arranged in clusters, each one with a leader. Every leader is connected to other leaders and most traffic is routed in this backbone. When streams arrive to the cluster leader they are forwarded to every peer through cluster.

It is particularly important to manage the streams mix at each level to avoid unnecessary bandwidth consumption, the echo problem and, of course, to let the participants speak and listen freely.

3.4.5 Tunneling

Due to the PariPari structure it is easy to imagine that many “parallel” sockets can exist between two peers. Any pair of peers can instantiate communications for every plug-in. This would not be a problem if, in the real world, each host was linked directly to the Internet. However many peers are behind proxies, NATs, firewalls. Hence they are reachable only if the system administrators can guarantee open ports to PariPari.

We are implementing a mechanism to mux (and demux) all the communications between two PariPari peers in a single socket. So only a single port will suffice and it will be very simple to exploit our NAT traversal facility.

Chapter 4

Application Layer Plug-ins

4.1 PariSync

PariSync is the plug-in needed to keep every peer in the network synchronized. This synchronization is absolutely necessary to allow transactions and to provide the credit module with a way to keep track of elapsed time. This work was published in [BBMP09].

Clock Synchronization is a problem that has been extensively studied over many years, both in the distributed systems community, and (for its great practical importance) by the networking community. In a nutshell, it consists in having all the nodes in a distributed system agree on a common virtual clock, ideally "sufficiently close" to real time (perhaps with a few "hints" from accurate, external sources) despite variable communication delays and misbehaving nodes.

The distributed systems community typically casts the problem of clock synchronization as follows. Assume a network of n nodes $v_1, \dots, v_n$, all with communication delays, all with clocks, initially perfectly synchronized but with potentially different speeds. Each node v_i witnesses a sequence of events $e_i^1, e_i^2, \dots$. The goal is to develop a communication protocol that allows nodes to reach a consensus on a virtual time $t(e_i^j)$ to be assigned to each event e_i^j so that no event witnessed by a node is timestamped "out of order". Ideally $t(e_i^j)$ should also lie within a multiplicative factor (bounded away from both 0 and ∞) of the real time at which e_i^j occurs, and the protocol should tolerate the largest possible number of "Byzantine" nodes - nodes that maliciously conspire, with perfect coordination and information, to make the protocol fail (a conservative model both of nodes controlled by actively malicious actors and of accidentally misconfigured nodes). Roughly speaking, protocols exist [LS86, DHS84] that tolerate a very large fraction of Byzantine nodes (from $\approx \frac{1}{3}$, to $\approx \frac{1}{2}$ if a shared digital signature system is available to all nodes). On the other hand, these protocols tend to require from each node considerable space and considerable bandwidth (both linear or polynomial in the size of the network), and so they are not really capable of scaling to networks with millions of nodes. The guarantees they provide on the timestamp assigned to each event are also relatively lax: many P2P applications require virtual time to remain within at most a few seconds from real time, and often less than a second.

The networking community has instead focused on less pessimistic models in terms of malfunctioning nodes and communication errors/delays ([MKKB89, SHLnLZ07, BOW05, LLW08]), and assumed the presence of a few trusted "correct" clocks (e.g., the tightly synchronized atomic clock servers of NTP) to provide lower communication overheads and tighter clock synchronization — indeed, some work aims at sub-millisecond and even, for LANs, sub-microsecond synchronization [Mac08]. Unfortunately, these algorithms tend to rely on tree-like structures that are less robust, since a single failure close to the root of the tree (particularly a "malicious" one) can cause disastrous effects on a large fraction of the network.

PariSync is a system for DHT based P2P networks that achieves the aforementioned goals. In a nutshell, PariSync is formed by two modules: a *topology* module that chooses, for each node, a small subset of neighbors with which to exchange timing information, and a *estimation* module, that chooses how to process that information to estimate the current time and speed of the global virtual clock and/or upper and lower bounds on them.

Two points are worth noting, in particular in terms of comparison with the very successful NTP (and similar protocols such as SNTP). First, although NTP access is often less restricted by network administrator policies than P2P traffic, by definition the only P2P systems on which one needs clock synchronization are those that do allow P2P traffic (it is pointless synching with a P2P network if one cannot communicate with it). Thus, *a system that piggybacks on existing P2P protocols will always be available to synchronize those protocols, whereas NTP may not be* - in fact, every large P2P network is virtually guaranteed to have a sizable fraction of all its nodes unable to employ NTP, whether because of misconfiguration, traffic restrictions, or lack of administrator privileges to run or configure the NTP daemon. Second, NTP and other similar protocols relying on a tree-like structure do allow peering between nodes at the same level of the tree to achieve greater robustness. However, none of these protocols defines the peer link structure, and showing how to set-up such a link structure in a way that allows scalable, effective synchronization is, indeed, one of the main contributions of this paper.

4.1.1 Algorithm

This subsection provides a description of the PariSync, motivating its design choices. The basic goal of PariSync is to allow PariPari to agree on a common virtual clock that should run as closely aligned to real time as possible. Each node maintains, at all times, its own clock, but also attempts to maintain (with good approximation) its offset and drift w.r.t. the global virtual clock, and/or upper bounds on this offset and drift.

The two fundamental difficulties in estimating offset and drift lie in "noisy" communication delays between nodes (as in classic clock synchronization) and in the fact that, since we want the algorithm to scale to networks of millions of nodes, each node should be allowed to communicate directly with only a small subset of the whole network, ideally piggybacking on the pre-existing DHT link structure. Thus, the main problem can be decomposed into two, almost orthogonal subproblems, that are, in fact, addressed by two independent software modules in PariSync:

1. A *topology* subproblem: have each node choose the appropriate neighbors with which to communicate.

2. An *estimation* subproblem: have each node estimate the current state of the global, virtual clock based on its own clock and the information received from its neighbors.

The latter aspect has received considerable attention in the literature; on the contrary, almost nothing is known about the former, and all solutions in the literature essentially either consider (non-scalable) fully connected graphs, or (fragile) trees or tree-like graphs. This paper focuses on the topology subproblem, adopting only a very basic solution to the estimation one — though more sophisticated strategies may be easily "swapped in" into the appropriate estimation module.

Two important estimation issues should be considered even when focusing on topology, however. First, ideally a sizable fraction of all nodes should have access to NTP or other means of accurately estimating their own time. It would be desirable for the global clock to be "anchored" to these accurate nodes. Second, a small but non-trivial fraction of all nodes should be expected to provide completely inaccurate results (e.g., due to misconfiguration or to an active attempt to disrupt the global clock). In the absence of some means to certify which nodes have access to accurate, external sources (and note that even if there is access to an accurate source, the intervening communication process may introduce large, undetectable errors) the two goals of anchoring the global clock to accurate external sources and of making it resilient to coordinated faults are conflicting: a minority of nodes whose clocks are perfectly consistent with each other, but widely divergent from those of the rest of the network, could either be the group of peers with access to accurate, external information, or simply a group of malicious peers trying to disrupt the network.

Since robustness to (coordinated) faults is probably a higher design priority than external clock anchoring for P2P networks — at least as long as the global virtual clock behaves "reasonably" — any robust system will have each node give little or no weight to the information provided by a small but non-vanishing fraction of "outliers". It is not difficult to see that this makes the natural scheme of having each node estimate the global clock on the basis of its DHT neighbors alone ineffective. Assume that the address space of a network of n nodes is partitioned into k segments, each holding approximately n/k nodes. The (sets of nodes in the) different segments will then never synchronize if they start with different offsets and drifts and if the estimation rejects a fraction at least $\frac{\log(k)}{\log(n)}$ of all outliers, since only a fraction $\frac{\log(k)}{\log(n)}$ of all links of a node falls outside its segment.

PariSync instead has each node increase its pool of long distance links by simply asking each neighbor (according to the DHT structure) to reply not only with its own time information, but also with the time information about a long distance neighbor of that node (i.e., a neighbor residing in the other half of the network). While this does introduce some extra noise due to the extra hop, it provides (as we shall see in the next subsection) a particularly well-behaved topology that offers rapid convergence of the consensus on the global clock; and it does so *essentially for free*, since each node in a DHT based peer to peer network will periodically contact its DHT neighbors anyway (to check liveness, to forward queries etc.), and adding time information about two nodes in such a communication exchange has a negligible impact on the performance of the system.

Once a node has retrieved from each of its neighbors the local time (and the local time of their long distance contact), the estimation module of PariSync evaluates its offset and drift from the virtual clock. Note that there exist some sophisticated schemes to do this, and to bound the resulting error (e.g. [Ber00, LLW08]). The focus of this paper, however, is on the topology module, and the estimation scheme provided below, while extremely basic, seems to work reasonably well — though certainly more sophisticated schemes could be introduced without having to modify the topology module.

Each node v at time t maintains a vector of T assessments of the global clock speed $s_v^t(t), s_v^{t-1}(t), \ldots, s_v^{t-(T-1)}(t)$ where T is ideally at least logarithmic in the size of the network. Informally, $s_v^\tau(t)$ measures the estimate, at time t, of the global clock speed at time τ. $s_v^t(t)$ is always set equal to v's own speed. $s_v^\tau(t+1)$, with $\tau \leq t$, is set equal to the average of all $s_u^\tau(t)$ such that u is a node providing time information to v that v had not rejected at time τ (note that v can easily estimate the speed of a neighbor relative to its own by comparing that neighbor's local elapsed time over an interval, and its own elapsed time during that interval, making the estimate slightly more accurate by taking into account communication delays estimated as half of the roundtrip time). Thus, the larger the gap between t and τ, the less up-to-date the information, but also the larger the set of nodes that contribute to it (note that, according to our experiments, clock speeds remain fairly stable over periods of at least a few hundred minutes). Instead of estimating the global clock speed, one could also — with the same mechanism — compute a lower and upper bound on it by simply taking the minimum and maximum instead of the average at each step.

Exactly in the same fashion, a node can obtain an assessment of the global clock *time* at (its own) timepoints $t, t-1, \ldots, t-T+1$, $g_v^t(t), g_v^{t-1}(t), \ldots, g_v^{t-(T-1)}(t)$ by averaging the times assessed by its neighbors. Again, an estimate relative to a recent timepoint may be not very accurate, because only few nodes have contributed to it. However, if (as it appears to be the case, see the next Section) the clock speeds of individual machines appear to be relatively stable over a period equal to the time between consecutive measurements (a few minutes) times the base 2 logarithm T of the size of the network (at most 20 to 30), a better evaluation of the current time of the global clock can be achieved by taking the estimate of that time at (local) time $t-T$, and correcting it by the elapsed local time adjusted by the (estimated) speed ratio over the intervening interval.

This section provides some preliminary experimental evaluation of PariSync. Subsection 4.1.2 shows how the standard procedure of estimating communication delay as half of the roundtrip time between nodes still holds even over multihop paths with large delays, such as those typically between peers in large P2P networks. Subsection 4.1.3 evaluates PariSync on a smallish ($100-200$ node) network of nodes dispersed throughout Europe (part of the AEOLUS testbed [aeo]). The resulting latencies, combined with clock drift data from several PCs, allow us to run with realistic parameters an extensive simulation of PariSync over a network of a million (simulated) nodes, detailed in Subsection 4.1.4.

4.1.2 Network Latency

NTP clock adjustments are based on the assumption that UDP packets travel from host A to host B in exactly one half of the Round Trip Time between A and B. Defining t_{AB} the time it takes for a UDP packet to travel from host A to host B, this means $t_{AB} = t_{BA}$. In a large P2P network the geographic distance between communicating hosts can be very changeable. Neighborhood in a DHT like Kademlia's is defined on an ID-oriented metric, and has nothing to do with geographic proximity. For this reason our system must take into consideration a large variety of links, ranging from LAN to intercontinental. IP's connectionless design makes it seem very likely to have packets following one route when going from A to B and a different one when going back to A. The longer the link, the higher the probability of having different hops in the two paths. We developed a plug-in for PariPari that

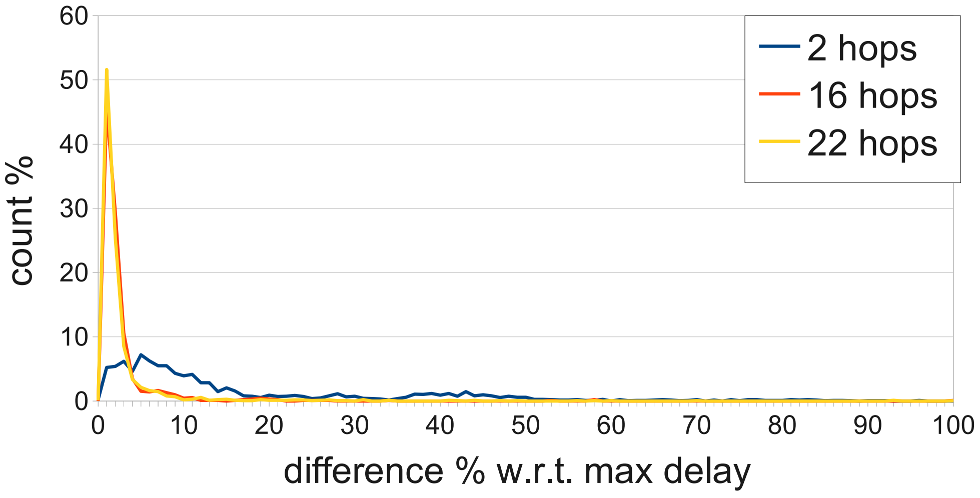


Figure 4.1: Differences in clockwise and counter-clockwise pings. Peak values for the 3 groups are: 7% count at 5% difference, 46% at 1%, 52% at 1%.

performs several *UDP pings* between hosts at regular intervals and logs results to perform statistical analysis. A *UDP ping* is performed by sending a small (a 64-bit signature) packet to a host using UDP and registering how much time passes before receiving a reply UDP packet from that host. In order to collect a significant amount of information, we ran these tests using AEOLUS [aeo] testbed. This testbed [tes] is a network connecting 16 European Universities, each providing a number of heterogeneous hosts, all of which run 1 to 10 (virtual) instances of JXTA. We split hosts into 3 groups of 3 hosts each, so that each host could compute $|t_{ABCA} - t_{ACBA}|$. Group 1 simulates A and B being in the same LAN, Group 2 simulates A and B being 14 to 16 hops apart (a continental link), Group 3 simulates A and B being 20 to 24 hops apart (an intercontinental link). The number of hops between hosts has been computed by running ICMP *traceroute* several times for each link. Hosts are placed in an overlay network in a token-ring fashion. A map of the ring instructing hosts about their neighbors is shared among hosts of the same group. A *UDP ping* begins with the first node appearing in the map starting a timer and sending a packet containing its signature to one of its neighbors. Every host that receives a packet from one of its neighbors forwards the message to the other neighbor, unless the packet has its own signature in it. If it does, the first ping is complete and the result is logged. A second ping is then performed by the first node, this time sending the packet to its other neighbor. The operation is repeated ten times before passing a token to the node's clockwise-next neighbor. A *round* is then complete. When a node receives the token, it waits for 10 minutes before starting its *UDP pings* in order to statistically separate rounds. After 500 rounds we analyzed the values of $|t_{ABCA} - t_{ACBA}|$ logged by every host. Figure 4.1 shows the distribution of these values for each group. In a pathological case, when hosts are in the same LAN, a certain amount of noise appears. This, though, seems more addressable to Java operations on packets than on changes on delivery time on the underlying network. Delays are in the order of half a millisecond, so a difference of 50% in the two pings means a difference of a quarter of millisecond, very close to our timer precision limit. We'll further investigate on this aspect relying on more precise Java timers, but from the figure it is evident that, for common links, the assumption made by NTP is valid and can be used also in very large networks.

4.1.3 PariSync for Real

Initially we ran tests on physical hosts using the AEOLUS testbed. We started each of the 103 hosts with different offsets and drifts choosing them according to a Gaussian distribution. We let our algorithm run with: 10 minutes as the interval between distributed synchronizations, 8 minutes as minimum interval between NTP synchronizations, α taken as 6 for distributed and as 40 for NTP synchronization. Each node polled 7 (log(103)) hosts per distributed synchronization. As we can see from figure 4.2 and 4.3 the consensus is achieved after just 11 synchronizations (110 minutes). After 450 minutes we turned off the NTP server and the stability of the system remained good, with an average offset of nearly 35 ms after 24 hours.

4.1.4 Simulations

Since PariSync is primarily designed to provide large P2P networks with synchronization we tested it on a simulated environment. We wrote a multi-threaded simulator and modeled it using data from real drifts.

We retrieved data from 6 hosts. `ntpd` was disabled on these hosts and hosts were rebooted to drop any `ntpd` drift correction. The drifts appeared different between hosts but reasonably constant (Fig. 4.4). We then modeled drift in the simulator assuming a gaussian distribution with the measured average and variance.

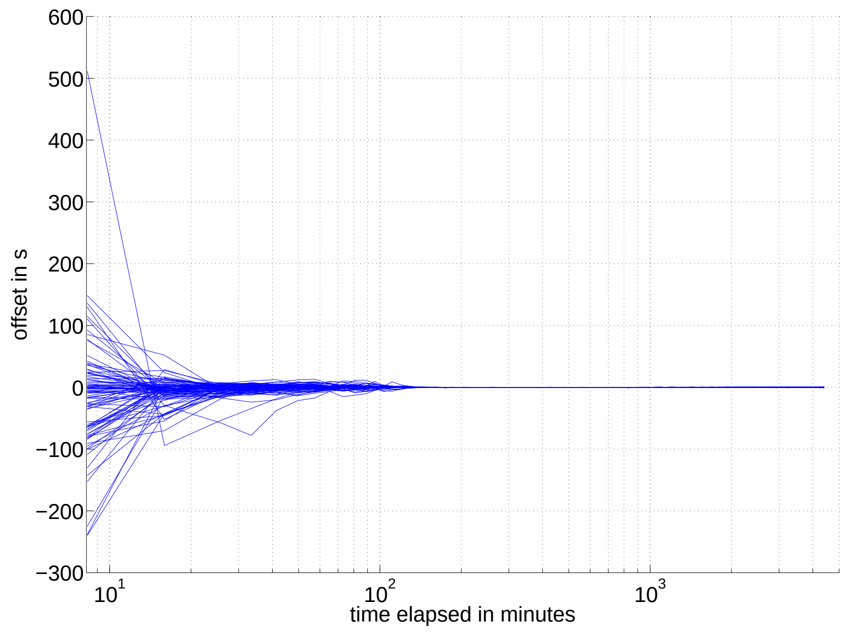


Figure 4.2: Offset converging on AEOLUS testbed.

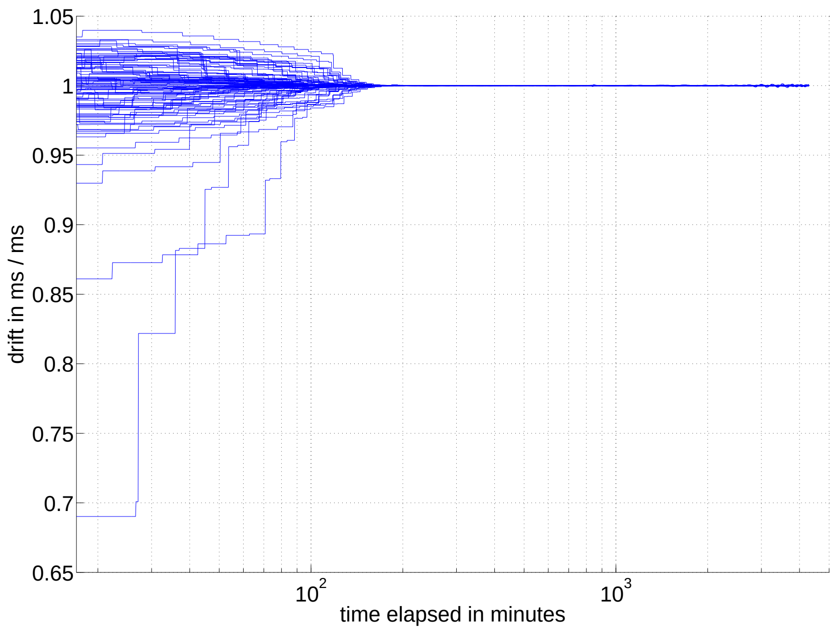


Figure 4.3: Drift converging on AEOLUS testbed.

As already stated, communication delay can be safely estimated from the measured Round Trip Time (possibly filtering and averaging to get more precise measures). As said above PariSync performs such measurements and corrects data received by other nodes with these estimated delays.

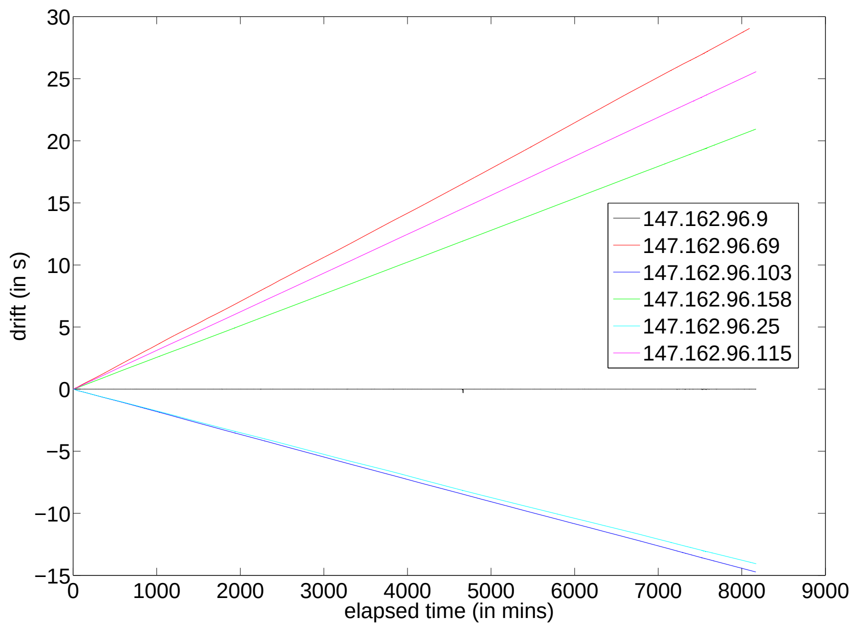


Figure 4.4: Drifts on 6 hosts: they range from -14μs/s to 52μs/s.

First of all we tried to discover, using a relatively small amount of nodes, the best parameters to let the network converge steadily and quickly. We run several trials changing the percentage of nodes to be accepted after the filtering process (i.e., `accepted nodes`) and the weight to be given to their average (i.e., `ppsweight`). In order to compare simulation performances, we consider the network to be *stable* as the distribution of the time values on all of its nodes reaches a standard deviation equal to or minor than 10^{-5}. Let t_{conv} be the time (in seconds) it takes to the network to become *stable*. Every simulation has been run for a total of 10000 seconds. The height of each bar in Fig. 4.5 represents $10000 - t_{conv}$, that is the amount of seconds left before the end of the simulation at the time the network became *stable* (hence higher is better). The figure clearly shows that the choice of parameters strongly affects network behavior. Filtering nodes with "strange" values proves to be very effective and increasing the average weight speeds up the convergence.

Moreover, PariSync exhibits a good behavior even in the presence of nodes with very different drifts. We ran several simulations setting the same incorrect clock value to some groups of nodes in order to stress the system. The higher the number of "bad" nodes the slower the system achieved the consensus.

We then tested PariSync's resilience to situations where nodes in the same segment of the ID space sport the same clock behavior, but this behavior varies between segments. We tested the simplest case, with all nodes with ID prefix 0 in one segment

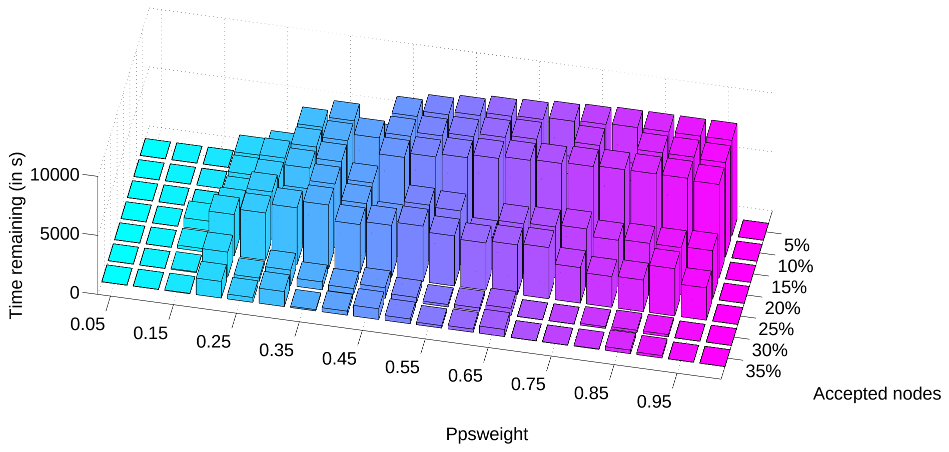


Figure 4.5: Seconds remaining to the end of the simulation at the time the network became stable for different values of `accepted nodes` - `ppsweight`.

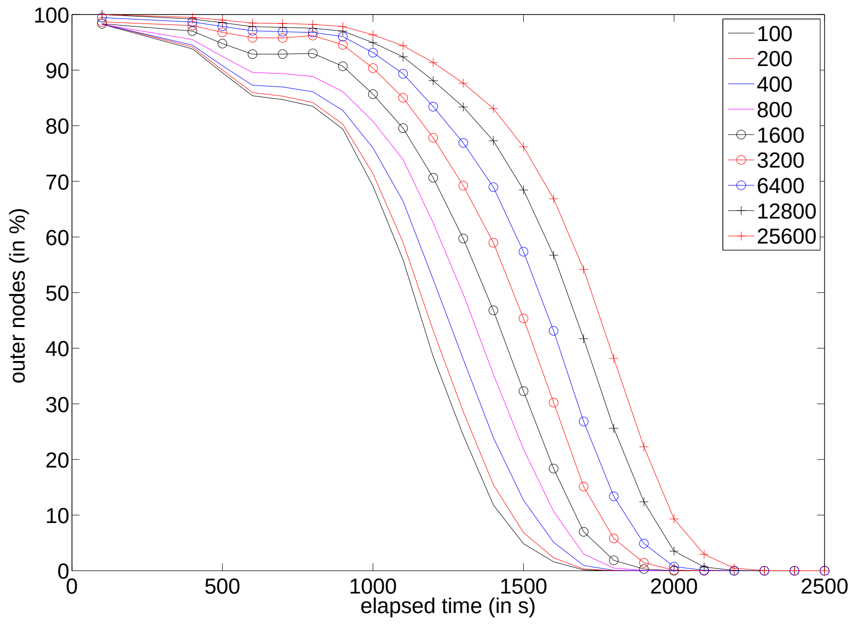


Figure 4.6: Outer nodes of 100K nodes in function of nodes with 0 ms/ms drift (`accepted nodes: 5%`, `ppsweight: 0.95`).

and all nodes with ID prefix 1 in another. Recall that, in this situation, simply averaging the information from DHT neighbors without including that from their long range contacts would result in slow or no convergence (see previous Section4.1.1). Fig. 4.7 shows the difference between standard PariSync, and PariSync without long range contact polling.

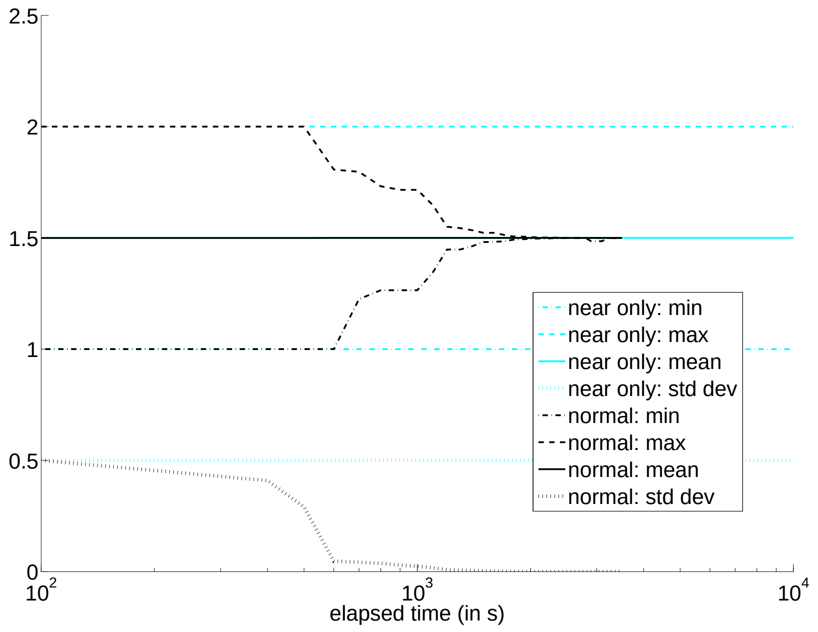


Figure 4.7: 1M nodes in a "split" network (`accepted nodes: 5%`, `ppsweight: 0.95`) with standard polling ("normal") vs. polling without long range contacts ("near").

Finally, we ran two simulations to test PariSync with and without NTP connection enabled. PariSync is slightly slower with NTP enabled but stabilizes on "real" NTP time. Fig. 4.9 shows mean and standard deviation evolution.

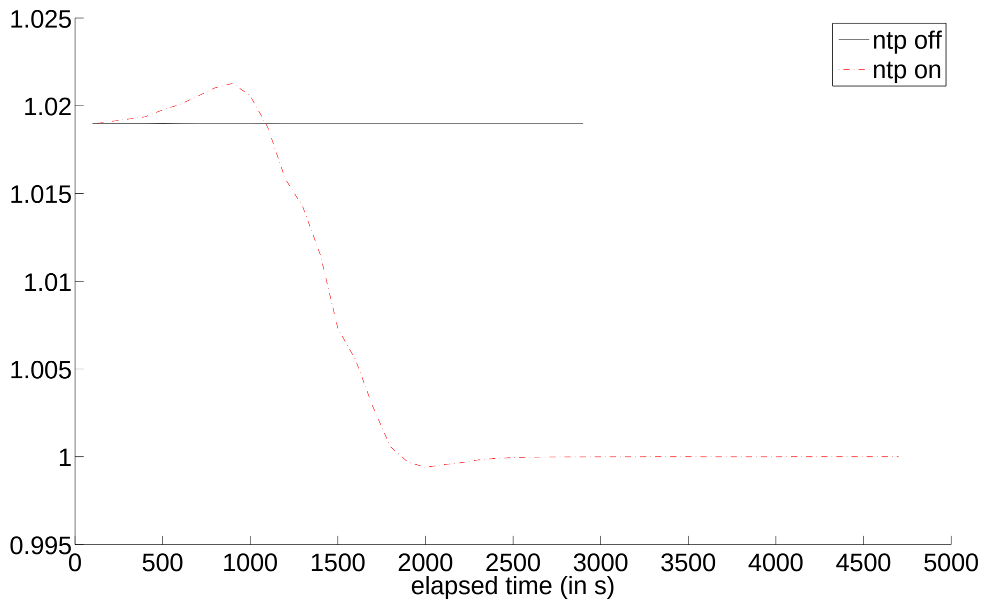


Figure 4.8: 1M nodes with and without NTP: mean.

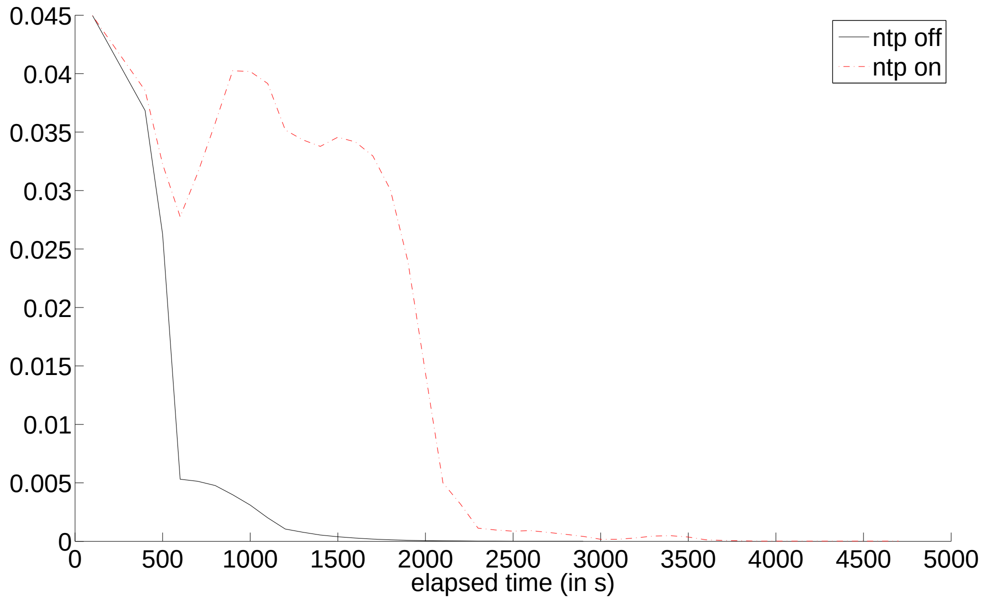


Figure 4.9: 1M nodes with and without NTP: standard deviation.

4.2 DiESeL

Distributed Extensive Server Layer (DiESeL) is a package to distribute a server onto the PariPari network. To be more precise, a normal server can easily become a distributed server using DiESeL. The basic idea is to create a generic interface that allows any plug-in to use the functionalities of the package without struggling to manage distribution and node communications. In particular, a server implementing the DiESeL interfaces, will be automatically distributed on a certain amount of capable peers and every content managed by the server will be accessible on each peer forming the distributed server. Hence, any client can connect to any peer forming the server while DiESeL keeps the contents consistency.

To achieve this objective, DiESeL provides three features:

- Outage detection
- Server Management
- Connection Redirection

4.2.1 Outage detection

In a server hosted on different nodes it is crucial to know at any moment whether a node fails (disconnects or crashes). To deal with this problem it is common practice to steadily flood the network with a large number of ping packets in order to quickly detect peers that are failing. The simplest method is to have each node ping periodically its neighbors. This method generates a lot of (useless) traffic, even though it yields an early outage detection.

In fact, in the usual keep-alive mechanism each connection is managed independently of all other connections in the network. For example, two nodes X and Y, both connected to a third node Z, do not share any information regarding their connection status. Since the same management task is performed twice, one can the keep-alive load between X and Y to achieve the common goal: outage detection of Z. Using the solution called Cooperative Keep Alive (CKA) presented in [DHS07] we have X and Y cooperate to decrease their keep-alive rate. Cooperation means that when X detects the outage of Z it informs Y and vice versa. Note node X and node Y do not need to maintain a permanent connection between them. If X detects an outage it does its best to inform Y. Even if Y does not receive the message from X it still has the chance to detect the outage of Z, although not so quickly. The role of node Z in this scheme is to inform X and Y that they are both connected to Z, and

	SKA	CKA
Storage	$O(d)$	$O(d^2)$
Traffic	$O(dN)$	$O(N)$
Delay	$O(k)$	$O(k)$ w.h.p.

Figure 4.10: CKA vs SKA comparison.

to determine the new keep-alive intervals for X and Y so that Z continues to receive keep-alive messages at a certain rate.

The effect of the cooperative approach described above is that it reduces the network traffic overhead by introducing additional overhead at the end nodes in terms of computing power and memory usage. In table 4.10 we can see a comparison between CKA and SKA. CKA guarantees a (maximum) desired delay (k seconds) generating an amount of traffic that depends on the numbers of peers (N) at the price of more storage used; CKA keeps information on d^2 peers where d is the peer degree.

4.2.2 Server Management

After solving the peer monitoring problem, the next challenge was dealing with the construction (and repair) of the distributed server, and with intra-server peer communication. To keep a random graph of n nodes connected with high probability we need at least $\log(n)$ edges for each node. We build the network of peers hosting the server keeping in mind these constrains. We can choose n as a function of the load of the distributed server and we can keep this n constant thanks to the CKA. In fact, whenever a disconnection is detected, DiESeL searches for new capable peers to join the managed network.

Moreover DiESeL keeps all peers updated with all the information received by each other peer. So, each peer of the server maintains the same status. This goal is achieved by broadcasting all data among peers and by replicating the contents (e.g., data files). In future releases we will use a smarter system based on Distributed Storage (see 3.3.2) and Multicast (see 3.4.4): multicast will warn the peers hosting the distributed server while all the contents will be read and written directly on the network using Distributed Storage.

4.2.3 Connection Redirection

In such a distributed environment we have to deal with churn. Also, peers hosting a distributed server can fail or disconnect even if clients are using their resources.

[Szy02] suggests three methods to solve this problem:

- Switching
- TCP Handoff
- DNS-based

The first two options do not fit in our environment. To be successful they need a gateway to route client requests to one of the peers of the distributed server. Nevertheless a P2P network as PariPari does not have static peers: every peer can join or leave the network at any moment. So we adopted the DNS-Based scheme. DiESeL sends to the DNS server the IP addresses of the best peers forming the distributed server. Each client needing to connect to the server can reach a working peer by asking the DNS server. The IP addresses list is continuously refreshed to keep the load of the peers balanced providing better performance for the clients.

The major drawback of this solution becomes apparent when the client keeps a stateful connection with a peer of the distributed server. In that case, whenever the peer disappears, the connection is reset and the client has to start again a new connection with another peer (readily served by the DNS). In the lucky case in which the client is a PariPari node and the peer disappearing is leaving "cleanly" the situation is manageable. DiESeL warns the connected peer (client) about the imminent disconnection also providing a new peer address to contact. This way the user should not notice the handover.

4.3 PariMessaging

One of the most popular kind of application among Internet users is the Instant Messaging (IM). IM is a form of real-time communication between two or more people based on typed text. In certain cases IM involves additional features such as the possibility to see the other parties (by webcam) or to chat directly for free. We decided to endow PariPari with IM and VoIP capabilities. Currently we are developing a framework to embed all possible Messaging Protocols in order to let third party users to add support for additional protocols to PariPari. To test our framework, and of course, to grant the users with the possibility to use one of the most popular protocols, we started implementing MSN.

At the same time we are developing an IRC client and server to test and stress the possibility of running a distributed server using DiESeL (see 4.2).

4.3.1 Messenger

The Messenger protocol and client are developed by Microsoft^{TM}so there is no public domain documentation. Our work is based on reverse engineering. We used many useful, but often incomplete, informations found on the World Wide Web to complement our results. Currently we support Version 15 of the Protocol.

4.3.2 IRC

Internet Relay Chat (IRC) is a form of real-time Internet text messaging (chat) or synchronous conferencing. It is mainly designed for group communication in discussion forums, called channels, but it also allows one-to-one communication via private message as well as chat and data transfers via Direct Client-to-Client. IRC is a popular protocol. Indeed, as of May 2009, the top 100 IRC networks served more than half a million users at a time, with hundreds of thousands of channels, operating on a total of roughly 1,500 servers worldwide. IRC operates in a standard client-server fashion.

IRC client Our IRC client currently supports all the basic features of the most popular clients. In particular it provides DCC and CTCP.

CTCP: Client-To-Client Protocol is a special type of communication between (IRC) clients. It extends the original IRC protocol by allowing users to query other clients for specific information.

DCC: Direct Client-to-Client is an IRC-related sub-protocol enabling peers to interconnect using an IRC server for handshaking in order to exchange files or perform non-relayed chats. Once established, a typical DCC session runs independently from the IRC server. Usually a DCC session is started using CTCP.

IRC server As stated above, our server is DiESeL-compliant. Hence, each user can run it locally, allowing other clients to connect to their server instance, or can enable DiESeL to join an existent IRC DiESeL distributed server or to create a new one. Actually one of the most serious problems experienced by the standard IRC server is the "netsplit". Servers, in fact, are linked with each other, in a tree-like fashion avoiding any loop (to eliminate the possibility of duplicated messages). Clearly whenever a server fails the network breaks into two or more parts and users from one part can not chat with users connected to other parts. In such a scenario many other problems can arise since servers will try to reconnect the split parts in a few minutes. For example the network has to manage carefully users with the same nick on two different parts when the parts will be joined again.

Our DiESeL-based IRC server implementation completely prevents netsplit, and all ensuing problems.

4.3.3 VoiP

Voice over Internet Protocol (VoIP) is a general term for a family of transmission technologies for delivering voice communications over IP networks such as the Internet. In PariPari we are working to implement a plug-in to allow users to speak with each other. Currently the plug-in uses JSpeex [jsp](a Java implementation of Speex [spe]) to encode and decode speech, and RTP to transmit data.

The plug-in handles small conferences using its own code that will be replaced by Multicast (see 3.4.4) when ready. Moreover we are working to enable video conferencing as well.

4.4 File Sharing

File sharing was the first purpose of P2P networks. Many networks have been created to share files among users. We can cite Napster, Gnutella, eMule, Bittorrent. PariPari already supports filesharing over two popular types of protocols We have written one eMule-compliant plug-in and one Bittorrent-compliant plug-in. In the next two sections we offer a description of the two plug-ins giving a necessary brief overview of the original protocols. The last section describes future work.

4.4.1 Mulo

mulo is the PariPari plug-in that operates on the eDonkey network.

Protocol

Unfortunately the protocol is not well defined. The basis is the protocol developed for the eDonkey network but in recent years clients using this network started to develop their own, not documented, dialects. eMule is the most widespread client and it adds many new features to the original protocol.

The eDonkey network is a decentralized, mostly server-based, peer-to-peer file sharing network best suited to share large files among users, and to provide long term availability of said files. Like most file sharing networks, it is decentralized, as there is no central hub for the network; also, files are not stored on a central server but are exchanged directly between users. The stability of the network depends on the availability of a number of servers that could be easily attacked. To overcome this problem the eMule Project also developed a Kademlia network called Kad. In addition, eMule includes a pure P2P client source-exchange capability, allowing a client to continue downloading (and uploading) files with a high number of sources for days, even after complete disconnection from the Kad or eD2k servers that handled the original requests. This source-exchange capability is designed to reduce load on servers.

Files on the eDonkey network are uniquely identified using an MD4 root hash of an MD4 hash list of the file. This treats files with identical content but different names as the same file, and files with different contents but the same name as different files. Files are divided in full chunks of 9500 KiB plus a remainder chunk, and a separate 128-bit MD4 checksum is computed for each of them. This way, transmission errors can be detected and corrupt only one chunk at the time instead of the whole file. Furthermore, valid downloaded chunks are available for sharing before the rest of the file is downloaded, speeding up the distribution of large files throughout the network. A file's identification checksum is computed by concatenating the chunks' MD4 checksums in order and hashing the result. In cryptographic terms, the list of MD4 checksums is a hash list, and the file identification checksum is the root hash, also called top hash or master hash. Files are searched directly on the network. To be more precise there are three possible types of search.

Local search the client asks for a file name to the server to which it is connected.

Global search the client asks for a file name to all known servers.

Kad search the client performs the search exploiting the Kad infrastructure.

Often, clients are hidden behind firewalls or NAT so they cannot accept incoming connections and thus share files. Such clients are denoted by a low ID, while clients accepting incoming connections are denoted by a high ID. The callback mechanism is designed to overcome this problem. The mechanism is simple: in case two clients A and B are connected to the same eMule Server and A requests a file that is located on B but B has a low ID, A can send the server a callback request requesting the server to ask B to call it back. The server, which already has an open TCP connection to B, sends B a callback request message, providing it with the IP and port of A. B can then connect to A and send it the file without further overhead on the server. Obviously, only a high ID client can request low ID clients to call back (a low ID client is not capable of accepting incoming connections). There was also a feature allowing two low ID clients to exchange files through their server connection, using the server as a relay. Most of the servers no longer support this option because of the overhead it incurs on the server.

eMule extended the protocol with some additional features:

AICH - Advanced Intelligent Corruption Handling is meant to make eMule increase the granularity of the corruption handling. SHA-1 hashes are computed for each 180 KB sub-chunk and a whole SHA-1 hash tree is formed. AICH is processed purely with peer-to-peer source exchanges. eMule requires 10 agreeing peers regarding the SHA-1 hash, so rare files generally do not benefit from AICH.

Credits are not global, they are exchanged between two specific clients. The credit system is used to reward users contributing to the network, i.e., uploading to other clients. The strict queue system in eMule is based on the waiting time a user has spent in the queue. The credit system provides a major modifier to this waiting time by taking the upload and download between the two clients into consideration. The more a user uploads to a client the faster he advances in that client's queue. The modifiers are calculated from the amount of transferred data between the two clients.

Protocol Obfuscation is a feature that causes eMule to obfuscate its protocol when communicating with other clients or servers. Without obfuscation, each eMule communication has a given structure which can be easily recognized and identified as an eMule packet by any observer. If this feature is turned on, the whole eMule communication appears like random data on the first look and an automatic identification is no longer easily possible. This helps against situations in which the eMule Protocol is unjustly discriminated against or even completely blocked from a network by identifying its packets. Note that this does not, however, provide anonymity to peers.

Secure User Identification Clients in the network are identified by a unique value called user hash. This user hash is stored locally and it enables a peer to favor other peers that have favored it in the past (by uploading files to it). eMule can use an asymmetric encryption to avoid manipulation of other users hash values. The method uses a private and a public key to secure the user hash and to ensure a proper identification of other clients.

Implementation

Given the almost complete lack of documentation, we spent considerable effort reverse engineering the eDonkey protocol and the eMule extensions. In fact, we believe ours to be the most complete and up-to-date documentation of the protocol currently available (at least publicly). Currently, the plug-in supports the main eD2k protocol features such as local and global search, callback and, of course, file download and upload. AICH and source exchange are also fully working. Our current goals are a reduction in resource consumption, an increase in download speed, and an improved server search.

4.4.2 Torrent

Protocol(s)

BitTorrent is a peer-to-peer file sharing protocol used for distributing large amounts of data.

BitTorrent protocol allows users to distribute large amounts of data without putting the level of strain on their computers that would be needed for standard

Internet hosting. A standard host's servers can easily be brought to a halt if extreme levels of simultaneous data flow are reached. The protocol works as an alternative data distribution method that makes even small computers (e.g., mobile phones) with low bandwidth capable of participating in large data transfers.

First, a user playing the role of file-provider makes a file available to the network. This first user's file is called seed and its availability on the network allows other users, called peers, to connect and begin to download the seed file. As new peers connect to the network and request the same file, their computer receives a different piece of data from the seed. Once multiple peers have multiple pieces of the seed, BitTorrent allows each to become a source for that portion of the file. The effect of this is to take on a small part of the task and relieve the initial user, distributing the file download task among the seed and many peers. With BitTorrent, not a single computer needs to supply data in quantities that could jeopardize the task by overwhelming all resources, yet the same final result (each peer eventually receiving the entire file) is still reached.

After the file is successfully and completely downloaded by a given peer, the peer is able to shift roles and become an additional seed, helping the remaining peers to receive the entire file. The community of BitTorrent users frowns upon the practice of disconnecting from the network immediately upon success of a file download, and encourages remaining as another seed for as long as practical, which may be days.

This distributed nature of BitTorrent leads to a viral spreading of a file throughout peers. As more peers join the swarm, the likelihood of a successful download increases. Relative to standard Internet hosting, this provides a significant reduction in the original distributor's hardware and bandwidth resource costs. It also provides redundancy against system problems, reduces dependence on the original distributor and provides a source for the file which is generally temporary and therefore harder to trace than when provided by the enduring availability of a host in standard file distribution techniques.

To share a file or group of files, a peer first creates a small file called a "torrent". This file contains metadata about the files to be shared and about the tracker, the computer that coordinates the file distribution. Peers that want to download the file must first obtain a torrent file for it, and connect to the specified tracker, which tells them from which other peers to download the pieces of the file.

The peer distributing a data file treats the file as a number of identically sized pieces, typically between 64 KB and 4 MB each. The peer creates a checksum for each piece, using the SHA1 hashing algorithm, and records it in the torrent file. When another peer later receives a particular piece, the checksum of the piece is compared to the recorded checksum to test that the piece is error-free. Peers that provide a complete file are called seeders, and the peer providing the initial copy is called the initial seeder.

Torrent files are typically published on websites or elsewhere, and registered with a tracker. The tracker maintains lists of the clients currently participating in the torrent. Alternatively, in a trackerless system (decentralized tracking) every peer acts as a tracker. Azureus was the first BitTorrent client to implement such a system through the distributed hash table (DHT) method. An alternative and incompatible DHT system, known as Mainline DHT, was later developed and adopted by the BitTorrent.

Later many other features have been introduced, we cite only those we have implemented (or we are implementing).

Multi tracker is an extension to the BitTorrent metadata format proposed by John Hoffman and implemented by several indexing websites. It allows the use of multiple trackers per file, so if one tracker fails, others can continue supporting file transfer. Torrents with multiple trackers can decrease the time it takes to download a file, but also lead to a larger network traffic.

Protocol encryption (MSE) is a protocol designed to provide a completely random-looking header and (optionally) payload to avoid passive protocol identification and traffic shaping. When it is used with the stronger encryption mode (RC4) it also provides reasonable security for the encapsulated content against passive eavesdroppers. It is a 3-way handshake where the initiating client can directly append its payload after his 2nd step (which globally is the 3rd). The responding client has to send one step (globally the 2nd) of the handshake and then wait until the initiating client has completed its 2nd step to send payload. To achieve complete randomness from the first byte on the protocol uses a D-H key exchange which uses large random Integers by design. The 2nd phase — the payload encryption negotiation — is itself encrypted and thus approximately random too. To avoid simple length-pattern detections various paddings have been added to each phase. This encapsulation protocol is independent of the encapsulated content.

Peer exchange (PEX) is a feature of the BitTorrent peer-to-peer protocol which can be utilized to gather peers. Using peer exchange, an existing peer is used to trade the information required to find and connect to additional peers. While it may improve (local) performance and robustness (e.g., if a tracker is slow or even down) heavy reliance on PEX can lead to the formation of groups of peers who tend to only share information with each other, which may yield slow propagation of data through the network, due to few peers sending information to those outside the group they are in.

Extension negotiation protocol is a meta protocol used to manage communication between peers. It allows any peer to choose to uses Azureus messaging protocol or Libtorrent extension depending on the other peer(s). We are almost the sole client to implement this protocol.

Implementation

In the beginning we started implementing the plug-in using a GPL library([Dub]) to quickly develop a working module. After succeeding in obtaining a stable code we completely rewrote the library to exploit PariPari features and unofficial Bittorrent extensions. Now we have a quite stable module with some important (unofficial) implemented features. In the next months we should easily add some more features to speed up the downloads.

4.4.3 Future work

We are working to speed up file transfer exploiting the features offered by PariPari. Our work progresses mainly along three lines.

Cooperative download PariPari can be seen as a set of different peers. Each peer can act individually downloading files from the eD2k or torrent network. Each peer can thus accumulate credit on these different systems with other users. However these peers can act as a unique smart meta client.

Each peer can contact any of its neighbors asking to download files (or chunk of files) from the network exploiting its credits, bandwidth or, at least, increasing the amount of peers downloading the file. Later the requesting peer can retrieve chunks and files from its cooperative neighbors. This download paradigm can be particularly effective in conjunction with the next two enhancements.

Multi-network download Each PariPari node can download files using different P2P clients such as Mulo and Torrent. Any user can choose the fastest network providing the same file. The innovative feature we are working on is the possibility to download the same file using chunks from any network.

File Merging It is often the case that essentially the same content exists on the network in different formats — for example, the same music file with different encodings, or the same video with different metadata. Our goal is to identify the most common cases when this occurs, and develop code that allows one to reconstruct the content from the different files. A similar, but more restricted, approach has been proposed in [PAK07]

4.5 PariWeb

A webserver is a computer program that is responsible for accepting HTTP requests from clients (user agents such as web browsers), and serving them HTTP responses along with optional data contents, usually web pages such as HTML documents and linked objects (images, etc.). Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. There are two major versions, HTTP/1.0 that uses a separate connection for every document and HTTP/1.1 that can reuse the same connection to download, for instance, images for the page just served. HTTP is a stateless protocol.

HTTP defines eight methods indicating the desired action to be performed on the identified resource.

GET Requests a representation of the specified resource. Note that GET should not be used for operations that cause side-effects, such as using it for taking actions in web applications. One reason for this is that GET may be used arbitrarily by robots or crawlers, which should not need to consider the side effects that a request should cause.

HEAD Asks for the response identical to the one that would correspond to a GET request, but without the response body. This is useful for retrieving meta-information written in response headers, without having to transport the entire content.

POST Submits data to be processed (e.g., from an HTML form) to the identified resource. The data is included in the body of the request. This may result in the creation of a new resource or the updates of existing resources or both.

PUT Uploads a representation of the specified resource.

DELETE Deletes the specified resource.

TRACE Echoes back the received request, so that a client can see what intermediate servers are adding or changing in the request.

OPTIONS Returns the HTTP methods that the server supports for specified URL. This can be used to check the functionality of a web server by requesting "*" instead of a specific resource.

CONNECT Converts the request connection to a transparent TCP/IP tunnel, usually to facilitate SSL-encrypted communication (HTTPS) through an unencrypted HTTP proxy.

4.5.1 Implementation

All the above methods are implemented except for `POST` — which is of very limited usefulness. Perhaps more importantly, currently, the web server does not support any scripting language nor dynamic functions. For what concerns the other potentially unsafe methods (`PUT`, `DELETE`) we provide the users with a simple authentication method known as *Basic Access Authentication*[1]. We chose it for the sake of compatibility and simplicity.

Local The plug-in can be run as a standalone local server. Each peer can run its own server regardless of other PariPari peers. Moreover each running instance can support a virtual host mode. Each PariWeb can handle simultaneously several websites with different data.

Distributed The plug-in can be run in a distributed fashion thanks to DiESeL. PariWeb retains all its functionalities even when running in a distributed fashion.

4.5.2 Future work

We are planning to embed in PariWeb a PHP engine and a more secure authentication method. PariWeb will use it to become “self hosting”.

[1]The basic access authentication is a method designed to allow a web browser to provide credentials — in the form of a user name and password — when making a request. Before transmission, the user name is appended with a colon and concatenated with the password. The resulting string is encoded with the Base64 algorithm.

Chapter 5

Plug-ins in Early Development Stages

This chapter provides a brief description of the PariCredits, PariLogin, PariDNS, PariDBMS and PariGUI plugins. These plug-ins are still in early stages of development, and their structure is in flux.

5.1 PariLogin and PariDNS

This Section gives a brief description of the preliminary versions of PariLogin and PariDNS. Both implement essentially the same function — a secure translation table piggybacking on (the insecure) PariDHT.

5.1.1 PariDNS

The Domain Name System (DNS) is a hierarchical naming system for computers, services, or any resource connected to the Internet. It associates various information with domain names assigned to each of the participants. Most importantly, it translates domain names meaningful to humans into the numerical (binary) identifiers associated with networking equipment. DNS works like a "phone book" for the Internet by translating human-friendly computer hostnames into IP addresses.

PariDNS could admit a trivial implementation over PariDHT, if not for two challenges. First, we need to implement a compatibility layer with the DNS protocol itself [Moc87], appearing indistinguishable from a "standard" DNS server to the end user (who may not be a PariPari user, but simply someone wanting to access the web page a PariPari user is hosting through PariWeb). Second, we must strengthen security and provide guarantees against DNS hijacking, which the current "best effort" implementation of DHT does not provide.

The second task is essentially the same that must be accomplished by PariLogin.

After a brief introduction to PariLogin in Subsection 5.1.2, a partial, temporary hack for both — with a number of drawbacks — is described in Subsection 5.1.3. It may well be that a future, more sophisticated version of DHT will make this redundant.

5.1.2 PariLogin

PariLogin associates a username/password pair to the state of a particular user (including storage permissions, encryption/signature keys, and current IP+port). Again, this requires simply the definition of an extensible representation of the user's state, and a level of protection from "identity theft" which is substantially the same that must be provided by PariDNS.

5.1.3 Identity Theft Protection

In a nutshell, for each web (sub)domain or username, we must maintain a record "owned" by that user on a DHT, with the following guarantees:

Ownership. The owner, and only the owner may modify or delete the record and read the private portions of the record.

Permanence. The record should remain constantly available.

Authenticity. Any user attempting to read the record (or the public portions of the record for users different from the owner)

The mechanism we are considering is simply to distribute the record over a sufficiently large number of peers, chosen by some pseudorandom hash function, taking the majority vote. This has some drawbacks we must still address. First, if an position in the DHT virtual address space is chosen, rather than a specific peer, a sufficiently determined adversary may adaptively try to conquer a position close to that of the record. Second, replication can have a substantial overhead. Third, in the case of PariDNS, one still faces the problem that a user outside PariPari looking for a DNS record will contact a single IP to retrieve the record (if that IP responds), so a majority vote technique will be ineffective if that one node is faulty.

5.2 PariDBMS

PariDBMS aims at being a distributed database management system that makes the distribution of the database transparent to the user. The plug-in is being designed to keep a collection of multiple, logically interrelated databases distributed over PariPari. We are using HSQLDB [hsq] both to manage the single database instances and to keep the whole structure connecting the distributed databases. Currently we are adapting the plug-in for the use of DiESeL and DistributedStorage. In the near future we plan to implement some smart features to speed up information retrieval and to arrange the database structure so that it can stand massive instantaneous loads (i.e., a high number of query or update requests) using distributed table caching.

5.3 PariGUI

The Graphical User Interface is one of the most important parts of many applications — particularly those aimed at the "average user". A good GUI is not simply an issue of eye-candy: it is also intuitive, easy to learn, and yet powerful enough even for the experienced user. This is particularly true for complex applications with many sophisticated functionalities — such as PariPari. This section briefly presents our plans for the future GUI of PariPari, and the transitional, extremely simple GUI currently embedded in PariCore.

5.3.1 PariPari's eventual GUI

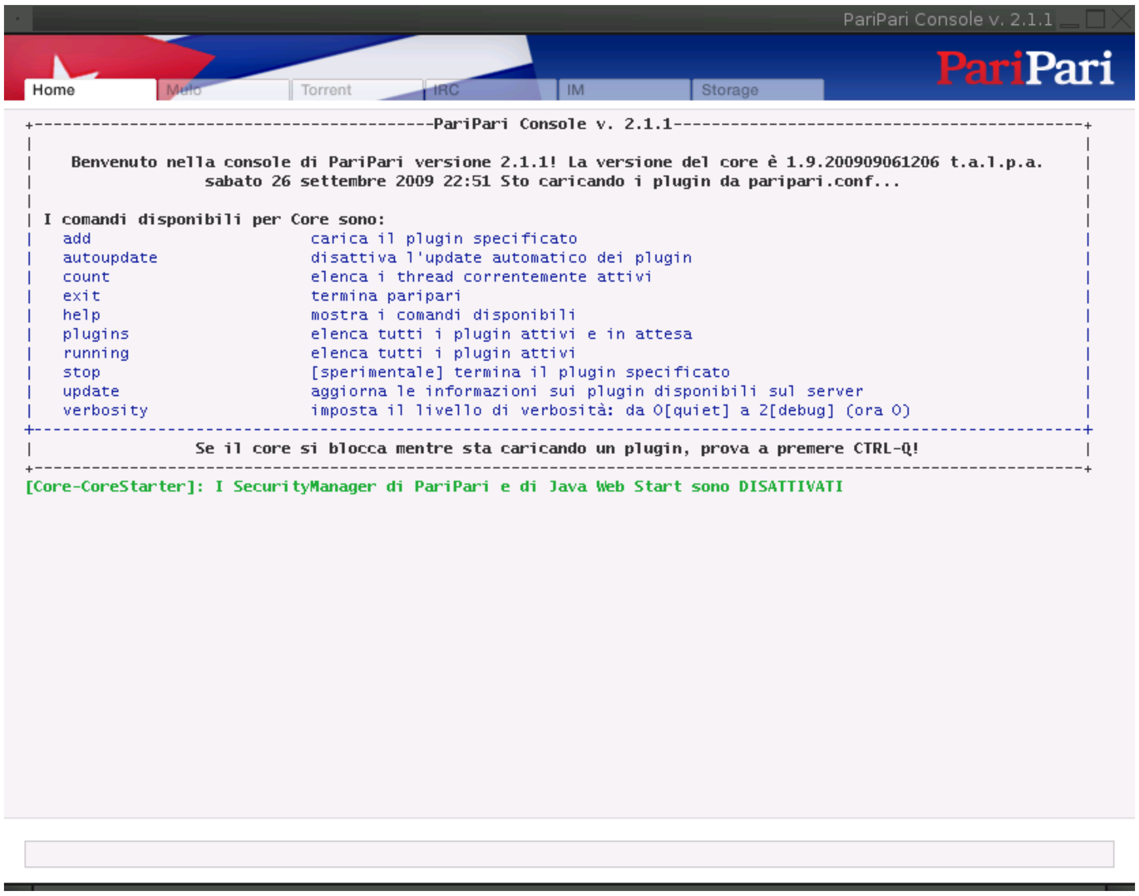
Our approach is based on the Model-View-Controller (MVC) [GHJV95, 14-16] architectural pattern. The GUI plug-in provides all other plugins with a number of abstract visualization methods (buttons, advancement bars etc.) through which they can query the user for information and constantly send him all information available. The GUI then also defines an actual visual representation corresponding to each method, as well as the organization of those individual representations on the screen (or, in our case, the browser's window). The visual representation can easily be changed (to adapt to different user tastes or constraints — e.g., the small screen of a mobile device) without having to modify the code of the plug-ins themselves. This also allows the GUI to easily produce multiple visualizations or simply a remote visualization (e.g., controlling PariPari, running on a server, from a mobile phone that is not sufficiently powerful to run it).

The GUI is only in its initial stages of development. We have already coded a small fraction of the abstract visualization methods, and have produced a number of possible actual visualization sketches. Over the next few months we intend to increase the collaboration between the GUI team and the teams of other plugins, to extend the visualization method base, and to refine the actual visualizations.

5.3.2 Simple GUI

A simple, transitional temporarily embedded in PariCore allows users and developers to test PariPari and its plug-ins without having to hard-code commands inside some configuration files. Currently this GUI is just an eye candy for the command line interface. However we added some tabs to launch some (already implemented) plug-ins.

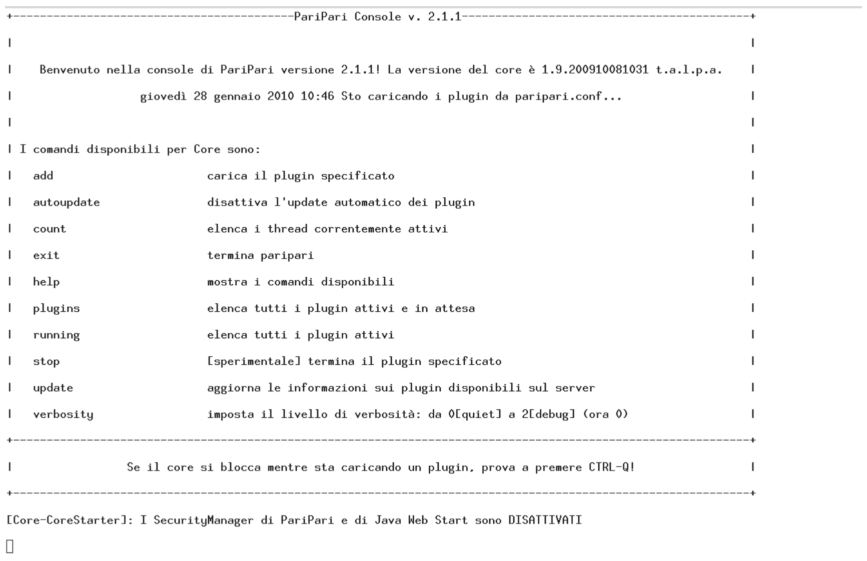
To allow remote access to PariPari (e.g., via `ssh`) we are also maintaining a TCP-CLI. When activated using its remote interface, PariPari opens a TCP socket



```
+-----------------------------------------PariPari Console v. 2.1.1-----------------------------------------+
|                                                                                                           |
|    Benvenuto nella console di PariPari versione 2.1.1! La versione del core è 1.9.200909061206 t.a.l.p.a.  |
|                  sabato 26 settembre 2009 22:51 Sto caricando i plugin da paripari.conf...                |
|                                                                                                           |
| I comandi disponibili per Core sono:                                                                      |
|   add                          carica il plugin specificato                                               |
|   autoupdate                   disattiva l'update automatico dei plugin                                   |
|   count                        elenca i thread correntemente attivi                                       |
|   exit                         termina paripari                                                           |
|   help                         mostra i comandi disponibili                                               |
|   plugins                      elenca tutti i plugin attivi e in attesa                                   |
|   running                      elenca tutti i plugin attivi                                               |
|   stop                         [sperimentale] termina il plugin specificato                               |
|   update                       aggiorna le informazioni sui plugin disponibili sul server                 |
|   verbosity                    imposta il livello di verbosità: da 0[quiet] a 2[debug] (ora 0)            |
+-----------------------------------------------------------------------------------------------------------+
|                  Se il core si blocca mentre sta caricando un plugin, prova a premere CTRL-Q!             |
+-----------------------------------------------------------------------------------------------------------+
[Core-CoreStarter]: I SecurityManager di PariPari e di Java Web Start sono DISATTIVATI
```

Figure 5.1: PariPari simple GUI (Cuba Release).

to listen to commands dispatched in a client - server fashion. The client can be a simple networking utility that reads and writes data across network connections using the TCP/IP protocol, such as `NetCat` [nc]



```
+------------------------------------------PariPari Console v. 2.1.1------------------------------------------+
|                                                                                                             |
|    Benvenuto nella console di PariPari versione 2.1.1! La versione del core è 1.9.200910081031 t.a.l.p.a.   |
|                   giovedì 28 gennaio 2010 10:46 Sto caricando i plugin da paripari.conf...                  |
|                                                                                                             |
| I comandi disponibili per Core sono:                                                                        |
|   add                       carica il plugin specificato                                                    |
|   autoupdate                disattiva l'update automatico dei plugin                                        |
|   count                     elenca i thread correntemente attivi                                            |
|   exit                      termina paripari                                                                |
|   help                      mostra i comandi disponibili                                                    |
|   plugins                   elenca tutti i plugin attivi e in attesa                                        |
|   running                   elenca tutti i plugin attivi                                                    |
|   stop                      [sperimentale] termina il plugin specificato                                    |
|   update                    aggiorna le informazioni sui plugin disponibili sul server                      |
|   verbosity                 imposta il livello di verbosità: da 0[quiet] a 2[debug] (ora 0)                 |
+-------------------------------------------------------------------------------------------------------------+
|                 Se il core si blocca mentre sta caricando un plugin, prova a premere CTRL-Q!                |
+-------------------------------------------------------------------------------------------------------------+
[Core-CoreStarter]: I SecurityManager di PariPari e di Java Web Start sono DISATTIVATI
▯
```

Figure 5.2: Command Line Interface.

5.4 PariCredits

One of the most innovative features of PariPari is a sophisticated pseudoeconomy to regulate access to resources. This functions at two layers: within each peer, to regulate access by different plug-ins to machine resources, and between peers, to encourage cooperation and discourage freeriding. While the two layers are fundamentally different, they are both managed by the PariCredits module which is tightly integrated with PariCore (the main reason for making PariCredits a separate plug-in is to isolate PariCore, which is already fairly mature and whose stability is paramount, from code changes in the PariCredits module).

5.4.1 Intra-peer Credits

The intra-peer layer has been almost entirely implemented. PariCore issues to each plug-in a steady stream of Credits with which it can purchase machine resources such as bandwidth, storage space and CPU cycles, competing in an auction system with other plug-ins for them. This can allow plug-ins to choose different strategies depending on the current cost of resources. For example, aggressive compression of transmitted data may or may not be worthwhile depending on whether CPU cycles are cheap or not compared to bandwidth — i.e., depending on whether the CPU's load is light or heavy compared to network's traffic.

The credit income that allows each plug-in to purchase resources varies both in time and between different plug-ins. It is essentially proportional to the priority of that plug-in at that moment. It can be modified by an experienced user directly; but we expect it far more often to be modified indirectly by a user signalling through PariGUI to PariCore his priorities (e.g., if being in haste to upload or download a particular file). Developing an expressive but intuitive way to signal different priorities is one of the most interesting directions of future research at the frontier between PariCredits and PariGUI.

5.4.2 Inter-peer Credits

The inter-peer layer is not yet past its design phase. We expect it to be much more difficult to design and implement than the intra-peer layer, for two fundamental reasons.

First, all plug-ins running in each peer are subject to a single authority (that of the user) and a common goal (the user's satisfaction). Thus, if we assume that the code base is safe, different plug-ins can trust each other and one need not consider faulty or malicious plug-ins. This is not the case in the inter-peer scenario: different peers will have different, often conflicting goals (e.g., when downloading data from a common source), and one can certainly expect a small but non-vanishing fraction of them to be faulty or actively malicious.

Second - as a consequence of plug-ins within the same peer all having the same goal and authority - the intra-peer pseudo-economy can run on a pseudo-currency generated by fiat by PariCore. This is impossible in the case of inter-peer exchanges. On the other hand [Pes] shows how it is possible to create an efficient barter based system, piggybacking on a DHT, that obviates the need of a shared currency and is robust against a small fraction of faulty or even actively malicious nodes. We plan to use it as the basis for the inter-peer layer.

Part II

PariPari: Management

Chapter 6

Human Resources

We invested considerable resources on the human factor. This was absolutely crucial because PariPari is a labour intensive project with a large but “amateur” pool of developers. Furthermore, these developers are virtually all unpaid student volunteers with many other commitments in life. It was, then, of the utmost importance to identify, cultivate and exploit the tiniest morsel of quality in our workforce - and at the same time to make every possible effort to maintain and reinforce its enthusiasm.

Section 6.1 discusses in greater detail the differences between our workforce and that of a “standard” software project. Section 6.2 deals into the crucial but slippery notion of the “quality” of the workforce. Armed with this knowledge, it is easier to understand our choices for the organizational structure of PariPari described in Section 6.3. Section 6.4 discusses our motivational strategies. Finally, Section 6.5 summarizes some “lessons” from the PariPari project and the implications on a) the current student curriculum in Computer Engineering at the University of Padova and b) the organization and management of large “amateur” software project.

6.1 Student vs Professional employees

The most evident difference between PariPari and any other software project is that the workforce of PariPari is entirely comprised of students (of the school of Computer Engineering of the University of Padova). We currently employ just one Ph.D student, several undergraduate students and and a few master students. This peculiarity strongly affects all the other aspects of the project. For example we have to deal with unavailability periods due to exam sessions. In the remainder of this section we review the main differences between students and professional employees.

6.1.1 High Turnover

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. Some of them, after their first level degree, continue working in PariPari. These students are very useful since they are already formed and can effectively help newcomers. Also, these experienced students often exhibit enough maturity, coding and design abilities, and commitment to the project that they can take on greater responsibilities.

Dealing with turnover is crucial. Since PariPari is a complex project, we found that the best way to pass on accumulated knowledge is through actual collaboration. Indeed, one of the worst fates that can befall a plug-in is losing its workforce before it can train a new generation. Although we spend a lot of time writing documentation about the project and code, we noticed that resuming work on a "dead" plug-in is very inefficient because a lot of information is passed only through a direct relationship.

6.1.2 Scarcity

PariPari is always hungry for new students. We can only reach students of the Department of Information Engineering at the University of Padova using flyers, word of mouth, etc. Although the number of reachable students could seem sufficient, we have to consider other factors that drastically reduce our potential workforce. First, only a subset of students are interested in PariPari. Second, only a smaller subset can work to PariPari due to their curricula studiorum. In fact, students work on PariPari only for their thesis. This time constraint considerably reduces the size of our potential workforce.

6.1.3 Low Cost and Low Quality

PariPari is a low cost project or, better, it is a no-cost project. Not having any money to spend to hire professional programmers, we are forced to use only students. This choice has two serious drawbacks: we have to teach students and we can have only a small leverage on them.

Most students that enter the PariPari project have insufficient experience with designing and writing good software. They only have a small coding experience. Their previous coding activities involve only very small one-man projects so they do not know how to behave in a programming team. More than half our students have received no formal training in software engineering, and none of the (two) courses in software engineering taught by our school involve actual, hands-on, coding.

To fill these gaps we have to spend some energy and time. On one hand this period further reduces the students productive time in PariPari, and on the other hand this teaching effort subtracts energies from the more experienced students, who must spend a fraction of their time training new recruits. It is crucial for the success of the project to evaluate on an individual basis whether a student will provide PariPari a net benefit, or instead drain more resources than he can give back to the project.

6.1.4 Unreliability

Students are unreliable. They also have numerous commitments that they often perceive as more important than PariPari: they have to attend courses, prepare exams and sometimes they even have a job. Often, they are not skilled enough to plan their time according to all their undertakings so they stop working for PariPari without or with little advance notice. This behavior obviously affects the productivity of all other team members slowing down their progress. Indeed, often students - not having been part of a large team effort before - often even fail to realize the extent to which they can cause damage to others by being unreliable.

It is very difficult to face and solve this kind of situations. We have only a small leverage on students. As part of the academic system we have to encourage them to attend courses and to study hard for examinations. We can only try to enhance their team relationships to lead them to change their priorities.

6.2 Personnel Quality

In section 6.1.3 we stated that students are a generally low-quality workforce. But what do we mean, exactly, with quality? Personnel quality strictly depends on context. For example, in a videogame design environment a high quality worker is one exhibiting creativity, coding speed, and the ability to deal with multimedia contents. On the other hand, for a security consultant precision, accuracy, and the ability to perform penetration tests are more important. So the same person can be of high quality in one environment and of low quality in another one. However, PariPari is so complex that many different qualities are required. Different plug-ins require students with several different qualities. Also, in general, we can define different roles in a single plug-in, each of which requires very different skills. Roughly speaking, we can state that a PariPari student must have at least one quality among relationship, design, and coding ability.

Since quality is so scarce it is crucial to identify it, cultivate it and exploit it, especially in newcomers.

6.2.1 Definition

We begin this section listing the main abilities required of our students.

Adaptability: the ability to get acclimatized in PariPari is very important. In particular, learning to use the adopted production tools and understanding the rules of the group in a few days represent a double advantage. First, students have more time to work on their tasks, and second, less effort is required of senior students, allowing them to continue working on their tasks.

Research: the will to understand a problem thoroughly is often the key factor to solve it in our environment. Most of the times, this skill is tightly linked with attitude to reverse engineering. This kind of approach is sometimes the only way to deal with problems in PariPari.

Design: the ability to keep in mind all the specifications of the project, all the problems, all the criticalities and to develop a good design is maybe the most important ability to look for in a student, particularly one who is to be entrusted with a leadership position.

Coding: designing and implementing good code is not a trivial task. Code has to be simple, fast, upgradable and quickly changeable (and, of course, correct!). Often, students tend to write code that seems good only to themselves. To address this issue we enforced object-oriented programming paradigms and adopted test driven development.

Relationship: the above are technicals skills, very useful to obtain good results and to work productively. However, we can not overemphasize the importance of relationship abilities. These kind of relationships are the mortar of the group and the bricks of the organization of work. It is not possible to work in an unfriendly environment. Besides, one particular skill is very important to our scope: leadership.

Leadership: [Bas02] defines leadership as a “process of social influence in which one person can enlist the aid and support of others in the accomplishment of a common task” - and organization is one of the most reliable indicators of success of a PariPari team. A good leader understands the capabilities and interests of his fellow workers, and can motivate them and adapt their workload so as to maximize fulfillment and hence productivity. We have noticed that good leadership is very often highly correlated with initiative; students showing initiative can often become excellent leaders.

Reliability: is the ability of a person to perform and maintain their undertaking. An unreliable worker slows down the work of the whole team, besides achieving their objective slowly. Hence, it is quite clear that a reliable student is a double advantage.

All the qualities above are important in PariPari students, and are crucial in many plug-ins — even though some plug-ins have more stringent requirements than others. However, the most important aspect dictating which qualities are important in a student is not the plug-in, but the role within the plug-in. Team leaders obviously cannot lack leadership. Design skill is also of greater importance for a team leader than for other members of the team. Indeed, a team leader with good design skills can usually obviate to a total lack thereof in the other team members.

6.2.2 Evaluation

Identifying quality in (potential) new members of the project is by no means an easy task. While a lengthy interview with the candidate can often provide sufficient information, this is neither always reliable, nor always possible. For example, most of the members of the project are recruited during large, sporadic recruitment events involving dozens of people.

To address these problems we adopt different strategies to evaluate newcomers. This evaluation is crucial because by identifying the quality of students we can assign them to the right role in the right team. Matching the students' attitude with our vacancies maximizes their satisfaction guaranteeing the best productivity for us.

Initially, we used coursework grades as a benchmark for a rough assessment of student quality. However, we soon discovered that this was an extremely unreliable benchmark. First of all, coursework obviously does not discriminate students based on one of the qualities we prize most — namely leadership. Second, even for "engineering" qualities like strong coding or design skills, coursework performance is often a poor indicator. We have seen many students with grades in the top 20% of their class that have terrible coding skills and, perhaps worse, no initiative at all. On the other hand, we have discovered excellent coding skills and (perhaps more surprisingly) excellent design skills in students with only average or below average grades.

Figures 6.1,6.2 and 6.3 provide a scatterplot of the relationship between a student's average grades and (respectively) coding and design skills, initiative and reliability for all students currently part of the PariPari project. All evaluations were performed by their respective team leaders, except for leaders who where evaluated by the head of the project. It is quite obvious that *there is almost no correlation between a student's grades and his design and coding skills, and only an extremely mild correlation between grades and initiative and reliability.*

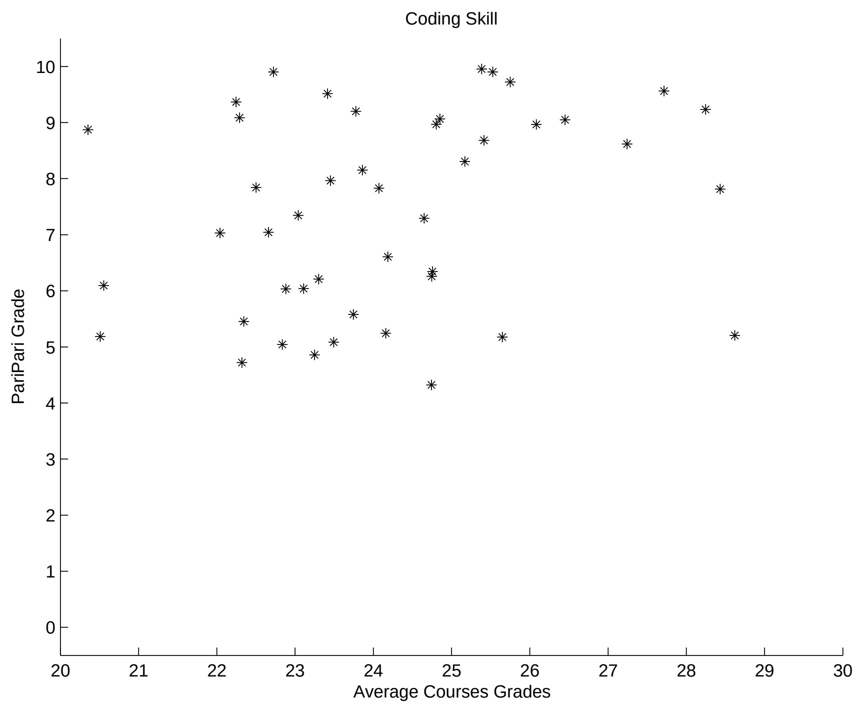


Figure 6.1: Course grade Vs PariPari quality: coding skill.

A more effective alternative that a number of teams have started to adopt is to quiz their new or prospective members with coding and theory tests developed by the current (student) team members. It is interesting that this methodology appeared "spontaneously" — i.e. it was not even suggested by the highest echelons of the PariPari hierarchy — and appears to have good results in predicting initiative, adaptability and research attitude of newcomers. While we still lack a reliable method to test for leadership potential, a student possessed of only these qualities is still an excellent addition to PariPari.

6.2.3 Cultivating Quality

It is obvious that cultivating the quality of our students is of paramount importance to the success of the project. Not only does this lead to a better workforce, but an

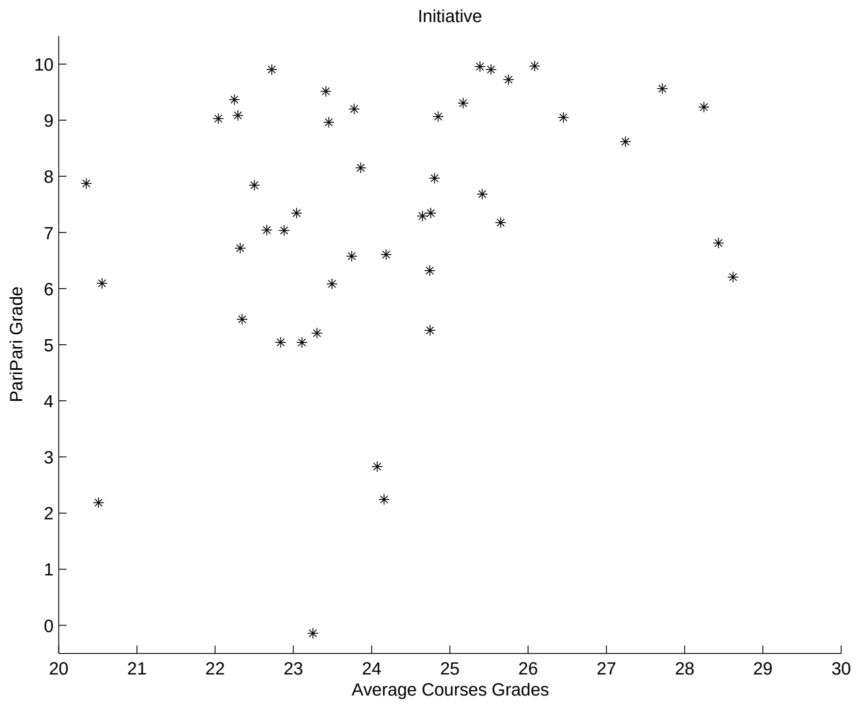


Figure 6.2: Course grade Vs PariPari quality: initiative.

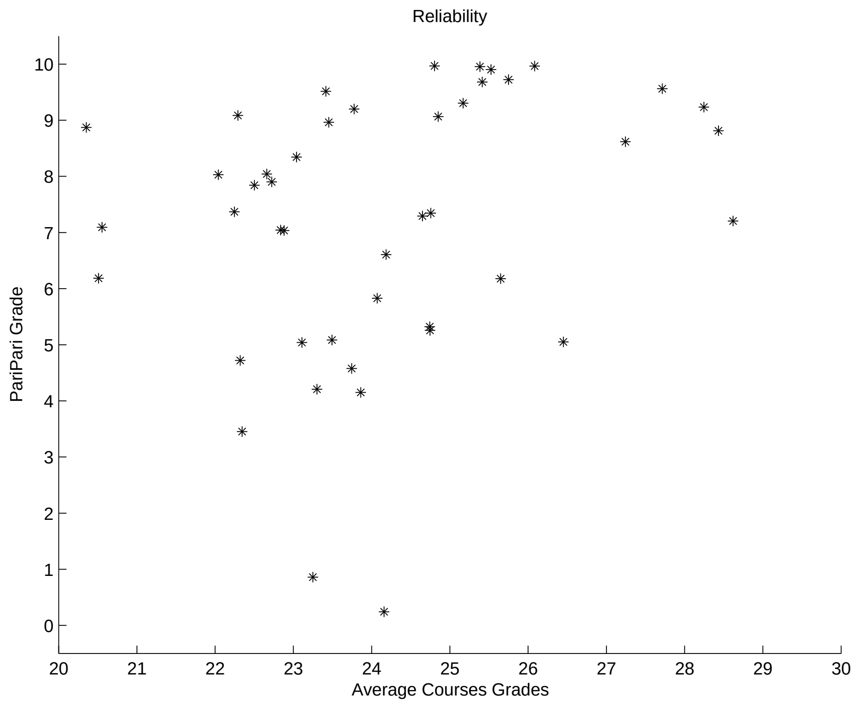


Figure 6.3: Course grade Vs PariPari quality: reliability.

increasing number of students are attracted to the project for its educational potential. Our strategy is twofold, entailing a direct approach aimed at strengthening the technical skillset of our students, and an indirect — but not less important — one aimed at cultivating the motivation, morale, confidence and other "social" aspects of the PariPari workforce. To improve the coding and design capabilities of our students we adopted some simple, but strict programming paradigms (see Section 7) These paradigms help students have a clear view on the coding process, from design to implementation. So, the source code written is relatively free of errors and, perhaps more importantly, readable, testable and simple. The fundamentals of these software engineering paradigms are taught in a small number of lectures; perhaps more importantly, more experienced students often provide one-on-one training to newcomers. Interestingly, many of the senior students enjoy providing this training and do not see it as a burdensome chore.

Cultivating the "social" aspects of our workforce is more complex. We created a feedback system that provides constant updates on the status of every plug-in team and of every member. Every two weeks we organize a meeting with all PariPari team leaders to check the personnel situation, to discover any criticalities as soon as possible, and to find together solutions to any problems that might arise. We also arranged a report system to force leaders to check the situation of their teams frequently. Every week such reports are filled with details on scheduling setback and personnel issues. Beyond these formal communications we allow and encourage direct contact with the project manager to report any trouble.

This approach not only provides us with a wealth of information that allows more prompt, effective solutions to problems. It also makes students feel "empowered" and part of the decision process (see Section 6.4 below). A crucial ingredient to maximize both advantages is to reward deserving students with increasing responsibilities.

6.2.4 Quality of Team

The quality of a team is not equal to the sum of the qualities of its members. A team formed by only very good coders will quickly produce simple code but it will not be able to fulfill any other objective.

A team needs to be formed by students with different abilities. A group composed of students whose abilities cover the set of all required abilities forms a team that can perform very well. An ideal team will involve just one leader with initiative and design skills and some students skilled in coding and testing. Another really important team characteristic, often neglected, is generational overlap. We try to form teams in such a way that the projected "tenure" of different members will partially overlap. In this way senior members can train junior ones, preserving a precious legacy of experience and training carefully built up over the months.

6.3 Organizational Structure

PariPari is a very large and complex project but its workforce is extremely "fragile" (see Section 6.2). This is the main constraint conditioning the organization of PariPari (and our "twist" on eXtreme Programming - see Chapter 7.2). This section presents the hierarchical structure of PariPari (Subsection 6.3.1) and its organization in terms of time (Subsection 6.3.2) and resources (Subsection 6.3.3).

6.3.1 Hierarchical Structure

PariPari is a software organized in plug-ins. To exploit all the benefits derived from this choice we organize students in teams. Each team develops a single plug-in, working semi-isolated from the rest of the PariPari group. Interfaces are written to fulfill the Test Driven Development requirements (see 7.3). They are the unique common point among plug-ins. Thus teams are quite independent of each other and almost always communication between teams can be narrowed down to simply reading each other's code documentation. However there exist plug-ins very tightly linked with each other. We chose to aggregate such plug-in teams in federations to simplify the work of the teams. In this way, according to the eXtreme Programming rules (see 7.2), we encourage communication between developers. This behavior should lead to a faster and, possibly, better solution to the problems we encounter. These groups are almost entirely self organized. However, a leader — called plug-in leader — is set for each plug-in team and a team leader is also nominated for each confederation.

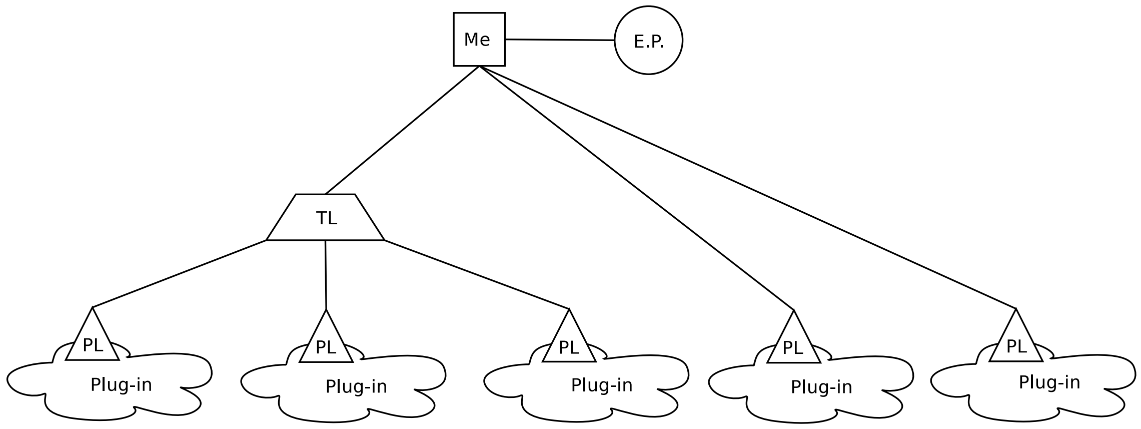


Figure 6.4: Hierarchical organization in PariPari: lines are the preferred communications

The project leader (actually myself) tries to draw together the threads of the whole PariPari project exploiting the information gathered from leaders and the scientific advice of other PhD students and faculty members.

Duties

In such a hierarchical structure it is very important to define the role of each person. We give a brief description of the main tasks for each role.

The Project Leader defines the main lines of the project. In particular, he decides the high level objectives and monitors progress of PariPari as a whole. He takes any decision about the roles of students (rewarding or punishing them) and proposes deadlines for pre-releases. Moreover, he tries to solve every personnel problem as soon as possible. All the above duties are performed exploiting all the information gathered from reports, meetings and chats in coordination with team and plug-in leaders.

Team Leaders coordinate the work of their plug-in teams. In particular, they help the plug-in leader in the design process. The team leader, in fact, takes advantage of their position to have a clear view of all plug-ins interactions. Occasionally they organize confederation meetings to promote communication and to coordinate efforts. They also schedule work, sharing these decisions with their workforce.

Plug-in Leaders have to organize the job assignment in the plug-in. If the plug-in is not part of any confederation, the plug-in leader performs the team leader role.

Evolution

We reached this organization after about two years of experiments and tuning. At the beginning there were only a few plug-ins with two or three students in each. It was quite simple to control and help everybody. In each team the plug-in leader acted almost as a router for communication. The project leader used to plan every task and controlled progress. Plug-in leaders forwarded guidelines coming from the project manager to all other students.

Whenever the group began to grow in numbers, plug-in leaders started helping the project leader collecting students requests and satisfying some of them (the most frequently asked ones - that they soon learnt how to handle). Then, along with the growth of the number of students and the birth of new plug-in teams, the role of plug-in leaders changed again. They began to organize work within their plug-in, assigning tasks to their students and solving minor issues.

Finally, we noticed that tightly linked plug-in teams sometimes require a lot of communication in order to avoid problems such as code duplication and incompatibility issues. To deal with this problem we grouped such plug-ins into “federations”, delegating the task of coordinating them to team leaders.

6.3.2 Organization of Time

Organization in terms of time is crucial in any large project. Very often, plug-ins develop mutual dependencies. If some piece of code is not ready at the right moment another plug-in using that code will be unable to progress. To address this problem we act on two levels: personal scheduling and team scheduling.

Individual Scheduling

Careful planning of the workload assigned to students is the first step to achieve good results. Leaders must also “share” deadlines with other members. While they may have a more accurate picture of the time constraints involved, they should avoid at all costs enforcing them by fiat, without consensus. Imposed deadlines will very likely be missed. Instead, a shared deadline (although more relaxed) will (probably) be hit.

Team Scheduling

On the basis of individual scheduling the project leader can quite safely build the scheduling for the team. Each leader makes a roadmap trying to satisfy constraints dictated by individual schedulings and by the project leader. These roadmaps aim at pre-releases that work as milestones. Pre-releases are prepared to provide synchronization points among plug-ins and to motivate students, giving them the opportunity to form a clear view of the global progress of PariPari. Clearly, pre-release preparations require a lot of additional work. Thus, we schedule release dates outside exam sessions, when students are busiest. This way students can concentrate their efforts on PariPari and on their studies without dangerous overlaps. In fact, we have been studying the interplay between exam sessions and productivity since the first years of PariPari. In Figure 6.5 we can see that before October 2008 during exam sessions the number of code commits decreased drastically. Thus, exam sessions are best avoided when setting deadlines that are likely to require an increased level of effort.

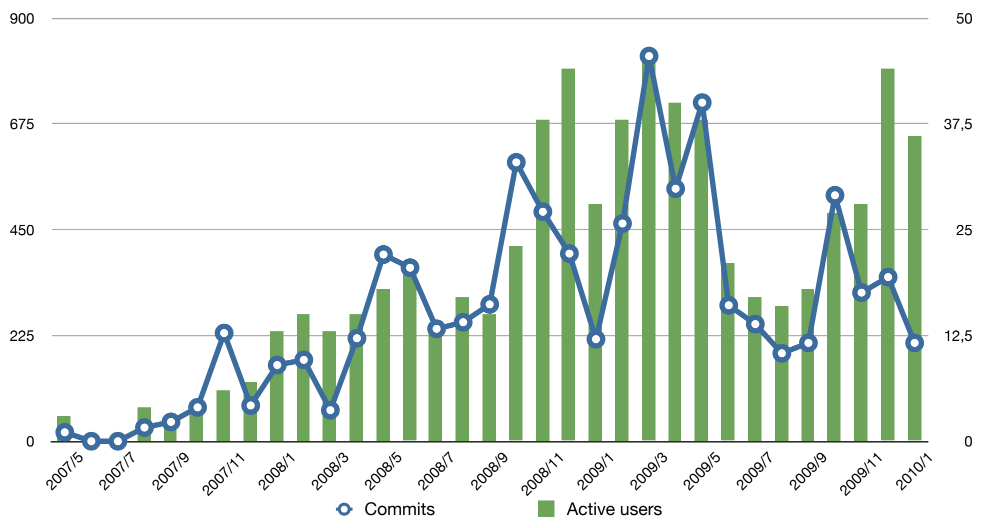


Figure 6.5: Commit on the SVN server: commit decrease during exam sessions.

Event Scheduling

Other important events to be carefully scheduled are recruitment sessions, report delivery dates, and meetings. Reports and meetings should be frequent enough to let the project leader know the current situation and to respond quickly to budding problems. At the same time they should not be too frequent, as they can represent a serious time drain for team and plug-in leaders. Currently meetings are scheduled every two weeks, and progress reports are due every week.

Recruitment session timing faces similar constraints. Organizing recruitment sessions requires considerable energy. Furthermore, they can involve bursts of "advertising activity" that, if too frequent, may annoy faculty and even prospective students. However, recruitment should take place with enough regularity to guarantee generational overlap (see above). Periods immediately following exam sessions are ideally suited to recruitment since these are the times when students typically begin looking for new projects to become involved in.

6.3.3 Organization of Resources

As stated in 6.1.3 PariPari is a very low cost project. We do have no dedicated spaces nor labs. Most communication takes place over instant messages and email. Our students work on their own, mostly using their own laptops. Frequently students work even from home. To support our workforce and organize code and communication we set up a Subversion Server and a mailing list (more details in 7.1).

Somewhat surprisingly, this loose, decentralized working environment actually increases student productivity. By decoupling the idea of work from that of a physical workplace, we encourage students to make full use of time between classes, on trains etc. and to work "on a whim" even when the our Department is closed.

6.4 Motivational Issues

Motivation in PariPari is crucial. Motivation is the main source of student productivity. We often noticed that motivated and ignorant students perform better than capable but unmotivated ones. In fact, motivated students can easily learn by themselves, unmotivated students tend to underestimate — and perform poorly at — any task. We try to cultivate student motivation in a number of ways.

6.4.1 Motivation from Joining

Clearly, the first necessary step for any student to become a productive member of PariPari is to join the project. We spend considerable effort to encourage student participation. We advertise that joining such a large software project is a unique opportunity to learn the tools and techniques used in the industry. This is particularly important because software engineering courses at the University of Padova — for lack of resources — often offer little or no practical training, and local companies offering internships are rarely large enough to tackle large scale, ambitious software projects. Also, the possibility of being part of a software project that may one day rival eMule, BitTorrent, MIRC or Wua.la holds an incredible appeal on young students.

We advertise PariPari using flyers, brief presentations in the intervals between classes, and word of mouth. Our main goal when advertising is to raise sufficient interest for students to attend one of our recruitment sessions. At these sessions we provide a more detailed presentation of the PariPari project and of the role prospective member would play in it. We have observed that well more than half the students who attend these sessions actually end up joining the project.

6.4.2 Encouragement

As mentioned before, we place a strong emphasis on inter-student encouragement. Roughly speaking, relationships between students in the project are of two types: peer to peer, and mentor/leader/senior student to junior member. Peer relationships often are synergistic with friendship. Friends often join the project together, usually in the same team. At the same time, members of the same team often become friends. We encourage the formation of strong peer-to-peer bonds between members, as it translates into a team mentality fostering collaboration and friendly intra-team and inter-team one-upmanship.

Senior-to-junior (and leader-to-team member) relationships are somewhat different, but equally important. Leaders help newcomers understand the structure, dynamics and conventions of PariPari, quickly infecting them with the project's "geist". While PariPari is a large project and it would be easy to feel just another cog in the machine, attention from their leaders makes students feel important — and pride in one's work can often boost productivity considerably. Finally, a leader provides an obvious role models for a junior members — and can subtly steer him to better performance with "carrot and stick" tactics carefully tailored to that member's personality.

6.4.3 Responsability

We borrow from the Scout method (see [BPoG51]) the crucial role given to responsibility and confidence. Students feeling confidence in their leaders are happier and are encouraged to test themselves discovering new possibilities and solutions to problems. To strengthen these feelings, leaders entrust new (small) responsibilities to deserving students. This behavior leads to a virtuous loop that often transforms a student into a new leader.

6.4.4 Special Events

Special events such as recruitment sessions and pre-releases are also important in terms of motivation. Both these events require quite a considerable effort from every student. Also, during recruitment sessions senior students present to potential newcomers the state of the art of their plug-ins. Being indirectly rewarded in this way and seeing their own code work after a release makes the student feel rewarded to be part of such a large project. These occasions reinforce the "esprit de corps" between members of the project, boosting morale and performance. Again, this often generates a virtuous loop of greater achievements leading to greater pride and viceversa.

6.4.5 The right student for the right plug-in

Perhaps the most obvious way to motivate students is to have them work on a sub-project they like. PariPari is a large project, and its plug-ins have many different "flavors". For example, students with a strong interest in graphics and ergonomy are naturally drawn to the GUI plug-in; students who want to learn the intricacies of the JVM tend to gravitate around the Operating System layer plug-ins and particularly PariCore. Additionally, plug-ins sport different degrees of maturity, leading to different challenges and rewards. Early in its stages of development, a plug-in is still in flux and a student can receive a considerable sense of empowerment from being able to shape its overall structure. Later, the plug-in's code base becomes more settled. At this stage, most of the work is refactoring and optimization. Some students dislike this phase of marginal returns; many relish the opportunity of developing a plug-in that is already functional, so that every small contribution leads to an immediate visible improvement.

6.5 Beyond PariPari

The lessons we learnt from Human Resource Management in PariPari have implications beyond it, both in terms of academic curriculum and employment opportunities for students, and in terms of large "amateur" software projects.

6.5.1 Computer Engineers and Computer Engineering skills

Our experience with PariPari suggests that the current program of Computer Engineering at the University of Padova is dangerously skewed towards a purely theoretical curriculum. Most graduates have only a superficial understanding of a single programming language — Java — and thus are not able to use it to full effectiveness, since they neither grasp its subtler points nor its strengths (and weaknesses) compared to other languages. Many students do not — in fact, can not — take any software engineering classes, and those who do face the subject from a purely theoretical standpoint, without writing any code, without becoming familiar with development tools, and without witnessing the practical advantages of adopting modern programming paradigms. Those few students with more extensive experience have almost invariably acquired it outside the formal curriculum.

This is even more serious at the Master and PhD level of the curriculum, where enrollment mechanisms strongly favor students with talent/interest in pure theory, and a student receives constant subtle (and not so subtle) messages about the intellectual superiority of pure theory. There is almost no perception that this effectively cripples what should be the elite of our alumni. In fact, even the laudable initiative of pushing students towards internships in companies is seriously hampered by the fact our interns — because of their perceived lack of "engineering maturity" — are often relegated to marginal roles of little educational value.

The fact that our Computer Engineering curriculum produces alumni with few Computer Engineering skills may well be one of the reasons why the average Computer Engineer graduate earns such a low starting salary — definitely less than the average plumber or baker. An additional, possible reason for this salary gap is described in the next subsection.

6.5.2 Grades and the market of lemons

The fact that the Computer Engineering curriculum at the University of Padova is so skewed towards pure theory has another deleterious effect. Since theoretical and engineering talent are often relatively uncorrelated (see Section 6.2.2) grade averages rarely reflect the skill of Computer Engineering graduate at Computer Engineering. Yet many companies in Italy still consider grade average a good indicator of a potential employee's quality; and indeed tend to restrict hiring to the higher grade cohorts.

This leads to a situation first described by George Akerlof in his celebrated paper "The market for lemons" [Ake70]. Akerlof observes that a market with strong information asymmetry — and in particular where the seller can assess more accurately than the buyer the quality of traded goods — is subject to a progressive deterioration of quality. The intuitive rationale is that buyers, who do not know the quality of a specific good, are willing to pay only average prices. This drives away from the market sellers of goods whose quality is above average, lowering the average quality — in a vicious cycle that ends with the market being populated almost exclusively by low quality goods traded at low prices.

From our experience with PariPari "alumni", the local Computer Engineering market seems indeed to be suffering from a "market of lemons" effect. Capable students with poor grades have difficulties finding adequate jobs in the local job market, and thus abandon it. This lowers the quality of the average Computer Engineering graduate witnessed by companies — whose response is to lower salaries, tighten grade requirements and generally offer progressively harsher hiring conditions, obviously increasing the “brain drain” even further.

6.5.3 The PariPari alumni network

Preliminary observations suggest that PariPari alumni, when attempting to recruit new software engineers for their companies, tend to come back to PariPari and inquire among recent graduates or soon-to-be-graduates. This is probably caused in part by the esprit de corps we have fostered. Yet the main cause (according to recruiters as well) is that the average PariPari member has stronger software engineering skills than the average Computer Engineering graduate, and that a positive recommendation from the senior members of the project is a much more reliable predictor of future performance than grade average. Of course, this often means that knowledgeable recruiters offer slightly better hiring conditions to PariPari members, creating a virtuous loop that partially counters the “market of lemons” effect described above.

PariPari is still a relatively young project, and only a small fraction of its alumni have been in the job market for more than one year. Thus, we still have insufficient data to obtain a significative quantitative assessment of the “network of alumni” effect we just described. We do, however, believe it to be an extremely promising direction for future research.

6.5.4 “Amateur” software projects

PariPari is the living proof that large, complex “amateur” software projects are feasible. Adequate management can squeeze productivity out of a workforce that is largely unpaid, working part time, and/or relatively unskilled. A correct organizational structure allows the training of a small seed of experienced workers to “infect” the rest of the project. Access to this training provides strong motivation to join the project even in the absence of direct, immediate monetary gain (see [LW05]). Enthusiasm for something perceived as “cool” or potentially famous is another strong incentive.

Ultimately the key to success is adopting the correct software engineering methodologies (see Chapter 7). This includes, importantly, avoiding rigid schemes and adapting to each individual situation and person, trying to identify (often hidden) potential and to make full use of it.

Chapter 7

Programming

To manage the code writtend by dozens of developers we chose to adopt the object oriented programming (OOP) paradigm. This paradigm yields:

- faster code development;
- easier maintenance;
- enhanced modifiability;
- simpler testing process;
- naturalness to deal with the classes.

All these features are clearly of crucial importance for the PariPari project, given its complexity and the nature of its workforce. In particular, in an environment prone to refactoring, OOP yield faster code writing and easier task management.

7.1 Tools

Supporting such a huge number of developers led to the adoption of some very popular tools. Below we simply cite these tools providing a brief description of their goals and functionalities.

Eclipse ([ecl]) is a multi-language software development environment comprising an IDE and a plug-in system to extend it. It is written primarily in Java and can be used to develop applications in Java and, by means of the various plug-ins, in other languages.

Subversion ([svn]) Subversion (SVN) is a version control system initiated in 1999. It is used to maintain current and historical versions of files such as source code, web pages, and documentation.

Googlegroup ([goo]) Google Groups is a service from Google that supports discussion groups, including many Usenet newsgroups, based on common interests. Google Groups offers, beyond the possibility to receive threads by email, a web based interface to the group. Using this interface it is possible to browse all the committed posts.

Bugzilla ([bug]) Bugzilla is a Web-based general-purpose bugtracker and testing tool.

Messengers To keep developers cohesive we make extensive use of instant messaging clients.

7.2 Extreme programming

Figure 7.1: eXtreme Programming Logo.

Extreme Programming is a software development methodology (born in 1996) to improve software quality and responsiveness to changing customer requirements. It advocates frequent "releases" in short development cycles, to improve productivity and introduce checkpoints where new customer requirements can be adopted.

We chose Extreme Programming as a software development standard for PariPari because of its interesting features (see below).

Responsiveness: XP empowers developers to confidently respond to changing requirements, even late in the life cycle.

Teamwork: managers, customers, and developers are all equal partners in a collaborative team. XP implements a simple, yet effective environment enabling teams to become highly productive. The team self-organizes around the problem to solve it as efficiently as possible.

Simplicity: XP keeps design simple and entrusts development of clean and clear code.

The most surprising aspect of Extreme Programming is its simple rules. Extreme Programming is a lot like a jig saw puzzle. There are many small pieces. Individually the pieces make no sense, but when combined together a complete picture can be seen. The rules may seem awkward and perhaps even naive at first, but are based on sound values and principles.

7.2.1 XP rules

We list all the XP rules focusing only on those that can be a little unclear and stressing those adopted by PariPari.

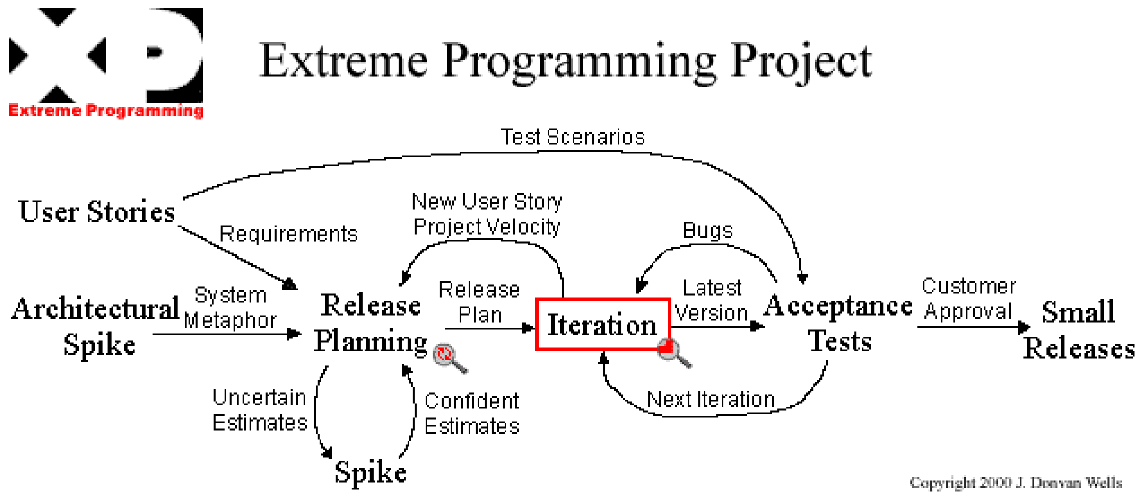


Figure 7.2: eXtreme Programming Project.

Planning

User stories are written. User stories serve the same purpose as use cases but are not the same. They are used to create time estimates for the release planning meeting. They are also used instead of a large requirements document.

Release planning creates the release schedule. A release planning meeting is used to create a release plan, which lays out the overall project. The release plan is then used to create iteration plans for each individual iteration. It is important for technical people to make the technical decisions and business people to make the business decisions. Everyone can commit to the release schedule.

Make frequent small releases. The development team needs to release iterative versions of the system to the customers often.

The project is divided into iterations. Iterative Development adds agility to the development process.

Iteration planning starts each iteration. An iteration planning meeting is called at the beginning of each iteration to produce that iteration's plan of programming tasks.

In PariPari, everybody participates in the scheduling task. This objective is achieved exploiting the hierarchical structure of the development group. Team leaders, Plug-in Leaders and the Project leader join the planning meeting considering the developers' opinions. The development proceeds by small frequent releases, and each plug-in leader assigns only small tasks to developers.

This is the most important behavior adopted in PariPari. Students rarely succeed in long term scheduling. This may have only limited functionalities, but it does work, and can be incrementally extended in small, simple steps.

Management

Give the team a dedicated open work space. Communication is very important: open space allows more communication paths to the team.

Set a sustainable pace. A sustainable pace helps planning releases and iterations and avoiding from getting into a death march (Adding more people is a bad idea when a project is already late) .

A stand up meeting starts each day. Communication among the entire team is the purpose of the stand up meeting.

The Project Velocity is measured. The project velocity is a measure of how much work is getting done on the project. To measure the project velocity it is enough add up the estimates of the user stories that were finished during the iteration.

Move people around. People have to move around to avoid serious knowledge loss and coding bottle necks. If only one person can work in a given area and that person leaves the project's progress reduced to a crawl.

Fix XP when it breaks. XP Rules are a good start, but it is possible to change what does not work

PariPari project does not have its own lab. But we try to set up something like a virtual open space for every developer. A PariPari-wide GoogleGroup allows students to talk to each other. This device provides a very good communication method: often senior developers working on different plug-ins mentor younger students. Moreover, each team shares knowledge using its own GoogleGroup. A meeting involving all leaders is kept every two weeks to check the progresses. To deal with the high turn over rate each team has a vertical structure, as explained above. Teams are formed by senior and junior students to ease the passage of notions and avoid a dramatic team death. In fact, to start working on a project it is often more effective to have a few words with its developers than to read directly its code and documentation.

Great importance has been attributed to working pace. Milestones and deadlines are always decided by consensus. This is particularly important because all the developers of PariPari have many other commitments in life.

Design

Simplicity. A simple design always takes less time to finish than a complex one.

Choose a system metaphor.

Use CRC cards for design sessions. Use Class, Responsibilities, and Collaboration (CRC) Cards to design the system as a team. The greatest value of CRC cards is to allow people to break away from the procedural mode of thought and more fully appreciate object technology.

Create spike solutions to reduce risk. A spike solution is a very simple program to explore potential solutions.

No functionality is added early. Keep the system uncluttered with extra stuff.

Refactor whenever and wherever possible. Refactor mercilessly keeps the design simple and avoids needless clutter and complexity. Keeping code clean and concise to easily understand, modify, and extend it.

The most important principle in PariPari is the K.I.S.S.[1] principle (as stated by Clarence Leonard “Kelly” Johnson). Every design or code has to be written in the simplest way. Once the simplest step is reached, incrementally, features are added. However, to axxomodate future extendibility, very simple projects are rarely possible. We moved most of the design complexity into the high level specification of PariPari, keeping the project extremely modular and tasks for individual developers extremely simple and clean. Frequent refactoring (particularly after spike solutions) also contributes to a clean code base.

Coding

The customer is always available. One of the few requirements of XP is to have the customer available.

Code must be written to agreed standards. Code must be formatted to agreed coding standards. Coding standards keep the code consistent and easy for the entire team to read and refactor.

[1]K.I.S.S. is an acronym for the design principle “keep it simple and stupid”, most commonly read as the backronym “keep it simple, stupid!”, or sometimes “keep it short and simple”.The K.I.S.S. principle states that simplicity should be a key goal in design, and that unnecessary complexity should be avoided.

Code the unit test first. Creating tests first, before the code, yields easier and faster code creation.

All production code is pair programmed. All code to be sent into production is created by two people working together at a single computer.

Only one pair integrates code at a time. Without controlling integration developers test their code and integrate it believing all is well. But because of parallel integration there is a combination of source code which has not been tested together before. Integration problems happen without detection.

Integrate often. Developers should be integrating and committing code into the code repository every few hours, when ever possible.

Set up a dedicated integration computer. A single computer dedicated to sequential releases works really well when the development team is co-located. This computer acts as a physical token to control the release process.

Use collective ownership. Collective Ownership encourages everyone to contribute new ideas to all segments of the project. Any developer can change any line of code to add functionality, fix bugs, improve designs or refactor. No single person becomes a bottle neck for changes.

In PariPari we use a modified pair programming paradigm. To successfully exploit the testing procedure and the power of pair programming we organized developers so that they test each other's code. To be more precise, one developer writes unit tests on code developed by the other and viceversa. Every time a single piece of code or test is complete (and hence can be successfully compiled) it is submitted to our SVN server.

Testing

All code must have unit tests.

All code must pass all unit tests before it can be released.

When a bug is found tests are created.

Acceptance tests are run often and the score is published.

A lot of effort is spent in testing. The next Section discusses this in greater detail.

7.3 Test Driven Development

Test-driven development (TDD) is a software development technique that relies on the repetition of a very short development cycle: first the developer writes a failing automated test case that defines a desired improvement or new function, then he produces code to pass that test and finally refactors the new code to acceptable standards.

1. **Add a test.** In test-driven development, each new feature begins with writing a test. This test must inevitably fail because it is written before the feature has been implemented. To write a test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through use cases and user stories that cover the requirements and exception conditions. This could also imply a variant, or modification of an existing test. This is a differentiating feature of test-driven development versus writing unit tests after the code is written: it makes the developer focus on the requirements before writing the code, a subtle but important difference.

2. **Run all tests and see if the new one fails.** This validates that the test harness is working correctly and that the new test does not mistakenly pass without requiring any new code. This step also tests the test itself, in the negative: it rules out the possibility that the new test will always pass, and therefore be worthless. The new test should also fail for the expected reason. This increases confidence (although it does not entirely guarantee) that it is testing the right thing, and will pass only in intended cases.

3. **Write some code.** The next step is to write some code that will cause the test to pass. The new code written at this stage will not be perfect and may, for example, pass the test in an inelegant way. That is acceptable because later steps will improve and hone it. It is important that the code written is only designed to pass the test; no further (and therefore untested) functionality should be predicted and 'allowed for' at any stage.

4. **Run the automated tests and see them succeed.** If all test cases now pass, the programmer can be confident that the code meets all the tested requirements. This is a good point from which to begin the final step of the cycle.

5. **Refactor code.** Now the code can be cleaned up as necessary. By re-running the test cases, the developer can be confident that refactoring is not damaging

any existing functionality. The concept of removing duplication is an important aspect of any software design. In this case, however, it also applies to removing any duplication between the test code and the production code — for example "magic" numbers or strings that were repeated in both, in order to make the test pass in step 3.

Starting with another new test, the cycle is then repeated to push forward the functionality. The size of the steps should always be small, with as few as 1 to 10 edits between each test run. If new code does not rapidly satisfy a new test, or other tests fail unexpectedly, the programmer should undo or revert in preference to excessive debugging. Continuous Integration helps by providing reversible checkpoints. When using external libraries it is important not to make increments that are so small as to be effectively merely testing the library itself, unless there is some reason to believe that the library is buggy or is not sufficiently feature-complete to serve all the needs of the main program being written.

7.3.1 PariPari's TDD

Very soon — when the number of students involved in the project increased over the initial five units— we understood that a strict code writing policy was needed. We adopted TDD because of the aforementioned features and for other two important PariPari-related reasons.

Test writing is relatively independent from code writing New developers require considerably more time and familiarity with PariPari to learn how to write good production code than to learn how to write good test code (assuming they are given high level specifications). Thus, an excellent way to "learn the ropes" of PariPari is to immediately start writing test code under the guidance of senior students.

TDD enforces a very good object oriented programming style Since everybody writes tests, everybody knows that object oriented code is mandatory for good quality tests. This enforces the adoption of an OOP style with all its attendant benefits.

Chapter 8

Conclusions

This work presents PariPari, a multifunction, extensible peer to peer network and development framework. In particular we detailed the design and implementation process (including the management of the human resources involved in the project).

Network Project and Development In the first part we detailed the current situation of the plug-ins. The operating system layer is ready. The main plug-ins, PariCore, PariConnectivity, PariStorage provide all the functionalities needed for running PariPari stand-alone and most of the functionalities needed for the networking. Hence, everybody can develop simple, working, plug-ins for our platform. We started building up our platform from these plug-ins following a bottom-up strategy. These plug-ins are invisible for the final users and maybe not very appealing for the developers. Nevertheless, in this way, we can effectively test our design and provide a good base for the other plug-ins.

The plug-ins needed to compete with state of the art file-sharing software. Mulo and Torrent implement all the functionalities offered by all our competitors. Torrent, besides, is the only torrent client featuring all those functionalities. These plug-ins, totally compatible with the existing eMule and BitTorrent networks, should let us gain popularity among the users. We expect filesharing will generate a large initial user base, allowing us to gain a sufficient critical mass to bootstrap other services (e.g., distributed backup). During the development of these plug-ins we experienced a strange information asymmetry. Both plug-ins deal with large established file-sharing networks but the documentation status of such networks is very different: while the BitTorrent protocol is very well described, the eMule protocol (derived by the eD2K protocol) has only outdated and fragmentary documentation. In this situation we made a large use of reverse engineering techniques.

Our net oriented plug-ins are ready and are under heavy testing. PariDHT and

Pari(Distributed)Storage are completed. We are testing them to remove any bugs or oddities. Since they will be used frequently, we are optimizing them to offer to the user a low latency experience using PariPari. Besides, we are also working to increase performance of these plug-ins and to incorporate advanced functionalities such as load balancing and (multi attribute) range queries. For these plug-ins it was crucial to follow the `K.I.S.S.` principle: only in this way, we could begin early on to implement other plug-ins using PariDHT and Pari(Distributed)Storage features.

PariWeb and PariCredits are almost ready: they offer very simple capabilities but we are working to enhance them with new interesting features (such as a PHP engine). Nevertheless, they are fully usable by other plug-ins and users.

Along with all the aforementioned plug-ins we have developed some "service-code": PariSync and DiESeL. These two plugins are not technically in the Operating System layer, but they are transparent to the end-user and provide their services to a number of other plug-ins. The first reserved us many surprises. Initially, it was meant to provide PariStorage with a method to keep a consistent time over the network but during the design phase we understood the complexity of this task. An interesting result is that we can piggyback the traffic needed to keep the network synchronized on the DHT. DiESeL, is a library providing basic services to manage a distributed server. We are still working on DiESeL to make it simpler to use by developers and to provide more useful features. We are spending considerable effort on this library because it will be the core of many, if not most, future (third-party) plug-ins.

Finally, plug-ins in their initial stages are under heavy development after an intense design phase. Hence, their interfaces are ready and other plug-ins can use them as stubs.

The main lesson learnt from this project is the importance of a careful, well-planned design phase. In such a large upgradable project it is hard but crucial to contemplate any possibility and to evaluate the effects of any possible dependency. Insufficient planning will invariably lead to conflicts within the code base and to a large amount of "wasted" code. A modular approach provides a partial solution.

Code and Workforce Management The second part of the thesis describes how we manage the students who form the workforce of PariPari, and which strategies and tools we adopted to help them in their work. We analyzed many different aspects of the human resources problem in PariPari. Our experience can be summarized in four main lessons.

The first is that the best way to know a code developer is through a personal relationship. Unfortunately this is often impossible. Delegating responsibility to experienced, motivated leaders is usually sufficient to ensure the project's continued success. Most other methods, like requesting frequent project reports, offer few marginal advantages.

The second lesson is that the only strategy to run a large project with low leverage on workers is to share decisions and schedules. This imposes a trade-off between time elapsed and hit rate of the deadlines. Seeking consensus on deadlines yields a marked increase in their hit rate (and thus in the good coordination of the whole project), at the expense of a somewhat increased time necessary to complete individual tasks.

A third, crucial lesson we re-discovered is that the best way to motivate a person is to give confidence - and that an excellent confidence building strategy is to gradually increase responsibilities as a reward for continued success and effort. Confident, competent leaders also instill greater confidence in the rest of the workforce, and provide a role model pushing everyone towards constant self-improvement.

The final, and perhaps most important, lesson is that it is possible to run a large software project relying only on a poorly trained workforce, even when one has only low leverage on it. The most important key to success is to avoid rigid schemes and to adapt to each individual situation and person, trying to identify (often hidden) potential and make full use of it.

Bibliography

[aeo] AEOLUS PROJECT. `http://aeolus.ceid.upatras.gr/`.

[Ake70] George A. Akerlof. The market for "lemons": Quality uncertainty and the market mechanism. *The Quarterly Journal of Economics*, 84(3):488–500, 1970.

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

[Bas02] B. M. Bass. *Cognitive, social, and emotional intelligence of transformational leaders.* Mahwah, NJ, Lawrence Erlbaum Associates, 2002.

[BBK02] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 205–217, New York, NY, USA, 2002. ACM.

[BBMP09] P. Bertasi, M. Bonazza, N. Moretti, and E. Peserico. Parisync: Clock synchronization in p2p networks. In *Precision Clock Synchronization for Measurement, Control and Communication, 2009. ISPCS 2009. International Symposium on*, pages 1–6, Oct. 2009.

[Ber00] Jean-Marc Berthaud. Time synchronization over networks using convex closures. *IEEE/ACM Trans. Netw.*, 8(2):265–277, 2000.

[BOW05] J. Fadel B. Ogden and B. White. Ibm system z9 109 technical introduction, 2005.

[BPoG51] Robert Stephenson Smyth Baden-Powell Baden-Powell of Gilwell. *Baden-Powell's scouting for boys.* C. A. Pearson, London :, 26th ed. / with an introd by lord rowallen, chief scout of the british commonwealth and empire. edition, 1951.

[BSFK06] S. Bianchi, S. Serbu, P. Felber, and P. Kropf. Adaptive load balancing for dht lookups. In *Computer Communications and Networks, 2006. ICCCN 2006. Proceedings.15th International Conference on*, pages 411–418, Oct. 2006.

[bug] Bugzilla. `http://www.bugzilla.org/`.

[CHXY08] Zhi Chen, Guowei Huang, Jing Dong Xu, and Yang Yang. Adaptive load balancing for lookups in heterogeneous dht. In *Embedded and Ubiquitous Computing, 2008. EUC '08. IEEE/IFIP International Conference on*, volume 2, pages 513–518, Dec. 2008.

[Com79] Douglas Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11:121–137, 1979.

[DDD+98] Joan Daemen, Joan Daemen, Joan Daemen, Vincent Rijmen, and Vincent Rijmen. Aes proposal: Rijndael, 1998.

[DHS84] Danny Dolev, Joe Halpern, and H. Raymond Strong. On the possibility and impossibility of achieving clock synchronization. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 504–511, New York, NY, USA, 1984. ACM.

[DHS07] Ivan Dedinski, Alexander Hofmann, and Bernhard Sick. Cooperative keep-alives: An efficient outage detection algorithm for p2p overlay networks. In *P2P '07: Proceedings of the Seventh IEEE International Conference on Peer-to-Peer Computing*, pages 140–150, Washington, DC, USA, 2007. IEEE Computer Society.

[Dub] Baptiste Dubuis. bitext. `http://sourceforge.net/projects/bitext/`.

[ecl] Eclipse. `http://www.eclipse.org/`.

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

[GDB05] Ashish Gupta, Peter Dinda, and Fabian Bustamante. Distributed popularity indices. In *in Proceedings of ACM SIGCOMM*, 2005.

[GG97] Volker Gaede and Oliver Gï£¡nther. Multidimensional access methods. *ACM Computing Surveys*, 30:170–231, 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[goo] Google Groups. `http://groups.google.com`.

[GRS99] David Goldschlag, Michael Reed, and Paul Syverson. Onion routing. *Communications of the ACM*, 42:39–41, 1999.

[GSBK04] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher. Adaptive replication in peer-to-peer systems. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 360–369, 2004.

[hsq] HSQLDB. `http://hsqldb.org/`.

[IMRV97] Piotr Indyk, Rajeev Motwani, Prabhakar Raghavan, and Santosh Vempala. Locality-preserving hashing in multidimensional spaces. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 618–625, New York, NY, USA, 1997. ACM.

[jsp] JSpeex. `http://jspeex.sourceforge.net/`.

[jsr] JSR 284. `http://jcp.org/en/jsr/detail?id=284`.

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

[KS03] M. Frans Kaashoek and Ion Stoica, editors. *Peer-to-Peer Systems II, Second International Workshop, IPTPS 2003, Berkeley, CA, USA, February 21-22,2003, Revised Papers*, volume 2735 of *Lecture Notes in Computer Science*. Springer, 2003.

[LLW08] Christoph Lenzen, Thomas Locher, and Roger Wattenhofer. Clock synchronization with bounded global and local skew. In *FOCS '08: Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 509–518, Washington, DC, USA, 2008. IEEE Computer Society.

[LMS$^+$97] Michael Luby, Michael Mitzenmacher, Amin Shokrollahi, Daniel Spielman, and Volker Stemann. Practical loss-resilient codes. In *In Proceedings of the 29th annual ACM Symposium on Theory of Computing*, pages 150–159, 1997.

[LS86] Leslie Lamport and P M Melliar Smith. Byzantine clock synchronization. *SIGOPS Oper. Syst. Rev.*, 20(3):10–16, 1986.

[LW05] Karim R. Lakhani and Robert G. Wolf. *Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects.* 2005.

[Mac08] T. Toriyama Machizawa, A. Iwawma. Software-only implementations of slave clocks with sub-microsecond accuracy. In *Precision Clock Synchronization for Measurement, Control and Communication, 2008. ISPCS 2008*, pages 17–22, 2008.

[MKKB89] Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Probabilistic clock synchronization. In *Distributed Computing*, pages 146–158, September 1989.

[MM02] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65, London, UK, 2002. Springer-Verlag.

[Moc87] P.V. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343.

[nc] The GNU Netcat project. `http://netcat.sourceforge.net/`.

[PAK07] Himabindu Pucha, David G. Andersen, and Michael Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *in Proc. 4th USENIX NSDI*, page 2007, 2007.

[Pes] E. Peserico. P2P Economies, SIGCOMM: Special Interest Group on Data Communications, 2006.

[PLGS04] Adina Crainiceanu Prakash, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram. Querying peer-to-peer networks using p-trees. In *In WebDB*, pages 25–30, 2004.

[RCFB07] Weixiong Rao, Lei Chen, Ada Wai-Chee Fu, and YingYi Bu. Optimal proactive caching in peer-to-peer network: analysis and application. In *CIKM '07: Proceedings of the sixteenth ACM conference on Conference* *on information and knowledge management*, pages 663–672, New York, NY, USA, 2007. ACM.

[RD01] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.

[RFS$^+$01] Sylvia Ratnasamy, Paul Francis, Scott Shenker, Richard Karp, and Mark Handley. A scalable content-addressable network. In *In Proceedings of ACM SIGCOMM*, pages 161–172, 2001.

[RMK$^+$96] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address Allocation for Private Internets. RFC 1918 (Best Current Practice), February 1996.

[SGG02] Stefan Saroiu, Krishna P. Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. January 2002.

[SHLnLZ07] Ahmed Sobeih, Michel Hack, Zhen Liu, and null Li Zhang. Almost peer-to-peer clock synchronization. *Parallel and Distributed Processing Symposium, International*, 0:21, 2007.

[Sho06] Amin Shokrollahi. Raptor codes. In *IEEE Transactions on Information Theory*, pages 2551–2567, 2006.

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

[SMK$^+$01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.

[spe] Speex. `http://speex.org/`.

[svn] SubVersion. `http://subversion.tigris.org/`.

[Szy02] Michal Szymaniak. A dns-based client redirector for the apache http server. Master's thesis, Vrije Universiteit, Amsterdam, The Netherlands, July 2002.

[Tan96] Andrew S. Tanenbaum. *Computer networks (3rd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[tes] AEOLUS PROJECT testbed. `http://aeolus.cs.upb.de/`.

[WS06] Xinfa Wei and Kaoru Sezaki. Dhr-trees: A distributed multidimensional indexing structure for p2p systems. In *ISPDC '06: Proceedings of the Proceedings of The Fifth International Symposium on Parallel and Distributed Computing*, pages 281–290, Washington, DC, USA, 2006. IEEE Computer Society.

[WXL07] Feng Wang, Yongqiang Xiong, and Jiangchuan Liu. mtreebone: A hybrid tree/mesh overlay for application-layer live video multicast. In *ICDCS '07: Proceedings of the 27th International Conference on Distributed Computing Systems*, page 49, Washington, DC, USA, 2007. IEEE Computer Society.