CLOUDMAN: A P2P FRAMEWORK FOR LARGE, DECENTRALIZED NETWORKS

Relatore: Alberto Montresor
Laureando: Erik Dassi

Anno Accademico 2006 - 2007
## Contents

1. **Introduzione**  
   1.1 Introduzione ................................................. 1  
   1.2 Stato dell’arte ............................................ 2

2. **Introduction**  
   2.1 Introduction .................................................. 4  
   2.2 Related Work ................................................ 5

3. **Cloudman Architecture**  
   3.1 The Cloud ...................................................... 7  
   3.2 Application Definition ...................................... 7  
   3.3 Middleware Layer .......................................... 8  
   3.4 Application life-cycle ...................................... 8

4. **T-Man**  
   4.1 Introduction ................................................. 10  
   4.2 Topology Construction Problem ............................ 11  
   4.3 Description .................................................. 11  
   4.4 Startup and Termination ................................... 13  
   4.5 Configuration ................................................ 14  
   4.6 Sample Topologies ......................................... 15

5. **Newscast**  
   5.1 Introduction ................................................. 16  
   5.2 Peer Sampling Service ...................................... 17  
   5.3 Protocol Description ....................................... 17  
   5.4 Configuration ............................................... 19

6. **Implementation Details** .................................. 20
CONTENTS

6.1 Introduction ................................................. 20
6.2 Cloudman Framework ...................................... 20
6.3 Configuration ................................................ 21
6.4 Protocol and EpidemicProtocol ......................... 21
6.5 Cloudman Epidemic Manager ............................. 23
6.6 Peer Sampling Service ..................................... 23
6.7 Newscast ..................................................... 24
6.8 Descriptor and TMandescriptor ......................... 25
6.9 T-Man ......................................................... 25
6.10 Rankers ...................................................... 27

7 Testing and Performances ................................. 29
  7.1 Introduction ................................................. 29
  7.2 LAN Testing ................................................ 29
  7.3 Configuration .............................................. 30
  7.4 Logging ...................................................... 32
  7.5 PlanetLab .................................................... 33
  7.6 Performances ............................................... 34

8 Conclusion ..................................................... 38

Bibliography .................................................... 40

List of Figures .................................................. 41
Chapter 1

Introduzione

1.1 Introduzione

Oggi internet offre un ampio spettro di possibilità per quanto riguarda i servizi e le applicazioni che vi possono essere eseguite. Una parte importante di queste è venuta alla luce con l’introduzione del peer-to-peer: eliminando i server e i relativi costi, questo paradigma di design ha portato ad un enorme sviluppo in molte aree: condivisione file, grid computing, ecc. Queste applicazioni sono diventate molto popolari, ma nonostante si stiano continuamente evolvendo soffrono ancora di una restrizione importante: i protocolli P2P ed i relativi sistemi sono adattati ad una specifica applicazione. Ciò significa che questi sistemi sono capaci di eseguire un solo tipo di applicazione, impedendo la coesistenza di differenti servizi e quindi sfruttando solo una minima parte delle potenzialità dei sistemi oggi disponibili.

Per eliminare questa restrizione ed ottenere quindi la flessibilità desiderata, è stata proposta l’idea di una descrizione dichiarativa delle applicazioni da eseguire (specificando quali componenti dovrebbero essere eseguiti e su quali risorse), completamente separata dalla sottostante rete di nodi. Ciò ha portato alla definizione di un intero framework centrato su questa proposta: Cloudman.

Cloudman è un framework generico per l’esecuzione di applicazioni in reti P2P, capace di gestire insiemi di nodi altamente dinamici e di dividere le risorse disponibili tra le applicazioni da eseguire. In questo framework i nodi sono lasciati auto-organizzarsi per portare a termine il compito richiesto, ricevendo soltanto una descrizione delle applicazioni e delle loro richieste. Una descrizione completa ad alto livello dell’architettura di Cloudman può essere trovata in [5].

Uno dei compiti principali del framework è la costruzione di reti overlay da zero in modo da poter organizzare un insieme di nodi distribuiti per partecipare all’esecuzione della stessa applicazione: il protocollo T-Man, implementato in questa tesi, offre questo tipo di servizio. Per fornire funzionalità di peer sampling (cioè l’abilità di
ottenere un campione casuale dei nodi che formano la rete quando richiesto), di cui vi è bisogno per l’esecuzione efficiente di protocolli di tipo gossip, è stato implementato come servizio di base per l’intero framework anche il protocollo Newscast.

Nella sezione 2 di questo capitolo presentiamo una descrizione dello stato dell’arte relativo a questa tesi, quindi passiamo ad introdurre la struttura del framework (capitolo 3) e descriviamo gli aspetti teorici di T-Man (capitolo 4) e Newscast (capitolo 5). Il sesto capitolo è dedicato alla spiegazione della nostra implementazione dei protocolli nel dettaglio, mentre quello successivo descrive il testing effettuato sul codice sorgente e le caratteristiche prestazionali del sistema (capitolo 7). Concludiamo quindi la tesi nel capitolo 8.

1.2 Stato dell’arte

Internet sta subendo una crescita esponenziale in dimensione, quantità di utenti e richiesta di applicazioni da poter vi eseguire. I sistemi con server centrale hanno iniziato a soffrire di problemi di scalabilità man mano che il numero di utilizzatori aumentava; per di più, la manutenzione dei server e delle relative infrastrutture è decisamente costosa. Per porre rimedio a questi problemi, è emerso un nuovo paradigma di programmazione: il peer-to-peer (comunemente noto come P2P).

Il termine “peer-to-peer” si riferisce ad una classe di sistemi e applicazioni che utilizzano risorse distribuite per eseguire una funzione in maniera decentralizzata[1]. In pratica, le reti P2P sono sistemi distribuiti dinamici e di grande dimensione in cui nodi sono collegati ad altri nodi in assenza di qualsiasi entità server. Questo paradigma è stato applicato per la prima volta in applicazioni di file-sharing quali Napster o Gnutella: è ora impiegato in molti tipi di scenari, anche scientifici (come, ad esempio, l’esecuzione di calcoli distribuiti).

Si presenta quindi la necessità di far sì che questi nodi siano connessi in maniera organizzata, per permettergli di portare a termine dei compiti, offrire servizi o semplicemente comunicare tra di loro. Una risposta a questo bisogno sono i popolari protocolli gossip-based: l’idea centrale di questa soluzione è che i nodi scambino continuamente messaggi (contenenti informazioni sui nodi conosciuti da chi spedisce il messaggio) con loro pari scelti in maniera casuale, fino a quando ciascun partecipante arriva a conoscere un particolare sottoinsieme della rete (questo insieme può essere determinato da fattori diversi come distanza, disponibilità di certe risorse, ecc) che rappresenta il migliore insieme di vicini per quel nodo.

Un semplice esempio di questa classe di protocolli è Newscast, che scambia periodicamente messaggi con altri nodi per aumentare la sua conoscenza della rete oppure per aggiornare le informazioni che la riguardano. Una descrizione completa del protocollo può essere trovata in [6].

Newscast può svolgere il compito di strumento che fornisce a ciascun nodo i suoi pari con cui scambiare informazioni: il peer sampling service, che permette anche di
inizialire i protocolli *gossip-based* con nodi selezionati in maniera casuale ed uniforme dalla rete. Una proposta di framework per il peer sampling service è descritta in [2].

I protocolli *epidemici* sono basati sugli stessi principi di gossiping ed utilizzano concetti estratti dalla biologia. Il loro obiettivo è di diffondere un aggiornamento dei dati (la cosiddetta *infezione*) il più velocemente possibile su tutti i nodi che compongono la rete. Inizialmente, un singolo nodo possiede i dati aggiornati (diciamo che è *infetto*): quindi entra in contatto con altri nodi in modo da infettare anche questi. I protocolli di questo tipo sono efficienti e robusti ma le garanzie sulle loro prestazioni sono esclusivamente probabilistiche e non deterministiche.

Quindi si presenta il problema di suddividere la rete seguendo diversi criteri, come il numero di nodi, le risorse disponibili, l’affidabilità di un nodo e il bisogno di affidabilità del servizio richiesto, ecc. Questo problema è noto come *slicing* e le sue funzionalità sono una parte importante di *Cloudman*. Una introduzione a questo problema può essere trovata nell’articolo introduttivo del framework [5].

Il naturale passo successivo consiste nell’unire tutti i nodi ottenuti tramite slicing, formando quello che è noto come *overlay network*. Ciò è lo scopo primario del protocollo *T-Man*, un componente fondamentale del framework *Cloudman*. Per una descrizione completa di questo protocollo leggere [4].

Un servizio che include il protocollo *T-Man* come componente fondamentale è il *bootstrapping service*, responsabile dell’avvio di una grande rete da zero, unendo insieme i nodi e le relative risorse, permettendo l’esecuzione delle applicazioni e gestendo massicce adesioni e dipartite dalla rete. Questo servizio è un’altra parte importante del framework *Cloudman* ed è proposto in [3].
Chapter 2

Introduction

2.1 Introduction

Nowadays the Internet offers a wide range of possibilities for services and application to be executed. An important part of these have come to light with the introduction of peer-to-peer computing: eliminating servers and relative costs, this design paradigm led to an enormous development in a lot of areas: file sharing, grid computing, ecc. These applications have become very popular, but nonetheless they are continuously evolving they still suffer from one important restriction: P2P protocols and related systems are tailored to a specific application. This fact means that systems are capable of running only one kind of application, blocking the co-existence of different services and thus taking advantage of a minimal part of today system’s potential.

To relax this restriction and thus obtain the desired flexibility, the idea of a declarative description of the applications to be run (specifying what components should be executed on what resources), completely separated from the underlying network of nodes was proposed. That gave rise to the definition of a complete framework centered around this idea: Cloudman.

Cloudman is a general-purpose framework for applications execution in P2P networks, able to manage highly dynamic nodes set and divide available resources between applications to be run. In that framework nodes are left to self-organize in order to execute the required tasks, only receiving a description of applications and relative requests. A complete high-level description of Cloudman architecture is [5].

One of the main tasks of the framework is the construction of overlay networks from scratch in order to be able to organize a set of distributed nodes to participate in executing the same application: T-Man protocol, implemented in this thesis, offers that functionality. In order to provide peer sampling functionalities (that is the ability to obtain a random sample of the nodes forming the network when requested), needed for efficient gossip-protocols execution, the Newscast protocol was also implemented...
2.2 RELATED WORK

as a basic underlying service for the whole framework.

In section 2 of this chapter we present a description of the related work for this thesis, then we move on introducing the structure of the framework (chapter 3) and we describe the theoretical aspects of T-Man (chapter 4) and Newscast (chapter 5). Chapter 6 is devoted to explaining our implementation of the protocols in detail, while the successive one describes testing done on the source code and performance characteristics of the system (chapter 7). We then conclude the thesis in chapter 8.

2.2 Related Work

The Internet is experiencing an huge increase in size, user quantity and requests for applications to be performed. Server centric systems have started to suffer scalability problems as user number increased; more, the maintenance of servers and their infrastructures is very expensive. In order to solve these problems, a new design paradigm has emerged: peer-to-peer computing (commonly known as P2P).

The term “peer-to-peer” refers to a class of systems and applications that employ distributed resources to perform a function in a decentralized manner[1]. In practice, P2P networks are large, dynamic and fully distributed systems, in which nodes are connected to other nodes without the presence of one or more server entities. This paradigm has been applied for the first times in file-sharing applications like Napster or Gnutella: it is now used in a wide range of applications, even scientific ones (like, for example, distributed calculations performing).

There is a basic need for these nodes to be connected in an organized way, in order to allow them to perform a task, offer a service or simply communicate with each other. A popular answer to this need are gossip-based protocols: core idea of this solution is that nodes continuously exchange messages (containing informations on the nodes known by the message sender) with random peers until every participant gets to know a particular subset of the network (this set may be determined by different factors, like distance, availability of certain resources, ecc) that is the best set of neighbours for that node. A simple example of this class of protocols is Newscast, that periodically exchange messages with other nodes in order to increase its knowledge of the network or to refresh some informations about it. A complete description of this protocol may be found in [6].

Newscast may be used as a tool that provide each node with the peers to exchange informations: the peer sampling service, that also allows to initialize gossip-based protocols with peers selected in random and uniform fashion from the network. A proposed framework for peer sampling service is described in [2].

Using concepts taken from biology, epidemic protocols are based on the same principles of gossiping. Their objective is to spread data update (the infection) as fast as possible to all the nodes forming the network. At first a single individual owns the updated data (we say it is infective): it then gets in touch with other
individuals in order to make other nodes infective too. The resulting protocols are efficient and robust but guarantees on their performances are only probabilistic and not deterministic.

Then come the problem of partitioning the network following different criteria, like number of nodes, available resources, reliability of a node and the need of a service for reliability, etc. This problem is known as slicing and its features are an important part of Cloudman. An introduction to this problem can be found in framework’s introductory paper [5].

The next natural step is to join together all the nodes obtained by slicing, forming what is known as an overlay network. That is primarily the scope of T-Man protocol, a fundamental building block of Cloudman framework. For a complete description of this protocol look at [4].

Including T-Man protocol as a fundamental building block is the bootstrapping service, responsible for bootstrapping a large network from scratch, gluing together nodes and their resources, allowing for application execution, massive joins and massive departures to a network. This service is another important part of Cloudman framework and it is proposed in [3].
Chapter 3

Cloudman Architecture

The P2P world is in a continuing growing phase: a lot of protocols are available for
the most disparate functions. But each one of them is tailored to a specific function
or service. What Cloudman aims to offer is a general-purpose framework allowing
to execute several applications independently across a very large scale and dynamic
network. The basic idea behind it is a list of applications description (specifying
what to run and what resources are needed) called Application Suite Description,
and a middleware layer responsible for applications management (starting and stop-
ping, building overlay topology for subset of the nodes, failures handling, etc) and
controlling that resources are subdivided as applications list prescribes.

3.1 The Cloud

We consider large groups of networked nodes, connected through a routed network.
These groups are seen as highly dynamic and unreliable. Nodes may join and leave
at any time: we call this environment a cloud. The cloud is formed and managed
by gossip-based protocols allowing nodes to get to know a subset of the other nodes
of the cloud. Every time a new application is started, a subset of the nodes called
sub-cloud is selected. Then an overlay topology is built on top of the sub-cloud, and
the application given access to the resources of the involved nodes.

3.2 Application Definition

Cloudman uses of a declarative description of a desired application that specifies what
services, components, etc., should be running on what resources. In our proposal, the
underlying system of nodes and the application descriptions are completely separated:
the descriptions are disseminated throughout the nodes who then self-organize to
ensure that the desired applications are simultaneously supported[5].
We introduce the Application Suite Description, containing the definition of all applications running on the network, and what resources are assigned to each of them. The description is independent of the actual set of nodes, that means description indicates proportion of the nodes to assign to a given application, and optionally desired properties for these nodes. All nodes knows, or have means of accessing to, the suite description, so everyone can see what applications compose the suite and how resources are divided.

3.3 Middleware Layer

To glue together the nodes (and their resources) and the suite description specifying how to subdivide and use this resources, we define a middleware layer that has to perform following services:

- **Slicing**: assigns right proportions of resources (by mean of nodes) forming the cloud to applications.
- **Bootstrapping**: starts up a new application from scratch, building overlay topology over the assigned sub-cloud.
- **Churn handling**: helps applications to cope with churn and failures. This service is responsible for adding and removing nodes from an application, and to recover from failures or massive nodes crashes.

In short the middleware has the task to connect the two entities, the cloud and the application suite description: this means that all nodes in the cloud have to know what applications to run and on what resources, and joining or leaving a particular application should be managed by this layer via slicing and churn handling using the procedures the current application defines.

3.4 Application life-cycle

Using these components connected together by middleware services, an application will undergo the following typical life-cycle:

- The user writes an application with an API that allows him to access Application Suite Description and middleware layer services (for starting and managing application).
- The Application Suite Description is updated inserting the user application and assigning it a portion of the cloud (ex: 20% of the nodes)
- The bootstrapping service will build up an overlay topology connecting all nodes assigned to the application, forming a sub-cloud and giving the application access to it.
The application code to be executed is spread to all nodes of the assigned sub-cloud, beginning from the user that inserted application. A node part of the sub-cloud that have not received the code, will actively try to get it from his neighbors.

When execution of the application is finished, it is simply removed from the suite description, end its nodes are released in order to be used by next application requesting resources.
Chapter 4

T-Man

4.1 Introduction

Overlay networks have become one of the most important instruments for implementing a large range of functions in fully decentralized systems. Nowadays every application has its own way of creating overlay, based on task-dependent factors and building different topologies (rings, trees, etc). As the task becomes more and more important, and in order to improve usability and efficiency, it would be desirable to have a unique, generic protocol for topology building: it could allow dynamic construction of overlay networks for every kind of new application, and allow easy maintenance and upgrades in the future.

*T-Man* is an algorithm for creating a wide class of overlay topologies from scratch: the main idea is the *ranking function*, that defines an ordering of the nodes dependent on the preference of a given base node to select the other nodes as desirable neighbors. We can so construct the most disparate topologies by simply changing this function with a new one, resulting in a very configurable and flexible system. The protocol is gossip-based, so all nodes exchange periodical messages with a randomly selected peer: informations exchanged are about the current known neighbors of the sender, so receiving these messages every node can improve his own set of neighbors, leading to fast and efficient convergence to the desired overlay topology.[4]

*T-Man* allows *Cloudman* to execute an application from scratch, building up an overlay network connecting all the nodes that agreed to participate in the task. The framework can then take advantage of the created overlay, in a fully distributed way, to coordinate application execution. T-Man finds its place in the middleware layer of *Cloudman*, precisely in *bootstrapping* service.

The protocol has been introduced and described by its creators in [4]; here I present a detailed description of it, needed to understand the implementation.
4.2 Topology Construction Problem

The main idea of T-Man is to connect all the nodes in a network to their best neighbors, in order to build the chosen topology. We consider a set of nodes in a network: each one has an address and a communication port, that are needed to communicate with other nodes across the network. Each node builds a view, that is an ordered set of node descriptors (a node descriptor is composed by address, communication port, and informations needed to rank the node, like for example geographical location or available resources). The descriptors contained in the view represents links between nodes in the resulting overlay network topology. We may describe the topology construction problem as follows:

- **input**: the inputs of the problem are the $N$ nodes that compose the network, the desired target view size $K < N$ and the ranking method to be used.

- **output**: the output consists in a target view composed of the $K$ nodes with highest ranking (according to the local node) of all the network. Formally we may say that the target view of node $a$ will consist of the first $K$ elements output by a call of $RANK(a, \text{all nodes - a})$, where $RANK$ is the ranking method provided as an input to the problem. An example of ranking method may for example calculate increasing distance from the base node and choose the nearest neighbors to be the best ones.

4.3 Description

Protocol rely on the ranking function, that is a function saying how to order a set of nodes, with respect to a given base node preference for choosing these peers as neighbors in desired topology. As an example, in a sorted ring topology, the first nodes of the ordered set will be the ones just following and preceding the base node in the ring. The rest of the nodes in the ordered set will be peers that are more and more far from the base node. By intuition we can affirm that this function defines completely an instance of the protocol, as it determines which topology will be built: ranking function indicates how to connect nodes to form the topology, by mean of a higher ranking of a given node with respect to the base one. This approach leads to a great degree of flexibility: by simply changing the implementation of the ranking function, and keeping constant the algorithm that is executed, we can change the topology that is to be built. This allows to think of T-Man being adapted to the most disparate application with little additional effort.

Each node has a descriptor, that contains its ranking id (used to compare it with other nodes) and informations needed to send it a message (ip address and protocol port number). The descriptor represent what is exchanged between nodes when executing the protocol: each node maintain a view, that is a set of node descriptors,
one for each known node in the network. By exchanging messages with peers, cycle by cycle, the View content is refined, always keeping at top the higher ranked nodes, until convergence is reached. Convergence happens when, for a fixed number of cycles, the first $\psi$ nodes from which we select a peer to communicate with don’t change, indicating that the node has already come to the list of his ideal peers.

The protocol executed by each node is the following:

- **Active Thread** is responsible for periodically initiate communication with a selected peer:
  1: loop
  2: \texttt{wait(}\Delta\texttt{)}
  3: $p \leftarrow \texttt{selectPeer}(\psi, \texttt{rank}(p, \texttt{view}))$
  4: \texttt{buffer} $\leftarrow \texttt{merge}(\texttt{view, myDescriptor})$
  5: \texttt{buffer} $\leftarrow \texttt{rank}(p, \texttt{buffer})$
  6: send first $m$ entries of buffer to $p$
  7: receive buffer$_p$ from $p$
  8: \texttt{view} $\leftarrow \texttt{merge}(\texttt{buffer}_p, \texttt{view})$
  9: end loop

This thread waits for a certain period of time, $\Delta$ seconds. This is a parameter needed for protocol configuration and indicates how frequently to start a communication. It determines convergence speed (the more frequently messages are sent, the quicker I expect to get to the desired topology) but also the cost of protocol execution (number of messages sent and occupied bandwidth). Every $\Delta$ seconds, the thread wakes up and ask for a peer $p$ to communicate with. The \texttt{selectPeer} method returns one of such. The node merge its descriptors with current cache, ranking the result with respect to the peer $p$ selected to receive the message. First $m$ elements of the resulting cache are inserted into data to be sent. The parameter $m$ is called \textit{messagewindowsize} and represents the maximum number of node descriptors that can be sent to peers in one message. The thread then sends the message and sit down waiting for response. Once received, it merges the cache sent by $p$ with its current one, and freezes again until next timeout that will wake it up.

- **Passive Thread** replies to a message received by another node, sending back its view content and merging the received one:
  1: receive buffer$_q$ from $q$
  2: buffer $\leftarrow \texttt{merge}(\texttt{view, myDescriptor})$
  3: buffer $\leftarrow \texttt{rank}(q, \texttt{buffer})$
  4: send first $m$ entries of buffer to $q$
  5: \texttt{view} $\leftarrow \texttt{merge}(\texttt{buffer}_q, \texttt{view})$

This thread is executed every time a message is sent to current node. First
of all the current cache is merged with node descriptor, ranked with respect to sender node and sent back to it (only first $m$ entries are sent back, as it is the maximum number of descriptors a message can hold). Then the received cache is merged with current node one, terminating thread execution.

We now go on to detail the functions protocol uses to accomplish its task:

- **selectPeer**: ranks the peers contained in node view, then randomly select one of the first $\psi$ nodes (not in tabu list) and returns it. The *tabu list* is a list of the most recent peers the node communicated with (the list size parameter is part of the protocol configuration). When a peer is selected, we first check if it is in the tabu list. If it is the case, we go on searching for another peer that is in the first $\psi$ nodes and that is not in tabu list. Otherwise, the peer is a valid choice and we select it for communication. The tabu list is updated by adding the chosen peer (so that in the next cycles we remember having recently communicated with it) in a circular fashion (after having filled the last available element in tabu list we return back to the first one for insertion of new peers). This improvement over the basic implementation of T-Man helps increase the diversity of selected peers\(^4\), counterbalancing the influence of a small $\psi$ that leads to frequent selection in a small set of peers.

- **rank**: given the cache and the base node, it returns the cache ordered with respect to the base node. This is the ranking function (defined above) and its implementation is exterior to the core protocol implementation. Everyone can write a class defining a ranking function (and implicitly an overlay topology to build), and then assign it to the protocol via configuration.

- **merge**: this method is defined as the set operation *union* of two sets, that is it merges the current cache and the received one eliminating duplicates and inserting nodes descriptors in order (considering their ranking id). It is implemented in the same class of the ranking function, as there could be the need to perform topology-dependent computations when merging.

### 4.4 Startup and Termination

When T-Man is started at a node, it is at first isolated, as it does not know any other node and no node knows it. The protocol definition indicates that when starting, the view of a node should be filled with a random sample of the network, provided by the *peer sampling service*. Here we consider this service being based on the *Newscast protocol*\(^6\). The main idea is that each node maintain a local set of random node addresses. Each node periodically sends the content of this set to a member of the set itself. When receiving a message, the local node keeps a fixed number of the most recent addresses (using a timestamp) from the local view and the view contained in
the received message, quickly eliminating nodes that crashed or disconnected from
the network. This protocol is extremely lightweight and low cost to execute.

Apart from the protocol being randomly initialized, other strategies are available
to start T-Man:

- **Fixed Node**: there is an always active node known by everyone in the network:
  when starting T-Man we contacts this node, getting a view message from it
  and so getting descriptors of some nodes participating, thus entering in protocol
  cycle.

- **Broadcast**: at the first there is only an active node called *initiator*. This node
  performs a broadcast with a special *wake-up* message: every node receiving
  this message immediately starts the protocol. Another source of activation for
  a node is receiving a T-Man message from a node that has already started
  executing the protocol.

We say that the protocol is terminated when the target overlay topology has
been built, that means when every node has evolved his cache to the correct list of
peers. It is quite difficult and above all, expensive, to detect when a topology is built.
So we need a mechanism to detect this situation (called convergence, as described
above) and terminate the protocol in every node independently from the others. The
creators of T-Man proposed the following: each node monitors its cache for changes
(addition of new nodes). If for a certain period of time no changes are detected then
the node suspends its active thread, going in a state called *suspended*. If a change in
the cache happens while in suspended state (in consequence to a message received
from an active node) then the node switches to active state and resets its suspended
state timers[4].

### 4.5 Configuration

Flexibility is one of the most important characteristic of the protocol, allowing it to
be used in a wide range of different situations without little or no changes at all. All
the parameters needed to run the protocol have been extensively described above,
we present here only a recapitulation of them:

- **ranking function**: indicates how nodes are ranked with respect to a given base
  node, defines the resulting overlay topology.

- **ψ**: indicates the number of cache nodes considered when selecting a peer (a
  peer is selected between the first ψ nodes of the cache).

- **Δ**: indicates how frequently the protocol starts a new communication. This
  parameter determines the speed of convergence to the desired topology but
  also the cost, in term of sent messages, of protocol execution.
4.6. SAMPLE TOPOLOGIES

- **message window size** $m$: maximum number of nodes descriptor a message can hold (the number of nodes descriptor sent in one message).

- **tabu list size**: size of recently selected peers list (in initiating a message exchange). The value determines the diversity of such peers.

- **$\Delta_{\text{suspended}}$**: indicates after how much time of the view remaining unchanged the active thread is suspended.

### 4.6 Sample Topologies

We describe here two very different simple topologies (and their ranking functions) that can be used with T-Man, that are:

- **Sorted Ring**: the target topology is a ring in which nodes are ordered by their ranking id (an integer of $n$ bits, randomly initialized by each node) except from the first and the last one (which will have biggest and smallest ranking id value), which are connected to each other to close the ring. The ranking function works as follows: the nodes are ordered with respect to the base node $x$ and a ranking value is calculated for each of them. This value represents the number of hops in the ring needed to go from ranked node to the base one. The nodes are then returned ordered by this value.

- **Binary Tree**: the target topology is a tree in which every node has a parent (or it is the root) and at maximum two children. The ranking ids of the nodes are defined as follows: supposing the network contains $N$ nodes, we assign the values between 1 (assigned to the root) and $N$ to the nodes. Using binary representation we say that numbers starting with 1 are assigned to leafs. Numbers starting with 0 like $0a_1...a_m$ are assigned to non-leaf nodes that have two children: $a_1...a_m0$ and $a_1...a_m1$. The nodes are sorted with respect to their ranking value that is calculated as the distance (based on the path length between the two nodes profiles) between current node and base node $x$. 


Chapter 5

Newscast

5.1 Introduction

The introduction of *T-Man* in the previous chapter introduced two new problems needing a solution: how to startup these kind of protocols with a uniform and random subset of the network and, in a more general way, how to provide a random peer when requested by them. We would like to have a distributed and lightweight protocol capable of aggregating data about the network, for example counting nodes and relative resources. A protocol responding to these requirements will be fast and runnable on a wide range of devices, even small and with poor resources.

*Newscast* is a p2p protocol created to address these issues. Its objective is to maintain and disseminate informations about the nodes forming the network, that may be unreliable and highly dynamic, without a central coordination. All nodes are equivalent and run the same algorithm: the basic idea is that nodes communicate with randomly selected peers at periodic intervals of time. The content of exchanged messages include *membership* informations like addresses of other participant nodes but also piggybacked application-specific informations (for example, in the case of *T-Man* a numeric ranking identifier). *Newscast* associate a timestamp with each information and, cycle by cycle, keeps only the most recent ones, thus quickly eliminating information on nodes that crashed or left the network. The cost of algorithm execution is little, and so is the occupied bandwidth because of low message sending rate, leading to a lightweight and efficient protocol, as required to fulfill the task.

*Newscast* is used in *Cloudman* to provide the initial set of peers needed to startup *T-Man*, also allowing, by the mean of exposed services, protocols to obtain single random peers from the network when needed. This protocol finds its place in the middleware layer of *Cloudman*, precisely in *bootstrapping* service.

The protocol has been introduced and described by its creators in [6]; here I present a detailed description of it, needed to understand the implementation.
5.2 PEER SAMPLING SERVICE

At the heart of the need for a peer-providing protocol lies the concept of the peer sampling service. Gossip protocols have been proved to be efficient and highly reliable, under the assumption that the peers they communicate with are randomly selected from the entire network. The aim of this service is to provide protocols with such peers, in a fully distributed way. The definition proposed in \[2\] essentially expose one method (apart from init method that we will not consider here as its function is intuitive): getPeer. Its role is to provide a peer address to communicate with, obtained as a sample from the known nodes set.

In the framework, Newscast is used as a peer sampling service, also providing startup peers to protocols requiring this facility (for example, T-Man): protocols register to the service in order to receive this set of node descriptors.

5.3 Protocol Description

The core idea of the protocol consists in nodes communicating at periodic interval of times with its correspondents (where a correspondent is a peer known by current node): each of them knows only a (continuously changing) small set of peers of which one is randomly chosen to exchange information\[6\] in order to increase their knowledge of the network and to mantain updated informations about it. Here we present a version of the protocol that is only interested in building a cache containing a subset of the peers forming the network, but Newscast may be used with little changes to collect informations like statistics on nodes efficiency, speed, and so on but also like resources available on each node. The protocol startups by contacting a fixed node in the network (node we assume always on): that node will send back its cache, allowing for Newscast on the local node to start communicate with an initial set of peers. While running, the protocol relies on three algorithms for executing its task:

- **Active Thread**: is responsible for periodically initiate communication with a randomly selected peer. Its code is the following:

```
1: loop
2: wait(Δ)
3: p ← getPeer()
4: send cache to p
5: receive cache_p from p
6: cache ← merge(cache, cache_p)
7: end loop
```

After having waited a fixed interval of time (Δ seconds) the thread requests a random peer via the peer sampling service method getPeer. It then sends the local cache to the obtained node and waits for the latter to reply by sending
its cache. Eventually the thread calls \texttt{merge} function in order to update the cache by mixing the local with the received one.

- \textit{Passive Thread}: the passive thread is responsible for handling and replying to messages received from other peers of the network. Its code is quite simple, and is the following:

\begin{verbatim}
1: loop
2: receive cache\(_p\) from \(p\)
3: send cache to \(p\)
4: cache ← \texttt{merge}(cache, cache\(_p\))
5: end loop
\end{verbatim}

The thread waits for incoming messages: when it receives one from peer \(p\) it replies by sending back the local cache to that peer. Then it proceeds to merge its cache with the received one and then sits down again waiting for another message.

- \textit{merge}: this function is responsible for merging two caches by considering both the presence of the same peers and their associated timestamps. Peers found in both caches will be saved once with the most recent timestamp in order to get the most updated informations. Here we present the most important part of the algorithm:

\begin{verbatim}
1: while cache and cache\(_p\) have items do
2: \(b ← \text{randomBoolean()}\)
3: if \(b\) then
4: if not contains(cache\(_\text{final}\), cache[i]) OR timestamp\(_i\) > timestamp\(_\text{final}\) then
5: add cache[i] to cache\(_\text{final}\)
6: \(i ← i + 1\)
7: end if
8: else
9: if not contains(cache\(_\text{final}\), cache[j]) OR timestamp\(_j\) > timestamp\(_\text{final}\) then
10: add cache\(_p[j]\) to cache\(_\text{final}\) if it is not local node
11: \(j ← j + 1\)
12: end if
13: end if
14: end while
\end{verbatim}

In this part of the algorithm we proceed to scan the two caches: we compare each peer we want to insert in the final cache to the ones that already in it. If we find a corrispondence we check the timestamp: if it is more recent than the one in the final cache we substitute it in order to keep the freshest
informations only. If the timestamp is older than the one in the final cache we just throw away the peer and proceed further until one of the two caches become empty. This way we are sure that we will obtain only one copy per peer in the final cache and that the timestamps will be the most recent ones. In the remaining part of the algorithm, not present in the code, we just fill the remaining available places of the final cache with the peers found in the cache that is not still empty. Eventually we return the resulting final cache.

As we can see the algorithms driving Newscast execution are quite simple and above all really lightweight: this means that protocol is scalable and efficient, allowing it to be run in huge sized networks and even in low-resources devices like phones, PDA or microcomputers.

## 5.4 Configuration

The protocol was created in order to provide flexibility and capability to be used in a wide range of different scenarios with no changes at all. To permit precise control over Newscast’s behaviour the following parameters are available:

- $\Delta$: indicates how frequently the protocol starts a new communication with a random peer taken from its cache. This parameter determines the time needed to obtain a complete cache and also the cost, in terms of messages sent, of protocol’s execution.

- cache size: this parameter represents the number of host descriptors that Newscast collects and maintain (when cache is full the protocol uses freshest timestamp criteria to decide which descriptors to keep).

- descriptor provider: its value is a reference to the protocol that provides piggy-backed informations (in the case of T-Man a numeric ranking id) later passed to the protocol receiving startup peers set or asks Newscast for a single peer.

- fixedNode: this parameter represent the address of a node (considered always on) that the protocol contacts at startup to obtain a set of peers to start communicate with.
Chapter 6

Implementation Details

6.1 Introduction

The protocol has been implemented as a fundamental building block of Cloudman framework, allowing it to form overlays with part of the nodes composing the cloud, and needed to start and execute distributed applications. As a part of the framework, it can take advantage of all the basic features offered by Cloudman core classes, easing the task of implementation. Another major advantage is that developers creating protocols of higher level than T-Man can easily use the latter as a facility for them: all they need to do is include it in configuration files and then use the resulting overlay topology. Chosen implementation language is Java, allowing the protocol and the whole framework to be deployed on all types of target machines without any effort (using the correct Java Virtual Machine for the target operating system suffices).

In this chapter we first present Cloudman architecture, then we move on describing core interfaces for protocols, a general epidemic manager, the peer sampling service, core T-Man, ranking functions, configuration system and utility classes implementation.

6.2 Cloudman Framework

The framework offers the basic services for the protocols: when launching Cloudman the configuration is read, the local node is initialized with a node manager class (itself specified in configuration) and all the protocols described in that file are started on the node in alphabetic order (otherwise specified with a particular parameter). The configuration system itself (relying on the properties) is provided by the framework.

The framework expose a class named CommonState that allows to obtain a reference to the local node, and in turn, to get the protocols that are executing on the node (for example, a protocol wishing to send a message over the network will
retrieve the transport protocol instance and use its methods). This class also offers some random number generation functions. The base framework contains also base class for node identification (NodeID class can store address and communication port for a node) and the Protocol interface described above: Cloudman in fact loads and handle the protocols by using references to these interface.

Last of all an utility for registering and managing timeouts is present: the class TimeoutManager allows to register a protocol with one or more timeouts in order to periodically trigger some actions (the manager will wake the protocol by invoking its timeout method).

6.3 Configuration

A protocol may need parameter to work correctly, and above all, the framework needs to know which protocols to load and execute at startup. In order to fulfill this requirements, a configuration system was setup in Cloudman.

It is based upon the properties system: a file is built with a section (enclosed by brackets) for each protocol, specifying the name for the category of the protocol (the "property", for example protocol.udptransport) that will be accessible to other sections of the configuration, and the name of the real protocol class that gives value to the property. Inside the section one can specify couples of sub-properties and values that will be used inside the protocol (for example, "period 5", indicating we have a property of the protocol named period, with value 5 that can be accessed by protocol's code, ecc.).

Every protocol that has a section in the configuration file specified when starting the framework will be loaded and executed by the latter, and once loaded will be able to retrieve the value of its sub-properties specified in the file.

6.4 Protocol and EpidemicProtocol

Every class aiming to be used as a protocol in the framework must implement the Protocol interface, which definition is in figure 6.1: We now describe the role covered

```java
public interface Protocol {
    public void receive(NodeID src, int pid, Object msg);
    public void timeout(Object msg, int pid);
}
```

Figure 6.1: Protocol interface code.

by the two methods of this interface:
• **receive**: it is called when the transport protocol receives a message from another node, forwarding the message to correct receiver. In this method protocols can react to request of other nodes and take the appropriate actions.

• **timeout**: if needed a protocol can register for timeouts when it is created. If this is the case, this method will be called whenever a timeout for that protocol expires, allowing to execute periodic actions like, for example, initiating a communication with other nodes.

We have also implemented additional facilities for epidemic protocols; examples of such protocols include T-Man, Newscast, and aggregation. Such protocols must implement a new interface: `EpidemicProtocol`, which extends the previously seen `Protocol` interface and exposes seven new methods (figure 6.2).

```java
public interface EpidemicProtocol extends Protocol {
    public void receivePeers(Descriptor[] peers);
    public Descriptor selectPeer(Descriptor lnode);
    public Message prepareRequest(Descriptor lnode, Descriptor rnode);
    public Message prepareResponse(Descriptor lnode, Descriptor rnode, Message request);
    public void merge(Descriptor lnode, Descriptor rnode, Message msg);
    public Descriptor getDescriptor();
    public void setDescriptor(Descriptor desc);
}
```

Figure 6.2: `EpidemicProtocol` interface code.

Let’s describe the meaning of the methods exposed by this interface:

• **receivePeer**: it is a callback method used by peer sampling service. The latter knows the id of the epidemic protocol it has to send the initial set of peers to. When the peer sampling service has obtained the required number of peers, it gets a reference to the epidemic protocol that needs them. Then the service calls this method, letting the protocol take the appropriate actions on the received peers.

• **selectPeer**: this method is called to obtain a new peer to communicate with, chosen between the ones in protocol’s cache according to the particular epidemic protocols which is run.

• **prepareRequest**: it is called whenever it is needed to start a communication with a peer. It will prepare and return a well-formed request message (according to particular protocol rules) ready to be sent.
• prepareReply: it is called when a message was received by another peer and the protocol must reply to the request. It will prepare and return a well-formed reply message (according to particular protocol rules) ready to be sent.

• merge: this method will merge the informations received by other nodes with the ones currently in protocol cache, according to protocol’s particular algorithm.

• getDescriptor and setDescriptor: these two are utility methods, needed to retrieve or to set the value of current protocol descriptor for the local node.

Basic framework services will handle classes implementing Protocol, while more specific utilities (like peer sampling, descriptors, ecc.) will use EpidemicProtocol classes.

6.5 Cloudman Epidemic Manager

The CloudmanEpidemicManager class was created to help handle a group of protocols behaving in the same way: epidemic protocols. Every one of these follow the same basic life cycle, using an active thread (invoked every $t$ seconds, where $t$ is the period of the protocol) for starting communication with other nodes in the network, and a passive thread, responsible for receiving and replying to incoming messages.

This class implements Protocol interface, as it behaves like a protocol: it registers for timeouts and uses the method for message reception from the transport protocol. When a timeout happens, the manager gets the identifier of the protocol to be waked up, and calls its active thread with this parameter, getting the destination protocol to furnish data to initiate a communication (in particular it will call selectPeer and prepareRequest methods, responsible for getting the next node to communicate with, and to prepare a well-formed message for destination peer). When a message is received, the manager simply looks for the destination node field into the message, then executes passive thread for the protocol to which it was addressed (in particular it will call prepareResponse and merge methods, which prepares a reply message for sender peer, and then merges the received informations with current ones).

A protocol wishing to use the epidemic manager should simply be inserted in the parameters of the epidemic manager, that will register it while being loaded and started by the framework.

6.6 Peer Sampling Service

The role of the peer sampling service is to provide a set of random peers to initialize protocols like T-Man. The service to be offered is the same for all protocols requiring a peer sampling, so a class wishing to be used for that purpose must implement PeerSampler interface, which code is presented in figure 6.3.
public interface PeerSampler
{
    public NodeID getPeer();
    public NodeID [] getPeers(int peersNumber);
}

Figure 6.3: PeerSampler interface code.

The role covered by these methods is the following:

- **getPeer()**: returns a single randomly selected peer (extracted from the cache the service built by communicating with other nodes).
- **getPeers()**: returns a set of randomly selected peers, its cardinality being passed as a parameter to the method.

It is important to note that this two methods may be called anytime by the various protocols that use the peer sampling service, but *T-Man* never use them, as it waits for the peer sampling service to provide the initial set of peers (via the receivePeers method of the EpidemicProtocol interface) and then goes on building its own cache alone.

### 6.7 Newscast

In my implementation, the protocol chosen to cover the role of peer sampling service is *Newscast*. By communicating with the other nodes, it builds up a cache consisting of network addresses and timestamps, updating it every time a message is received and keeping only a fixed number $n$ of nodes, the ones having the freshest timestamps (doing that we ensure that the nodes eventually furnished as peers are still connected and alive with an high probability, reducing the risk of returning peers that are unusable).

public class Newscast implements Protocol, PeerSampler
{
    public NewscastNodeID getPeer() {...}
    public NewscastNodeID [] getPeers(int peersNumber) {...}
    public void mergeCaches(NewscastNodeID [] receivedCache,
                            int receivedDegree) {...}
    public void receive(NodeID src, int pid, Object msg) {...}
    public void timeout(Object msg, int pid) {...}
}

Figure 6.4: Newscast exposed interface.
Figure 6.4 illustrates the interface of Newscast. Behaving as a protocol (of which it implements the interface), it uses receive and timeout methods to respectively receive and periodically send messages to the nodes in the network. The core method of the protocol, doing the real work, is mergeCaches (used only internally), responsible for updating the cache when a new message is received. This method will compare local addresses and timestamps with received ones and when will return a combination of the two, following the rules described above.

Eventually Newscast will fill its cache completely: when this happens, it will pass the so obtained peers to the registered epidemic protocol (by calling the receivePeers method on it), that will use them to start communicate.

6.8 Descriptor and TManDescriptor

The Descriptor class represent the basic node description tool for epidemic protocols. In this protocols there is the need to rank and classify peers according to some of their properties (maybe an id, resources available, proximity, ecc.). This class provides only identification with network properties (that is IP address and communication port) which may seem poor, but that will be understandable by all protocols. By extending Descriptor each of them can add its own ranking informations, as a payload for the class. The message sent will contain a reference to a Descriptor: upon reception a protocol will cast it to the correct specific type, gaining access to the informations it need to rank the node.

The TManDescriptor class is the extension of the base descriptor carrying the informations T-Man needs to rank nodes. In particular the payload consist in an long ranking id (8 bytes wide), that will be numerically compared to id’s of other nodes to be ranked. So the messages exchanged by nodes executing T-Man will contain caches of TManDescriptor.

6.9 T-Man

The T-Man class implements the EpidemicProtocol interface, so exposing all its five methods, and it is managed by the EpidemicManager, providing it messages for request and reply, and merge functionalities. Figure 6.5 shows the methods exposed by protocol interface. Now we go on describing the most important characteristics of our implementation of T-Man, that are:

- **start**: when the protocol is loaded by the framework, it fetches all the parameters from the configuration (ranker class, periods of time, cache sizes, and so on), generates a random node id to identify itself among other nodes executing T-Man, and eventually initialize the cache to an empty one. After this steps the protocol lies idle, waiting for the peer sampling service to call its receivePeers method. When the method is called, the protocol receives an initial set of peers
public class TMan implements EpidemicProtocol
{
    public Descriptor getDescriptor() {...}
    public void setDescriptor(Descriptor desc) {...}
    public Descriptor selectPeer(Descriptor lnode) {...}
    public Message prepareRequest(Descriptor lnode, Descriptor rnode) {...}
    public Message prepareResponse(Descriptor lnode, Descriptor rnode, Message request) {...}
    public void merge(Descriptor lnode, Descriptor rnode, Message msg) {...}
    public void receivePeers(Descriptor[] peers) {...}
}

Figure 6.5: T-Man exposed interface.

(which size is part of the parameters of peer sampling protocol used) and can start communicate with them, in order to form the list of its best neighbors.

- termination: the protocol continues to communicate with other peers, sending requests and getting replies, but when the cache remains unchanged for a fixed number of cycles (another parameter for the protocol, so to suit best to network conditions) its active thread is stopped: that way the protocol lies idle listening for eventual requests coming from other nodes. If the size of local cache changes while merging it with received one, it means that new nodes have been discovered, so the active thread is restarted and the protocol starts to communicate actively again, in order to detect and rank all the new peers. When no new peers appear for a certain number of cycles, all the nodes will eventually shutdown their active thread, leading to the complete termination of the protocol.

- tabu list: the latest peers the protocol communicated with are stored in a list, the tabu list, that is checked every time we need to furnish a new peer, in order to avoid to send requests again to a recently contacted node. The size of the list is a parameter of the protocol and is fetched from the configuration. This list helps to increase difference in peers selection, and reaching more nodes in the network.

- peer selection: is done by ranking the peers currently in local cache, then selecting one of the first $\psi$ nodes, and checking that it is not in the tabu list. Once the peer is selected it is inserted in that list in order to avoid to select it again in a few number of cycles.

- message preparation: the messages prepared for request or reply to other peers are built in the following way: sender and receiver of the message are set, then,
using the ranker fetched from the parameters, the protocol ranks the cache according to that node, selecting only the first $m$ peers ($m$ is the message window size parameter) and fills the message cache with them.

- **message reception**: when receiving a message, a reply message built as described above is prepared and sent, then *merge* method of the obtained ranker is called, passing in the local and the received caches. The result is the updated cache containing also the new nodes found in received cache. On the next peer selection turn, this nodes will participate and eventually be selected to start a communication.

### 6.10 Rankers

The ranker is a fundamental part of T-man protocol as it implements the ranking function, classifying nodes according to their properties with respect to the base node. Every ranker offers the same services, so it must implement an interface describing them: its name is *Ranker* and consists of two methods (shown in figure 6.6):

```java
public interface Ranker {
    public TManDescriptor[] rank(TManDescriptor node,
                                  TManDescriptor[] cache);
    public TManDescriptor[] merge(TManDescriptor localNode,
                                  TManDescriptor[] cache,
                                  TManDescriptor[] received);
}
```

**Figure 6.6:** Ranker interface code.

- **rank**: taking the cache to rank and the base node as parameters, it returns the cache ranked according to the particular rules of current ranker and with respect to the node passed as the base one. So for example, one could rank its cache passing the peer selected for communication as the base node, and then send the so ranked cache to the peer.

- **merge**: this method will merge, according to current ranker rules, passed local and received cache, ordering nodes by their ranking identifiers and avoiding to insert duplicates.

In order to provide a wide basis for testing and experimenting with T-Man, three different rankers were implemented: *SortedListRanker*, *SortedRingRanker* and *TreeRanker*. Here is a description of how the function *rank* behaves for each one of them:

- **SortedListRanker**: builds a linked list of nodes, ordered according to node’s ranking id. The cache of a generic node participating in the list contains
the preferred neighbors (the ones whose ranking id is the closest to node’s one) taken as follows: at first the closest peer before and after the node, than the second peer after and before the node, and so on. This ranker builds a topology with a big linear extension.

- **SortedRingRanker**: builds a linked list of nodes as the SortedListRanker does, but here the peers at start and end of the list are linked to each other (so in both caches were will be the ranking id of the other peer), in order to form a closed ring. The topology built in this way has a large diameter.

- **TreeRanker**: builds a binary tree, were the lowest ranking id is the root and the highest ones are the leaves. The best neighbors for a node obviously are calculated as its sons and its father, giving rise to a topology with quite small extension, opposite to the sorted ring ranker.

In all the three rankers merging is done by simply comparing ranking id’s to avoid inserting duplicates in cache: topology characteristics do not affect merging of two caches, and a successive ranking of the cache will return the correctly ordered set of neighbors for current peer.
Chapter 7

Testing and Performances

7.1 Introduction

Once protocol’s implementation is finished, it is needed to thoroughly test and prove the code correctness in a wide range of real situations. First of all, the software has to be tested with basic experiments in a friendly environment in order to discover and eliminate major flaws, then more refined experiments have to be conducted under conditions closely approaching to operative ones, were more subtle errors could be found, also allowing for investigation on implementation’s reliability and robustness (like, for example, being sure the protocol does not crashes on known potential problems). Data has to be collected while executing these experiments, so logging is needed at various degrees of completeness. These informations are analyzed after each experiment and the still remaining bugs are found and corrected. Then the tests are repeated and output data is investigated again. Once testing is completed, a series of experiments has to be run to retrieve data about speed, efficiency and reliability of the protocol. A single experiment is characterized by a particular set of parameters representing some of the possible real use configurations (for example, one could investigate linear topology with large window size, while another may test other kinds of topologies and different communication parameters).

In this chapter we will first explain in detail how testing was done, in friendly LAN environments and then in PlanetLab. Afterwards, system performance data will be illustrated by the mean of the experiments conducted for this purpose.

7.2 LAN Testing

The first environment in which the protocol was tested is a LAN. The reasons of this choice are obvious: in a LAN you eliminate almost at all problems due to traffic congestion, intermediate nodes crashes and similar, allowing to focus on discovering
bugs related to, for example, the execution cycle or the ranker functions.

Testing was conducted by selecting a number of computers, and then launching one or more instances of Cloudman on each of them as follows:

1. An SSH connection is established between the known host computer and the each of the participants.

2. The JAR file containing the protocol implementation and the configuration file needed to run T-Man and Newscast on the node are shipped to each computer involved via SCP.

3. One instance of Cloudman is launched on each participant computer by the means of an SSH command.

Then launched instances were let run for a period of time (some minutes), then terminated and the resulting log files were fetched on the known host computer.

### 7.3 Configuration

We now see and analyze an example of the configuration file used to launch Cloudman on a node with the required protocols. The same file is shipped to each node starting the framework. The only difference appears on the fixedHost, where the relative parameters are left empty (as this host does not start active communication but waits for other nodes to contact it):

```
1: protocol.netTransport TCPTransport
2: {
3: port 14000
4: }
5:
6: protocol.peerSampling Newscast
7: {
8: port 14000
9: transport netTransport
10: descriptorProvider tMan
11: cacheSize 10
12: delta 3000
13: fixedHost austud01
14: fixedHostPort 16000
15: }
16:
17: protocol.tMan TMan
18: {
```
In this file we can individuate four protocols. Only the protocols found in configuration file will be started and run by the framework:

- **netTransport**: in lines 1-4 the protocol covering the role of low level message sender is configured, the class `TCPTransport` is assigned to this role (meaning that the framework will instantiate it when `netTransport` is needed) with the network port it uses.

- **peerSampling**: in lines 6-15 we find the protocol covering the role of peer sampling service. The class of `Newscast` protocol is assigned to that role indicating that we will use this protocol as a peer sampler. In its parameters we can see that it uses `netTransport` defined before as transport protocol, `tMan` that will be defined below as descriptor provider for the node and two parameters defining the `fixedHost`, that is the host that is considered always on and that will be contacted first by every node starting `Newscast` in order to start building its cache.

- **tMan**: in lines 17-25 we can see the `T-Man` configuration section. The role of this protocol is covered by the class object of our implementation and all the parameters described in chapter 3 are can be found there. We may remark that for this configuration the `SortedRing` ranker class was chosen.

- **epidemicManager**: configured in lines 27-32, it is not a real protocol. It is responsible for the life-cycle of the protocols it manages, forwarding messages and invoking their methods when it is needed to accomplish a task. Its configuration includes the transport protocol it uses and the list of the protocols it manages (parameter `protocol`).
7.4 Logging

A logging facility was needed in order to understand what was going on while executing the protocols and to help detect bugs: in each important section of code a string describing current activity was printed on the standard output stream or on the standard error stream (depending what was happening). The two streams were redirected to two distinct files when launching Cloudman: `cloudman_out.log` for output stream and `cloudman_err.log` for error stream. Once Cloudman was terminated the files were fetched and analyzed along with the ones from all the other participating computers.

We now see and analyze a log file obtained when testing the protocols included in the configuration file described in the previous section. Messages from each protocol are prefixed with protocol name, in order to help distinguish and individuate them quickly:

1: NEWSCAST: Timeout Fired, sending cache to austud32.
2: TRANSPORT: sending message from austud20 to austud32.
3: EPIDEMIC MANAGER: Active Thread for protocol 2.
4: TRANSPORT: Receiving message from austud32 to austud20.
5: NEWSCAST: Merging my cache (degree 1) with received (degree 3).
6: TMAN: received peers from Peer Sampling Service.
7: NEWSCAST: Timeout Fired, sending cache to austud32.
8: TRANSPORT: sending message from austud20 to austud32.
9: SORTEDRINGRANKER: destnode is -8546419059292728165
10: SORTEDRINGRANKER: ranked cache:
11: -8518326836019163397
12: -8409262892178403394
13: EPIDEMIC MANAGER: Active Thread for protocol 2.
14: TMAN: Selected Peer -8409262892178403394
15: TMAN: Prepare Request.

In this chunk of the log file we can see a typical example of execution flow for all the involved protocols. The TRANSPORT lines are messages written by the transport protocol, informing that a message was sent or received, including also sender and/or receiver. The NEWSCAST lines represent peer sampling service output. In particular we may remark the merge action of a received cache with the local one and the activation of the protocol following a timeout expiration. The EPIDEMIC MANAGER lines just point out what kind of actions the manager ask a particular protocol(indicated by its pid) to perform. The TMAN lines represent core T-Man informations. We can see typical actions of the protocol, like selecting the next node for communication and preparing request for that peer. Note also the line 6, where T-Man informs that it has been provided the initial set of peers by the sampling service. From there on it starts to communicate and to build its view, as it was
previously waiting for the peer sampling service to provide it some peers to talk with. Finally `SORTEDRINGRANKER` are messages from the used ranker, telling that it is performing its actions, like ranking a cache to be sent (lines 10-12). The resulting cache is also written on the file to allow a check on the correctness of ranker’s algorithm.

3: TMAN: Merge.  
4: SORTEDRINGRANKER: merged cache:  
5: -8518326836019163397  
6: -8467099834827118947  
7: -8409262892178403394  
8: TMAN: Stopping Active Thread.

This chunk depicts a situation where the protocol is reaching its termination. In particular, line 8 informs us that T-Man has reached convergence and that it is going to stop active search for new peers. The `SORTEDRINGRANKER` lines indicate that the ranker is merging the local cache with the received one, finally writing down the resulting cache. T-Man then detect that cache has not changed for a number of cycles and so stops its active thread.

Error detection was performed by analyzing these logs, searching for incongruent situations like corrupted messages, cache being smaller after a merge, bad ranking of cache according to destination peer, node sending messages always to the same peer and so on. These kind of error search was performed with little network sizes in order to keep complexity and length of log files under control. Once this done, controls were done with bigger network size, by checking congruence of cache sizes and first peers in ranking.

### 7.5 PlanetLab

*PlanetLab*([www.planet-lab.org](http://www.planet-lab.org)) is an environment for distributed application execution and testing. Each university willing to participate in the program provides one or more machines connected to the internet were experiments can be run(actually a total of around 800 machines are available to users). Each user owning an account on *PlanetLab* can create one or more slice: each of them corresponds to a virtual machine on each of the computers forming the network. The environment provided is a basic *Linux* distribution. This virtual machines act as ordinary computers, so each user may install the needed applications or libraries, run experiments and so on. Access to nodes in the slice is done via SSH connections. As all the computers of the slice are connected only through internet, *PlanetLab* provides a large-scale testing en-
vironment with a real and unfriendly network (with lost packets, network failures, node crashes, ecc.), allowing to test distributed application in conditions closely related to everyday use.

A lot of open source tools have been created by users in order to ease the task of preparing the nodes and running the experiments: among all these tools PlMan was chosen. It allows to execute the same commands on a subset of all available nodes, seeing the corresponding output and also offering functionalities to upload and download files. In order to run the experiments two steps are needed:

- **JAVA install**: The first thing needed to run Cloudman on the nodes is to install the Java Runtime Environment. The rpm package was downloaded independently by each node, issuing a wget command with the url of the package as a parameter. Then the installation was ran on each node, eventually checking the result by trying to invoke the virtual machine

- **execution**: After having uploaded the Cloudman JAR and the configuration file on each node, the experiments were run via PlMan by invoking the java virtual machine on the class files (the precise command is java -jar Cloudman.jar Configuration.cfg). After a fixed interval of time the produced log files were fetched back and analyzed.

Even if Cloudman was run each time for some minutes only, the PlanetLab environment can also handle services running for a long time, allowing to test the whole framework once it is completed.

### 7.6 Performances

Once testing is completed, having proved the implementation of the protocol is reliable, we are interested in analyzing its performances under different sets of parameters, in order to check it behaves as expected. We so run a number of experiments, collect the resulting data and build some graphs illustrating the obtained results.

We are above all interested in evaluating the speed of the protocol: the performance measure we use to do so is convergence time, that is the time needed to obtain the desired target topology (the best set of neighbors at each node). The experiments are run with both Sorted Ring and Tree topology, then average values are taken to build the graphs. The unit of time will be seconds, easier to measure as convergence time is determined by doing the difference between convergence and starting time of TMan at each node. Another possible unit of measure could be cycles, but we will not use it in these experiments.

Experiments are run, as testing, in the PlanetLab environment and configuration files are shipped to participating nodes on each experiment varying the needed parameters. We run the protocol on a subset of available machines, launching more instances per node if needed to achieve an high number of participating nodes.
First of all we measure convergence time by varying the network size: if the protocol behaves correctly we expect to obtain a curve quite similar to the logarithmic one, as when the number of nodes doubles the average convergence time increases less than twice, in a non-linear fashion. If that was not the case, the protocol would suffer of important performance degradations when the number of participating hosts grows, thus making it unusable and crash prone in these situations. The network size ranges between 4 and 1024 with successive step consisting in doubling the previous value (so we have values of convergence time at 4, 8, 16, 32, 64 nodes and so on).

![Figure 7.1: Convergence time curve for T-Man protocol. By varying number of participating hosts the result is a logarithmic-like curve.](image)

As we can see the assumption we made about convergence times are true, with the protocol behaving, as expected, in a logarithmic fashion. This is an important fact about T-Man performances, as it means that the protocol is scalable, allowing for big quantities of nodes to participate in topology construction without crashing or leading to high convergence times.
Afterwards, we proceed to analyze the impact of tabu list introduction on T-Man performances. Tabu list contains the most recently contacted peers: the protocol will avoid to select one of these for communication, thus increasing the difference of selected peers. Because of that, we expect to see a decrease in convergence time, above all with network containing hundreds or even thousands of nodes (where contacting always the same small set of peers may even mean that convergence is never reached).

![T-Man and Tabu List](image)

Figure 7.2: T-Man convergence time curves for basic and tabu list version showing increase in performance for the latter.

As showed by the graphic the version with tabu list obtains an important increase of performances, especially with big number of hosts. With little network sizes the difference between the two versions of the protocol is near to zero, as in this situations it is highly probable that each node knows a big part of the hosts, so it can reach convergence even by always contacting the same small fraction of them. By contrast we can see that the difference between the two curves keeps increasing while augmenting hosts number, thus confirming our theoretical expectations and the effectiveness of tabu list in speeding up the protocol.
The last measure we do is again the convergence time of \textit{T-Man}, but this time we vary another important parameter of the protocol: \textit{m}. The latter is the \textit{message window size} that is the number of nodes descriptor sent in one message. By reducing this parameter too much we may degrade the convergence time in a dramatic way, as the number of messages, and therefore of cycles, needed to know the part of the network representing the best set of neighbor for a node, would increase in an exponential way. We so expect to obtain a curve where convergence time decreases as \textit{m} increases, eventually reaching a minimum and then increasing again for the effect of the growing probability of view changes when \textit{m} grows too.

![T-Man varying parameter m](image)

Figure 7.3: \textit{T-Man} convergence time curve when varying \textit{m} parameter.

The resulting curve confirm our assumptions: with little values of \textit{message window size} convergence time is high; when \textit{m} increases the convergence time rapidly decreases until a point where it starts to increase again. The value of \textit{m} is not strictly correlated to network size, as if it increases in parallel with the number of hosts, the message size quickly become huge, leading to high communication costs and, for the reason mentioned above, convergence time increase.
Chapter 8

Conclusion

Cloudman is a framework for executing distributed application in large and decentralized networks. It is based on the cloud concept, that is all the nodes and resources composing the network, and on the subcloud that is a subset of the cloud’s nodes that perform a task. T-Man is one of the core protocols of the framework: its role is to build an overlay topology linking together all the nodes participating in the cloud, allowing them to cooperate in applications execution. The part of Cloudman realized for this thesis covers bootstrapping including in particular overlay network construction and peer sampling service.

We have implemented T-Man inside the basic facilities of the framework, providing also basic utilities and services needed. We chose and wrote the peer sampling service in the form of Newscast protocol to provide starting facility for the protocol, and used a simple policy of cache monitoring to detect T-Man convergence to the best set of neighbours. During this task we also implemented some helper classes needed to provide timeout services to the protocols.

We then proceeded to thoroughly test all the features of the protocol, first in friendly medium size local networks and then in PlanetLab environment (consisting of hundreds of machines located all around the globe and connected via internet), covering different sets of parameters and network size in order to detect implementation errors and refining the details of the protocol.

Afterwards we ran a set of experiments varying network size and the most important parameters. With the data collected we produced some graphics in order to show the most important qualities of T-Man and to confirm major assumptions on its behaviour and performances (like, for example, convergence time curve properties).

What resulted from the experiment’s data confirmed our expectations. First of all we obtained a logarithmic-like curve for convergence time of T-Man. In other words the time needed for the protocol to reach convergence does not double its value when doubling network size, in fact it increases less and less while augmenting
the number of participating nodes. This is an important fact, as it means that the protocol is scalable and will therefore behave well in big size networks consisting of hundreds or even thousands of nodes.

Then we analyzed the usefulness of the tabu list introduction in T-Man. The curve of the basic version reveals significantly higher convergence time than the tabu list version; the major differences occur with big network sizes, also underlining that the difference keeps growing while augmenting the number of participating nodes. This confirm that tabu list brings an effective improvement over T-Man basic version.

The last assumption was on the curve of convergence time obtained while varying the number of nodes descriptors sent in a message, called $m$. We expected an important shortening of convergence time in a first phase and in a second phase, from a certain value of $m$ on, a progressive augmentation of that time (due to the incidence of message size an to increasing probability of cache changes when sending big quantities of nodes descriptors in each message). We really noticed that behaviour, leading to the conclusion that the parameter is effective in influencing convergence time for T-Man.

Future work should focus on protocol refining and tuning, as the rest of the framework is completed and is so testable in a complete environment. In particular, completing the whole bootstrapping service and integrating T-Man in it by exposing primitives needed to use the built overlay. Further developments may also consist in implementing different rankers using more complex criteria for peer ranking (like, for example, available resources on a node) rather than simple id distance.
Bibliography


# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Architecture of Cloudman framework</td>
<td>9</td>
</tr>
<tr>
<td>6.1</td>
<td>Protocol interface code</td>
<td>21</td>
</tr>
<tr>
<td>6.2</td>
<td>EpidemicProtocol interface code</td>
<td>22</td>
</tr>
<tr>
<td>6.3</td>
<td>PeerSampler interface code</td>
<td>24</td>
</tr>
<tr>
<td>6.4</td>
<td>Newscast exposed interface</td>
<td>24</td>
</tr>
<tr>
<td>6.5</td>
<td>T-Man exposed interface</td>
<td>26</td>
</tr>
<tr>
<td>6.6</td>
<td>Ranker interface code</td>
<td>27</td>
</tr>
<tr>
<td>7.1</td>
<td>Convergence time curve for T-Man protocol. By varying number of participating hosts the result is a logarithmic-like curve</td>
<td>35</td>
</tr>
<tr>
<td>7.2</td>
<td>T-Man convergence time curves for basic and tabu list version showing increase in performance for the latter</td>
<td>36</td>
</tr>
<tr>
<td>7.3</td>
<td>T-Man convergence time curve when varying m parameter</td>
<td>37</td>
</tr>
</tbody>
</table>