Alma Mater Studiorum · Università di Bologna

CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

# DESIGN AND IMPLEMENTATION OF AN ANONYMOUS PEER-TO-PEER IAAS CLOUD

Tesi di Laurea in Sicurezza delle Reti

Relatore:
GABRIELE D'ANGELO

Correlatori:
MORENO MARZOLLA
STEFANO ZACCHIROLI

Presentata da:
MICHELE AMATI

Sessione III
Anno Accademico 2013-2014

# Sommario (Italian)

In questo lavoro tratterò del problema della privacy e dell'anonimato nel mondo del Cloud Computing. Un mondo che recentemente ha conosciuto una considerevole espansione e non riguarda più solo le grandi aziende, ma anche tutte le persone comuni che abbiano un computer o anche solo uno smartphone. Forse non ne siamo tutti consapevoli, ma quando inviamo una e-mail attraverso una Web Mail, quando il nostro smartphone esegue il backup automatico delle foto sul nostro account (Google, Apple, o Microsoft che sia), quando usiamo una qualsiasi Web Application, noi stiamo sfruttando dei servizi Cloud. Servizi dei quali, spesso e volentieri, non sappiamo praticamente nulla e ai quali affidiamo ciecamente i nostri dati. Possiamo essere sicuri che il Cloud Service Provider di turno non si prenda troppe libertà? Che la confidenzialità dei nostri dati non venga, in un modo o nell'altro, violata? Domande avvalorate dalla recente scoperta delle attività di 'spionaggio' su larga scala ad opera della National Security Agency americana [9][10].

L'anonimato è un modo per raggiungere la privacy, e non è chiaramente un'invenzione recente, è già stato applicato in passato a cose fondamentali come ad esempio le comunicazioni. Più il Cloud diventa un'esigenza, più la possibilità di avere un servizio cloud anonimo diventa importante. Ed eccoci arrivati al punto, in questo lavoro partirò da un software prototipale per la costruzione di un'infrastruttura cloud peer-to-peer[5], e farò in modo di renderlo anonimo attraverso l'utilizzo delle reti di anonimizzazione, come ad esempio Tor[14]. Il risultato sarà dunque un prototipo in grado di creare un'infrastruttura per servizi cloud nella quale non solo gli utenti saranno anonimi, ma anche tutte le singole componenti della rete (Peers) non si conosceranno se non attraverso indirizzi fittizi.

Procederò nel seguente modo: inizialmente mostrerò lo stato dell'arte per quanto riguarda il Cloud Computing, soffermandomi sulle differenze tra le architetture più centralizzate adottate dai grandi vendor del settore, e quella distribuita del prototipo a mia disposizione. Successivamente farò un confronto tra le reti di anonimizzazione esistenti, approfondendone il più possibile il funzionamento. A questo riguardo faccio presente che non si tratta di 'strumenti' magici che proteggono la nostra identità sempre e comunque. Vanno capiti ed usati con giudizio, altrimenti l'unica cosa che si ottiene è un calo prestazionale, sì, perché a seconda del livello di anonimato desiderato c'è un prezzo da pagare in termini di risorse e questo porta ad un degrado più o meno significativo delle prestazioni.

Le prestazioni saranno appunto un elemento che valuterò dopo aver apportato le necessarie modifiche al prototipo, confrontando quelle della versione originale con quelle della versione anonima. I test che lancerò consisteranno nell'esecuzione del prototipo su un certo (grande) numero di macchine, e nella valutazione dei tempi di esecuzione di alcune operazioni. Per avere un numero sufficientemente alto di macchine mi rivolgerò ai servizi di Amazon EC2[28]. Attraverso uno script automatizzerò la creazione delle istanze (macchine), l'avvio dei prototipi, la chiamata alle varie API interessate dai test e tutte le altre operazioni necessarie. I risultati sono esposti al capitolo 4 nella sezione 4.3.

# Introduction

Cloud services are becoming ever more important for everyone's life. Cloud storage? Web mails? Yes, we don't need to be working in big IT companies to be surrounded by cloud services. Another thing that's growing in importance, or at least that should be considered ever more important, is the concept of privacy. The more we rely on services of which we know close to nothing about, the more we should be worried about our privacy. In this work, I will analyze a prototype software based on a peer to peer architecture for the offering of cloud services[5], to see if it's possible to make it completely anonymous, meaning that not only the users using it will be anonymous, but also the Peers composing it will not know the real identity of each others. To make it possible, I will make use of anonymizing networks like Tor[14].

I will start by studying the state of art of Cloud Computing, by looking at some real example, followed by analyzing the architecture of the prototype, trying to expose the differences between its distributed nature and the somehow centralized solutions offered by the famous vendors. After that, I will get as deep as possible into the working principle of the anonymizing networks, because they are not something that can just be 'applied' mindlessly. Some de-anonymizing techniques are very subtle so things must be studied carefully.

I will then implement the required changes, and test the new anonymized prototype to see how its performances differ from those of the standard one. The prototype will be run on many machines, orchestrated by a tester script that will automatically start, stop and do all the required API calls. As to where to find all these machines, I will make use of Amazon EC2[28] cloud services and their on-demand instances.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction to cloud computing and anonymizing networks

In this first chapter I'll introduce the basic concepts of Cloud Computing and of Anonymizing Networks. Both topics are quite wide and complex so I'll focus mainly on the aspects relevant to my work.

## 1.1 Cloud computing

As defined by the National Institute of Standards and Technology (NIST) the Cloud is: "A model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [1]

There are many types of Cloud services but they all share the same main idea that somewhere, somehow, there are computer-related resources that a user can use through a public interface to carry out his tasks. He can use them wherever he wants and whenever he wants paying only for what he uses. Extreme scalability makes it easy to follow the business needs allowing to get more resources almost instantly and release them as fast as they were got when they are not needed anymore, without having to deal with lot of expensive hardware laying unused and getting old. With this in mind it's easy to understand why the Cloud business has spread so much and it's still expanding. The ever improving quality and diffusion of Internet connections also played a big part in making this possible since Cloud without Internet is as impossible as real clouds without the sky.

So is Cloud the best solution for everyone? No, the Cloud has requirements and downsides too. The main requirement is that the user has to be able to reach it in a fast and reliable way, which means, he needs an appropriated Internet connection. Most of the other downsides derive by the fact that the user looses some of the control he has over his process making use of the Cloud. He can only trust the Cloud Service Provider will do as written in the *Terms of Service* (ToS). Most of the times he knows nothing about how the Cloud System really works behind the public interface, or even where that system and the user data are located. The more security is important for the user, the more this aspect of the Cloud gets problematic because there can't be security as long as one or more parts of the system are unknown. This issue is partially solved in Private and Hybrid Clouds described in this document at 1.1.2.

### 1.1.1 Types of service

Cloud services can be divided into three categories, ordered ascending by the level of abstraction from the physical resources:

**Infrastructure as a Service (IaaS).** The user gets remote access to the fundamental computing resources, e.g. processing, storage, network... This is the level that leaves the user the most freedom at the cost of an increased difficulty in handling the whole thing because it's up to him to install, configure and maintain software (and that may include the Operative System).

**Platform as a Service (PaaS).** The user gets a platform, which is a software that hides the lower infrastructure level and makes it easier for the user to build his own service like e.g. allowing software packages to be installed and configured in a graphical drag and drop way. The user has less freedom than in the IaaS model, but he hasn't to worry about updates, configurations and software compatibility.

**Software as a Service (SaaS).** The user gets the possibility to use a software remotely. He doesn't know what components form it, nor he knows anything about the lower infrastructure level on which it runs. There are many commonly used SaaS services we use every day, e.g. all the cloud storage services, all the web apps used for streaming music or for managing emails.

### 1.1.2 Deployment models

Cloud services can also be classified according to the users that can access them. They can be:

**Public.** Anyone who can pay for them can use them. It's the most widespread version.

**Private.** Usually built by big organizations, they can only be used by a restricted group of authorized user (that are often members of the organization that build them).

**Community.** Similar to the Private one but used by a community made of organizations that share the same concerns.

**Hybrid.** A mix of the three above, e.g. part of the Cloud could be Private and can be used to store/process sensitive data, while the rest could be Public.

There are many reasons why Clouds that are at least partially Private are a lot better at security:

- As I have already mentioned, having one or more parts of the system that we don't know how they are built or how they work, makes us unable to design an appropriate security system. This unknown factor is lower or zero in Private Clouds. It depends of who own, manage and operate the Cloud. A Private Cloud is not necessarily owned, managed and operated by the organization itself, if a third part does it, there might still be an unknown factor.

- The isolation level is maximum. Isolation level is a 'measure' of how good the shared resources are kept separated among the multiple users of the Cloud. In a Public Cloud our data might be stored on the same physical machine where the software of a malicious user is running, trying to break the virtual isolation to reach others data.

- The chance of getting malicious insiders is much lower since the access to the cloud is restricted.

### 1.1.3 Classic cloud vs P2P cloud

The architecture of a Cloud System can be designed in many different ways. Vendors often refuse to show how their systems are designed but we can look at one of the many Open Source projects to get an idea. An example is OpenStack.

"OpenStack is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter, all managed through a dashboard [...]" [2]. In the OpenStack architecture we can find the following components:

**Horizon.** Is the dashboard through which users can administrate the system by doing API-Calls.

**Heat.** Is a template-based orchestrator. Templates are files that describe specific cloud applications (resources and configurations).

**Nova.** The computing module.

**Glance.** Provides and manages disk images.

**Swift.** An unstructured data storage system.

**Neutron.** Manages networking and network interface devices.

**Cinder.** Manages persistent storage.

**Keystone.** Manages identities and permissions.

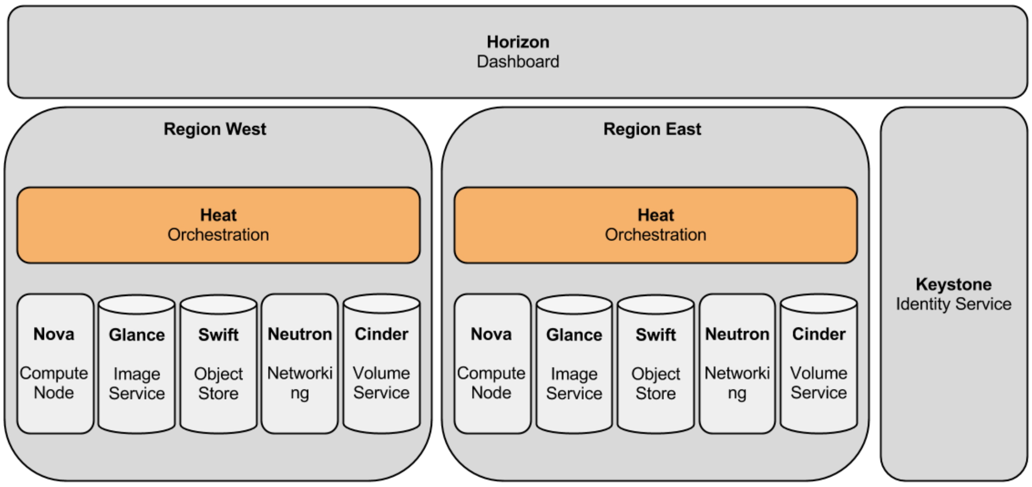

**Figure 1.1:** OpenStack Components Diagram [3]

In figure 1.1 we can also see the concept of *Region*, which is a way to point out the physical location of parts of the system. It can be useful if we want to be sure some sensitive data do not leave the country, or to implement fault tolerance.

OpenStack makes large use of virtualization, *Nova* works with virtual CPUs (VCPUs), *Neutron* uses virtual network interfaces, *Cinder* works with virtual disk volumes. Virtualization is essential for the 'rapid provision' property of the cloud.

To be able to use more than a physical machine, to create a cluster on which OpenStack runs, one or more nodes have to become *Cloud Controller*.

In the OpenStack documentation is written that: “The cloud controller provides the central management system for OpenStack deployments. Typically, the cloud controller manages authentication and sends messages to all the systems through a message queue.” [4] In other words, *Cloud Controllers* provide the required coordination.

Getting deep on how OpenStack (or other similar Cloud System) works is out of the scope of this work. I mentioned it just to be able to show the differences between a ‘classical’ architecture and a P2P architecture. The important points to keep in mind are: modularity, virtualization and hierarchy between the machines forming the system.

The concept of Peer to Peer (P2P) was born before the Cloud. P2P denotes those architectures where the software entities that take part to the system are somehow connected (e.g. Internet) and are ‘peer’, with no centralized control and/or hierarchy. How exactly this entities communicate and how can they work together to produce some kind of result is up to the specific application. P2P architectures usually offer great scalability, can make use of many low-power heterogeneous machines and are often great at fault tolerance but are also harder to design due to the different way coordination must be achieved. Recently some work has being done to design a Cloud system that uses a P2P architecture, from now on I’ll take as an example of such work the thesis *Progettazione e Sviluppo di un Sistema Cloud P2P* (Design and Implementation of a P2P Cloud System) [5] [6] by Michele Tamburini.

We will call *Node* or *Peer* an instance of the P2P Cloud System software running on a machine. There can be more than one node on a single physical machine but it would not make much sense in a real scenario. The nodes are connected by an overlay network[1] upon a standard routed network like the Internet. Coordination among nodes is achieved in a completely decentralized way, by decisions taken locally in each node with the local available

[1] Overlay Network: a network build on top of another network where links between nodes can be logical or virtual and correspond to paths on the ‘real’ underling network.

information. The users can request the cloud's resources through a public interface that aims to be as close as possible to the one offered by the 'standard' cloud systems.

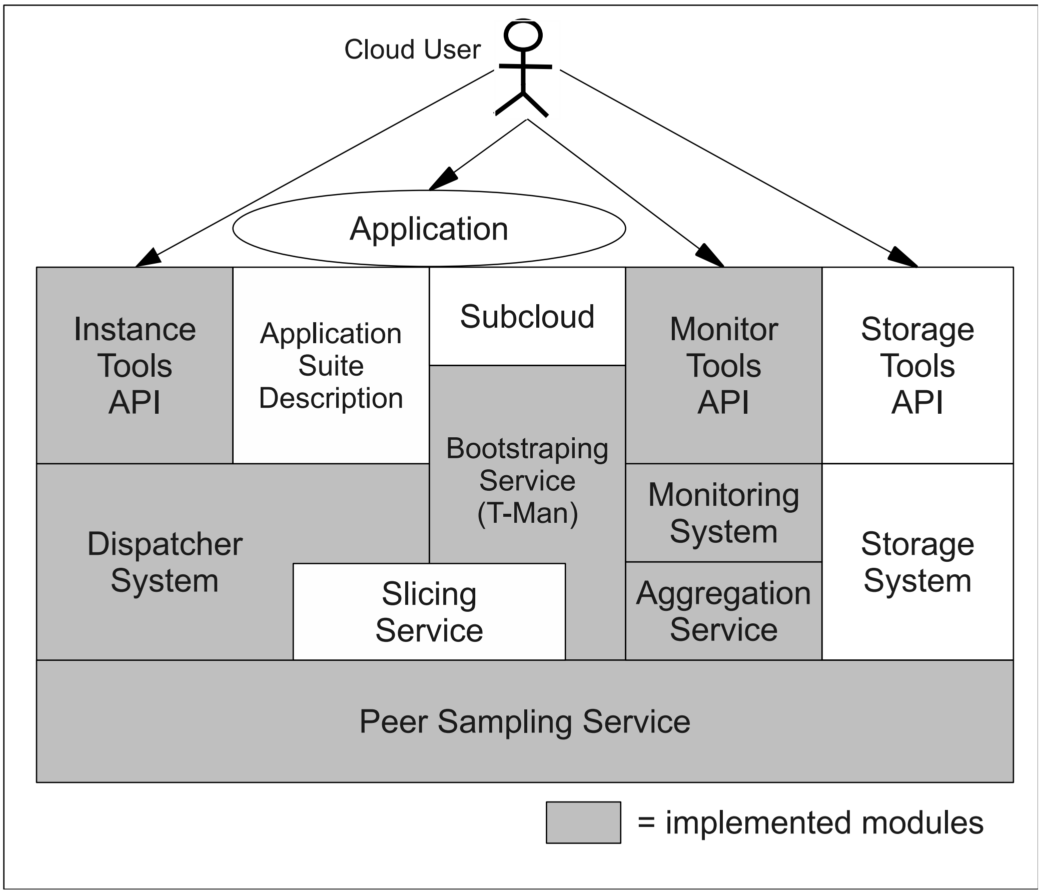

**Figure 1.2:** P2P Cloud System Architecture as designed by Michele Tamburini [7]. The white components are those that have not been implemented in the prototype attached to his thesis.

Let's take a closer look at his architecture, from the bottom to the top (Figure: 1.2) we have:

**Peer Sampling Service (PSS).** Since the number of nodes forming the cloud could be very high, it's not possible for each node to keep a list of the addresses of all the other nodes. This is where the PSS come in use. Every node keeps a partial view, a small list with the addresses of few other nodes. The PSS uses a gossip algorithm to periodically exchange information regarding the partial view with other nodes (contained in the partial view), and uses them to update the content of the local partial view itself. This way every partial view will behave like a random subset of all the nodes in the network. Given that the time between taking samples is high enough, picking a random node from the partial view will be like picking a random node from all the nodes in the network.

**Slicing Service.** It allows to get a slice (a subset) of nodes that match some kind of query, like e.g. those that are best at computing.

**Aggregation Service.** Using the PSS and a gossip algorithm, it allows each node to get global knowledge by exchanging local information with other nodes, e.g. if every node is assigned a number, it's possible for each node to compute the average of the values of all nodes in the network just by exchanging information with other nodes and changing their own value to the average of it and the value of the other node. Again, getting deep on how it works is not the purpose of this document, see *Gossip-Based Aggregation in Large Dynamic Networks* [8] for a more detailed description of this issue.

**Bootstraping Service.** It's used to get sub-clouds by creating overlay networks (different from those created by the PSS). It uses the T-Man gossip algorithm.

**System Modules.** The Monitoring System is a resource monitor, it knows how many free and busy resources we have. The Storage System takes care of the storage. The Dispatcher System is of some interest to us because among other tasks it handles and dispatches communications, something we will have to look at in deep when we'll talk about how to anonymize the system.

**Interface Modules.** It includes all the APIs that the user will be able to call to use the system.

## 1.2 Anonymizing networks

In this section, I'll talk about anonymity in the Internet world. What identify an entity on the Internet? What does it mean to be anonymous? Why should we want it, we have nothing to hide isn't it? We'll see Anonymizing Networks as tools that could help us reach a certain degree of anonymity and we'll use them later to make the P2P Cloud System anonymous.

### 1.2.1 Introduction to online identity and the importance of anonymity

Everything that's connected to the Internet can be uniquely identified by its IP address[2], it's like the Italian *Codice Fiscale*, or the USA *Social Security Number*, it's something we have just for being part of the system and it's actively required to communicate so we can't just throw it away. But that's not all, even if we find a way to hide it, our identity can still be inferred by what we do! Just think at this, you have succeeded in crafting a fake identity and the one you are talking to do not know who you really are, but then you inadvertently tell him something real about you and he starts to be suspicious because it does not fit well in your fake identity. He will start to pay much more attention and everything you do or even how you look will help him not only understanding that you're not who you said you are, but it might even be able to narrow it down until he finds your real identity.

So, we have to solve both problems, find a way to communicate without

[2] Actually, thanks to the *Network Address Translation (NAT)*, it is possible for two or more devices to have the same 'private' IP and still be able to communicate through the Internet. NAT maps each packet that passes through it (directed to the Internet) to a new packet with a different IP (and port numbers) using as the new IP a public unique one. Responses are also mapped back to the 'private' IP.

using our real identity, and behave in a way that will not give out any information that could betray us. In a system that tolerates anonymity it's not a problem if the other finds out that you are trying to hide, but there are also systems that would just 'kick you out'.

But why should we want anonymity? We are not spies or terrorists, so why should we care? There could be many reasons:

**Privacy.** It's not the same thing as anonymity but they are strictly related concepts. Having privacy means being able to do something without anyone knowing it. Being anonymous means being able to do something others will notice but will not be able to connect to you. From the user's (real identity) point of view anonymity imply privacy. The "Nothing to hide" argument is so frequently used when talking about privacy and anonymity that it even has a dedicated Wikipedia page [12]. We might tend to agree with that but think at this example by Adam D. Moore: "Imagine upon exiting your house one day you find a person searching through your trash painstakingly putting the shredded notes and documents back together. In response to your stunned silence he proclaims "you don't have anything to worry about - there is no reason to hide is there?"" [13]. I'm sure most would dislike such thing, and that's what happens every day with everyone's online activities.

**Freedom of speech.** Here in Italy, the freedom of speech is (most of the time) granted, but think at all the totalitarian regimes around the world where people risk their life if they say the 'wrong thing'.

**Anonymous complaints.** There could be situations in which we feel we should report or denounce something but we do not want to be involved.

**Hide physical location.** Given an IP address it's possible to know where the machine that's using it is physically located. This allows an attacker to physically harm that system by e.g. cutting the cables that bring it power or the Internet connection, or even to totally destroy it. In a military scenario this is totally possible. Hiding the real IP will prevent all of this.

**Ensure a safe future.** This is something most people forget to think of, actions that are perfectly legit and safe now are not granted to stay the same in the future! Are you totally sure that the opinion you are expressing now will not cause you problems, after some years, with the new 'Big Boss' whose opinion will be exactly the opposite of yours?

**Any kind of bad things.** Anonymity is a feature, it can be used for both good and bad purposes.

Recently it has emerged that the *American's National Security Agency* (NSA) has being spying on every citizen of the world in such a deep way that no one would have believed it to be possible [9]. They said it is for protecting themselves from terrorists, and that could be true, but is it all? And even if it is, is it right that everyone has to pay with his privacy for it to be possible? How do we know they are not spying on other countries stealing their industrial secrets? What if they 'data-mined' our private data and knew what we think and what we do and this way they were able to 'manipulate' us? That would be quite easy, it's the daily bread of those who do marketing. In a single question: Who watches the watchers? No one, or maybe similar agencies of other countries, but that's of no help to us.

Back to the main topic of this work, communications can already be anonymized, Internet browsing can already be anonymized but to be able to do anonymous computation, something like an anonymous cloud system is needed and P2P architectures are the best starting point because of their properties (no centralized control, all nodes are peers).

### 1.2.2 Basic principle behind anonymizing networks

For a connection[3] between two entities on the Internet to be possible, one must know the address of the other, and once the connection starts, the second one learns the address of the first one. A basic way to avoid revealing an entity's real identity is to use a relay. Consider three entities on the Internet called A, B, and C. If A wants to talk to B without revealing its identity it can connect to C and tell C to connect to B. C will act as a relay for the communication between A and B, so that B will think to be talking with C instead of A. This solves part of the problem, but still A needs to know B. Let's add a fourth node D. To protect itself but still be able to offer its service, B can create a fake identity, an alias that we will call 'ServiceB' and write it somewhere in a public area, something like: "To get ServiceB you have to ask to node D". This way, D will act as a relay exactly like C did. This is a very basic and weak anonymizing strategy, what if C or D were corrupted[4]? Real anonymizing networks are a lot more complex.

An interesting concept strictly related to how anonymizing networks works is that *"Anonymity Loves Company"* [11]. One may think that reaching anonymity means being as far as possible from anyone who can see or hear you but that's wrong! It's easy to spot someone who's isolated, much harder is to find someone dressed as normal as possible in a place full of people. With the way anonymizing networks work, it's possible to tell if a service is being accessed by someone (UserA) who's using an anonymizing network, and it's also possible to tell if someone else (UserB) is using them, but what makes them safe (or not) is the number of the anonymizing network users. The more they are, the harder it will be to tell if UserA and UserB are the same person.

[3] Here by 'connection' I mean a bidirectional communication. The majority of the connections over the Internet are bidirectional but there are also unidirectional connections that would not require the receiver to know the sender. Broadcast communications are also possible and they would not even require the sender to know the receivers.

[4] Corruption: The target looses its integrity because of unauthorized modifications that make it function in an unintended manner.

### 1.2.3 A comparison of existing anonymizing networks

At the time of writing, there are only two anonymizing networks famous enough to be worth looking at: *The Onion Router* (Tor) [14] and the *Invisible Internet Project* (I2P) [15]. They share many similarities but there are also some fundamental differences. I'll start with the things they have in common. Unfortunately they use different terminology to refer to the same concepts so it's useful to take a look at the comparison Table 1.1 kindly provided by the guys of I2P.

| **Tor** | **I2P** |
|---|---|
| Cell | Message |
| Client | Router or Client |
| Circuit | Tunnel |
| Directory | NetDB |
| Directory Server | Floodfill Router |
| Entry Guards | Fast Peers |
| Entry Node | Inproxy |
| Exit Node | Outproxy |
| Hidden Service | Eepsite or Destination |
| Hidden Service Descriptor | LeaseSet |
| Introduction Point | Inbound Gateway |
| Node | Router |
| Onion Proxy | I2P Tunnel Client * |
| Relay | Router |
| Rendezvous Point | Inbound Gateway + Outbound Endpoint * |
| Router Descriptor | RouterInfo |
| Server | Router |

**Table 1.1:** Comparison of Tor and I2P Terminology [16]. Items marked with '*' are not an exact match but more of a similar concept.

Both projects start from the idea presented at 1.2.2 of avoiding direct communications, by building a path of nodes through which making them pass. How they choose the nodes is the first notable difference between the two.

In Tor, a client (Onion Proxy - OP) that needs to build a path (Circuit) asks few special nodes (Directory Servers) for information about the nodes (Onion Router - OR) in the network, and then it chooses the ones whose declared performances better suit its needs. Directory Servers are a small group of well known servers[5] that collect signed state information from all the OR in the network and use them to create a sort of global view of the network (Directory) which OPs can fetch.

In I2P, the client (I2P Tunnel Client) asks special routers (Floodfill Router) that have access to a distributed network database (NetDb) which contains information equivalent to those of the Tor's Directory, information about the nodes (Router). The big difference is that any node that matches some basic performance criteria can and will automatically become a Floodfill Router making things a lot more distributed and decentralized than Tor's Directory Servers. Floodfill Routers collect signed state information from the routers in the network the same way Directory Servers do with the ORs. Once the path is created data can be sent. They both make use of encryption to protect data confidentiality and here we have another difference.

Tor uses a technique called *Onion Routing* to pack, encrypt and send data. The name is taken from the onion (vegetable) because of its layered structure. Every time a message (Cell) has to be sent, the OP encrypts it a number of times equal to the number of nodes it will cross, using a set of symmetrical keys created during the circuit creation. Each key is shared with one of the nodes in the path. It does it in reverse order so that the outer layer of encryption is the one encrypted with the key of the first node, the second outer layer uses the key of the second node and so on. Every time

[5] About trusting Directory Servers: "[...] directory servers must be synchronized and redundant, so that they can agree on a common directory. Clients should only trust this directory if it is signed by a threshold of the directory servers. [...] Tor only needs a threshold consensus of the current state of the network." [17]

the Cell passes through a node, a level of encryption is removed exposing the information needed to know what's the next hop. The last node removes the last level and exposes the unencrypted data. This way every node can only know information regarding the previous and the next hop and even if they are compromised they can't do much[6].

I2P uses *Garlic Routing.* It can be seen as an evolution of Onion Routing. The concept of layered encryption is almost the same, what changes is that the packet that is sent through the tunnel is not made of a single message like in Tor's Cells but it's a bundle of messages (Cloves) instead. All messages have to reach the end of the tunnel so there's no reason to keep them separated, this also makes it harder to carry out traffic analysis attacks[7].

Given that paths creation is hard and time consuming due to the latency and the multiple use of public key encryption[8], both Tor and I2P create them as soon as they can, before they are actually needed and in a transparent way to the user. Paths are valid for a certain amount of time then they expire and must be replaced. The same happens if they just break due to one or more failing nodes, or if the user wants to change them even if they are still valid and working. Many different connections can use the same path. Another big design difference is that Tor's Circuits are bidirectional so one circuit is enough to carry out a communication while I2P's tunnels are unidirectional so it needs at least an Inbound Tunnel and an Outbound Tunnel to communicate. This has many implications regarding security and performance but things gets really complicated and not very useful for my work so I'm not going to explain them here.

Another thing that's worth focusing on is how Tor and I2P handle the creations of anonymous services. At 1.2.2, I talked about the fact that, having two entities A and B, for A to be possible to connect to B, A must know

[6] Unless almost every node is compromised or the attacker controls both the entry and exit node.

[7] Attacks based on trying to infer information by looking at the data (patterns, timings) even if it's encrypted and unreadable.

[8] Used to create the set of shared symmetrical keys.

B. We will now see how B can use Tor or I2P to offer its services without exposing itself.

Tor calls them *Hidden Services*, anyone participating in the Tor network can create his own hidden services. If 'B' wants to host one it will do as follows:

1. B generates a public key pair that will identify his service.

2. B chooses some nodes in the network that will work as *Introduction Points* to its service, then it creates circuits to them. After that it creates a *Descriptor* containing information about the introduction points and its public key, and signs it with its private key. Descriptors are associated a string name in the form "XYZ.onion" where XYZ is a 16 character name derived from the public key.

3. B uploads this descriptor to the *Hidden Service Directory Servers* (HS-Dirs).

Now the service is advertised, to reach it 'A' needs to know the .onion address and will do as follows:

1. A downloads B's descriptor from the HSDirs.

2. A chooses a node in the network that will serve as a *Rendezvous Point* (RP) and builds a circuit to it[9].

3. A connects to one of B's introduction points and instructs it to tell B to meet A at the RP.

4. If B wants to talk to A, it will build a circuit to the RP.

5. A and B can now communicate through the RP.

[9] Actually, the RP is also provided with a *rendezvous cookie* by A, to be able to recognize B later. The same cookie is passed to B through one of its introduction points.

In I2P we have something similar but without the Rendezvous Point and the centralized (redundant) HSDirs. When the service takes the form of a web server it is called *EEPsite* but it's also possible to host other types of services. Things are a bit easier to understand compared to Tor because of the concept of the Inbound Tunnels. What Tor does by choosing introduction points and creating circuits to them, in I2P is done be creating one or more Inbound Tunnels. The node of the tunnel that's further away from where the service is hosted (Inbound Gateway) is like a Tor's introduction point. To advertise a service, the hosting node publishes a *LeaseSet* (like Tor's descriptors) in the NetDb through a Floodfill Router. Every LeaseSet is identified by an address made of 516 Base64[10] characters that can also be expressed in Base32 in the form "{52 chars}.b32.i2p". Anyone in the I2P network that wants to access the EEPsite have to know the address and use it to get the LeaseSet from the NetDb. It can then simply connect to one of the Inbound Tunnels listed in the LeaseSet through one of his Outbound Tunnels.

There are two big differences between Tor and I2P yet to be discussed. The first one regards what types of traffic can actually pass through them. Tor has been designed to work with *Transmission Control Protocol* (TCP) streams and can't in any way carry *User Datagram Protocol* (UDP) packets. UDP support has been theorized and would require some deep design change but the bigger problem is that it would be possible to tell if someone is using UDP and this would make UDP users easy to deanonymize because soon after the change, UDP users will be very few compared to the TCP users. I2P does not suffer from this problem and can carry both UDP and TCP. The last difference I'll describe is about the communications between someone who's inside the anonymizing network and another one who's outside, in the 'normal' Internet. In Tor, ORs can be configured to relay communications only with other nodes inside the network or also with anyone outside. Those who can relay communications with the outside Internet are called

[10] It is a binary-to-text encoding scheme. Binary data is divided into groups of six bits and than encoded using the Base64 index table [18].

*Exit Nodes*, it's up to every OR to decide if they want to be exit nodes or not[11]. In I2P, exit nodes (OutProxies) are special nodes running a different application, hosted by volunteers, and there are very very few of those. A standard I2P node only relay communications inside the I2P network. This makes communicating with the outside Internet very slow and unreliable, I2P's goal are intra-communications.

Back to the main purpose of this document, If I had to choose one for building the Anonymous P2P Cloud System I would choose Tor. It has a much larger pool of users and it has been studied a lot more than I2P so we can assume it's more secure. But the good news is, I don't really have to choose![12] Given that they both need connections to be 'routed' to a local proxy (luckily the same type of proxy) and that they both use string addresses that are handled entirely by the proxy, I can design the system to be independent from the specific anonymizing network. In Chapter 2, I'll give a better idea of how the system is designed.

[11] Being an exit node can be very dangerous because its real IP is what the outside Internet sees so if a Tor user does something bad the exit node is the one who'll be blamed.

[12] Unfortunately, during the implementation phase I've encountered a limitation in I2P support for SOCKS that made it impossible for the prototype to work with it without some important modification, so I'll just go on with Tor.

# Chapter 2

# Architecture of a P2P anonymous cloud system

Now that I have introduced the two fundamental components (anonymizing networks and P2P cloud architecture), we can start to see how they can be brought together to create a P2P cloud architecture in which the peers do not know the real identity of each other.

## 2.1 Recap of the P2P cloud system architecture

In order to facilitate the comprehension of the anonymizing process, I will now get back to Figure 1.2 and propose a different version of it, this time built around the *Communication Handler*[1] component. For now, I will just ignore the parts that were not implemented in the prototype.

Every module of the node that is interested in receiving messages, creates a personal *Network Manager* and registers it to the *Message Dispatcher* of the Communication Handler.

[1] In Figure 1.2 it's not shown but it's the core component that handles communications, almost every other component uses it.

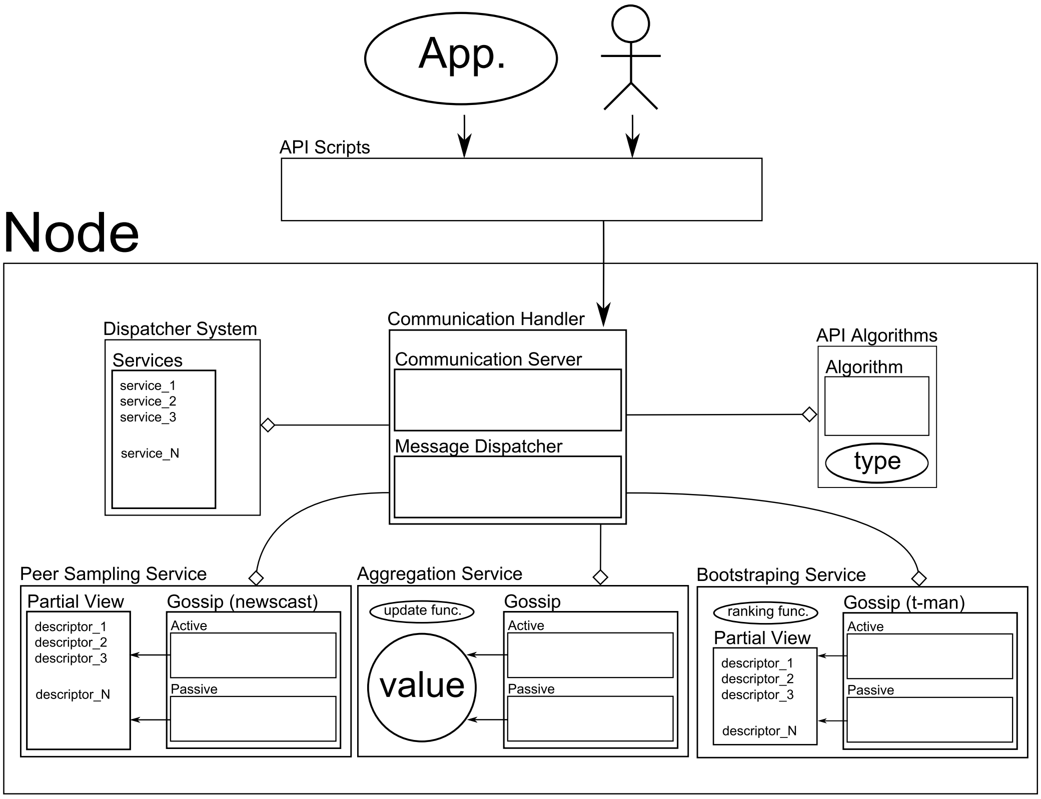

**Figure 2.1:** P2P Cloud System Architecture scheme focused on the communications.

As we can see in Figure 2.1, applications and users interact with the system through a set of APIs. The APIs send messages to the Communication Handler of a node that will function as an entry point for the cloud system. This node is not special, it's not important what node is chosen to be the entry point. Once the API message reaches the Communication Handler, it searches through the registered Network Managers and forwards the message to the appropriated one. In case of API messages, the appropriate module is the *Dispatcher System*.

In Figure 2.2 we can see a simple API call that has no consequences on the node, and just returns some information. It's also possible for API calls to alter the structure of the node they are sent to, by adding or removing modules (also called services). A good example of such behavior is the API

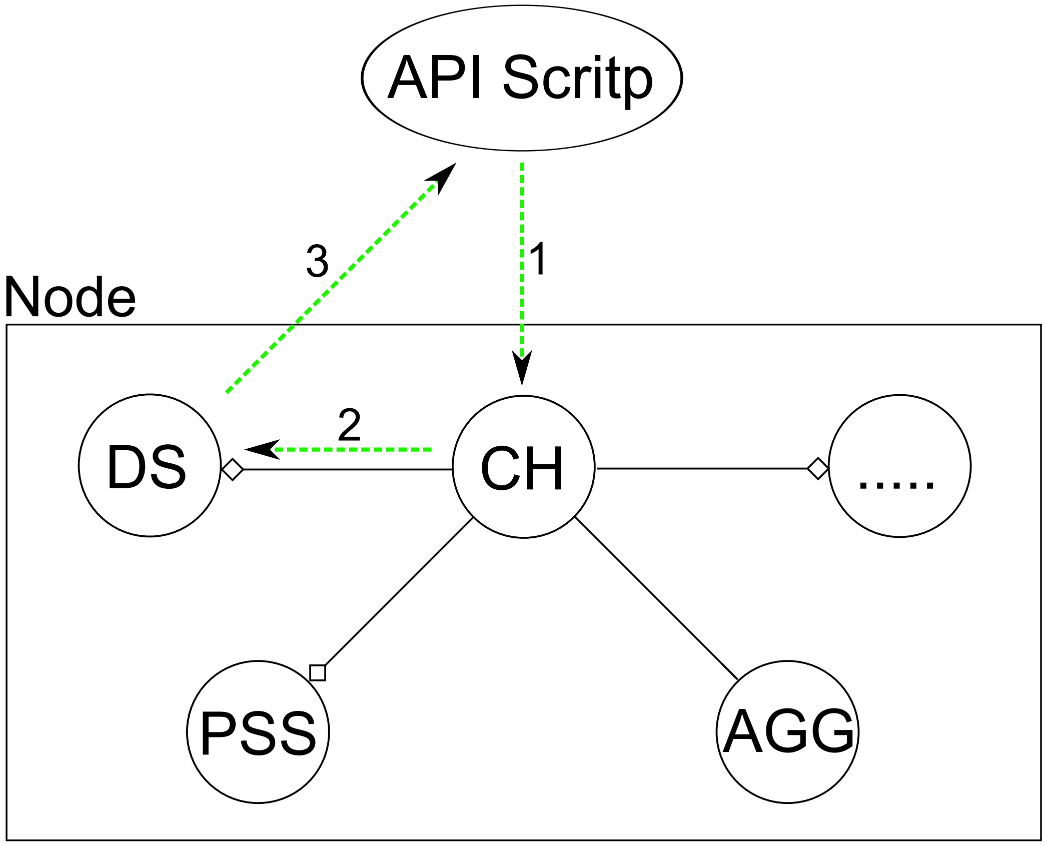

**Figure 2.2:** Example of the simplest API call possible. Communication Handler (CH), Dispatch System (DS), Peer Sampling Service (PSS), Aggregation Service (AGG).

*RunNodes*, that tries to create a sub-cloud with a certain topology[2] using the Bootstraping Service that it adds to the node.

API calls are communications between the nodes and something that is outside of the cloud. There are also a lot of communications between nodes of the cloud. In Figure 2.3 we can see the communication between two PSS modules of different nodes.

Intra-cloud communications represent the majority of the communications. They are mostly generated by the gossip algorithms that power the basic cloud services. By design, gossip algorithms need to frequently ex-

[2] The arrangement of the nodes in the sub-cloud. Depending on the way they are connected, they can form various topologies e.g. Star, Ring, Mash, Tree... [19]

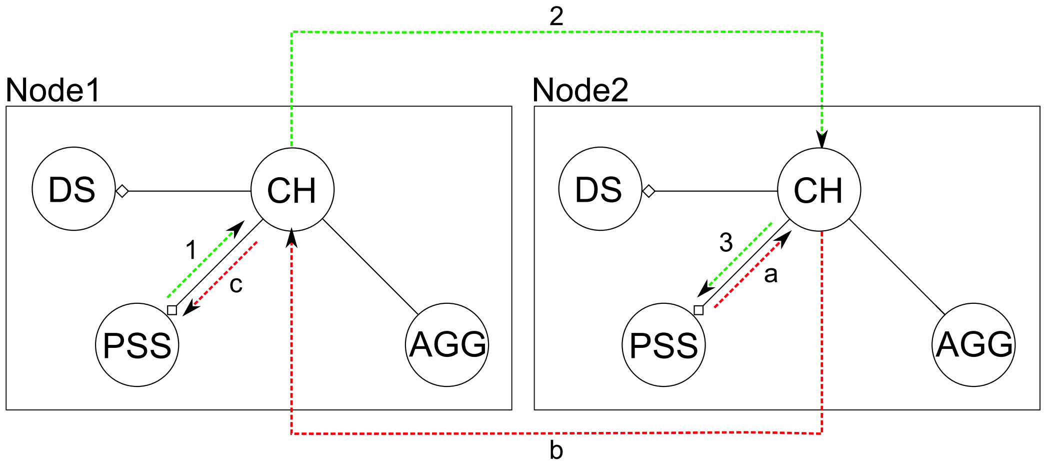

**Figure 2.3:** Communication flow between two PSS.

change (or at least push or pull) information. How frequently is not fixed, a reasonable range can go from less than a second to half a minute, it really depends on the specific algorithm and the desired performances, not to mention the performances of the means of communication (like the Internet connection speed and latency).

Every communication between two nodes follows the same pattern: local module –> local Communication Handler –> remote Communication Handler –> remote module. Replies, if needed, can use the same connection (the one initialized by the remote node) or can also take the form of a second communication, similar to the first one but in the reverse order. If it was not yet clear, the Communication Handler is the core point of every communication and that's why I'll focus on it to anonymize the system.

## 2.2 Towards anonymization (what needs to be changed)

So, once again, to anonymize the system we will have to look at every connection and make sure they are redirected to pass through the anonymizing network. We will also have to check that the data sent through those connections do not reveal anything that could help an attacker in the de-anonymization process.

### 2.2.1 Obvious problems (direct connections)

That's the easy part. Normally, this task would require to find every point in every module where a communication is needed and change it to use the anonymizing network. But wait, we have the Communication Handler where every connection leaves and enters the node so we only need to make one 'big' change there and we are done[3]!

For such a modification to be possible, we also need to make a little change at how the nodes identify themselves. Prior to this change, every node was identified by a *Name* and an *Address*. The Name is not that important, it is used to distinguish the nodes if there are two or more on the same machine (and for other minor things). The Address is the public real identity of the node, it can be directly an IP address or a domain name, and we can no longer use it if we want to make the system anonymous. From now on, every node is identified by a Name (unchanged) and a *Fake Address*.

One last thing. The communications originated by the APIs, used by the applications and the users, do not pass through a 'local' Communication Handler. So we have to reproduce the same changes we made to the Communication Handler to them, in order to make them use the anonymizing network.

### 2.2.2 Hidden problems (leaks)

Much harder to spot are the possible information leaks. They can be everywhere and do not necessarily be limited to a single communication. A communication can leak just a little useless piece of information, but lots of pieces can be brought together to get an ever increasing chance of successful

[3] Atually, we still need to look at every point in every module where connections are initialized to find possible leaks. See 2.2.2.

guessing. Information can also be obtained by combining the pieces of communication of different modules. Timing and communications order can also lead to leaks.

Before starting to look for leaks it is useful to define exactly what is our threat model and who could be the attacker. We are trying to design an architecture for anonymous computation, our main asset (that we must defend) is the anonymity of the nodes and of the users. The threats we are interested in are those caused by the possible flaws in the design, that if exploited would result in the de-anonymization of one or more entities of our system. We don't care if a node willingly gives out information. What we care is if there's something in the design that someone could use to obtain information that are not supposed to be exposed. Here I will also assume that the implementation will be perfect and bug free.

The attacker could be someone outside the cloud (user), or someone inside the cloud (malicious node). Given that there will be no centralization and peers will be anonymous, it will be very hard to detect and remove a malicious node so the chances of that happening is much higher than in other more standard clouds. I'm now going to list and analyze all the communications of the basic modules of the system and of the APIs, results are summed up in Table 2.1. While the basic modules (PSS, BS, AGG) could be analyzed in a purely theoretical way, the APIs are more implementation dependent so I'll use those that are implemented in the prototype. A brief description of the APIs (scripts and algorithms) will follow, but I suggest to see [5] for more details.

All the APIs have a *script* part and an *algorithm* part. The script part is the one executed outside the cloud and contains the instructions to start a communication with a node. The algorithm part is what's executed inside the node (thus inside the cloud) after it receives the corresponding script communication. Scripts names are lower case words separated by underscores, algorithms names contain the same words of the corresponding script but they use CamelCase[4] style.

| Module | Detail | Action | Loop | Leaks |
|---|---|---|---|---|
| API (script) | run_nodes | send / receive | NO | minor |
| API (script) | terminate_nodes | send / receive | NO | minor |
| API (script) | add_new_nodes | send / receive | NO | minor |
| API (script) | describe_instances | send / receive | NO | NO |
| API (script) | monitor_instances | send | NO | NO |
| API (script) | unmonitor_instances | send | NO | NO |
| API (algorithm) | RunNodes | send | NO | minor |
| API (algorithm) | TerminateNodes | send | NO | NO |
| API (algorithm) | AddNewNodes | send | NO | minor |
| API (algorithm) | DescribeInstances | send | NO | NO |
| API (algorithm) | MonitorInstances | send | NO | NO |
| API (algorithm) | UnmonitorInstances | send | NO | NO |
| PSS | gossip-active | send / receive | YES | medium |
| PSS | gossip-passive | send / receive | YES | medium |
| BS | gossip-active | send / receive | YES | minor |
| BS | gossip-passive | send / receive | YES | minor |
| AGG | gossip-active | send / receive | YES | NO |
| AGG | gossip-passive | send / receive | YES | NO |
| SS | gossip-passive | send / receive | YES | medium |
| SS | gossip-passive | send / receive | YES | medium |

**Table 2.1:** A list of all the communications of the system, with their type and threat level.

**run_nodes.** It connects to a node and asks it to create a sub-cloud with a certain *name* and a certain *size*. Then it waits for one response from the node that served as an entry point to the cloud, and for a number

[4] It consists of writing compound words such that each next word or abbreviation begins with a capital letter [20].

of other responses equal to *size* from the nodes that have formed the sub-cloud. The only information an attacker could get from this is the number of free nodes[5], because the system will return an error if there are not enough nodes to create the sub-cloud. An attacker could just keep trying bigger sizes until he gets the error. This assuming the system does not limit the number of nodes anyone can get.

**RunNodes.** Executed on a node, it uses the local PSS to get the required number of (free) nodes and then it tells them to join the sub-cloud by starting their BS modules. The only interesting information I can think of here, comes from the ability to check if a node is free or it's already part of a sub-cloud.

**terminate_nodes.** It connects to a node participating in a sub-cloud and asks it to remove one or more nodes from that sub-cloud. The leak-level of this, depends mostly on the verbosity of the errors. If the system returns an error in case the contacted node is not in a sub-cloud or/and if one or more nodes to remove are not there, the attacker could find out if a node is in use and what other nodes it is linked to.

**TerminateNodes.** It sends a message to the nodes to be removed, basically telling them that they have been removed, and also it sends messages to the remaining nodes telling them to update their views after the other nodes removal. I don't think there's anything that could leak here.

**add_new_nodes.** Almost like run_nodes, but instead of creating a new sub-cloud it adds nodes to an existing one. Same leaks implications.

**AddNewNodes.** Almost like RunNodes, but instead of creating a new sub-cloud it adds nodes to an existing one. Same leaks implications.

[5] Nodes that are not participating in any sub-cloud.

**describe_instances.** It connects to a node and asks it its sub-cloud id and its neighbors. An error is returned if the node is not part of a sub-cloud. Well, this one doesn't need to be exploited to give out information.

**DescribeInstances.** It just gathers the required local information and sends them to whom who asked for them.

**monitor_instances.** It connects to a node and asks it to start its 'monitoring system' that uses the AGG to estimate the total number of nodes in the cloud. No leaks here as far as I know.

**MonitorInstances.** Starts polling the AGG module to estimate the total number of nodes in the cloud. No undesired leaks here.

**unmonitor_instances.** Opposite of monitor_instances.

**UnmonitorInstances.** Opposite of MonitorInstances.

In no way I consider this analysis complete. This APIs are too 'rough', too much different from those of a real usable cloud system so it's not worth spending time on them since even a small change will lead to completely different leak situation. On the contrary, the basic modules are quite 'stable':

### PSS

In Algorithm 1 we can see the algorithm used by the PSS. For a detailed explanation see [21], but to us the only important things to know are the following. There are two threads[6], an active one that starts the communications and a passive one that waits for them. In every PSS module there are both the active and the passive threads. Every loop, the active thread chooses a peer to communicate with from the local partial view, and sends it a subset of its view. This subset is made of random peers chosen among all but the $H$ oldest peers. In the most frequently used configuration (push + pull), the active thread also awaits for the receiver to send it its subset,

[6] Independent logical control flow.

obtained in the same way. Then they both merge their view with the obtained subset (method *select*), following certain rules to remove duplicates and older nodes, and also giving priority to the peers contained in the subset (parameter $S$ of the algorithm controls the priority).

**Algorithm 1** The Peer Sampling Service Algorithm [21].

(a) Active Thread

```
loop
    wait(T time units)
    $p \leftarrow$ view.selectPeer()
    if push then
        // 0 is the initial age
        buffer $\leftarrow$ ((MyAddress,0))
        view.permute()
        move oldest H items to the end of view
        buffer.append(view.head(c/2-1))
        send buffer to $p$
    else
        // empty view to trigger response
        send (null) to $p$
    end if
    if pull then
        receive buffer$_p$ from $p$
        view.select(c,H,S,buffer$_p$)
    end if
    view.increaseAge()
end loop
```

(b) Passive Thread

```
loop
    receive buffer$_p$ from $p$
    if pull then
        // 0 is the initial age
        buffer $\leftarrow$ ((MyAddress,0))
        view.permute()
        move oldest H items to the end of view
        buffer.append(view.head(c/2-1))
        send buffer to $p$
    end if
    view.select(c,H,S,buffer$_p$)
    view.increaseAge()
end loop
```

I can see a leak threat here. A node that has just exchanged its peers information with another node, knows some of the peers that the other node is likely to be still ‘using’. This piece of information alone is not that useful but it could be used together with something else, like e.g. asking that node to create a sub-cloud soon after the exchange will probably lead to the subcloud having at least one of the known peers. Luckily, this works only for a very limited time after the exchange.

### BS

In Algorithm 2 we can see the T-Man algorithm used by the BS module. For the detailed explanation see [22]. As with the PSS there are an active and a passive thread, and both are needed in every node. Given a set of nodes (in our case, the nodes of a sub-cloud), every node will run the BS module and will keep a partial view[7] containing a sub-set of the nodes of the sub-cloud. Continuous exchanging, and sorting of the nodes of the partial views with a certain *rank* function, leads to the creation of a specific network topology dependent of the function. The active thread selects a peer among the peers of its view that it prefers the most (according to the rank function), and sends it the first $m$ peers of its view sorted by the same function to be those that it will prefers the most. Then it waits for the receiver node to do the same and 'reply' with its $m$ entries. Finally they both merge their views with the received list of peers.

**Algorithm 2** The T-Man Algorithm [22].

(a) Active Thread

```
loop
    wait(T time units)
    // select rand. among first $N$ ranked descriptors
    $p \leftarrow$ view.selectPeer($N$,rank(myDescriptor,view))
    buffer $\leftarrow$ merge(view,{myDescr})
    buffer $\leftarrow$ rank($p$,buffer)
    send first $m$ entries of buffer to $p$
    receive buffer$_p$ from $p$
    view $\leftarrow$ merge(buffer$_p$,view)
end loop
```

(b) Passive Thread

```
loop
    receive buffer$_q$ from $q$
    buffer $\leftarrow$ merge(view,{myDescriptor})
    buffer $\leftarrow$ rank($q$,buffer)
    send first $m$ entries of buffer to $q$
    view $\leftarrow$ merge(buffer$_q$,view)
end loop
```

About the possible leaks, the *rank* function itself is revealing, but as long as it is chosen in a way that we don't care what it reveals, there should be no problems. To be honest, I can't think of a function that would be of any use in our case[8] and that also reveals something we would want to avoid revealing.

[7] Not to be confused with the partial view of the PSS. The concept is very similar but they are not the same 'object'.

[8] A ranking based on latency could have revealed something about the locations of the nodes, but the anonymizing network's onion/garlic routing flattens out latency.

### AGG

In Algorithm 3 we can see the basic aggregation algorithm that allows each node to get to know some global property by just continuously updating their local values based on other nodes values. Again we have an active and a passive thread, the active one gets a neighbor from the PSS and send it its value. The passive thread gets it and sends back its value. Finally they both update their local values. How exactly they compute the updated value depends on the type of aggregation they are trying to achieve. For the detailed explanation about gossip-based aggregation see [8]. The aggregated value itself tells something about the system but it's supposed to be something we want the nodes to know or we would not be calculating it. I can't see any problem here.

**Algorithm 3** The Gossip-Based Aggregation Algorithm [8].

(a) Active Thread

```
loop
    wait(T time units)
    $q$ ← getNeighbor()
    send $s_p$ to $q$
    $s_q$ ← receive($q$)
    $s_p$ ← update($s_p$,$s_q$)
end loop
```

(b) Passive Thread

```
loop
    $s_q$ ← receive(*)
    send $s_p$ to sender($s_q$)
    $s_p$ ← update($s_p$,$s_q$)
end loop
```

### SS

The *Slicing Service*, not yet implemented in the prototype but worth mentioning because of its possible leaks. Given a set of nodes, we might want to group them into slices with certain characteristics e.g. network bandwidth, processing power, storage capacity... To do so in a decentralized way we need every node to be able to place itself in the correct slice, but this implies that the nodes must be able to tell how good they are at something (e.g. processing power) compared to the others. A solution has been proposed [23]: it relies on an algorithm that's quite similar to the one used by the PSS, with the difference that every peer descriptor contains also a pair of values [*attribute*; *random-number*] where *attribute* is the value of the characteristic we are considering, while *random-number* is a uniform random number generated over a fixed interval. The goal for the nodes, is to exchange the random numbers in a way that the position of the numbers in the fixed interval reflects the position of the nodes holding them in a hypothetical list of nodes sorted by the value of the attribute. As an example, let's say we are trying to slice the nodes based on their processing power. If a node ends up with number 1 in the interval [1;9], it means that it is very bad at processing and it belongs to the slice of the lowest processing power nodes. Slices themselves have to be defined, they might be something like: poor [1;3], average [4;6], good [7;9]. This is great but how are the numbers exchanged? We can see that in Algorithm 4, at line 8. The active thread has just obtained some new fresh peers the same way the PSS would have done, and now it searches them to find one, such that exchanging its random number with it would improve the 'sorting'. There is an improvement if the inequality $(attribute_{remote} - attribute_{local})(random_{remote} - random_{local}) < 0$ is satisfied.

**Algorithm 4** The Newscast Sorting Protocol [23].

```
(a) Active Thread
1:  loop
2:      wait(T time units)
3:      $p \leftarrow$ random peer from view
4:      buffer $\leftarrow$ view $\cup$ {myAddress,ts,$x_q$,$r_q$}
5:      send buffer to $p$
6:      receive buffer$_p$ from $p$
7:      view $\leftarrow$ youngest $c$ entries of buffer$_p$ $\cup$ view
8:      $i \leftarrow$ peer from view such that $(x_i$-$x_q)(r_i$-$r_q)<0$
9:      send $(x_q,r_q)$ to $i$
10:     $r_q \leftarrow r_i$ from $i$
11: end loop

(b) Passive Thread
1:  loop
2:      receive buffer$_q$ from $q$
3:      buffer $\leftarrow$ view $\cup$ {myAddress,ts,$x_p$,$r_p$}
4:      send buffer to $q$
5:      view $\leftarrow$ youngest $c$ entries of buffer$_q$ $\cup$ view
6:  end loop
```

*ts = time stamp*
*x = attribute value*
*r = random value*

A possible leak here is that every node gets to know the values of the attribute of other nodes. Such values cannot be correlated to the nodes' real identities most of the times, but in case of few exceptional node that are a lot better (or worse) than the other, it might be possible. Let's say that the cloud is made of 1000 nodes, but only 10 have over 100GBps bandwidth. An attacker that knows the 1000 real machines but does not know the association between fake and real identities, could use the attribute values to make a guess. The more the attribute's value is uncommon, the less real machines candidates remain, thus the easier the guessing will be. Of course, the assumption that the attacker knows most of the cloud's real machines is unrealistic. The two proposed anonymizing networks can't hide the fact that someone is using them, but it should not be possible to know that they are being used to anonymize this cloud system[9].

### Other security threats

Those are all the leaks I was able to find, I can't really be sure that there aren't any other. Unfortunately, there's not a particular way to find them other than thinking of how you would attack your own system. Obviously, assuming that the 'leak-resistance' of the basic system is good, it will be up to the designer of any future modules to ensure their safeness.

I would also like to point out that I've only talked about the security threats that could have resulted in the de-anonymization of the nodes. There are a lot of other security problems that should be resolved before this system could really be used. I'll just spend few words about the most important one. Without any kind of authentication, anyone can inject in the cloud an arbitrary number of modified nodes to do all kind of bad things e.g. poisoning the PSSs with false data, 'sponsoring' some nodes while totally cutting down the communications with others and much much more.

## 2.3 The final P2P anonymous cloud system architecture

After all that's been told, Figure 2.4 should not be a surprise. The previous architecture was already almost ready to be anonymized, and just needed

[9] Unless some heavy traffic analysis is done and some classic pattern of the cloud system (like e.g. using the timing of the gossip algorithms) is found.

the addition of the anonymizing component. The leaks I've found are not serious enough to justify any major changes, but they should be kept in mind when developing future modules.

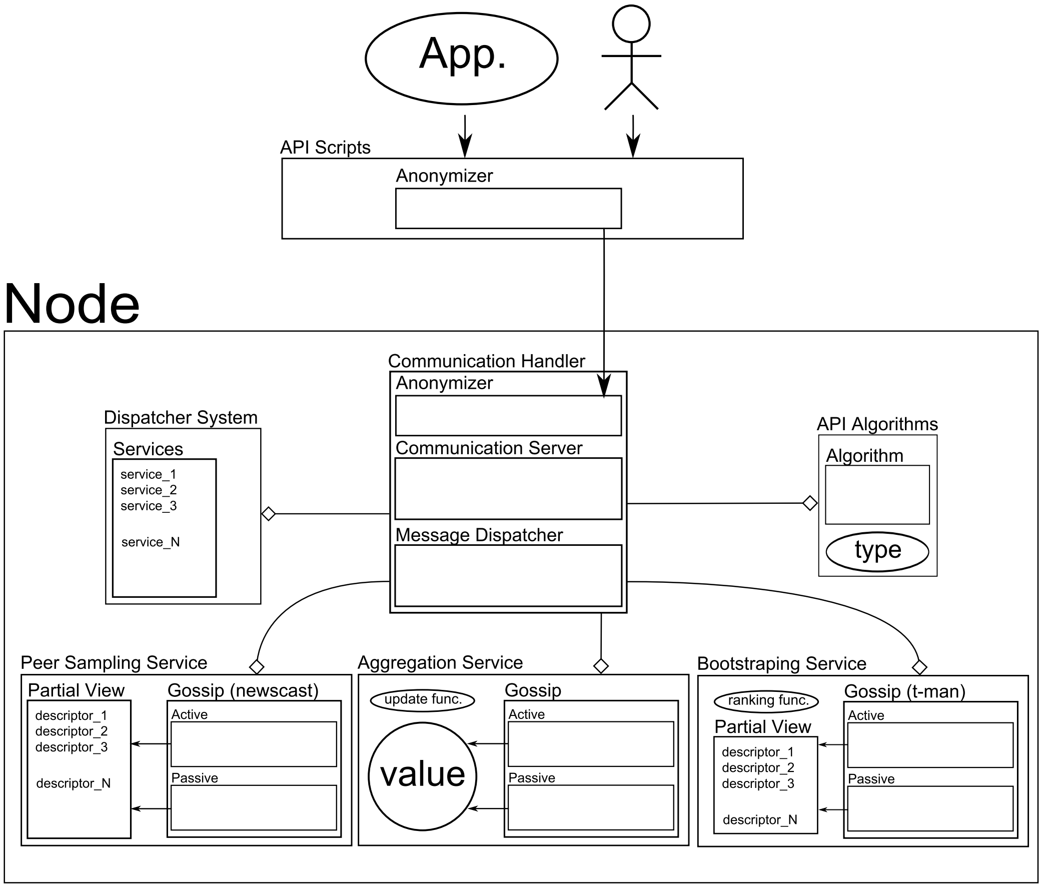

**Figure 2.4:** The Anonymous P2P Cloud System Architecture.

# Chapter 3

# Implementation

We will now see how the proposed changes have been implemented. This chapter is divided in two main parts. The first one shows the changes in the Java code of the prototype, the second one regards the creation of the fake identity (Hidden Service) using the tools provided by the anonymizing network (Tor).

To make it easier to compare the performances of the anonymous system with those of the non-anonymous version, I've implemented it in a way that allows switching the 'anonymity mode' flag *on* or *off*. When the flag is set to off, the system behaves like it did before the changes. It's not possible at the moment to have some node that are anonymous and some that are not, but it would not make much sense. Every nodes would still need to run Tor even if they are not anonymous to be able to talk with anonymous nodes. Anonymous node would still need to use Tor also for connections with non-anonymous node or they could be easily de-anonymized. The only direct connection in such an hybrid system would be between two non-anonymous nodes. Performance-wise, this could make some sense, but the rest of the system would still need to be re-designed to allow users to choose if they want to use all the cloud or only the anonymous part. I'm not going to carry on with this idea in this work.

## 3.1 JRMI and SOCKS proxy

The prototype uses Java Remote Method Invocation (JRMI) to carry out communications. Using JRMI it is possible to bind Java objects to 'keys' in a way that anyone requiring the objects could get them just by connecting and providing the right key. This sounds easy but there are quite a few problems in getting it to work with firewalls, NATs and hidden services, and I'm now going to explain you why. The complete process of binding and retrieving an object using JRMI is as follows:

1. The RMI Registry must be created specifying the port at which it will listen.

2. Anyone (Java program) wanting to sign up an object to the RMI Registry must export it, connect to the Registry (be it remote or local) and bind the exported object to a key. The object must be serializable. Once an object is exported, it is associated an RMI Server that will listen at a port automatically chosen by the JRMI system. RMI Registry and RMI Servers can be (and often are) on the same machine but could also be on different machines.

3. Anyone (Java program) wanting to retrieve the object, must connect to the RMI Registry and provide the key. Then the RMI Registry points the client to the RMI Server hosting the required object. The client can then connect to it and interact with said object.

The biggest problem here is the creation of the RMI Servers at *random ports*. In any realistic scenario there will surely be firewalls and NATs that will need to be configured to allow incoming connections to certain ports. But you can't open ports you do not know. Luckily there is a way to manually assign ports to RMI Servers by giving a pair of socket factories to the object during the exportation phase. One socket factory will be used to create the RMI Server, while the second one will be passed down by the RMI Registry to the client that will use it to connect to the RMI Server. This way it will work, but what if we have to bind more than one object? What if it's not possible to know how many objects we will be needing at any certain given time? That's exactly the case of the prototype I'm modifying. What I've ended up doing has been deciding a pool of ten known ports. Every time a new object needs to be exported, a small algorithm looks for a free port among those of the well known pool. If there are no ports available, an exception is thrown. The pool has been sized to make it unlikely that a single node will need to host more than that many objects (ten) at the same time.

Another minor problem with JRMI is that when you export an object you must do so having in mind who will need it, or more precisely, *where* will the users be. If the intended user is remote, then the object must be exported with a socket able to connect to the server from remote. If you need an object to be accessible from both local and remote then you have very few options: you can duplicate the object and bind the former configured with a 'remote-able' socket and the latter with a 'local-able' socket, or you can use just a remote one and hope the underling system will be smart enough to connect locally even if the socket tells otherwise. Things get worse with Tor and the need of sockets that must talk with the local SOCKS proxy. In this case, in no way the system will be able to see that the RMI Server is local, and the connection will be forced to pass through Tor anyway.

A small note about performances. As we've seen, for a node to interact with a remote object, two connections are needed, the first to the RMI Registry, the second to the RMI Server. This will surely impact performances, especially in anonymous mode, because the increase in latency will be paid twice. There's really few I can do about it, the system should be re-designed not to use JRMI, but the time at my disposal does not allow me to do it.

Back to how the prototype works, every machine hosting a node has to run an RMI Registry. In case of multiple nodes on the same machine (but that's not going to happen) they will all use the same RMI Registry. Communications start by connecting to the remote[1] RMI Registry, then a well known key is provided and the reference to the remote node's Communication Server is obtained. The sender can now put its message in the remote Communication Server (think of it like a mailbox) by connecting to the RMI Server hosting it. When a new message arrives, the Message Dispatcher awakes and dispatches it to the appropriated module.

In practice, what we have to do to anonymize the system is change the way the connections to the remote RMI Registry and RMI Servers are done, to make them pass through Tor. Connections to the RMI Registry are done using the *java.rmi.registry.LocateRegistry* [24] class. Connections to the RMI Servers are implicit and occur whenever interacting with the remote object. Here's an example:

```
// Connecting to the remote registry
Registry registry = LocateRegistry.getRegistry(hostname, port);
// Retrieving a reference to the remote object (Communication Server)
CommSrvRemoteAPI remoteCommServer = (CommSrvRemoteAPI) registry.lookup(key);
// Delivering the message (Implicit connection to the RMI Server)
remoteCommServer.receiveMessage(msg);
```

**Code 3.1** Connecting to a registry, getting and using a remote object. *hostname* is the real identity of the remote machine, *port* is the number of the port at which the RMI Registry is listening, *key* is a string that identify the Communication Server, *msg* is the message to be delivered.

The way Tor uses to interface with applications is a SOCKS[2] proxy. We will see how to configure it at Section 3.3 when we'll talk about the Hidden Services. Here the Java code modified to use the proxy:

[1] Most of the times it's remote, but it could also be local if there are multiple nodes on the same machine.

[2] Socket Secure (SOCKS), it's a (de facto) standard for circuit-level gateways [25].

```
/**
 * Returns the registry hosted locally.
 */
public static Registry getLocalRegistry() throws RemoteException {
  return LocateRegistry.getRegistry("localhost", RMI_REGISTRY_PORT, null);
}

/**
 * Returns the registry hosted at hostName.
 */
public static Registry getRemoteRegistry(String hostName) throws RemoteException {
  // This two vars will be set depending on the state of ANONYMITY_ON
  RMIClientSocketFactory socketFactory = null;
  int port = 0;
  if (!ANONYMITY_ON) {
    // Direct
    socketFactory = null;
    port = RMI_REGISTRY_PORT;
  }
  else {
    // Anonymous, a factory that will provide sockets that will use the given proxy
    socketFactory = new RMIClientSocketFactory() {
      @Override
      public Socket createSocket(String host, int port) throws IOException {
        // Local proxy data
        SocketAddress proxyAddr = new InetSocketAddress("localhost", TOR_PORT);
        Proxy proxy = new Proxy(Proxy.Type.SOCKS, proxyAddr);
        Socket socket = new Socket(proxy);
        socket.connect(new InetSocketAddress(host, port));
        return socket;
      }
    };
    port = RMI_REGISTRY_HIDDEN_PORT;
  }
  // Having socketFactory == null behaves like not having it at all
  return LocateRegistry.getRegistry(hostName, port, socketFactory);
}
```

**Code 3.2** Connecting to a RMI Registry, be it local, direct remote, or behind a hidden service. *hostName* can be an IP/domain name or an onion address, *RMI_REGISTRY_PORT* is the number of the port at which the real RMI Registry is listening, *RMI_REGISTRY_HIDDEN_PORT* is the number of the port of the hidden service linked to the real RMI Registry, *TOR_PORT* is the number of the port of the local Tor SOCKS proxy.

```
/**
 * Returns a socket connected using the appropriate settings.
 */
public Socket createSocket(String host, int port) throws IOException
{
  Socket socket;
  if (!ANONIMITY_ON) {
    // Direct
    socket = new Socket();
    socket.connect(new InetSocketAddress(host, port));
  }
  else {
    // Anonymous
    SocketAddress proxyAddr = new InetSocketAddress("localhost", TOR_PORT);
    Proxy proxy = new Proxy(Proxy.Type.SOCKS, proxyAddr);
    socket = new Socket(proxy);
    socket.connect(new InetSocketAddress(host, hPort));
  }
  return socket;
}
```

**Code 3.3** A piece of code from the client socket factory used to export objects. Depending on the anonymity mode in use, *host* could be an IP/domain name or an onion address, *port* is the number of the port at which the real RMI Server is running, *hPort* is the port of the remote Tor Hidden Service , *TOR_PORT* is the number of the port of the local Tor SOCKS proxy.

Another modification I've done regards the API scripts and their parameters. The original prototype used the Java networking APIs to get the IP address of the local node, this address was included in the requests done by the node and allowed the receiving nodes to know whom to reply to. Obviously this can't be done anymore, the fake identity must be provided instead of the IP address, but there's no Java API to get it. Using Tor, such fake address can be found in a file (we'll see it better at Section 3.3), in I2P it's on a local configuration web page. In both cases it's not easy to get it in an automatic way, especially considering that there could be many anonymous node on the same host, thus many fake identities. Since this is just an implementation problem and it does not add much value to my work, I've opted for the easier solution, that is, the fake identity must be provided as an input parameter in all the scripts.

```
// Old version
$./startNode.sh −n <node_name>
// New version
$./startNode.sh −n <node_name> −hostname <address>
```

**Code 3.4** Example of the new *hostname* parameter. The *address* could be the real address or the fake address, depending on the anonymity mode in use (on/off).

## 3.2 Other secondary but necessary changes to the code

What I've shown in Section 3.1 are just the most meaningful changes. There were lots of other minor modifications spread all around the prototype that needed to be done in order for it to work. The most important one regards the fact that the prototype used only IP addresses. If there were any hostnames in the initial partial views of the nodes, they were resolved as soon as possible and only the resulting IP were saved. That's impossible with onion addresses. An onion address is at some point resolved to an IP but that link is valid only for a short amount of time. Also, the 'resolution' of the onion address has to be done by Tor, not by a standard Domain Name System (DNS) query[3].

Fixing this has not been 'hard', but it took quite a bit of time because even if the high-level architecture was well documented, the code documentation relied only on Javadoc[4] comments. Don't get me wrong, Javadoc is great, but it does not help a lot if you are interested in something like: find-

[3] DNS queries of onion addresses is one of the most frequent and easiest way to get de-anonymized. They are not encrypted so a attacker can see what onion addresses a user is connecting to.

[4] The (de facto) standard for documenting Java classes by formatting comments following a certain set of rules.

ing all the places where an IP is stored, understanding how such IPs are set and how they are used. In addition, the project makes large use of the practice of marshalling[5] and sending runnable[6] objects so that they are executed somewhere else. This opens up new design possibilities but it also lowers the code comprehensibility, requiring you to first understand where and under what circumstances it will be executed. What I ended up doing has been searching the code around the parts where I knew a connection was likely to be necessary, also I've searched the whole project for keywords like "hostname", "address", "*Inet4Address*"[7]. Once I've found and understood all those parts I decided that the best solution would have been to keep strings with hostnames (domain names or onion addresses) instead of IPs, moving the resolution process to right before they are needed. Applying this also took quite a bit of time because every changed part brought the necessity to do other minor modifications on the parts that depended on them and so on, I'm sure you know what I'm talking about.

To sum it up, I've changed every part of the system that used an IP to use an hostname instead. Address resolution (be it a standard domain name or an onion address) is done on the fly when needed. This will probably adversely affect performances but as far as I know there's no way to avoid it. Note that it's still possible to use IP addresses to identify nodes in the non-anonymous mode just by inserting them as strings, Java is smart enough to recognize an IP in string form, assuming it's well formed.

## 3.3 Setting up the hidden services

Assuming Tor is correctly installed and working on the hosting machine, to create an hidden service we'll do the following. Locate the *torrc* Tor configuration file. Its location depends on the operative system we are using

[5] The process of transforming the memory representation of an object to a data format suitable for storage or transmission [27].

[6] Objects that are intended to be executed by a thread.

[7] The Java type for IPv4 addresses.

and on how we installed Tor. This topic is covered extensively in Tor's documentation [26] so I will not explain it here. Once located, edit it and add the lines:

```
HiddenServiceDir /path/to/a/directory
HiddenServicePort fake_port real_address:real_port
```

**Code 3.5** An example of part of a torrc file. *HiddenServiceDir* is the directory where important files of the hidden services are stored. *HiddenServicePort* defines a mapping between the real service and the hidden service.

For it to work, it's not important what HiddenServiceDir is chosen as long as the Tor process can read/write in it. But since that directory contains very important files that must remain secret, attention must be paid to who can access it. When Tor is first started after editing the torrc file, a key pair is created for each hidden service and the private key is stored in the file *private_key* in the HiddenServiceDir. That key must remain secret or someone else could impersonate the hidden service. A file called *hostname* containing a short summary of the public key is also generated, the content of that file is the fake identity, the address others will use to connect to the hidden service. This one can (should) be given out.

```
duskgytldkxiuqc6.onion
```

**Code 3.6** An example of hostname file.

About the HiddenServicePort, the *fake_port* is the one used together with the fake address to connect to the hidden service. Let's say our fake port is 81, the connection to the hidden registry would be:

```
LocateRegistry.getRegistry("duskgytldkxiuqc6.onion", 81, socketFact);
```

**Code 3.7** Connecting to a hidden registry.

Tor takes care to 'redirect' the connection to the real service, that in our case is hosted locally, let's say at port 8181. A torrc file configured this way would look like:

```
HiddenServicePort 81 127.0.0.1:8181
```

**Code 3.8** Another example of torrc file.

# Chapter 4

# Performance evaluation

In this chapter, we will see how the modified prototype has been tested. Tests are aimed at showing how much difference (if any) there is between the performances of the standard version and those of the anonymous version. To run the tests we will need a lot of machines on which to run our prototype, and the easiest way to get and orchestrate them is to make use of a cloud service. Do not get confused about it, the fact that we will use a cloud service has nothing to do with our cloud system, we need it just for the tests. It would have been the same[1] if we could have had access to many personal computers around the world instead of using a cloud service.

## 4.1 Amazon EC2

As you could have guessed by the title of this section, we will use Amazon EC2[28]. The reason is that it's one of the most used, thus it's easy to find documentations, examples and frameworks for many languages. Google Cloud[29] or OpenStack[2] based solutions would have worked too. The basic idea is that I will ask Amazon to give me a certain number of machines

[1] The possibility of running many instances of the exact same machine (including firewall settings) makes it actually a lot easier for the system to work, than just using random computers. But that's just a matter of compatibility and configurations, apart from that there would be no differences.

(virtual machines) which I will control through Secure Shell[2] (SSH). I will also write a script that will orchestrate those machines to do the tests by starting, stopping and calling the prototype's APIs. Quite simple isn't it? Yes the concept is simple, but realizing it is not. Keep in mind that EC2 services are not free, every resource has a cost that depends on the type, power, location, and of course on the amount of time you will use it. A 'trial and error' brute-force approach would have been quite costly so I had to spend some time exploring all possible solutions, in order to minimize the cost of each resource.

So, what resources are we talking about? The minimum number of resources that every machine has to have to be useful for our tests is four, and they are:

**Instances.** An instance is a virtual machine, with a certain computational power and memory. Amazon EC2 lets you choose among several 'tiers' of instances[31], each one with a different combination of CPU and RAM. You cannot build an instance with an arbitrary CPU and RAM, you are bound to choose a tier. Since our prototype needs very few resources we will go for the lowest tier possible, which at the moment of writing is the *t2.micro*[3]. Obviously, an instance also needs a volume (virtual disk) from which to load the operative system. Every tier has support for some types of volume, but the volumes themselves are stand-alone resources and we will talk about them later. An instance can go through several states[32], the most important ones are: *Stopped*, *Running*, *Terminated*. A stopped instance does not cost anything, it's just a mere bunch of settings, it does not consume any real resources. Once completely started, the instance will be in the running state and

[2] It's a network protocol that allows the creation of a secure channel between a client and a server[30]. One of its most common uses is remote command execution, and that's exactly what I will use it for.

[3] The t2.micro tier comes with a single virtual CPU 2.5GHz, 1GB of RAM, support only EBS volumes, has 'Low to Moderate' network performances and can only be used with HVM AMIs[31].

you will start to pay for it. Costs are counted on an hourly basis, which means that you will be charged for every hour the instance will be running. It's also important to know that you will pay for a full hour even if you use it for just a minute, one hour is the minimum unit of time. The 'hours counter' will start every time the instance is started so if you stop and start it many times within an hour you will pay a full hour for every time the instance was started. This has influenced the design of the tests a lot. The final state, terminated, is reached when the user choose to permanently destroy the instance.

**Volumes.** Volumes are virtual storage devices. They can be divided in two main categories: the ephemeral ones (Instance Store[33]) and the persistent ones (Elastic Block Storage[34]). All data inside Instance Store (IS) volumes are permanently lost when the related instance is stopped or terminated, while Elastic Block Storage (EBS) volumes can live on without being attached to a running instance. The main cost factor of volumes is how big they are, you pay for the number of Gigabyte reserved to you. Note that if you have a 100GB disk with just 1GB used, you will still pay for all the 100GB so choosing the right size is important. The *t2.micro* instances tier that I will use supports only EBS root volumes so I don't really have a choice here.

**AMI.** AMI stands for Amazon Machine Image[35], it's the standard used by Amazon for boot-able disk images. Inside an AMI there's an operative system and any possible software that the creator of the AMI decided to put in it. AMIs and Volumes are strictly related, every AMI is bound to a specific type of volume. EBS-backed AMIs will run only on EBS volumes, while IS-backed ones will need IS volumes. When an instance is started using an AMI, an appropriated volume will be created. Note that who creates the AMI also decide the size of the disk, and you'll be forced to have a disk of at least that size. Creating an AMI from scratch is possible but quite complicated, the best way to go is often to start from an existing AMI and modify it to your needs. Once you're done with the modifications you can use an Amazon tool to make a new AMI from what you ended up with. Another important thing to know is that Amazon EC2 supports two types of virtualization[36]: Paravirtualization[37] (PV) and Hardware-assisted virtualization, also called Hardware virtual machine[38] (HVM). Each tier of instances may support one or both of them, while AMIs are made to support only one of them. To make it all work you must choose an AMI with the virtualization type supported by the tier you want to use. The *t2.micro* tier only supports HVM instances and that caused me some troubles because the Debian[4] AMI[39] I decided to use only support PV. I solved using an experimental version[5] of the Debian AMI. Last but not least important thing to know about AMIs is that they have a cost. Most of it comes form the operative system they contain, but can also come from the rest of the software and/or from the possibility to get official support. Unix based AMIs are mostly free and I didn't have to use any shareware software so I totally avoided AMI related costs.

**Network.** Every instance has a network interface, it's not exactly 'required' for the instance to exist, but without it, it would not be possible to connect to it and since it's in the cloud, it would be quite useless[6]. My instances will have a public and a private network interface. The private one will be used to communicate with other instances inside the same region, the public interface is needed for me to be able to connect to them from outside the cloud. The cost of networking is determined by how much data flows from and into each network interface. A communication between two instances of the same region can be carried out using the private interface, and costs much less that the traffic between

[4] It's a widespread free operative system. It uses the Linux or the FreeBSD kernel, most of its tools come from the GNU Project[40].

[5] The exact AMI I started from is the one with the id: *ami-698cdf59*[41].

[6] Instances used for working on big amounts of data, that do not need any input/output, might make sense even without network interface.

an instance and something on the internet, that must use the public interface.

Prices for all the resources I've listed are ever changing and can be found on the Amazon Web Services (AWS) site[42]. Another thing to know is that users cannot ask for an arbitrary amount of resources. Every account has default limits[43] that may be lifted by posting an 'increase limit request' to Amazon's support. Those limits are there because, with all the 'digital power' that comes with the relatively cheap cloud resource, an inexperienced or malicious user could digitally harm someone. Also, it's not that hard to make mistakes when trying to automate things (like instances creation) and that may lead to unexpectedly high bills. I filled a limit increase request, asking for the possibility to start 1000 *t2.micro* instances but due to the 'youth' of my account they could only lift my limits up to 50, thus tests will be done with 50 machines.

## 4.2 Setting up the tests on EC2

50 machines might not be that much but it's still too much to be handled manually. We need to automate things to be able to put out some meaningful tests. Let's see what we need to do with some pseudo-code:

**Algorithm 5** Tester script pseudo-code

```
create instances
loop
    $parameters \leftarrow$ readParametersFile()
    if parameters is null then
        break
    end if
    start/restart remote prototypes with parameters
    start/restart local prototype with parameters
    start the monitor-instances API and wait for the network to initialize
    start the first run-nodes API
    start the second run-nodes API
end loop
terminate instances
```

We are interested in testing how much time it will take for the network to initialize (line 9), how much time a request for the creation of a sub-cloud will take to complete (line 10), and how much time it will take for a second sub-cloud to form (line 11), when almost half the nodes are already used by the first sub-cloud. All the tests will be done on both the standard and the anonymous version, we will also see how things change when varying the size of the local view of the nodes. Parameters, which are: anonymity mode and view size, will be read from a file (line 3).

Something that has not yet been written is how the system will bootstrap. We know that all the nodes are kept together by the overlay formed by the Peer Sampling Service, making use of the information of the local views. The problem is that we don't know the addresses of the nodes (be them IP or .onion) up until run-time, so we can't fill the views with meaningful information. The adopted solution consists of using a special node with a well know IP and .onion address, that will be put in all the views of the other nodes. It will serve as a linking point for the network and even though it will be very 'busy' soon after it is created, its central position will fade over time as nodes will start to know each others. This will also be useful for our tests, because the special node will naturally function as a barrier for the whole network. We can start all the other nodes whenever we want, knowing that they will stay isolated until the central node is started. The best place for this central node to be, is the same machine where the tester will be running. The tester will start the local well known node after all the other nodes will be ready, and will start counting the time from that moment. The network will be considered initialized when the API *monitor-instances* will report an estimated number of nodes in the network, close enough to the real number[7].

Back to the pseudo-code, we can think of it as the union of three different parts: managing the instances, managing the execution of the remote

[7] The reason why we don't wait for the exact real number of nodes, is that the estimation process is not perfect. With a non-trivial number of nodes it will probably never stabilize on the exact value. Not to mention that nodes may crash, and even if we could estimate it perfectly, just one dead node would cause us to wait forever.

commands on them, and managing the local prototype and the API calls on it. The last one is quite easy with some bash scripting, but the other two might prove more challenging. Still they are quite common needs so there exist a lot of frameworks to ease our work. After some studies I've decided to use Python language for my tester, since two of the best and most used frameworks to do the things I just wrote up here are for Python, and it's full of useful examples on the Internet.

### 4.2.1 Boto

Boto[44] is a powerful Python framework for interacting with Amazon's EC2 APIs. I will use it for the creation/termination of the instances and for retrieving information about them, such as their status and their hostname. The hostname will be used for executing remote commands as we will see later at 4.2.2. Here an example of instances creation with boto:

```
conn = boto.ec2.connect_to_region("us-west-2", profile_name="michele.amati")
res = conn.run_instances(image_id="ami-15240025",
                         min_count=49,
                         max_count=49,
                         key_name=KEY_PAIR_NAME,
                         security_groups=[SECURITY_GROUP_NAME],
                         instance_type="t2.micro",
                         monitoring_enabled=False,
                         disable_api_termination=False,
                         instance_initiated_shutdown_behavior="terminate",
                         client_token=ONE_TIME_TOKEN,
                         dry_run=False)
```

**Code 4.1** An example of instances creation. First we get a connection to the Amazon's region in which we want to start the instances, then we specify how do we want them. Important parameters are: the id of the AMI, the number of instances we want, the cryptographic key pair that we will use to access them, the security group (firewall settings), the type of instances, and what should they do when they are shutdown.

For Boto to work, it must be configured with the credentials of the user's AWS account. They are not those used to log into Amazon's services, but special ones created expressly for EC2 using the AWS web console. They must be placed in the file:

```
> cat /home/user/.aws/credentials
[michele.amati]
aws_access_key_id = XXXXXXXXXXXXXXXXXXXX
aws_secret_access_key = XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

**Code 4.2** The credentials configuration file.

If the call at Code 4.1 succeeds, we obtain what they call a 'reservation', which is like a set of instances. Obviously the call might also fails, for configuration errors, something wrong at Amazon's side or simply because the resources we are asking for are not available, exceptions must be handled. Also remember that even if Boto is great, you still need to know what you're doing, because nothing prevents you from forming invalid requests, like trying to ask for an instance with an impossible configuration.

### 4.2.2 Fabric

The Fabric[45] Python framework will help us executing the remote commands needed to start/restart the prototypes on all the instances. Another option would have been to terminate and re-create them every time (with the prototype starting automatically at boot) but considering the way costs are calculated it is much better to keep the instances up and just restart the prototypes. The main advantage of using Fabric instead of other lower level SSH libraries like Paramiko[46], is that Fabric allows you to do the following: give it a *task* (command to be executed) and a list of *hosts*, it will automatically execute the *task* on all host, with the desired degree of parallelism. Errors must be handles somehow, because with remote calls there are lots of things that might go wrong, but still the overall management of remote executions is much simplified.

```
@parallel(pool_size=5)
def reset_prototype_task(commands):
    tmp = ""
    for cmd in commands:
        tmp = tmp + "{}{}".format(cmd, ";")
    return run(tmp)

results = execute(reset_prototype_task, commands, hosts=hostnames)
```

**Code 4.3** Example of remote command execution with Fabric. The list of hostname is obtained using Boto, the commands are just strings chained with the ';' character that will cause them to be executed in order, always waiting for the previous one to terminate. The @parallel annotation tells Fabric that it can execute a maximum of 5 commands in parallel.

To be able to connect to the various hosts Fabric must be given the appropriated credentials, that might be a *user* and a *password* or a *private key file*. Amazon forces you to use the latter for SSH connections, remember the *KEY_PAIR_NAME* parameter that we set during the instances creation? That's what it is used for. Here at Code 4.4 are some other settings that might be worth looking at, especially when working with many hosts.

```
# Automatically accept hosts fingerprint
env.reject_unknown_hosts = False
# Ignore hosts to which connections are impossible
env.skip_bad_hosts = True
# Try connecting X time before giving up
env.connection_attempts = 3
# Do not abort execution if something on some host fails
env.warn_only = True
# The user to use
env.user = "admin"
# The path to the private key file
env.key_filename = ["/path/to/the/privatekey.pem"]
```

**Code 4.4** Useful Fabric settings when dealing with multiple hosts.

## 4.3 Tests results

Tests have been run with the following sets of parameters:

```
{
  anonymity_mode: off
  view_size: 20%
},
{
  anonymity_mode: off
  view_size: 50%
},
{
  anonymity_mode: off
  view_size: 90%
},
{
  anonymity_mode: tor
  view_size: 20%
},
{
  anonymity_mode: tor
  view_size: 50%
},
{
  anonymity_mode: tor
  view_size: 90%
}
```

**Code 4.5** The anonymity mode ‘off’ means the standard version, while ‘tor’ is the anonymous version. The size of the view is expressed as a percentage of the total number of nodes. Each one of the six tests has been executed ten times.

The initial idea was to consider the network initialized when the estimated number of nodes reaches a value as close as possible to the real number of nodes. But during the first tests with a number of nodes (50) larger than the one used for developing/debugging (5), I realized that such estimated value was quite unstable. I ended up accepting as ‘good’ any value in the range $[RNN - RNN * 0.3; RNN + RNN * 0.3]$, where $RNN$ is the real number of nodes. That’s a pretty ‘generous’ range, but still sometimes it’s not large enough and the prototype fails to stabilize. So, I consider the network initialized if the monitoring API gives me five consecutive values in that range. I must admit that I came out with that ‘five’ with no theory proven reason, I just saw that less than five causes the prototype to be considered stabilized too soon most of the times, while more than five has a great chance to lead to a never ending stabilization process. This stabilization problem has quite bad consequences for all the tests. It makes them considerably longer (thus more expensive) but can also false the results, because subsequent calls (run-nodes) might fail if they are done on a not yet completely initialized network.

About the run-nodes API, I’ve made the calls ask for forming sub-clouds of size determined by: $MIN(RNN * 0.45, VS)$, where $VS$ is the size of the views. The reason why each call cannot get more than 45% of the network is that nodes might crash or become unreachable, so leaving out 10% of the nodes makes the tests more likely to complete. Also, limiting them to the size of the views makes the first call able to complete without having to wait for the Peer Sampling Service to change the view, allowing us to make some potentially interesting considerations in the performances comparison phase.

In the following pages I’ve reported the results. Please note that tests can fail to yield a result, leaving us with missing values. This can cause calculations like: average and standard deviation, to be done on sets of values of different sizes.

| Parameters | Completed | AVG(s) | DEV(s) | MIN(s) | MAX(s) |
|---|---|---|---|---|---|
| off, 20% | 10/10 | 141.13 | 55.37 | 85.07 | 225.19 |
| tor, 20% | 3/10 | 330.29 | 70.06 | 280.26 | 415.37 |
| off, 50% | 10/10 | 104.58 | 48.66 | 50.04 | 220.18 |
| tor, 50% | 2/10 | 377.83 | 17.67 | 365.33 | 390.32 |
| off, 90% | 10/10 | 179.16 | 98.26 | 65.05 | 335.28 |
| tor, 90% | 1/10 | 175.15 | – | 175.15 | 175.15 |

**Table 4.1:** The time taken for the network to initialize

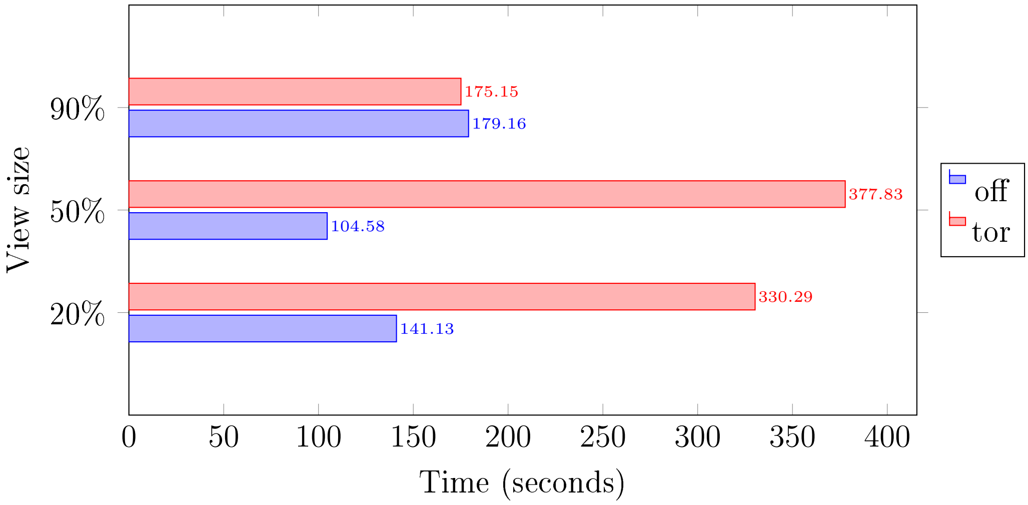

**Figure 4.1:** Average network initialization times compared

As we can see, the anonymous version takes more than twice the time used by the standard version. About the results with the *90%* view size, they are very close, but consider that only 1 out of 10 anonymous version tests completed successfully, and that might have been a very lucky case. The timeout is set at 500 seconds so if we considered the failed tests as tests that would have required more than 500 seconds to complete, we will get a much higher average, in line with the results of the tests taken with the other two view sizes.

| Parameters | Completed | AVG(s) | DEV(s) | MIN(s) | MAX(s) |
|---|---|---|---|---|---|
| off, 20% | 10/10 | 1.80 | 0.42 | 1.00 | 2.01 |
| tor, 20% | 9/10 | 227.57 | 28.29 | 198.21 | 284.31 |
| off, 50% | 10/10 | 2.00 | 0.01 | 2.00 | 2.01 |
| tor, 50% | 9/10 | 536.12 | 48.36 | 496.53 | 641.67 |
| off, 90% | 10/10 | 10.51 | 14.97 | 2.00 | 49.05 |
| tor, 90% | 9/10 | 563.26 | 35.27 | 526.56 | 620.66 |

**Table 4.2:** The time taken for the first run-nodes to complete

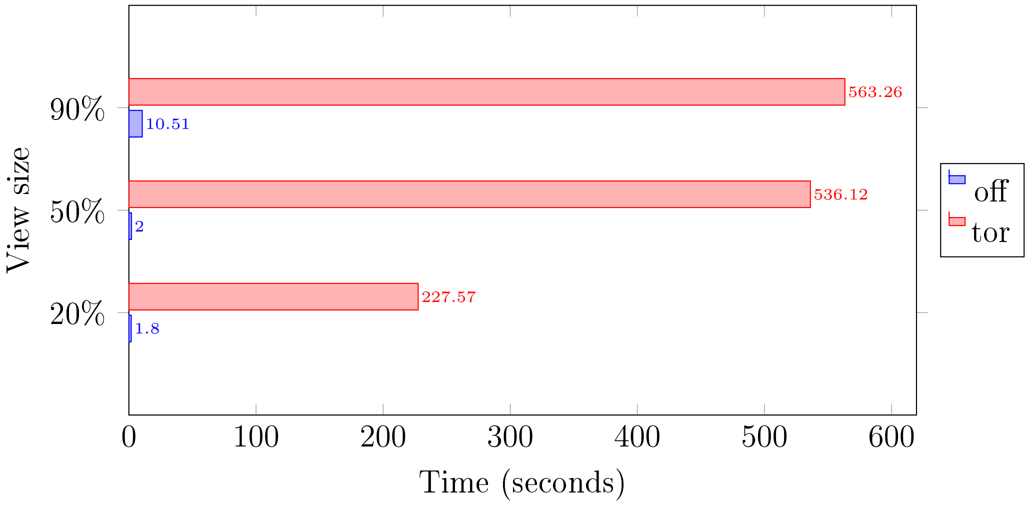

**Figure 4.2:** Average time to complete the first run-nodes call compared

The difference between the two versions is even higher in the first run-nodes tests. The prototype looks for a new node to add to the sub-cloud only when the previous node has replied (positively or negatively), so, even assuming that all the contacted nodes will be able to join, the last node will be contacted only after 'paying' two time the latency for each previous node. Remember also that we are using JRMI that alone doubles the number of connections required to carry out a single communication, resulting in the latency been paid four times for each node. The standard prototype uses the EC2 internal network with an average latency of 1ms, so every node will 'cost' less than 5ms. The anonymous version's latency really depends on the Tor network status, and might go from 300ms to 1500ms. If we consider an average of 900ms, than every node would required 3600ms to join. This alone cannot explain the huge amount of time taken by the anonymous version. The only other thing I can think of is that the high amount of failures in determining if the network was initialized or not (in the anonymous version), might have lead to the start of the run-nodes without the network being totally initialized. Thus the assumption that the first run-nodes would have had all the information required ready in the view would be true only for the standard version.

| Parameters | Completed | AVG(s) | DEV(s) | MIN(s) | MAX(s) |
|---|---|---|---|---|---|
| off, 20% | 10/10 | 3.30 | 1.16 | 2.00 | 5.01 |
| tor, 20% | 10/10 | 247.16 | 24.01 | 210.20 | 292.31 |
| off, 50% | 8/10 | 12.26 | 9.70 | 5.00 | 34.04 |
| tor, 50% | 7/10 | 929.54 | 156.33 | 757.74 | 1179.25 |
| off, 90% | 10/10 | 70.77 | 77.90 | 3.00 | 168.18 |
| tor, 90% | 8/10 | 822.58 | 70.78 | 708.69 | 894.86 |

**Table 4.3:** The time taken for the second run-nodes to complete

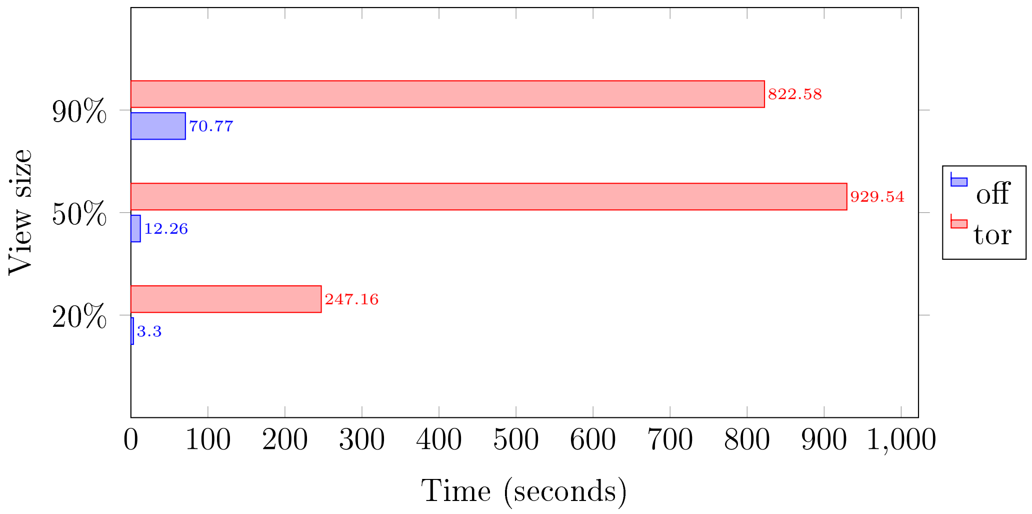

**Figure 4.3:** Average time to complete the second run-nodes call compared

The second run-nodes behaves almost like the first one, just with bigger numbers for both versions, which is easily explained by the fact that the number of 'free' nodes is less. An interesting thing to notice is that the *20%* view size anonymous version test doesn't differ that much from the same test of the first run-nodes. This might be another sign that most of the anonymous tests started doing the run-nodes without been properly initialized, because having in the views nodes that are already in the first sub-cloud (thus cannot take part in the second) should behave almost like having the fake non-existing nodes used to fill the fixed-size views prior to their complete initialization.

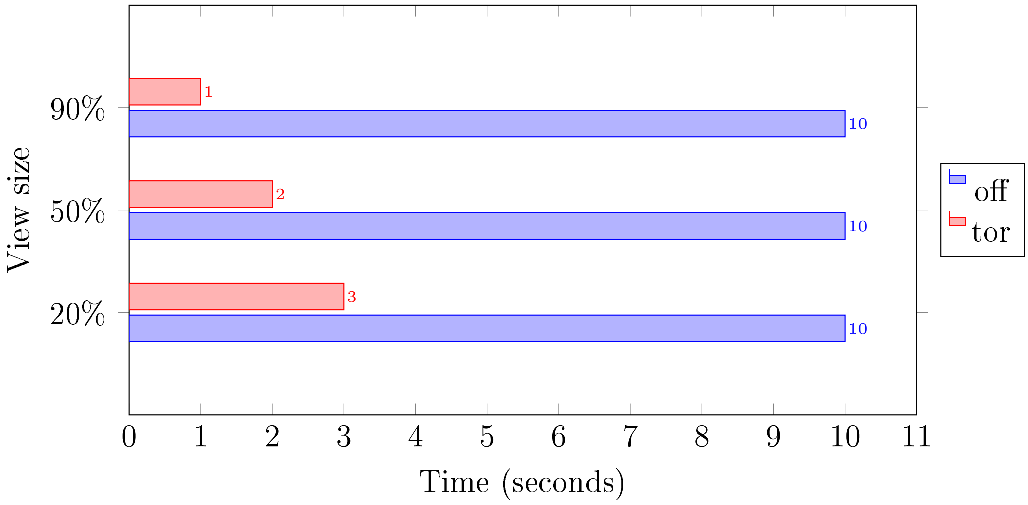

**Figure 4.4:** Initializations completed without incurring in the timeout, max is 10

The results seem to suggest that increasing the time limit for the initialization might help, but during some preliminary tests (not reported in this document) without any kind of timeout, I've seen it failing to stabilize most of the times, even after three time the current timeout. I've checked the code that does the estimation and it seems to be implemented correctly, according to [8]. I suspect that the epochs length might not be optimal for Tor's latency.

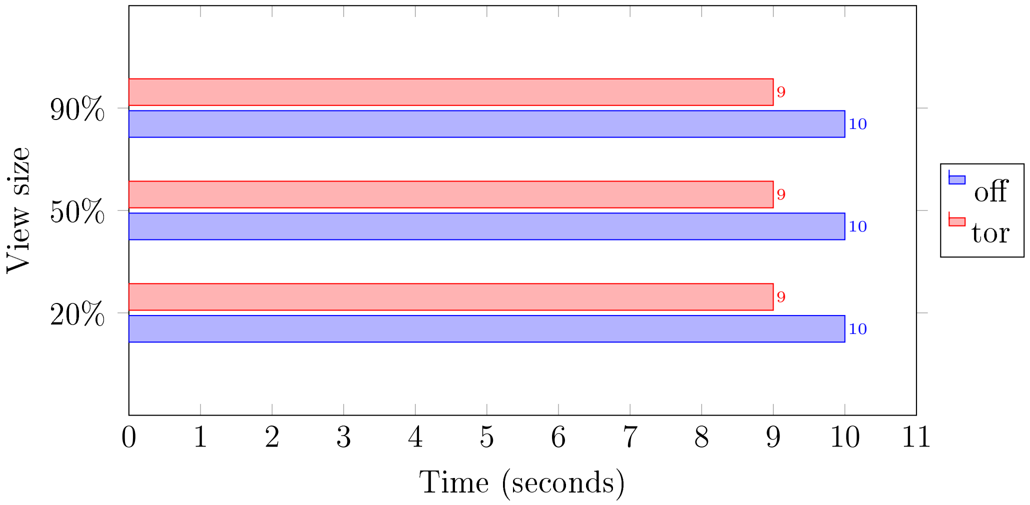

**Figure 4.5:** (first) Run-nodes completed without incurring in the timeout, max is 10

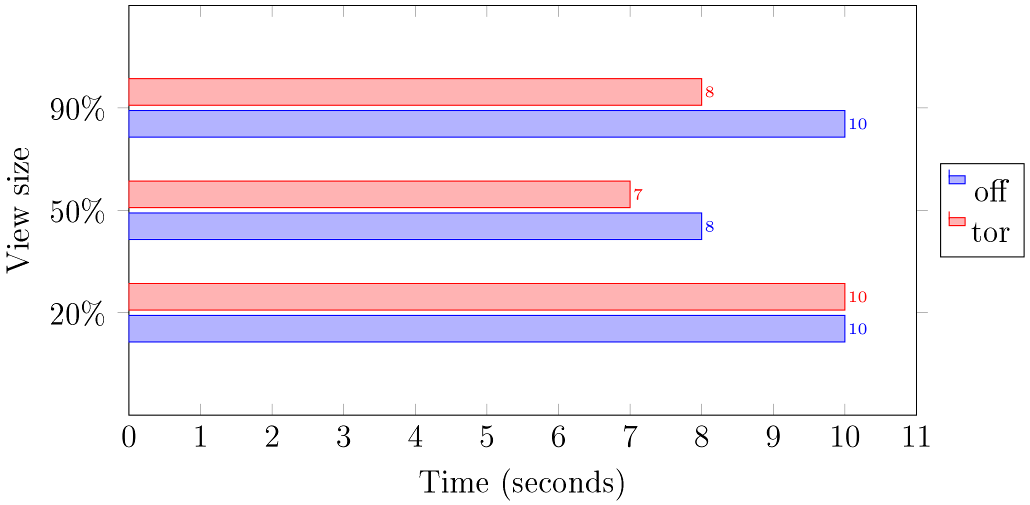

**Figure 4.6:** (second) Run-nodes completed without incurring in the timeout, max is 10

# Conclusions

We wanted to see if it were possible to create an infrastructure-level software for the offering of anonymous cloud services, starting from an existing prototype based on a peer-to-peer architecture. We did it, the prototype has been successfully adapted and now it's anonymous. Performances has been tested and they are worse that expected. There where good reasons to think of a decrease in performances, starting from the intrinsic delay of the anonymizing networks, followed by the use of Java Remote Method Invocation (that doubles connections), but still I wouldn't have thought them to be that bad.

Another thing to notice is that the original prototype was (and still is) not very failure resistant. There are many situations in which, if something goes wrong, the whole system will hang. Due to the way it is implemented it would be hard and quite time consuming to 'fix' it, and since my work is not about the cloud system itself but just its anonymization, I could not spend time in doing it. This instability might have been the cause of some of the failed tests.

About the tests part, studying Amazon EC2 took a lot of time, almost 1/3 of the time I had. When I started, I knew close to nothing about it, and even if now I can't surely be called an expert, I can say that I know what I'm doing. I might have spent that time better, like in improving the prototype, but knowing that there would have been costs, especially considering that the first idea was to test it with 1000 instances, made me wanted to be very sure of what I was doing.

My personal opinion is that any future development of this project should start by revisiting the core of the prototype communications, to substitute the Java Remote Method Invocation with something more ‘direct’. Soon after that I would advise to try improving its resilience to failures or at least design some kind of fall-back mechanism to avoid ending up in inconsistent states.

# Bibliography

[1] National Institute of Standards and Technology, *The NIST Definition of Cloud Computing*, NIST Special Publication 800-145, Peter Mell, Timothy Grance, September 2011

[2] OpenStack cloud operating system, website, `www.openstack.org`

[3] File:Components Diagram.png, wiki.openstack.org, website, `https://wiki.openstack.org/wiki/File:Components_Diagram.png`

[4] OpenStack Operations Guide, October 28 - 2014, 3. Designing for Cloud Controllers and Cloud Management, page 33

[5] *Progettazione e Sviluppo di un Sistema Cloud P2P*, Michele Tamburini, 2010-2011, thesis

[6] *Design and Implementation of a P2P Cloud System*, Ozalp Babaoglu, Moreno Marzolla, Michele Tamburini, Technical Report UBLCS-2011-10, September 2011, Department of Computer Science, University of Bologna

[7] Diagramma dell'architettura a livelli interna al nodo (Node's internal architecture diagram), *Progettazione e Sviluppo di un Sistema Cloud P2P*, Michele Tamburini, 2010-2011, thesis, page 69

[8] *Gossip-Based Aggregation in Large Dynamic Networks*, M. Jelasity, Alberto Montresor, and Ozalp Babaoglu

[9] *How the NSA Almost Killed the Internet*, Steven Levy, 01/07/14, web article, `http://www.wired.com/2014/01/how-the-us-almost-killed-the-internet/`

[10] Glenn Greenwald - *No Place To Hide*, website, `http://glenngreenwald.net/`

[11] *Anonymity Loves Company: Usability and the Network Effect*, Roger Dingledine, Nick Mathewson, The Free Haven Project, January 2 - 2005

[12] Nothing to hide argument, Wikipedia, `https://en.wikipedia.org/wiki/Nothing_to_hide_argument`

[13] *Privacy Rights: Moral and Legal Foundations*, Adam D. Moore, page 203

[14] *Tor Project: Anonymity Online*, website, `https://www.torproject.org/`

[15] *The Invisible Internet Project*, website, `https://geti2p.net/`

[16] I2P Compared to Tor, Comparison of Tor and I2P Terminology, `https://geti2p.net/en/comparison/tor`

[17] *Tor: The Second-Generation Onion Router*, 6.3 Directory Servers, page 11, Roger Dingledine, Nick Mathewson, Paul Syverson

[18] *The Base16, Base32, and Base64 Data Encodings*, RFC 4648

[19] Network topology, website, `http://en.wikipedia.org/wiki/Network_topology`

[20] CamelCase, website, `http://en.wikipedia.org/wiki/CamelCase`

[21] *Gossip-Based Peer Sampling*, M. Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, Maarten Van Steen, 2007, Article 8 / 5

[22] *T-MAN: Gossip-based fast overlay topology construction*, M. Jelasity, Alberto Montresor, Ozalp Babaoglu, April 2009

[23] *Ordered Slicing of Very Large-Scale Overlay Networks*, M. Jelasity - University of Bologna - Italy, Anne-Marie Kermarrec - INRIA/IRISA - Rennes - France

[24] *java.rmi.registry.LocateRegistry*, Oracle Java documentation, website, https://docs.oracle.com/javase/7/docs/api/java/rmi/registry/LocateRegistry.html

[25] SOCKS Protocol Version 5, RFC 1928, March 1996, https://www.ietf.org/rfc/rfc1928.txt

[26] Tor FAQ, website, https://www.torproject.org/docs/faq.html.en#torrc

[27] Marshalling, website, http://en.wikipedia.org/wiki/Marshalling_%28computer_science%29

[28] Amazon EC2, website, http://aws.amazon.com/ec2/

[29] Google Cloud Platform, website, https://cloud.google.com/

[30] Secure Shell (SSH), website, http://en.wikipedia.org/wiki/Secure_Shell

[31] Instance Types, website, http://aws.amazon.com/ec2/instance-types/

[32] Instance Lifecycle, website, http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-lifecycle.html

[33] Instance Storage, website, http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/InstanceStorage.html

[34] Elastic Block Storage, website, http://aws.amazon.com/ebs/

[35] Amazon Machine Image, website, http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html

[36] EC2 Supported Virtualization Types, website, http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/virtualization_types.html

[37] Paravirtualization, website, http://en.wikipedia.org/wiki/Paravirtualization

[38] Hardware Virtual Machine, website, http://en.wikipedia.org/wiki/Hardware-assisted_virtualization

[39] Debian AMI on the AWS Marketplace, website, https://aws.amazon.com/marketplace/pp/B00AA27RK4/ref=sp_mpg_product_title/178-2204224-4541844?ie=UTF8&sr=0-2

[40] Debian, website, https://www.debian.org/

[41] Jessie AMI, website, https://wiki.debian.org/Cloud/AmazonEC2Image/Jessie

[42] EC2 Pricing, website, http://aws.amazon.com/ec2/pricing/

[43] AWS Account Limits, website, http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html

[44] Boto, website, https://boto.readthedocs.org/en/latest/

[45] Fabric, website, http://www.fabfile.org/

[46] Paramiko, website, http://www.paramiko.org/