Valid Query Answers exploiting Node Virtualization
A robust architecture for Interval Valid queries in dynamic distributed systems, formal proofs, experimental evaluation and cloud application
AUTHOR'S ADDRESS:
Adriano Cerocchi
Dipartimento di Ingegneria Informatica Automatica e Gestionale
“A. Ruberti”
Sapienza Università di Roma
Via Ariosto 25, 00185 Roma, Italy
e-mail: cerocchi@dis.uniroma1.it
www: http://www.dis.uniroma1.it/~cerocchi
TBA...
Abstract

This thesis studies the problem of answering system-wide queries, satisfying the interval validity semantics, in a distributed system prone to continuous arrival and departure of participants. The interval validity semantic states that the query answer must be calculated considering contributions of at least all processes that remain in the distributed system for the whole query duration. It is impossible to satisfy this semantics in systems experiencing unbounded churn due to the lack of connectivity and path stability between processes. This thesis presents a novel architecture, named Virtual Tree, for building and maintaining a structured overlay network with guaranteed connectivity and path stability in settings characterized by bounded churn rate. The architecture includes a simple query answering algorithm that provides interval valid answers. The overlay network generated by the Virtual Tree architecture is a tree-shaped topology with virtual nodes constituted by clusters of processes and virtual links constituted by multiple communication links connecting processes located in adjacent virtual nodes. A bound has been formally proved on the churn rate for interval valid queries and it has been carried out an extensive experimental evaluation that shows the degree of robustness of the overlay network generated by the virtual tree architecture under different churn rates. The thesis also provide an application scenario of the proposed architecture in the field of the volunteer computing. The application scenario shows that Virtual Tree represents a fundamental building block for the architecture of a volunteer computing platform and its capability of offering valid queries enhances the performance of the system providing a better accuracy in the task-scheduling procedure.
Contents

1 Introduction 1
   1.1 What a dynamic distributed system is .......................... 3
   1.2 Querying issues in dynamic distributed systems ............... 5
   1.3 Contribution and organization of the thesis .................. 7

2 Background on dynamic distributed systems 11
   2.1 Formal definition of a dynamic distributed system ............ 12
   2.2 Formal definition of churn .................................... 14
       2.2.1 Churn characterization ................................. 14

3 Overlay networks 17
   3.1 State of the art ............................................. 20
       3.1.1 Unstructured topologies ............................... 20
       3.1.2 Structured topologies .................................. 22
       3.1.3 Hybrid topologies ...................................... 25

4 Building an overlay network by using Attractors and Virtual
   Nodes 29
   4.1 Can physical node based overlay obtain valid queries? ...... 30
   4.2 What are the attractors and the virtual node ................. 32
       4.2.1 Related works ......................................... 33
   4.3 How virtual nodes can “drive” the dynamism .................. 35
   4.4 Virtual Tree .................................................. 35
       4.4.1 The Virtual Tree Graph ............................... 37
       4.4.2 Overlay management protocol ........................... 39

5 Obtaining a valid query by using Virtual Tree 41
   5.1 How to obtain a valid query ................................... 42
   5.2 A querying algorithm for the Virtual Tree ..................... 43
   5.3 Deterministic warranties about validity ....................... 45
       5.3.1 Algorithm correctness ................................. 45
Chapter 1

Introduction

“In recent years, researchers have investigated best-effort algorithms to efficiently process aggregate queries [...] unfortunately, query semantics for best-effort algorithms are ill-defined, making it hard to reason about guarantees associated with the result returned”


In the last years we observed a growing attention to the world of distributed computing due to the introduction of the concept of cloud computing and to the continuous enhancement of the performance exhibited by the single machines. Distributed computing is a continuous evolving science composed by a large variety of open research fields: cloud/grid computing, sensors network, peer to peer, consensus, query answering, transactional memory, and others. Historically query answering is one of the first issue that attracted the interest of the researchers due to its strong relationship with the database. Query answering is a problem that tries to retrieve the information related to a given question. In informatics query answering was born with monolithic databases and evolved and still evolves toward distributed application.

Today’s query answer in the field of relational database represents a solid building block for ICT, but it still remains an interesting research field due to the evolution of the NoSQL databases and the key-value storages in the field of cloud computing and distributed storage such as Apache Cassandra [66],
Dynamo [35] or MongoDB [76]. Relational databases and NoSQL databases are all designed to run in a managed or semi-managed environment; in these settings the dynamism is predictable and it is possible to ensure some validity properties for the results. Relational databases (e.g. Oracle [33] or MySQL [32]) ensure the famous ACID properties (atomicity, consistency, isolation, durability), the NoSQL databases, such as Cassandra, ensure the eventual consistency. When the dynamism is not predictable, as in peer to peer systems, the query answer protocols are so far from respecting any validity property. The validity of the results, obtained by a query issued in a large scale dynamic environment, is assumed to be best effort. In order to understand the problem we suggest the following sketch:

![Figure 1.1: An 8 segment, 360 degree panoramic image of Trafalgar Square](image)

Figure 1.1: An 8 segment, 360 degree panoramic image of Trafalgar Square

Imagine you are in Trafalgar Square in London, the environment should appear like in figure 1.1, at a given time $t$, a man asks you: “Hi guy! I can give you one million dollars if you are able to count all of the people in the square without leaving it. Are you able to do that?” Imagine you reply “ok let’s start!”, and the man replies “ok, I will wait for you here”. At time $t$ you remain in the square and you start your counting. Let’s assume that you are able to count the people with an algorithm. At a given time $t'$ you obtain the result and you go back to the man telling your result. The man replies “ok guy, good job, but in the interval $< t, t' >$ I see a lot of people entering and leaving the square; which kind of semantic has your result got? Is it related to a particular instant or to something else?”. You cannot reply to this question because you could not stop the people from entering and leaving the square; they were not under your control and the square was too large to be to be totally watched. Then you reply “I did my best to count the people but I cannot relate the result to a particular instant or to a particular semantic”. The man replies
This nice sketch perfectly snapshots the problem of the dynamic system in the case of the query. More generally dynamic systems represent a challenge scenario for the distributed applications; the models considering continuous and uncontrolled arrivals and departures of the participants represent an interesting perspective to evaluate novel solutions that are currently developed in a controlled environment.

Numerous protocols have been designed for dynamic distributed systems, few papers (such as [3] and [14]) strive to present models suited to such systems, and extremely few dynamic protocols have been proved correct. Up to now the most common approach used to address dynamic systems is mainly experimental, a precise model is needed to find the limits that make implementations work correctly despite arrivals and departures of nodes. Hence, providing theoretical bounds on the maximum churn rate tolerated by applications is currently an issue.

The design and the implementation of distributed computing platform have to deal not only with asynchrony and failures, but also with the system dynamicity. In distributed systems affected by failures, there are very precise bounds on fault tolerance (e.g. [70]). On the contrary, bounds on the arrivals and departures of processes do not exist considering the same abstractions in a dynamic system with continuous churn. In particular, given an implementable abstraction in a static distributed system under some timing assumption, it is not possible to say if it is still implementable (under the same timing assumptions) in a dynamic distributed system. For example there are no solutions for the consensus operation in dynamic environment; the consensus is evaluated among a well-known set of elements. What happens if the set changes during the time? Can we still talk about consensus?

In the following sections we deal with the dynamic distributed systems, then we explain in details which kind of problems are related to a querying system that run in a dynamic environment, and we conclude with the contributions proposed by the thesis. The formal definitions about the distributed system itself and the dynamism will be treated in chapter 2.

1.1 What a dynamic distributed system is

Due to the evolution of technologies and communication paradigms, modern distributed systems are evolving towards novel computational models. The last few years have been characterized by an always increasing growth of computational capabilities of devices, bandwidth availability and diffusion
of wireless communications, that let distributed systems move from classical models to new classes of systems and applications. Vehicular Area NETworks (VANETs), peer-to-peer networks (P2P), Wireless Sensor Networks (WSNs), Smart Environments, Cloud Computing are just few examples of emerging systems and applications. These systems have, in common, the impossibility to be modelled as it has done in common distributed systems. It is due to the uncertainty on the set of processes participating to the application computation. As a consequence, all the problems that have been proved correct under a well defined set of assumptions are newly under discussion when instantiated in these new settings.

As an example, let consider the Skype [87] context; Skype provides a messaging and VoIP service, built on top of a peer-to-peer overlay network where users cooperate by sharing their resources (CPU, bandwidth, memory space) in order to implement the service [19]. Applications developed in such context are characterized by the complete autonomy of processes; they can decide autonomously and independently when entering the computation and when leaving it. Therefore it becomes mandatory to define procedures able to manage unexpected and large changes, due to the arrival and departure of processes (a phenomenon also called churn). In order to guarantee applications availability, such procedures have to be implemented in a distributed fashion; on the contrary, the responsible for the application management can become quickly overloaded. In other words, the system has to be self-managed to quickly react to changes and to provide availability despite continuous changes.

The example of Skype perfectly answers the question What is a dynamic distributed system? in fact it introduces the concept of continuous arrivals and departures of processes, i.e. the churn phenomenon, and it remarks the need of distribution of the operation among the processes composing the system. A distributed system can be defined as an application in which two or more entities contribute to the computation of the final result; a dynamic system can be defined as an environment in which the entities composing it are autonomous with respect to the system; thus they can join and leave independently from the computation, consequently a dynamic distributed system is an application composed by several autonomous entities in which the computation of the final result is independent from the continuous arrivals, departures and failures of them.

Many protocols that deal with dynamic distributed systems assume that the churn is quiescent [58], [75], [92], i.e. the arrivals and departures of processes eventually end and the system converges to its target. However, such assumption could be not completely realistic for some scenarios, like Skype, where processes continuously change, although the different arrival and departure rates [51]. In general the larger is the number of entities composing the system and the less realistic the assumption is. In this elaborate we always
consider a “continuous” dynamism model: processes never stop to join and leave.

The formal definitions of a dynamic distributed system and the churn are left to the next chapter; the chapter also shows a characterization of the churn phenomenon with respect to the real systems.

1.2 Querying issues in dynamic distributed systems

As already said today’s large scale distributed systems are characterized by strong dynamics caused by the inherent unreliability of their constituting elements (due to churn). This continuous dynamism has a strong negative impact on distributed algorithms designed to work on such systems.

In-network query algorithms [22] run on top of such systems (e.g. in large scale sensor networks) to collect query answers in a scalable manner. A correct answer for a specific query can only be found by extensively checking data stored on every entity of the system and by collecting the results [43] [54] [45]. However, this procedure inevitably clashes with the high level of dynamism characterizing the distributed dynamic network. Strong network dynamism can severely affect the correct functioning of such protocols. As an example, consider the representation of a sensor network reported in Figure 1.2; assume that at time $t_1$ sensor $a$ issues a query in the network (left box) to collect some aggregate data (e.g. the sum of all sensed values); the query will be routed following a simple broadcast/convergecast scheme throughout the network (center box) and the results aggregated on their way back to the source $a$ (right box) where it will complete at time $t_2$. What happens to the query result if a sensor fails during the time interval $\{t_1, t_2\}$ required to calculate it? If node $b$ fails this does not constitute a problem as only its contribution to the result will be missed. Conversely, if node $c$ fails an entire branch of the query broadcast tree will be pruned and the query result will thus be severely affected by missing contributions of the nodes that are still part of the network. In the end, $a$ will receive some results, but it will be unable to assign a precise semantics to them.

![Figure 1.2: aggregate query in a sensors network](image-url)
The problem of defining a precise semantics for in-network query answering in large-scale dynamic networks was introduced by Bawa et al. in their seminal work [20]. The paper presents a set of validity properties whose aim is to formally define which entities are expected to contribute to the calculation of a query in a system characterized by churn: snapshot, interval and single-site validity.

The snapshot validity property captures the intuitive definition of query answering: all the entities and only those that constitute the system at the time when the query is issued, must participate to its calculation. The original definition is actually slightly relaxed allowing the participation of any set of nodes that constituted the system during the execution of the query.

The execution of a query in a distributed system where processing times and network latencies are not negligible, takes a certain amount of time, during which the system can change its composition due to churn; if the amount of churn is not limited it is impossible to solve the query with the snapshot validity semantic, because no algorithm is able to calculate a perfect snapshot of the system population at a specific time instant [20]. We can thus relax the validity property by allowing an answer to be calculated from a set of processes constituted by, at least, all nodes that remain in the system during the whole query calculation time, and at most by all the nodes that appear in the system, at least for a while, during the same query calculation. This definition got the name of interval validity. It is easy to see that even this relaxed validity property cannot be enforced if churn is not limited [20]. Consider, as an example, entity B involved in the query calculation previously represented in Figure 1.2; this process is connected to the querying node (at the root of the tree: the node A) through a network path passing through process C; the query is expected to reach B through C. Suppose that C unexpectedly leaves the system; B is still connected to the systems, but there is no way it will ever reach A; in this case B is part of the system during the whole query execution time, but it cannot contribute because C has left. This violates the interval validity. The only property that can be enforced with such churn model is the single site validity property: those entities that remain connected to the root by a path that does not change (i.e. a stable path) for the whole duration of the query are expected to participate; the stable path is a network path composed by entities that do not leave during the computation.

In [14] Baldoni et al. introduce the notion of dynamic validity that extends the concept of single-site validity. It forces the final result to keep in charge the contributions of the entities that, during the whole query time, remain connected to the root through some paths. Basically, with reference to the scenario presented in figure 1.2, dynamic validity states that if - for example - an edge directly connecting B to the root A appears during a query before the leave of C, then the contribution of B must be considered in the final result,
1.3. CONTRIBUTION AND ORGANIZATION OF THE THESIS

due to the fact that B is always connected to the root with some paths.

Bawa et al. [20] propose a work without any kind of churn bound, in realistic scenarios [51] [83], as remarked by the spikes analysis proposed in 2.2.1; the churn trend is commonly known, then in some way it is predictable i.e. it is possible to define bounds.

1.3 Contribution and organization of the thesis

In the previous paragraph we propose an overview about the issues that can be encountered during the implementation of a querying algorithm in a dynamic environment. The main obstacle is clearly represented by the network uncertainty introduced by the dynamism. A system-wide query clashes with two main factors: the connectivity of the network and the capability of routing the query in the network. The first one is easy to figure out: if the network is not connected, (i.e. it is partitioned in some sub-graphs) it is not possible to reach all the nodes with the query; the second factor is instrumental to the goal: the connectivity of the network represents a necessary but not sufficient condition. If the network is connected and a node A is able to broadcast the query in the network, all of the nodes, receiving the query, reply with a new broadcast message containing their contributions; A eventually receives all of the contributions but it is not able to state when this operation will be concluded. Moreover the broadcast does not represent an efficient way to distribute the query: in a large network the naive broadcast algorithm cannot be applied due to scalability. The example given shows that the query routing capabilities represent a fundamental building block for the querying system.

In addition even assuming that the naive algorithm solves the querying problem in a connected network, nothing can be said about the validity of the result: it still remains a best-effort algorithm.

Scope of the thesis is to propose a solution to obtain interval valid queries in a large scale distributed dynamic system characterized by bounded churn. A complementary scope is also the application of the proposed solution to a large scale volunteer computing system. In the presented scenario the contribution of the thesis can be resumed in the following points:

1. a novel managing procedure called attractor applied to the virtual node in dynamic systems: in dynamic environment the churn phenomenon represents the main issue for implementing complex overlay architecture supporting the querying algorithm. A virtual node is composed by a small clique of nodes with a shared state and a management algorithm named attractor procedure or more easily attractor. The churn phenomenon acts on the physical nodes; if the virtual node observes a
large number of disconnections, it triggers the attractor procedure that moves the nodes among the cliques in order to preserve the survival of the virtual one. The novelty of the approach is represented by the attractor procedure. In the past virtual nodes were used with the aim of standard replication, now the attractor represents an improvement of the state of the art in distributed dynamic system.

2. a tree-shaped overlay network implemented by using the abstraction of virtual nodes: virtual nodes allow to implement complex virtual topology exploiting their capability to resist to the churn, driving the erosion introduced by it toward the larger virtual nodes. Tree-shaped topology represents a challenge scenario for a dynamic environment because they can be disconnected by simply removing a non-leaf node. Usually tree-shaped architecture for dynamic environment works by fixing the disconnection introduced by the churn by re-arranging the network or by using replication. It is very hard not to encounter disconnection in a tree prone to churn; due to this reason these systems always deal with “repairing period” that temporarily denies the access to the tree. Exploiting the abstraction of virtual nodes Virtual Tree is proposed: an always available tree-shaped overlay able to resist to the churn erosion.

3. a query algorithm designed for the overlay network introduced above: having a tree-shaped topology it is easy implemented a scalable per-step querying algorithm able to run on top of the Virtual Tree.

4. a formal proof of the interval validity of the results obtained by the querying algorithm on the tree-shaped overlay network: the tree-shaped topology together with the querying algorithm presented represent a sufficient condition for implementing a valid query in dynamic network. The result is motivated by formal proofs and an extensive experimental evaluation that shows that the couple Virtual Tree - query algorithm ensures valid results also over the threshold shown in the proofs. An interesting result is that the connectivity of the Virtual Tree is a sufficient condition for obtaining a valid results.

5. a large scale volunteer computing platform in which is applied the presented querying system: a fast, reliable querying system has a very large application set. Virtual Tree represents just a solution in a large bibliography. However, thanks to its capability of preserving the connectivity (and then the querying capability) of the structure even in presence of strong dynamism, it represents an interesting solution for all the environment in which the other systems fail because their architec-
ture is based on physical node. A relevant scenario is represented by the large scale volunteer computing in which thousands of nodes continuously join and leave the system imposing continuous changes in the resource list. It is the contest of Cloud to Peer: a cloud/grid computing system based on volunteer participants built on top of Virtual Tree. Cloud to Peer exhibits good performance with respect of well known system such MapReduce [36].

The contributions listed above and the basic ideas from which this work started are partially contained in the following papers: [17] [15] [16] and take inspiration by the concepts introduced in [20] [37] [95]. In particular the first four contributions were partially published in [17] [15] [16] and were inspired by [20] [37]; the last contribution was inspired by [95].

The rest of this work is organized as follows: chapter 2 introduces the formalism used in the rest of the thesis required to correctly figure out the concept of dynamism with particular reference to real system; chapter 3 presents the concept of overlay network required for understanding the needs of the virtual node abstraction and the architecture proposed by Virtual Tree; chapter 4 explains in detail how virtual nodes work and how they can be used in the implementation of Virtual Tree; the chapter also deals with the related works in the field of virtual nodes and tree-shaped topology. Chapter 5 proposes the query algorithm providing the formal proof of the validity of the results also proposing the extensive experimental evaluation; chapter 6 proposes an application scenario for Virtual Tree in the field of cloud computing and chapter 7 concludes the work.
Chapter 2

Background on dynamic distributed systems

“it is hard to derive good requirements on churn-resilience”


This chapter provides some formal definitions about a distributed system and the churn. In the following will be detailed how a distributed system can be modelled and how the churn (i.e. the dynamism) can be modelled and characterized. In this work are usually used expressions like “high/low level of churn”, “high/low dynamism” scope of this chapter is to give to the reader the correct idea about what we mean with that. In order to do that we introduce the model for a dynamic distributed system with reference to the timing models used in this thesis and successively we propose a formal model for the churn characterizing it in relation with real dynamic systems.
2.1 Formal definition of a dynamic distributed system

The distributed system is composed by an infinite set of processes (also called nodes) \( V = \{p_1, p_2, \ldots, p_i, \ldots \} \), each having a unique identifier and a finite memory space for local computation. The time-lapse in the system is measured by a common global clock, represented by an infinite set of integer \( \{t_0, t_1, \ldots, t_i, \ldots \} \), not accessible by the processes. At each time unit, the number of processes composing the distributed system is finite. Processes of the distributed system, at each time unit, are connected with each other through an Overlay Network, i.e. a logical network built on top of the physical one.

At every time \( t \), each process \( p_i \) has a partial view on the set of processes currently part of the network i.e. it knows only a subset of the process identifiers in \( V \) and stores them in its local view called neighbourhood. Note that if a process A is in the local view of process B it does not imply that B is in the local view of A.

Processes can communicate by exchanging messages on top of perfect point-to-point channels (i.e. messages are not created nor duplicated and if both sender and receiver do not leave the system, each message is eventually delivered). Every process \( p_i \) can exchange messages only with its neighbors, i.e. with processes in its local view.

Considering the set of processes and their local views, at each time \( t \) the system can be represented as a graph \( G(t) = (V_t, E_t) \), where \( V_t \) is the set of processes part of the network at time \( t \) and \( E_t \) is the set of edges \( e_{i,k} \) connecting the processes in \( V_t \). The graph \( G(t) \) represents the topology of the overlay network interconnecting processes in \( V_t \).

Processes decide autonomously when joining the computation and when leaving the computation: in this sense the computation is dynamic. A process \( p_i \), belonging to the system, that wants to participate to the distributed computation has to execute the \texttt{JOIN()} operation. Such operation, invoked at some time \( t \), is not instantaneous: it consumes time, during that time a bootstrap service, provides \( p_i \) with a local view [59]; the bootstrap service can be seen as a server that is continuously informed about the current composition of the network, thus it can provide the initial local views. Note that from time \( t_0 \), the process \( p_i \) can receive and manage messages related to the computation sent by any other process that belongs to the system and that participates to the computation.

When a process \( p_k \), belonging to the distributed computation, does no more want to participate, it leaves the computation. In our model the leave operation is performed in an implicit way and \( p_k \) leaves the computation for-
ever and not longer sends messages. A leave can thus be considered as a crash failure [50] and in the rest of the elaborate we will refer to these two kinds of event as leaves. Without loss of generality, we assume that if a process leaves the system and later wishes to re-enter, it joins the system with a new identity.

### Timing Assumptions

Distributed systems usually fall in three timing assumption categories [50]: Synchronous Systems, Asynchronous Systems and Eventually Synchronous System.

**Synchronous Systems** can be characterized by the following properties:

- Synchronous computation: there is a known upper bound on processing delays
- Synchronous communication: there is a known upper bound on message transmission delays

The main fallback of this properties is the *point-to-point timely delivery*: in the case of no message loss, there exists an integer $\delta$, known by processes, such that if $p_i$ invokes “send m to $p_k$” at time $t$, then $p_k$ receives that message by time $t + \delta$ if it has not left the system by that time.

The **Asynchronous Systems** at the opposite is characterized by the absence of any timing assumption about processes and channels. Processes have no access to any bounds on processing and communication delays cannot be assumed, consequently the *point-to-point timely delivery* is modified as follows: let $p_i$ be a process invoking a “send m to $p_k$”. If both $p_i$ and $p_j$ do not leave the system, then $p_j$ eventually\(^1\) receives that message.

Generally, distributed systems appear to be synchronous. More precisely, for most systems, it is relatively easy to define physical time bounds that are respected most of the time. However, periods where the timing assumptions do not hold are possible. In order to deal with such an issue, in static systems, an **eventually synchronous system**\(^2\) is modelled as a system that after an unknown but finite time behaves synchronously [27] [38]. In that model the *point-to-point delivery* results as follows: there is a time $t$ and a bound $\delta'$ such that any message sent at time $t' \geq t$, is received by time $t' + \delta'$ to the processes that are in the system during the interval $[t', t' + \delta']$.

Another remarkable timing model was introduced by [20] and is named **relaxed asynchronous model**. Such model states that there exist known upper bounds on (i) process execution speeds, (ii) message transmission delays

---

\(^1\)the result can be achieved by a naive “re-transmit forever” algorithm [50]

\(^2\)sometimes also called partially synchronous system
and (iii) clock drift rates. The model, even if it is named asynchronous by the authors, is clearly equivalent to the synchronous model.

2.2 Formal definition of churn

The dynamicity due to the continuous join and leave of nodes in the computation is a phenomenon identified under the name churn. This section introduces a general model able to characterize such dynamicity. The model proposed here is mainly based on the definition of two functions: (i) the join function \( \lambda(t) \) that defines the join of new processes to the system with respect to time and (ii) the leave function \( \mu(t) \) that defines the leave of processes from the system with respect to time. Such functions are deterministic discrete functions of time:

**Definition 1.** (Join function) The join function \( \lambda(t) \) is a deterministic discrete time function that returns the number of processes that invoke the join operation at time \( t \), i.e. it defines the in-churn.

**Definition 2.** (Leave function) The leave function \( \mu(t) \) is a deterministic discrete time function that returns the number of processes that have left the system at time \( t \), i.e. it defines the out-churn.

The two functions are expressed as percentage of processes (rounded down) with respect to \( N_0 \) that represents the initial size of the system \( N_0 = |(V)| \). Given the in-churn and the out-churn functions, the number of processes that join and leave at each time unit is represented respectively by the numbers \( \lambda(t) \cdot N_0 \) and \( \mu(t) \cdot N_0 \). We assume that churn is continuous, i.e. it does not exist a time instant \( t \) after which churn ends [79] i.e. \( \lambda(t) = \mu(t) = 0 \).

2.2.1 Churn characterization

This brief sub section has the aim to give to the reader the idea about the influence of the churn on a system and which can be considered realistic levels of it. In [83] Saroiu et al. propose an analytical studies on the most common large scale distributed dynamic protocol for file sharing such as Gnutella [47] or KaZaA [62]. File sharing systems are characterized by a large amount of system-wide queries used to retrieve the owners of the desired files, so they are perfect to figure the churn characterization that we are looking for. In these
### 2.2. FORMAL DEFINITION OF CHURN

<table>
<thead>
<tr>
<th>First Author</th>
<th>System Observed</th>
<th>Session Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saroiu[83]</td>
<td>Gnutella, Napster</td>
<td>50% &lt;= 60 min</td>
</tr>
<tr>
<td>Chu[30]</td>
<td>Gnutella, Napster</td>
<td>31% &lt;= 10 min</td>
</tr>
<tr>
<td>Sen[84]</td>
<td>FastTrack</td>
<td>50% &lt;= 1 min</td>
</tr>
<tr>
<td>Bhagwan[21]</td>
<td>OverNet</td>
<td>50% &lt;= 60 min</td>
</tr>
<tr>
<td>Gummadi[52]</td>
<td>KaZaA</td>
<td>50% &lt;= 2.4 min</td>
</tr>
</tbody>
</table>

Table 2.1: Observed session times in various peer-to-peer systems. The median session time ranges from an hour to a minute.

Systems the main part of the users join the system just for the time required for searching the desired file and download it. This is a severe churn model, in fact other systems, like the sensor networks, are not characterized by this “opportunistic” behaviour, so we consider it for this characterization as a very representative churn scenario.

Table 2.2.1 summarizes some realistic session times: they go from one minute to an hour, usually a session time of 40 minutes is considered realistic. The discrete functions $\lambda$ and $\mu$ express the percentage of nodes that join or leave per time unit. A time unit is an abstract entity that does not correspond to any specific standard timing (such seconds, hours, etc...), however it is usually related to the update frequency of the local view of the processes (see 2.1) that is usually measured in seconds, thus a time unit can be assumed as 1 second. If a time unit is one second it follows that the whole node set can completely change every 40 minutes, i.e. each 2400 seconds, then each second the $0.04\%$ ($1s/2400s \simeq 0.04\%$) of the nodes change i.e. join/leave. The rational is useful to figure out realistic levels of churn, in the following we will consider as realistic level of churn a $\lambda$ $\mu$ rates lower than $0.1\%$.

Another important characterization is represented by the spikes in join/leave of the nodes. Since the beginning of the elaborate we state that the churn is continuous, the statement can be also detailed as follows: $\forall t > 0 \rightarrow \lambda(t) \neq 0 \land \mu(t) \neq 0$, usually $\lambda(t) \approx \mu(t)$ however some spikes are possible i.e. $\lambda(t) \gg \mu(t)$ or $\mu(t) \gg \lambda(t)$. The analysis of the real system traces helps to correctly understand the importance of them. Figure 2.1 plots the join and the leave rates of the nodes participating to the Gnutella network. In the graph the highest spike is represented by 6 joins on an average of about 1-2 nodes per time unit, considering the size of the Gnutella network (thousands of nodes) the observed spikes are practically negligible, so the churn can be
considered continuous and - moreover - constant.

Figure 2.1: Churn in the gnutella [61] traces. Full time span of 3600 one minute cycles and zoomed in to cycles 2250 to 2750.
Chapter 3

Overlay networks

“Overlay network is a network in which the nodes are formed by the processes and the links represent the possible communication channels”

A.S. Tanenbaum and M. Van Steen @ Book: Distributed System Principles and Paradigms (2006)

This chapter presents the concept of Overlay Network and why it represents a fundamental building block for any distributed dynamic system.

A distributed dynamic system, as introduced in 1.1, is composed by a set of nodes (also known as processes) connected to a physical network such as a LAN or the internet. The physical connection allows uniquely to address a node in the network; a logical connection between two nodes A and B is represented by the physical path connecting the two nodes, anyway, the routing of the message and the physical path connecting them is in charge of the physical network. In the following we will never mention the physical path and we will always assume that A and B are directly connected by the network.

An overlay network is a graph $\mathcal{G} = <\mathcal{V}, \mathcal{E}>$ composed by the nodes $\mathcal{V}$ and the edges $\mathcal{E}$ representing the logical connection among the nodes. Each node $n$ in $\mathcal{V}$ (see 2.1) is equipped by a local view that is a subset of nodes $\in \mathcal{V}$. The local view represents the logical connections of $n$. Note that the edges defined by the local views are directed, then the abstraction of the local view does not ensure that if node B is in the local view of A then also A is in the local view of B. The responsibility of the bi-directional edges is in charge of the algorithm that rules the overlay network (for example Virtual Tree,
presented in chapter 4 ensures this property). Due to the local views a node A is said to be connected to B if B is in the local view of A or if exists a logical path connecting A to B. For example in figure 3.1 nodes A and C are directly connected by the local view of A, but also A and B are connected through the path A-C-B.

An important property of an overlay network is the connectivity. The connectivity property states that a network is connected if and only if \( \forall n_1, n_2 \in \mathcal{V} \) exists at least one path connecting the nodes. The connectivity has to be evaluated on the directed graph, for example in figure 3.1 the overlay network 1 is not connected, in fact there is no path from B to A. At the opposite the overlay network 2 respects the connectivity property. Sometimes [92] the connectivity is evaluated on the “related undirected graph”, this can be an interesting investigation but it does not really represent the connectivity of the network (for further information please refer to 5.5.4).

![Overlay networks](image)

Figure 3.1: two examples of overlay networks

As introduced in chapter 2 in dynamic system the nodes composing the overlay change during the time, due to churn. Dynamic distributed systems are characterized by continuous churn (see 2.2), the phenomenon is well described by the *infinite arrival model* [74] where, in each run, infinitely many processes constituting the *system population* \( \mathcal{V} = \{ p_1, p_2, \cdots, p_i, \cdots \} \) may join/leave the system. However, at each time unit \( t \), the distributed system is effectively composed only by a finite subset of the population, denoted as \( \mathcal{V}_t \), including all the processes that have joined but have not left yet (i.e. \( \mathcal{V}_t \subseteq \mathcal{V} = \{ p_i \in \mathcal{V} \} \) a time \( t, p_i \) has joined the system and it has not left yet\).

Scope of an overlay network is to maintain the network connected despite churn. The churn is one of the biggest handicap to develop large scale distributed application. To clearly understand the issues introduced by the churn the following metaphor is proposed: imagine you want to build a high tower
(i.e. your connected overlay) with your building blocks (i.e. the local views). The churn could be seen as an adversary able to continuously affect the blocks wherever in your building; your effort is much more increased: you have to (i) build your tower, (ii) continuously find where the adversary has broken your tower and (iii) repair the damages (if possible). Even in presence of limited churn if the adversary is very expert it could be possible that you will never be able to build your tower. The metaphor puts in evidence which is the bad side of this phenomenon: the out-churn. This intuition could be explained in the following formal way: the network is a graph, the connectivity of it is fundamental, banally if the graph is not connected (i.e. it is partitioned) there exist at least two entities that cannot communicate; let us assume that at time $t$ the graph is connected, when a new entity joins it expands the graph adding a node and some edges (at least one) without removing nothing for the pre-existent graph. If the graph was connected it has to remain connected; at the opposite if an entity leaves the system it removes a node and some edges from the graph (at least one); if the leaft node was the only connection point between two sub-graphs the graph loses its connectivity: it is partitioned. This rational proves that in dynamic environment the bad side of the churn is the out-churn, due to the fact that it is the only\footnote{from the in-churn/out-churn point of view} thing able to compromise the connectivity of a connected network.

An overlay network can be structured or unstructured, in the first case the local views are organized with well-known rules, and the graph respects some explicit topology (e.g. circles [89], cubes [5], trees [44], etc). On the contrary unstructured networks do not appear to respect any explicit topology, they seem more like a tangle, in fact they look very similar to the random graphs. The pros ans cons related to the two approaches are mainly related on two aspects: (i) churn resistance and (ii) message routing. Structured topologies allow the designer to use deterministic strategy to route the messages among the network. One of the main factor of success of the distributed hash table [89] is in fact their capability or routing the messages directly to the recipient in at most $\log(n)$ steps; an unstructured network makes impossible to do that because the paths connecting the nodes do not follow any specific organization. On the other hand the unstructured topology exhibits a strong resilience to the churn, unstructured topologies remain connected even in the case of high level of churn. Structured solutions can have some important points of failure, for example thinking to a tree with reference to the metaphor previously used, the adversary can kill the structure starting from the root in place of the leaves. In an unstructured topology the adversary cannot have a disconnection strategy suggested by the topology, in fact the admissible churn rates for an unstructured network are orders of magnitude higher then a structured topology [79] [92].
A possible trade-off between the structured and unstructured topology can be represented by some hybrid solutions such as [53] [13]. These solutions use an underlying unstructured topology able to ensure the global connectivity of the network and some small structured topologies built on top of the unstructured one with several goals (for example managing of interest groups [13], improving the availability of come indexes [53], ...). The hybrid solutions exhibit better performance in churn resistance with respect of the structured ones, so they can represent a possible solution for the systems that experience a relevant level of churn but require some explicit topology.

In the rest of the chapter we present some famous structured, unstructured and hybrid topologies with the aim of listing both their capability of handling the churn and both - if exists - their facilities in matter of query routing. The chapter concludes with a section that lists some clear motivations about the impossibility of having a valid query in an overlay network based on single nodes.

3.1 State of the art

In this section we present some famous overlay networks that belong to the categories of structured, unstructured and hybrid topologies. For each architecture we discuss the performance in terms of connectivity and query routing. As reference query we use a general system-wide query that implies the participation of all of the nodes in the system.

In the following we present cyclon [92] as example of unstructured topology, Chord [89] and Thicket [44] as relevant examples of structured topology and Tera [13] and Kelips [53] as example of hybrid solutions.

3.1.1 Unstructured topologies

Unstructured topologies do not follow specific architecture for organizing the local views: they can be assumed composed at random. Due to their random topology all the protocols belonging to this family are very similar; the differences are mainly related on the way the local views are updated, however the large part of the protocols use the local view exchange technique. In order to better understand the technique we present Cyclon [92]: one of the most famous protocol implementing an unstructured network.

Cyclon relies on the extremely simple idea of shuffling [61]: each entity knows a small, continuously changing set of other entities, called local view or neighbourhood. Occasionally each participant contacts a random node in the local view in order to exchange part of its neighbourhood with it. More
formally, each entity maintains the local view in a small, fixed-sized cache of \( c \) entries (with typical value 20, 50, or 100). A cache entry contains the network address (i.e., IP address and port) of another entity in the overlay. Each entity \( E \) repeatedly initiates a neighbour exchange operation, known as shuffle.

On reception of a shuffling request, a node randomly selects a subset of its own neighbours and sends it to the initiating entity.

The neighbours are continuously updated, so they can be considered as a random set of entities; by this end Cyclon is also used to implement a sampling service: in order to select a random node a participant has only to randomly select an entity within the neighbours.

Due to the size of the neighbourhood and the randomness of the local views the unstructured topologies usually result to have (i) a limited - but unpredictable - diameter\(^2\) and (ii) a strong resistance to the churn.

Implementing a querying algorithm on top of Cyclon is quite hard: the network appears like a random graph so there are no strategies - beside the broadcast - to distribute and collect the query. Knowing that the broadcast is an infeasible solution due to scalability issues, two techniques are commonly used to issue a query in a network: the flooding and the random walk.

Query flooding is a method to search for a resource within a network. It is simple but scales very poorly and thus it is rarely used. Early versions of the Gnutella [47] [80] protocol operate searching queries using flooding.

If an entity wants to find a resource in the network, it could simply broadcast its search query to its immediate neighbours. If the neighbours do not have the resource, it then asks its neighbours to forward the query to their neighbours in turn. This is repeated until the resource is found or all the entities have been contacted, or perhaps a network-imposed hop limit is reached (this case is named limited horizon flooding). Query flooding is simple to implement and it is practical for small networks with few requests. Flooding contacts all reachable nodes in the network but, in case of huge network, it has to be implemented with a limited horizon thus the results are just probabilistic.

Some enhancements (like limited horizon) had been introduced to flooding but in any case it is still considered an expensive solution [69] [80]. Anyway, if the number of entities is very limited the flooding is probably the best approach due to its simplicity.

On the other hand a Random Walk [72] consists in sequential random jumps among the entities of a network. Like the flooding the random walk is a methodology used to find a resource in a set of entities. A sketch of algorithm is proposed below:

\[^2\text{the diameter of a network is defined as follows: the largest minimum path length connecting two nodes in the network}\]
1. the entity $E$ throws a query $q$ within the network, it selects a random entity $E'$ then it forwards $q$ to $E'$

2. once received the query if $E'$ knows who has the resource it replies to $E$ the result, otherwise $E'$ selects a new random entity $E''$ forwarding $q$ to it

3. the procedure goes on until a stop condition is verified or the resource is found

Also the Random Walk suffers of scalability issues, its big problem is represented by the query time: in a big network the number of required jumps could be very high. This problem could be solved by using multiple random walks but in any case the query time remains too long. Moreover, issuing a system-wide query by using a random walk is practically impossible: a random walk broadcasts the query in the network in a huge time and moreover its distribution is even probabilistic!

Unstructured topologies are a good choice if the network is characterized by a strong churn and there are no requirements of having system wide queries.

### 3.1.2 Structured topologies

Structured topologies are characterized by a precise local views organization, the resulting graph appears like a regular graph with a clear topology. In the large family of the structured topologies we prefer to present the ones that provide a fast access to a spanning tree. A spanning tree is a tree covering the whole network, due to this reason a spanning tree is a perfect topology (see chapter 5) for running a query: it represents a ready-to-use architecture to distribute the query in the network and collect the results.

The most classic example of structured topologies is represented by the distributed hash tables DHTs.

Distributed hash tables (DHTs) are a class of decentralized distributed systems that provide a lookup service similar to a hash table; (key, value) pairs are stored in a DHT, and any participating entity can efficiently retrieves the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the entities, in such a way a change in the set of participants causes a minimal amount of changes. This allows a DHT to scale to extremely large numbers of entities and to partially handle the churn. Today due to their capability of efficiently managing a key-value, the DHTs represent a very important architecture in the field of peer to peer and cloud computing and they are a comparison reference for the main part of the structured topology.
Historically the first DHT was Chord [89], then it was followed by Pastry [23] [24] and successively by Bamboo [79]. In the following, in order to figure out the way a DHT works we propose Chord.

Using the Chord lookup protocol, entity keys are arranged in a circle. The circle cannot have more than $2^m$ nodes ($m$ is a system wide parameter). The circle can have IDs/keys ranging from 0 to $2^m - 1$. IDs and keys are assigned an $m$-bit identifier using consistent hashing. The SHA-1 algorithm [39] is the base hashing function for consistent hashing.

![Chord DHT Architecture](image)

Figure 3.2: A 4-bit Chord DHT architecture with the finger table of node 1

Chord requires each entity to keep a finger table containing up to $m$ entries. With such a finger table, the number of entities must be contacted to find a successor in an $N$-node ($N < m$) network is $O(\log N)$. The entries of the finger table are used to store the responsibility of the keys associated to the nodes and the references to the successor nodes with the associated keys. The references are organized with the aim of mapping the entire key-space. Each node, receiving a query, checks if it belongs to the key-set associated to himself, otherwise it forwards the query to the successor node that results the closest to the searched key. The finger table organization allows to reach all of the keys (and then the other nodes) in $O(\log N)$ steps. The goal is obtained by the following trick: each node is responsible for every key that comes before it in the ring (the node associated to a key is calculated by the function $\text{successor}(k)$). The $i$-th row in the finger table belonging to node identified by $X$ contains nodes returned by $\text{successor}(X + 2^i(i - 1))$. Figure 3.2 shows an example of a small DHT with a finger table. Note that starting from whatever node using the finger tables it is possible to quickly build a spanning tree of
the network useful for distributing a query.

Since entities may disappear from the network (because of failure or departure), each entity stores a whole segment of the circle adjacent to it, i.e. the \( r \) nodes preceding it and the \( r \) nodes following it. Theoretically, even if the network has a high churn, Chord ring should work fine, practically chord works fine only in presence of limited churn \[79\]. The most recent solutions like Bamboo \[79\] have only partially solved the problem.

A DHT architecture is quite good for general queries, in fact (i) it offers an already implemented mechanism for key-values queries and (ii) due to the capability of keys reachability\(^3\), a DHT can provide a sort of multiple spanning trees always ready to use at each node. Anyway there is not warranty about the maintenance of these spanning trees during the query time, then some partial results can be lost. Moreover the key-value paradigm limit too much the expressiveness of a query, due to this reason some special hash functions \[1\] \[2\] can be used in order to enhance the expressiveness a little bit. These hash functions allow ordering and/or comparison among different keys: the more the prefix of two keys is equal and the more is the relevance between the two values.

In very large systems the DHT-based is the most used approach due to the good trade-off between scalability and the expressiveness of the query. Novel approaches, in order to be considered as a valid alternatives, should allow more expressiveness, more scalability and mainly an enhanced churn resistance.

Another remarkable example of a structured network is represented by Thicket \[44\] a novel decentralized algorithm to efficiently build and maintain multiple trees over a network.

Thicket addresses a relatively unexplored region of the design space: building multiple trees in a decentralized manner on top of an unstructured network. The resultant overlay appears to be composed as a collection of spanning trees: due to this reason is classified as a structured topology.

Thicket operates by employing a gossip-based technique to build \( T \) divergent spanning trees (\( T \) is a fixed protocol parameter), where most of entities are interior in a single tree and leaf in all other trees. The \( T \) trees are assumed defined from the beginning, in any case there are a lot of methods to build spanning trees in unstructured network \[96\] \[54\]. Furthermore, Thicket uses the remaining overlay links for the following purposes: (i) ensuring that all nodes in the system are connected to all trees. Notice that to ensure this, some entities may be required to be interior in more than a single tree; (ii) detecting and recovering from tree partitions when nodes fail; (iii) ensuring that tree heights are kept small, despite the existing dynamics of the system; and finally, (iv) ensuring that the forwarding load of each participant is limited by

\(^3\)each entity should always be able to reach all of the available keys
3.1. STATE OF THE ART

a protocol parameter named \textit{maxLoad}.

The \textit{maxLoad} parameter must be low enough in order to limit the forwarding load imposed to each entity, avoiding overloading situations. However, if the chosen value is too low, entities might be unable to coordinate among themselves in order to generate trees with full coverage (i.e. connecting all entities). Following epidemic theory \textit{maxLoad} should be logarithmic with the number of nodes in the system.

Event if Thicket seems perfect for running a query, it is biased by the same problem examined in a DHT: there is not warranty about the maintenance of these spanning trees during the query time, in fact they can be temporarily disconnected by churn, then some partial results can be lost. In order to solve this issue the same query could be parallel thrown in each spanning trees. The probability that a disconnection event happens in each one of them is quite low. However the churn resistance exhibited by Thicket is equivalent to a DHT, so it appears poor in the case of strong churn rate.

3.1.3 Hybrid topologies

In this subsection we mention two interesting protocols with hybrid organization: TERA [13] and Kelips [53]. Hybrid topologies are usually designed ad-hoc for a specific goal, the proposed protocols are in fact designed to deal with interest groups.

TERA [13] is a topic-based publish/subscribe system [42] [18] designed to offer a robust and efficient event diffusion service for very large scale systems, made up of autonomous and collaborative participants. In TERA we do not make any distinction between clients of the publish/subscribe service and brokers implementing it: each client, either publisher or subscriber or both, also acts as a broker. In the following we will refer to any TERA participant simply as a node of the system.

All nodes in TERA are organized in a logical two-layers architecture (see figure 3.3).

Each group of nodes, belonging to either the lower or upper layer, is kept connected using an instance of an overlay maintenance protocol. The overlay network representing the lower level group contains all TERA nodes and is called global overlay. Overlay networks representing upper level groups are used to maintain connected all nodes sharing the same subscription (i.e. each overlay represents a topic), and are called topic overlays. All these overlay networks are completely independent, and work concurrently, thus every node maintains a separate view for each overlay it is part of. The OMP employed to manage general and topic overlay must only provide a simple mechanism to allow neighbor nodes to communicate (it is implemented with PS service).
A node subscribing to a topic $t$ must join the corresponding topic overlay containing all the other nodes subscribed to $t$. A node publishing an event $e$ for topic $t$, in order to diffuse it toward all the target nodes, must forward $e$ to one of the nodes subscribed to $t$. Once this node receives $e$ it will first notify locally the event, and then broadcast $e$ in the topic overlay associated with topic $t$, where all other subscribers of $t$ are connected.

Various problems must be addressed in order to realize this form of event diffusion. The most important is how a node can find another node in the system that is subscribed to a specific topic $t$ (if such node exists); in other words: we need a service able to return a node that can be used as an access point for a specific topic in order to publish events or subscribe to that topic. TERA solves this problem through a topic sampling service that is used by each node to maintain a cache of access points. This service requires an overlay maintenance protocol to manage the global overlay, in order to work correctly. Access to a set of neighbors at each moment represents a uniform random sample of the whole system population. TERA works perfectly in the case of uniform sampling, unfortunately the churn severely affects the uniformity of this service corrupting the correct execution of the protocol. The churn rates tolerated by TERA are higher than a structured topology like a DHT (see section 3.1.2).

From the query point of view TERA proposes an interesting mechanism based on the idea of the random walk: when a node needs to find a resource it starts a random walk in the system. The caching of the access points at each node allows to boost the random walk obtaining the result in very few steps; obviously the larger is the size of the cache and the faster is the random walk. For the same reason given for the random walk (see 3.1.1) the search system
3.1. **STATE OF THE ART**

implemented in TERA is not suitable for a system-wide query, anyway the caching system introduced represents an interesting strategy for the querying systems, it is in fact also used to speed-up the flooding and the random walk.

The last protocol that we present is Kelips. Kelips consists of $k$ virtual affinity groups, numbered 0 through $(k - 1)$. Each node lies in an affinity group determined by using a consistent hashing function to map the node identifier (IP address and port number) into the integer interval $[0, k-1]$. Let $n$ be the number of nodes currently in the system. The use of a cryptographic hash function such as SHA-1 ensures that with high probability, the number of nodes in each affinity group is around $\frac{n}{k}$.

Each node in Kelips stores the following entries: (i) affinity Group View: a (partial) set of other nodes lying in the same affinity group. Each entry carries additional fields such as round-trip time estimate, heartbeat count, etc. for the other node; (ii) contacts: for each of the other affinity groups in the system, a small (constant-sized) set of nodes lying in the foreign affinity group. Entries contain the same additional fields as in the affinity group view; (iii) filetuples: a (partial) set of tuples, each detailing a file name and host IP address of the node storing the file (called the file homenode). A node stores a filetuple only if the file homenode lies in this node affinity group. Filetuples are also associated with heartbeat counts.

Kelips uses the same strategy introduced by TERA: dividing the group the nodes in interests structured sub-groups. At the opposite of TERA, Kelips uses consistent hashing for solving the search operation, for this reason it results close to a DHT. In TERA each subgroup is independent, in Kelips all of the sub-groups are connected through affinity policies, this easy but smart trick allows Kelips to efficiently solve the search operation like in a DHT but at the same time it resists to the erosion of the churn thanks to the redundancy of the link among the sub-groups. Like in the Chord ring, Kelips is not suitable for system-wide query, anyway it introduces the interesting concept of sub-group interconnection to resist to the churn erosion.
Chapter 4

Building an overlay network by using Attractors and Virtual Nodes

“United we stand, divided we fall”

Aesop @ The Four Oxen and the Lion (∼600 BC)

An overlay network has the aim of connecting all of the nodes of a network preserving their connectivity. The previous chapter has introduced several overlay networks showing the various alternatives in terms of topology and related capability. Chapter 3 has shown for each overlay category its capability of handling the dynamism and the possibility of having a query algorithm on top of it. The classic overlays are not able to provide valid queries because they do not provide clear mechanism for ensuring the validity of the results.

In this chapter we present a novel procedure called attractors that we apply to the abstraction of virtual node. In the last three years the virtualization concept has obtained a large research interest due to the cloud computing but it has never moved toward the field of distributed dynamic systems as a design principle; virtual nodes and the attractor procedure are an attempt to do that with the aim of implementing complex structure offering high level facilities
that on the contrary could not be implemented with ongoing solution in a dynamic environment with the single physical nodes.

4.1 Can physical node based overlay obtain valid queries?

The previous section presents some famous overlay networks for dynamic systems. For each of them we sketch the query algorithm usable on that network discussing about the related performance. The proposed analysis on the state of the art shows that having a system-wide query is not an easy task due to the network dynamism. Structured overlays seem to be a good solution for the queries. Exploiting their well-know architecture they can implement a query algorithm in an efficient way, for example a DHT is able to perform a query in logarithmic time, moreover its particular architecture allows each node to have an easy access to a spanning tree useful to propose system-wide queries.

A query usually imposes a broadcast/convergecast schema to obtain the results. With the broadcast all of the nodes in the network are informed about the request, then through the convergecast phase the results are collected. In a large network having a broadcast is a costly operation, classic algorithms such as the reliable broadcast [50] are not suitable to this end; probabilistic solutions can represent a valid light-weight alternative, anyway they do not offer warranties about the query delivery into the entire network. This elaborate focusses on the problem of valid queries, if a protocol does not ensure to distribute the query in the whole system - even in the case of zero churn - it automatically compromises any hope of deterministically obtain a valid result. In order to efficiently broadcast the query in the network a distribution strategy is required in order to limit the message wasting. The strategy followed by the broadcast in order to reach all of the nodes is called distribution schema; a spanning tree is the classic example of what a distribution schema can be.

The reader is invited to note that any distribution schema can be always seen as a spanning tree, even in the case of redundant paths. In this case the distribution schema can be seen as an overlapping of multiple spanning trees as shown in figure 4.1. Due to this reason, without loss of generality, in the following of this section we will refer to spanning tree or distribution schema indifferently.

Having a distribution schema, it is easy to have a valid query in a static environment: the distribution schema can be used both for distributing and both for collecting the results, the distribution schema reaches all of the nodes, so all of them can participate to the final result satisfying the property of snapshot validity (see 1.2). The only problem that makes impossible to have
4.1. CAN PHYSICAL NODE BASED OVERLAY OBTAIN VALID QUERIES?

Figure 4.1: an example of distribution schema composed by two spanning trees

A valid query in a dynamic system is represented by the churn because it can disaggregate the distribution schema during the query compromising both the broadcast and the convergecast.

In the case of unbounded churn there is no way to obtain a valid query [20], however in the realistic case of bounded dynamism nothing is proved. Intuitively a system-wide query must rely on a distribution schema, i.e. on a spanning tree. A spanning tree can be build ad-hoc before the query or it can be provided by the overlay network, but in any case it is continuously prone to churn and then it is continuously disaggregated. As we introduced in chapter 1 each process of the distributed dynamic system is independent from it, thus it can leave the network also during the computation and the computation should be able to conclude independently from the failures and the voluntary leaves. If the distribution schema is based on the single physical nodes it is impossible to ensure that the entities composing the schema remain in the system with the aim of preserving the computation liveness because it violates the dynamism concept itself. It follows that the distribution schema must rely on reliable entities that are not independent from the computation because they have to remain in the network till the end of it. In order to solve this issue in this chapter we propose a novel abstraction called “virtual nodes”.

The simple replication is not able to deterministically ensure the liveness of the computation. The time required to solve a query can vary in function of the complexity of it, the network delays and so on; if the churn is continuous it follows that for each time unit a given number of nodes are disconnected by the system, thus, in the case of large query time, every replicated spanning tree will be eventually disaggregated corrupting the computation of the query. A proactive approach able to maintain the distribution schema with ongoing solution is required to solve the problem.
CHAPTER 4. BUILDING AN OVERLAY NETWORK BY USING ATTRACTIONS AND VIRTUAL NODES

4.2 What are the attractors and the virtual node

A virtual node [vn] is a small group of physical nodes fully connected (i.e. a clique) in which are clearly defined two policies:

- the interconnection toward the other virtual nodes that defines the virtual overlay
- the attractors i.e. the rule for moving the physical nodes among the virtual ones

The virtual node size is included in an interval defined by min-size and max-size. If the virtual node reaches the max-size it stops to receive other nodes diverting the new ones toward other virtual nodes. On the contrary if it reaches or it goes beyond the min-size it starts to attract new physical nodes from other virtual nodes accordingly to the specified policies. We call this operation attractor procedure of simply attractor or attractors. The attractor procedure of a Virtual Node has to contain a deterministic rule that for each time unit defines from which other vn or vns can get the nodes for having |vn| > min-size. For example in figure 4.2 vn A is allowed to get the physical nodes from vn B, assuming a min-size of 2. Once node c disappears the attractor procedure immediately moves node d from vn B to vn A fixing the hole introduced by the churn.

The attractors represent a proactive procedure able to ensure the survival of the virtual node. Thanks to this operation the survival of a vn is no longer related to the single physical nodes, allowing the designer to assume that a virtual node does not disappear from the system independently from the computation and from the dynamism. During the computation the physical nodes that compose the vn at the beginning of the computation can completely change; anyway, the liveness of the operation is preserved by the presence of the information maintained inside virtual node by the physical nodes that alternatively compose it. Virtual nodes and attractors define a reliable virtual architecture that allows to design the protocol assuming it will be not disaggregated by the churn.
The concept of Virtual Node intrinsically defines the concept of Virtual Link [vl]: a vl is the collection of all of the links connecting two virtual nodes. Each node of a virtual node can contact all of the nodes present in another vn connected through a vl. This can be obtained simply connecting all of the node of a vn with all of the nodes of the other one.

The idea of the virtual nodes was firstly introduced in [15] and then expanded in [17] [17].

A clique of nodes, in order to be defined as a vn, must include the attractor procedure, on the contrary it can be assumed as simple replication because it does not include any procedure able to preserve the survival of the group. In the past several works tried to introduce similar concepts but no one included the idea of the attractors, for further details please refer to the following related works subsection.

4.2.1 Related works

Several works [15, 35, 37, 43, 67] leveraged clusters of nodes in order to improve the robustness of an overlay network.

In [37], the authors introduce the notion of Virtual Mobile Node. This paper represents the first application of this abstraction. In the work the virtual nodes are introduced in field of mobile ad-hoc network. In this setting predicting how the nodes will move in the space is impossible because all of them are independent, thus if at time $t_1$ two nodes A and B are at transmission distance at time $t_2$ it could be no longer true. The problem faced by the authors is to move an information from a point X to a point Y of the space relying only on the physical nodes. In order to do that a Virtual Node is defined in X using the nodes that are close to that point. The Virtual Node is then moved toward Y asking to the nodes that are closer to Y and at transmission distance, to join the virtual node pushing out the the nodes that are moving in the opposite directions of Y. The authors show that if the route between X and Y is reasonably reach of nodes, even if they are moving in random directions, the Virtual Node reaches Y. The Virtual Nodes presented in that work deeply differ from the ones introduced in our work because there is not attraction procedure and there are no relations between the Virtual Nodes. In [37] the Virtual Node moves itself among several physical nodes, in our approach the physical nodes are moved among the virtual ones.

Another interesting application of the Virtual Node is represented by [43]. In that work Eyal et al. use the virtual nodes in order to tolerate byzantine behaviors. The scenario addressed by the paper is represented by a large collection of sensors in which is required to collect aggregated results evaluated over the entire network. In the paper the authors divide the network in several
small clusters that are used to partially aggregate and filter the values read by the sensors. If some values result clearly far from the average of the others they are pruned out. The filtering introduced by the various clusters (the authors do not explicitly use the term “virtual node”) allows to perform aggregation and filtering with a very good scalability. Also in this case the work differs from the one proposed in this chapter because there is no clear procedure able to preserve the survival of the clusters.

An interesting work, probably the closest to the concept of Virtual Node introduced in this thesis is Overnesia [67]. It introduces virtual nodes (called super peers) defined as small cliques of processes. Given two super peers, they are connected through multiple probabilistic links. The main difference between Overnesia and the virtual nodes lies in the fact that the former does not provide any mechanism to prevent super peers from disappearing but they rather repair the overlay after this has happened. As a consequence, the network can experience temporary disconnections.

The idea of using Virtual Nodes has been adopted in Amazon’s Dynamo [35] as well. However, Dynamo defines virtual nodes as virtual replicas in a logical key-space of “physical” nodes, and exploits them to balance the load in its architecture. Dynamo represents an important example of the other face of the Virtual Node: in this thesis we show how Virtual Node can be used in order to implement complex (virtual) structures in presence of dynamism, so we put the focus mainly on the reliability introduced by the vn; with Dynamo Amazon puts the focus on the load balancing. Since the information contained in a vn is replicated in the nodes composing it, a virtual node can autonomously balance the work load among the group without the need of a centralized coordination improving the performance of the overall system. Amazon is a managed environment, so it is not exposed to the strong dynamism of a large scale dynamic network, in fact, also in this case the concept of attractors is completely missed.

Analytical studies [51] [19] proved that also another big company uses a concept close to the virtual node to improve the performance of its architecture: Skype. The Skype architecture uses node clustering for ensuring the connectivity of the network; as in this elaborate Skype puts the focus on the reliability introduced by the virtual nodes. In Skype - the world’s largest voip telephony system - the nodes are grouped in small groups coordinated by a so called super peer. A super peer is a node that exhibits good network and cpu performance, and thus it can coordinate the rest of the group. The group reliability is related to the super-peer, if it fails the group is temporarily disconnected; in the virtual nodes introduced here this is not possible: the vn liveness is in charge of all the nodes of the clique, so there is no single point of failure. In the past, a similar approach was envisioned also for the Gnutella [47] and KaZaA [62] networks [93].
4.3 How virtual nodes can “drive” the dynamism

Virtual nodes represent a powerful abstraction useful to manage complex structures as a tree in dynamic environment. As detailed in section 4.2 a virtual node has to know from which other vns can got the physical nodes in order to fill its group.

The attractor procedure can be easily completed without the need of coordination: once a node detects that the size of the group violates the min-size it independently selects a node (or more than one) from another virtual node. Without any coordination all of the nodes of the clique potentially start multiple concurrent attraction procedure, anyway this does not represent a trouble because the node/s to be moved are selected using a unique selection policy (e.g. the first nodes in an alphabetical order, so each node of the vn will select the same node/s to move). Each node of the vn has a large local view composed by (i) the nodes in its vn, (ii) the nodes in the adjacent vns, in particular the nodes of the vns from which they can attract the physical nodes.

The attractor procedure can start a chain reaction into the network that moves the nodes in the network in order to respect the min-size condition of all of the virtual nodes. What happens if all of the virtual nodes have a size equals (or lower) than min-size? This question puts in evidence an important point: an “end-point” for the attraction procedure have to be known a priori. An end-point is represented by a virtual node that cannot attract nodes, for example in a tree the leaves represent the end-point of the structure, because the leaves can disappear from the system without disconnecting any sub-tree. In this sense the designer of the vn network can decide a-priori which vn can be disconnected by the system. Knowing a priori the virtual nodes that will be disconnected by the system the applications designed to run on top of a vn-network can take into account the event preventing the introduction of errors due to unpredictable disconnection preserving the computation. Using the virtual nodes the churn is bridled inside the virtual nodes and it is driven toward the area of the network in which it cannot introduce faults.

In the following section we detail Virtual Tree: a tree-shaped overlay network composed by virtual nodes. In the discussion of Virtual Tree are also treated the strategies for connecting the virtual nodes and the algorithm used to implement the attractor procedure.

4.4 Virtual Tree

In the previous sections and chapters it is remarked how the unstructured networks show a good robustness to dynamicity but are quite expensive when used to support information dissemination and retrieval. This is due to the
CHAPTER 4. BUILDING AN OVERLAY NETWORK BY USING ATTRACTORS AND VIRTUAL NODES

absence of clear facilities to gather and disseminate information in the network. Structured networks are suitable to efficiently disseminate and retrieve information but they are not as resilient to the churn as the unstructured solutions. Defining a spanning tree of the network is actually the most efficient way to support queries dissemination on top of a distributed system. However, the spanning tree fragility, with respect to node failures, limits its employment in dynamic environment. A possible approach to combine together (i) the efficiency in the information dissemination provided by structured network with (ii) the robustness characterizing unstructured network, is represented by the abstraction of virtual nodes introduced in this chapter.

In order to run an in-network query answering protocol, we must first define the overlay network connecting processes that participate to the system. Several overlay network schemes are suitable, but tree-shaped topologies offer some clear advantages in the form of (i) low diameter (useful to quickly disseminate the query and collect its results), (ii) good scalability and (iii) the possibility to easily define protocols with deterministic termination conditions. However, three-shaped topologies are strongly susceptible to faults and dynamism. In order to provide query answers complying with the IV semantics two necessary conditions [20] must be met: (P1) the overlay network must always be connected and (P2) any process that does not leave the system during the query execution must have a stable path (a path that does not change) that connects it to the query source.

We solve the first problem by introducing Virtual Tree [VT] graph that exploits virtual nodes to improve its resilience to system churn. In order to address the second problem, we implement the overlay management protocol (OMP) by including the attractor procedure that migrates processes at run time from the lower levels of the VT graph to the upper ones in order to let churn impact only its leaves. Through this technique, the OMP can guarantee, as long as churn is bounded by a given constant, that the VT graph will meet conditions P1 and P2.

In figure 4.3 is provided a sketch of the architecture of the Virtual Tree, the formal details are provided in the following subsection.

From an high level point of view Virtual Tree appears like a \( k \)-ary tree of virtual node (\( k > 1 \) is a system wide parameter) fully connected one to one, then two virtual nodes directly connected define a clique.

The Virtual Tree topology can be considered part of the family of cluster based overlays. Some examples are represented by [68] and [5]. eQuus [68] makes use of small cliques (full graphs) of nodes to enhance performance and reliability in a DHT. Differently from our proposal, eQuus solves the problem of maintaining node clusters through merge and split operations; this solution was not applicable in our case due to fact that the merge operation introduces a change in existing virtual paths thus violating the required path
stability property. PeerCube [5] builds and maintains an hypercube of virtual nodes constituted by cliques; also in PeerCube virtual nodes can merge and split. Recently the PeerCube structure was applied to the problem of isolating targeted attacks [6].

4.4.1 The Virtual Tree Graph

A VT graph is constituted by virtual nodes (VN) and virtual links (VL) arranged in a tree-shaped topology. A virtual node $VN_i = (V_i, E_i)$ is a subgraph of VT constituted by a set of nodes $V_i$ interconnected by a set of edges $E_i$ in a full graph (i.e. a clique). A virtual link $VL_{i,j}$ connecting two virtual nodes $VN_i$ and $VN_j$ is defined by the set of all the edges $e_{h,k}$ connecting any pair of nodes $p_h$ and $p_k$ such that $p_h \in VN_i$ and $p_k \in VN_j$. Let us note that nodes belonging to two adjacent VNIs define a fully connected subgraph of the VT graph. As an example, in Figure 4.3, $VN_i$ and $VN_r$ represent two adjacent VNIs constituted by nodes $I, L, M$ and $A, B, C$ respectively; $VL_{r,i}$ is the VL interconnecting them and it is constituted by links connecting each node in $VN_i$ with every node in $VN_r$.

Now we can formally define the structure of a VT graph as follows:

**Definition 3** (Virtual Tree graph). Let $VN = \{VN_1, VN_2, ..., VN_x\}$ be a set

$^{1}$Note that, unless otherwise stated, in the following we refer to processes with the generic term node and to a virtual node with the acronym VN.
of virtual nodes and $\mathcal{VL}$ be a set of virtual links. A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a Virtual Tree (VT) graph if:

1. $\mathcal{V} = \bigcup_{i=1}^{x} V_i$;

2. $\mathcal{E} = (\bigcup_{i=1}^{x} E_i) \cup \mathcal{VL}$;

3. each $VN_i$ is associated to an integer defining its level in $\mathcal{G}$;

4. $\mathcal{G}$ contains a single virtual node at level 0 (root $VN$);

5. $\forall VN_i, VN_j : V_i \cap V_j = \emptyset$;

6. every virtual node $VN_j$ at level $l$ is connected through a virtual link $VL_{j,k} \in \mathcal{VL}$ to one single virtual node $VN_k$ at level $l-1$; in this case, we say that $VN_j$ is child of $VN_k$ and $VN_k$ is father of $VN_j$;

7. no virtual link exists in $\mathcal{VL}$ connecting two virtual nodes at the same level;

Given a VT graph, it is possible to define paths connecting any two virtual nodes:

**Definition 4** (Virtual Path on VT graph). Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a VT graph. Let $\mathcal{VN} = \{VN_1, VN_2, ..., VN_x\}$ be the set of virtual nodes of $\mathcal{G}$ and let $\mathcal{VL}$ be its set of virtual links. Given two virtual nodes $VN_i$ and $VN_j$, a virtual path $\mathcal{P}_{i,j}$ on $\mathcal{G}$ between $VN_i$ and $VN_j$ is a sequence of virtual nodes such that for any two consecutive virtual nodes there exists a virtual link in $\mathcal{VL}$.

From Definition 4, it follows that given a VT graph $\mathcal{G}$ and a virtual path between two virtual nodes $VN_i$ and $VN_j$, there exists at least one path between any pair of nodes $p_i$ and $p_j$, such that $p_i$ belongs to $VN_i$ and $p_j$ belongs to $VN_j$. At the same time, given a path between two nodes $p_i$ and $p_j$, such that $p_i$ belongs to a virtual node $VN_i$ and $p_j$ belongs to a virtual node $VN_j$, there exists a virtual path between $VN_i$ and $VN_j$. 
4.4. VIRTUAL TREE

4.4.2 Overlay management protocol

The OMP has two fundamental goals: (i) it must position joining processes in the overlay network maintaining a VT graph and (ii) it must guarantee that only leaf VNs can possibly disappear due to out-churn. This latter requirement stems from the observation that if a leaf VN is removed from the VT graph, the graph is still connected and none of the VNs still present in the graph will see a change in the paths that connect them to the root VN; therefore, both P1 and P2 will be preserved. The OMP relies on a view-maintenance algorithm [29] to keep local views on nodes up-to-date and consistent with respect to each VN population. In accordance with the relaxed asynchronous model assumption made in Section 2.1 here we assume that the view maintenance algorithm is able to guarantee deterministically consistent local views despite node joins/leaves. In Section 5.4 we will relax this assumptions explaining how a probabilistic approach can be employed for the maintenance of the VT graph.

In the following, we will assume that such view maintenance algorithm will update the local views periodically.

**Join Algorithm.** New processes joining the system can, in principle, be accommodated in any VN (a join, in fact, cannot impact graph connectivity or path stability, see 3). However, being each VN a fully connected graph, it makes sense to maintain its population small in order to reduce the overhead incurred for the maintenance of local views. Therefore, the OMP allows up to a maximum of $N_{\text{max}}$ processes in a VN, where $N_{\text{max}}$ is a configuration parameter. A VN that reaches this threshold will delegate the acceptance of joining processes to one of its children VNs. Note that if the selected child cannot accept the request because it reached the max size it can (i) create a new child composed by the only joining process (if it has less than k children) or (ii) send the join request to one of its own children.

In particular, when a new process $p$ wants to join the system, it obtains a pointer to a process $p_{ap}$ already part of the system from the bootstrap service [59]. Afterwards, $p$ sends a join request message to $p_{ap}$ that acts as follows:

- if the size of $p_{ap}$’s VN is smaller than $N_{\text{max}}$, then it sends back to $p$ an acceptance message containing all the information concerning the population of the VN it belongs to, its father VN, all the children VNs (needed by $p$ to join all the groups containing $p_{ap}$). In addition, $p_{ap}$ also sends all the information related to concurrently running queries to ensure the termination of the query despite continuous process arrivals and departures.
- if the size of $p_{ap}$’s virtual node is equal or larger than $N_{\text{max}}$, $p$ can either
forward the join request to a process in a child virtual node (if it already has $K$ children) or create a new child virtual node containing only $p$. $K$ is a configuration parameter defining the maximum number of children that a virtual node will create. Note that $K$ affects the total number of edges in the $VT$ graph and thus impact the cost incurred for maintaining local views.

Due to the dynamism induced by churn, the join procedure could possibly not terminate (e.g. because $p_{ap}$ leaves the $VT$ graph before correctly placing $p$ in a virtual node). To avoid this problem joining nodes can use a timeout to reissue failed join request to a new bootstrap node.

**Attractor Procedure.** The attractor procedure has two goals: (i) maintaining the $VT$ graph connected and (ii) ensuring virtual path stability. These can be both achieved by ensuring that at any time all VNs (with the exception of leaves) are composed by a non-zero population. This should be guaranteed when nodes leave the system too. To this aim, whenever the size of a certain virtual node $VN_i$ shrinks below a given threshold $N_{min}$ the OMP starts migrating nodes from its children $VN$s and moves them in the father virtual node $VN_i$ to reconstitute a “safe” size, thus avoiding its disappearance from the $VT$ graph. When applied to the whole $VT$ graph, this procedure can create a cascade effect such that nodes are progressively moved from leaf $VN$s to the upper levels. The $N_{min}$ threshold is a function of the maximum allowed churn rate and, intuitively, must be calculated with the aim of giving “enough time” to the OMP to migrate nodes from the lower levels of the graph toward a non-leaf $VN$ that is currently experiencing a local churn surge.

When nodes belonging to a certain virtual node $VN_i$ detect that its size $|V_i|$ is smaller than $N_{min}$, they take the following steps:

1. each process $p_i$ selects, according to a deterministic rule, $N_{min} - |V|$ processes, called helpers, among all those belonging to children $VN$s. Afterwards, $p_i$ sends a HELP message to each helper; such a message contains all the information needed by the helpers to migrate in $VN_i$.

2. delivering a HELP message, a process $p_j$ updates its local view according to the information received.

Note that the maintenance procedure is always running and helper processes continue to be attracted from children $VN$s until the virtual node size $|V_i|$ grows larger than $N_{min}$.
Chapter 5

Obtaining a valid query by using Virtual Tree

“A distributed query plan however requires us to deal with dynamism in the network, and the semantics of the final answer returned”


In [20], Bawa et Al. propose three different semantics, namely snapshot validity, interval validity and single-site validity, defining when the result computed by an aggregate query can be considered valid. The authors also prove that in an unstructured dynamic network, with unbounded churn, the strongest semantics that can be deterministically satisfied is single-site validity. Successively in [14] Baldoni et al. define one further semantic, namely dynamic validity, to take into account the effect of churn due to both join and leave operations. The same work also provides an aggregate query answering algorithm enforcing dynamic validity semantics.

Several algorithms have been proposed to provide query answers but most of them provide just best effort semantics [31], [55], [71] i.e., the algorithm does its best to gather all the values maintained by the processes part of the network despite their continuous arrival and departure. Another remarkable example of query answer protocol is represented by [45] and [82] in which the
authors propose mechanisms to distribute database functionalities on top of a peer-to-peer network, in particular, in [45] Furfaro et al. consider historical queries. Even in this solution, the results remain in the “best effort class”.

In this chapter we present a query algorithm able to ensure the interval validity [IV] property. The IV states that a result is said to be interval valid if and only if it includes all of the contributions of the nodes that remain in the network for the entire query interval. The property is achieved by exploiting the Virtual Tree overlay introduced in the previous chapter. Virtual Tree [VT] builds and maintains a tree-shaped topology making use of the virtual nodes [vn]. The vns allow the Virtual Tree to maintain stable virtual paths useful to distribute and gather the query despite churn.

The chapter is organized in two main sections: the theoretical and the practical one. In the first one, introducing the realistic assumption of bounded churn, we prove that under the relaxed asynchronous model (the same of [20]) it is possible to prove that (i) the query algorithm eventually terminates and (ii) the results obtained are always interval valid; in the second section we prove through an extensive experimental evaluation that the theoretical results are also maintained into an asynchronous environment. The theoretical section introduces also an important achievement: independently from the churn rate the connectivity of the Virtual Tree (together with the query algorithm) is a sufficient condition for obtaining a valid result, this important property cannot be deterministically ensured also in the case of asynchronous environment, however the condition is not contradicted by the experimental evaluation, proving that the couple VT-query algorithm is a reliable solution for having valid queries in large dynamic networks.

5.1 How to obtain a valid query

Obtaining a valid query means to have an algorithm able to consider the contributions of all of the nodes connected to the network. It means that (i) the nodes must be always able to receive the query and send their contributions to a collection point and (ii) the same collection point must be able to evaluate a termination condition. The two main issues to be solved are then the reachability and the termination. Solving the reachability issue means to ensure that each node can be contacted in a finite time; solving the termination problem means that exists a time $t$ such that at time $t' > t$ all of the contributions required for the query are collected and the result will be eventually carried out. In the following, without loss of generality, we will assume that if the contributions are available, the result can be surely computed in a finite time, so the termination problem only consists in a termination condition about the contributions collection. In the following we introduce a naive solution that
5.2. A QUERYING ALGORITHM FOR THE VIRTUAL TREE

will be used for explaining the complexity of having a valid result.

Having a connected environment and assuming that each node can be reached by a broadcast message in a finite time $\alpha$ it is possible to use a trivial broadcast/convergecast algorithm for having a valid query. The broadcast/convergecast algorithm consists in two phases: a broadcast phase in which the query is distributed in the entire network and a convergecast phase in which the nodes reply with their contribution. The process that starts the query in the network is called starter. The algorithm works as follow: the starter distributes the query (broadcast phase), then it waits $2\alpha$. In the meantime the broadcast issued by the starter reaches the entire network in $\alpha$ time, then each node - receiving the query - can reply with another broadcast message that will reach the starter (and also the other nodes) always in $\alpha$ time. The starter, receiving the messages, collects the contributions in a contribution set. Finally, after waiting the timeout of $2\alpha$, the starter has surely collected all of the contributions in the contribution set and then it can compute the final result on it.

If the assumption about the reachability is respected the query is interval valid. All of the contribution must reach the starter in a finite time, then the starter is able to express a termination condition.

In a realistic environment the trivial algorithm presents several issues: first of all defining the constant $\alpha$ on the timely delivery of the messages is too hard, in fact in order to do that the channels should be timely, the paths toward each node should be stable and the maximum path length should be known. Deterministically ensuring all these things in a dynamic environment prone to churn is very hard; moreover the broadcast operation is hard to scale over a large network beside the case of overlays that allow to broadcast a message with low message wasting (e.g. a spanning tree).

The trivial algorithm introduced will be never used in a real large scale dynamic system, anyway it is introduced with the aim of presenting the logic (broadcast/convergecast) of a system-wide query and the issues (reachability and termination) that has to be solved in order to ensure an interval valid result. In following section we present a query algorithm designed to run on top of the Virtual Tree overlay. Virtual Tree allows to maintain stable paths toward the root of the tree solving the reachability problem; moreover the tree-shaped topology allows to implement an easy termination condition solving also the second issue.

5.2 A querying algorithm for the Virtual Tree

The query processing algorithm is an adaptation of a broadcast/convergecast approach with partial result aggregation, modified to run on the VT graph
CHAPTER 5. OBTAINING A VALID QUERY BY USING VIRTUAL TREE

topology. Without loss of generality, we assume that all queries are started by the root VN\(^1\) that disseminates the query throughout the VT graph. Starting from the leaf VNs partial results are aggregated in intermediate VNs and forwarded to the upper levels until they reach the root of the VT graph. The absence of disconnections in the VT graph and the stability of virtual paths (both provided by the OMP), together with the structure of the query protocol guarantee that the returned result will include contributions from all processes that remained in the system for the whole query duration and will thus comply with the IV property.

More specifically, the query processing algorithm is started from the node \(p_i\) belonging to the root VN\(_r\) that issues the query. When a query \(q\) is issued by \(p_i\) the following steps occur:

1. \(p_i\) takes a snapshot of nodes belonging to VN\(_r\) (including itself)
2. \(p_i\) stores the lists of the current children VN identifiers at level 1.
3. \(p_i\) sends a message \texttt{QUERY}(\(q\)) to all processes part of VN\(_r\) and its children.
4. When a node \(p_j\), belonging to a VN\(_x\) at some level \(i\) receives the \texttt{QUERY}(\(q\)) message, it:
   
   (a) repeats step 1, 2 and 3 at its own level;
   (b) sends \texttt{QUERY_REPLy}(\(q,\text{value}_j\)) to all the nodes in VN\(_x\);
   (c) waits until:
       (i) it collects \texttt{QUERY_REPLy}(\(q,\text{value}_k\)) messages from each node \(p_k\) belonging in the snapshot of VN\(_x\) and
       (ii) it delivers a \texttt{CHILD_QUERY_REPLy}(\(q,\text{ag}_{VN_y}\)) messages from at least one process in every children VN\(_y\) at level \(i+1\);
   (d) evaluates the query on the set of values collected from its snapshot and on the partial aggregated results coming from the lower level;
   (e) sends a \texttt{CHILD_QUERY_REPLy}(\(q,\text{ag}_{VN_x}\)) message to all the nodes in the father virtual node at level \(i-1\).

For any virtual node VN\(_i\), its snapshot is updated every time that a local view change occurs due to a leave and the corresponding process is removed. Nodes belonging to VN\(_i\) snapshot are required to participate in the query evaluation, while other nodes joining VN\(_i\) after the snapshot is taken are only required to participate in the aggregation process in order to guarantee that partial aggregated results flow toward the root VN. In this way, even if a

\(^1\)Other nodes can delegate nodes in the root VN if needed.
5.3 Deterministic warranties about validity

5.3.1 Algorithm correctness

In the following, we will first prove that given an overlay network structured as a VT graph it remains connected as long as the churn rate is below a certain threshold (Lemma 1). Then we will show that in any connected virtual tree topology there always exists a stable virtual path between any pair of virtual nodes (Lemma 2) and in every interval of $2\delta$ time there also exists a stable path between processes belonging to different virtual nodes (Lemma 3). Finally, we will show that this conditions are sufficient to ensure that the query algorithm introduced in the previous section always return interval valid results (Theorem 1).

Lemma 1. Let $G = (V, E)$ be the overlay network at time $t_0$. Let $T_{move}$ be the upper bound on the time needed by any process to be part of a new view. If (i) $G$ is a virtual tree graph and (ii) at any time $t$, $\sum_{i \in [t, t+T_{move}+1]} \mu(i) < N_{min}/N_0$, then $G$ is always connected.

Proof. $G$ is connected as long as a path exists between any pair of nodes. Considering that at time $t_0$ $G$ is a virtual tree (and thus it is connected), connectivity can be broken only due to the effect of churn. The join procedure has no impact on the connectivity of the overlay network as the algorithm adds new edges to $G$ while old ones are unaffected. $G$ can be partitioned if and only if all nodes belonging to a non leaf VN leave the system and no new node replaces them. Let us consider the worst case scenario where the out-churn affects a single VN while in-churn is 0. Without loss of generality, let $VN_i$ be the first virtual node whose size reaches the threshold $N_{min}$ and let $t$ be the time when this is detected by processes belonging to $VN_i$. Let

virtual node population is completely “refreshed” between the query dissemination time and the partial aggregate evaluation delivery time, the query aggregate result can still proceeds toward the VT graph root.

During the query execution nodes can be attracted toward VNs at the upper levels. When a node moves to a new VN it keeps listening to messages exchanged in its old VN until all the pending query processing procedures are completed. In this way, a moving node will always answer to a running query either in its original VN or in the VN it is moving to avoiding the loss of its contribution.
us denote as $V_i(t)$ the set of processes belonging to $VN_i$ at time $t$ and let $|V_i(t)| \geq N_{min} + 1 - (\mu(t) \cdot N_0)^2$.

According to the rules of the virtual tree maintenance algorithm, at time $t$, each process in $VN_i$ selects, through a deterministic function, $N_{min} - |V_i(t)| + 1$ helper processes among those belonging to $VN_i$ children, and asks them to move in $VN_i$. Considering that $T_{move}$ represents the maximum amount of time needed for a node to move from a child $VN$ to the father, the selected helper processes will complete their movement at time $t + T_{move}$ at the latest. In the meanwhile, for each time $t'$ between $t$ and $t + T_{move}$, $\mu(t') \cdot N_0$ processes leave and, in the worst case, they leave from $VN_i$ (i.e. the size of $VN_i$ shrinks to $N_{min} + 1 - N_0 \sum_{i\in[t,t+T_{move}]} \mu(i)$). To avoid disconnections, we must guarantee that during $T_{move}$, at least one process remains in $VN_i$ despite the out-churn. In the worse case, all processes leaving at time $t + 1$ are all helper processes selected at time $t$. However, the maintenance algorithm will continue to select new helper processes, until there exists at least $N_{min}$ processes in $VN_i$. Considering that, at each time $t'$ the number of selected helper processes is $N_{min} + 1 - |V_i(t')|$ and that at time $t + 1$ the number of selected processes is greater than $\mu(t) \cdot N_0$, we have that, $N_{min} + 1 - N_0 \sum_{i\in[t,t+T_{move}+1]} \mu(i) > 1$ from which the claim follows.

Lemma 2. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be the overlay network at time $t_0$. If (i) $\mathcal{G}$ is a virtual tree graph and (ii) at any time $t$, $\sum_{i\in[t,t+T_{move}+1]} \mu(i) < N_{min}/N_0$, then there always exists a stable virtual path connecting any two virtual nodes.

Proof The proof trivially follows by considering that (i) the graph is connected (Lemma 1) and (ii) at time $t_0$, $\mathcal{G}$ is a VT graph. As a consequence, no virtual node, except leaves, can disappear from the graph. Therefore, for any two virtual nodes the virtual path connecting them never changes and thus the claim follows.

Lemma 3. Let $VN_i$, $VN_j$ and $VN_k$ be three virtual nodes such that $VN_i$ is father of $VN_j$ and $VN_j$ is father of $VN_k$. Let $p_i$ and $p_k$ be two nodes in $VN_i$ and $VN_k$ respectively. If, for any time $t$, $\sum_{i\in[t,t+T_{move}+1]} \mu(i) < N_{min}/N_0$, then at least one of the paths connecting $p_i$ and $p_k$ is stable for at least $2\delta$ time units.

\footnote{Let us remark that, at time $t - 1$, at least $N_{min} + 1$ processes were part of $VN_i$ and $\mu(t) \cdot N_0$ is the number of processes that leave at time $t$.}
5.3. **DETERMINISTIC WARRANTIES ABOUT VALIDITY**

**Proof** The proof trivially follows from Lemma 1 and Lemma 2 considering that there exists at least one process in $VN_j$ that does not leave its virtual node until an helper process completed the movement toward $VN_j$. $\square$ 

Note that, Lemma 3 guarantees that messages can flow from $VN_i$ to $VN_k$ (and vice-versa) despite churn. Intuitively the migration of the helper nodes toward the root virtual node (induced by the maintenance algorithm) “moves” the effect of churn only in the leaf VNs. Observing the structure from a virtual-node point of view it is possible to see that only leaves disappear, while non-leaf virtual nodes remain stable preserving virtual paths. As a consequence, the query can safely be diffused in the virtual tree collecting all of the contributions from nodes that remain in the tree for the entire query duration.

**Theorem 1.** Let $p_i$ be the process issuing a query $q$ at time $t$ and let $G = (V, E)$ be a VT graph at time $t$. If $G$ is always connected then $q$ terminates and satisfies the interval validity semantics.

**Proof** In the following we fist show that every query eventually terminates and its results satisfies interval validity.

**Termination.** The query termination is guaranteed by the fact that eventually, each process will be unblocked from the wait state. Due to the view maintenance mechanism, in fact, every process that left the system is eventually removed from the view of any other process and, due to the reliability of communication, every $\text{QUERY\_REPLY}(q, r(q))$ message will be eventually delivered.

**Interval Validity.** Let us suppose, by contradiction, that $G$ is connected, it is a VT graph, $q$ terminates but it does not satisfy interval validity semantics. If $q$ does not satisfy IV, it means that there exists at least one process $p_j$ that has been part of the system for the whole period of the query evaluation while its contribution has not been considered in the query result. Let $p_i$ be the process of the root virtual node that issued $q$ (i.e. $p_i \in VN_r$). If the contribution of $p_j \in VN_j$ has not been considered, two cases could have happened: (i) $p_j$ has never received the query or (ii) the contribution of $p_j$ was not received by $p_i$.

**Case 1:** $p_j$ never receives $q$. If $p_j$ does not receives $q$, it means that the $\text{query}(q)$ message has been lost while traveling from $p_i$ to $p_j$. Due to the assumptions, there always exists at least one path connecting $p_i$ and $p_j$. Considering the shortest path between two such nodes, two cases can happen: (i) the shortest path has length 1 or (ii) the shortest path has length greater than 1.

In the former case $p_j$ and $p_i$ are directly connected. Due to the reliability of the communication primitives, we have that any message sent from $p_i$ to $p_j$ will be eventually delivered. Thus, $p_j$ never receives the query if and only
if either $p_i$ or $p_j$ leave during the query execution. None of these two cases impact IV, thus leading to a contradiction.

In the latter case, there exists at least one virtual node $VN_x$ on the path between $VN_r$ and $VN_j$. Let us consider the generic case of three virtual nodes, namely $VN_1$, $VN_2$ and $VN_3$, belonging to the virtual path connecting $VN_r$ and $VN_j$. Let us recall that any message $m$ sent at time $t$ will be eventually delivered; as a consequence, the query message broadcasted at time $t$ from a node $u \in V_1$ will be eventually delivered to any node belonging to $V_2$, the message will be processed and forwarded to any node in $VN_3$ that will eventually receive the query. As a consequence, due to the fact that nodes can only move from children $VN$s to the father $VN$, i.e. $p_j$ can only reduce its distance to $p_i$, at any time the query moves along the virtual path always closer to its destination $p_i$, and thus this is true on all disjoint paths connecting two arbitrary nodes belonging to $VN_1$ and $VN_3$, respectively. Considering that the virtual path between $VN_r$ and $VN_j$ can always be decomposed in sub-paths of length 2, it is possible to iterate the reasoning above and find again a contradiction.

**Case 2:** $p_i$ never receives the contribution sent by $p_j$. This case trivially follows from the previous reasoning by substituting $p_i$ and $p_j$. \(\Box\)

### 5.4 Engineering Aspects

The algorithm introduced in Section 5.2 strongly relies on a view-maintenance algorithm able to keep consistent local views among processes belonging to the same $VN$. Considering the relaxed asynchronous model inherited from [20], view maintenance algorithms can ensure consistency as long as the number of processes in each view does not grow too much. Considering the topology imposed by the virtual tree graph, this limitation can be handled by properly defining the parameter $N_{max}$ used in the join procedure. Actually, each process maintains a limited set of views (one associated to the virtual node, one to the father and one for each virtual node child) whose size is bounded by $N_{max}$.

While the relaxed asynchronous model well fits the characteristics of application running in large-scale datacenter based on low latency local area networks, most large-scale systems are deployed in WAN environments where communication links cannot be assumed timely and reliable. In such scenarios, deterministic view-maintenance algorithms fail in maintaining multiple views coherently up-to-date while processes concurrently join and leave the groups [10, 26]. However, our solution can be adapted to work in a WAN-based large scale setting, by using probabilistic view-maintenance solutions [10]. Such distributed
protocols employ highly resilient probabilistic approaches (e.g. gossiping) to maintain connected groups. The coherence of local views in this case is guaranteed only in “long enough” stability periods characterized by a well behaving communication infrastructure and limited system dynamism. Note however that such systems always guarantee connectivity within the group with very high probability, even in strongly dynamic and unreliable scenarios. Connectivity within groups maintained in local views is used by our OMP to both handle the membership of VNs and maintain the connectivity among virtual nodes in the VT graph. As long as such connectivity is maintained the virtual tree OMP guarantees both the entire network connectivity and the virtual path stability, and thus the query processing algorithm provides interval valid results. For each test performed in the following section all of the query completed without disconnections in the structure were interval valid. Section 5.5 provides an experimental evaluation where our solution is stressed in dynamic settings that well go beyond the limits devised in Section 5.3.1 and is shown able to still maintain connectivity and provide interval valid results.

In addition, the view maintenance procedure is usually configured to run periodically and check the consistency of information stored in each local view. The update frequency is usually fixed and must be defined as a function of the maximum churn that the system expects to experience. This means that a careful evaluation of the expected churn rate must be done before starting the system and that the latter must be configured as to tolerate large churn surges even if churn rates will be mild on average for most of the system lifetime. In order to improve the efficiency of our solution we thus introduced a dynamic view-management frequency whose purpose is to dynamically adapt by locally monitoring at run time churn rates experienced in each single VN. The dynamic frequency adaptation works as follows: let $f$ be the current update period. After $f$ rounds, if the local view was not updated, then the new update period will be $f = f + 1$, otherwise it will be $f = f / 2$. In order to avoid too large update periods, that can easily lead to the loss a VN in case of a local churn surge, the value of $f$ is limited by an upper threshold defined as a configuration parameter.

Also the join procedure can be enhanced by applying an heuristic based on a locality principle (e.g. latency) to make sure that nodes participating to a same VN experience low reciprocal communication latencies. The rationale behind this choice is that locality helps the correct and efficient functioning of the view-maintenance algorithm used within the VN. In this case a joining node would be given multiple possible bootstrap nodes and will join the VN whose nodes are “closer” (locality-wise) to it.
5.5 Performance evaluation

In this section we provide a detailed evaluation of the Virtual Tree architecture through simulations. The simulations were performed in a multi-thread ad-hoc simulator. The usage of an ad-hoc simulator was dictated by the need of testing the proposed approach in both very large scale and dynamic settings.

The evaluation is focussed on three distinct aspects:

- **overlay network connectivity**: we tested the robustness of the virtual tree overlay network topology under different levels of churn. In this set of experiments we have highlighted the main effects of the algorithm parameters on the connectivity of the overlay network.

- **message overhead**: this set of experiments assesses the cost of the virtual tree overlay management protocol in terms of messages needed to maintain the topology.

- **comparison with alternative approaches**: this set of experiments compares the Virtual Tree architecture with other solutions with respect to their ability to provide interval valid results in dynamic settings.

Note that we do not present the results obtained for the valid queries because for each query completed without disconnection in the VT the result was interval valid. The graphs that we can propose for the interval validity result to be a copy of the ones obtained for the connectivity, thus we decided to avoid to show them because they do not provide any additional information.

5.5.1 General Settings

The Virtual Tree architecture has been implemented in a round-based simulator used to simulate concurrent process activities and message network transmission delays. Processes are equipped with a probabilistic view-maintenance algorithm. All the experiments start from an initial configuration with $N = 13500$ processes arranged in a complete VT graph (i.e. each virtual node, except the leaves, has the same number $k = 4$ of children\(^3\)). Each virtual node is initially populated with $N_{\text{max}}$ processes with values changing from test to test. When the simulation runs the VT graph is stressed with churn that lasts for the entire test duration (1000 rounds). All reported values are the result of 10 independent runs. Standard deviations are not shown as their values were always smaller than 5%.

\(^3\)Other tests have been performed for different values of $k$ without sensible differences in the results.
Churn is modeled as a continuous periodic triangle-shaped process: during the first half period, processes enter the system (growing phase), while processes are progressively removed during the second half period (shrinking phase). This churn model tries to reproduce the characteristic periodic oscillations observed in real large-scale dynamic distributed systems [90].

**Metrics and Parameters.** In the experiments we collected the following metrics:

(i) $\%$ correct tests representing the percentage of runs completed with a single connected component in the VT graph (i.e. no partitioning),
(ii) *virtual tree height* measuring the maximum path length between the root and leaf nodes in the VT graph,
(iii) *virtual node size* representing the average size of virtual nodes in the VT graph,
(iv) *message overhead* measuring the number of messages produced for the correct maintenance of the VT graph.

All the previous metrics have been evaluated by varying the following parameters:

(i) *virtual node size*: virtual node size is defined according to the lower threshold $N_{\text{min}}$ that triggers the migration of processes from the lower level of the VT graph and the upper threshold $N_{\text{max}}$ that imposes the forwarding of a join request to children VNs. Several configuration have been tested and each configuration is identified by a pair $VN(N_{\text{min}}, N_{\text{max}})$.

(ii) *churn level* $c \in [0, 1]$: it represents the percentage of processes that join/leave the system at each round.

During the experiments, we also collected measures about the percentage of interval valid queries. However, the results showed that only queries ran in graph experiencing disconnections lead to results violating interval validity. For this reason, we are going to show only pictures related to the connectivity evaluation.

### 5.5.2 Connectivity Evaluation

The first test checked graph connectivity at various churn rates for different settings of $N_{\text{min}}$ and $N_{\text{max}}$. Figure 5.1 shows how the overlay network connectivity is affected from churn under different virtual node size configurations.

The virtual tree topology shows a typical bimodal behaviour: connectivity remains stable at 100\% until a certain threshold for $c$ (that depends on $VN(N_{\text{min}}, N_{\text{max}})$) is met; from that point on, the VT graph connectivity quickly drops to 0.

Note that the threshold is way larger than the limit for deterministic connectivity as defined in Section 5.3.1; if we consider, for example, the curve
CHAPTER 5. OBTAINING A VALID QUERY BY USING VIRTUAL TREE

Figure 5.1: Graph connectivity vs. churn level for different configurations of $N_{\text{min}}$ and $N_{\text{max}}$.

$VN(4, 9)$, the amount of churn needed to disconnect the graph is $c = 10^{-2}$; conversely, with the same settings the deterministic threshold inferred by the formula proposed in Lemma 1 is about two orders of magnitude smaller.

Figure 5.2: Impact of $N_{\text{min}}$ and $N_{\text{max}}$ on graph connectivity.

In Figure 5.2 we have tried to separately evaluate the effect on the connectivity of the variation of the two parameters $N_{\text{min}}$ and $N_{\text{max}}$, for different churn rates. $N_{\text{min}}$ regulates the robustness of $VN$s with respect to out-churn while $N_{\text{max}}$ has no direct connection to the graph robustness as it only regulates the acceptance of new processes in “large” $VN$s. However, $N_{\text{max}}$ could have an indirect impact on the $VN$ resilience to out-churn surges, as it defines
how large a VN can grow. Therefore, we tested this parameter as well.

Figure 5.3: Virtual node sizes at different levels of the VT graph.

The three solid curves show the algorithm behaviour by varying $N_{\text{min}}$ only ($N_{\text{max}} = 25$): as expected they closely resemble the behaviour already reported in Figure 5.1. Conversely, the three dotted curves show the behaviour by varying $N_{\text{max}}$ only ($N_{\text{min}} = 5$): all three cases report almost identical performance. We can thus conclude that the connectivity performance can be controlled only by varying the $N_{\text{min}}$ parameter; this result doesn’t come as a surprise as Section ?? already showed how connectivity is dependent only on the churn level, the time needed by the protocol to attract processes from low level virtual nodes, and the $N_{\text{min}}$ parameter. We can thus conclude that the indirect impact of $N_{\text{max}}$, if present, is negligible.

Figure 5.4: Non leaf virtual node size vs virtual leaf size.
Figure 5.3 shows the ability of the Virtual Tree OMP to move the effect of churn toward the leaf virtual nodes. The plot reports the average virtual node size at different levels of the VT graph for several $N_{min}$ values with $c = 10^{-2}$. The plot shows how virtual nodes always maintain a stable size that is slightly larger than $N_{min}$; the only exception is represented by virtual nodes positioned at the lower levels of the tree that with high probability are leaves: these nodes cannot attract processes from children nodes, and are thus condemned to sizes that are way below the $N_{min}$ value.

The concept is also remarked by figure 5.4 in which we plot the average size of a non leaf virtual node versus the average size of a leaf virtual node. As the previous graph, the comparison shows that the managing procedures are able to constantly maintain the size of all the non-leaf virtual larger than $N_{min}$.

Figure 5.5 shows how the maximum tree height varies with churn. From the different curves it is possible to catch a general behaviour: as the values of $N_{min}$ and $N_{max}$ are increased, the maximum height tends to decrease; this is an obvious consequence of the larger amount of processes that fit in virtual nodes at the highest level of the tree. The curves for small $N_{min}$ values are truncated as experiments with larger churn levels reported disconnections in the graph. Curves for large $N_{min}$ values (i.e. $N_{min} \in \{8, 13, 15\}$) show a rather unexpected behaviour as the max height first increases as churn grows, then reaches a maximum and starts to decrease for high churn levels. The initial growing phase is a consequence of growing churn that tends to increase the average number of children VNs that are created in the VT graph. After this initial growth the churn start to be so intense that many nodes are not able to finish their join procedure before being removed from the graph; as a
consequence, the average number of nodes in the VT graph starts to shrink and its height shrinks as well.

### 5.5.3 Overhead Evaluation

The Virtual Tree architecture relies on the presence virtual nodes whose membership is maintained by a view-maintenance algorithm. The test reported in Figure 5.2 showed us that $N_{\text{min}}$ controls the resilience of VNs to out-churn. Conversely, $N_{\text{max}}$ is responsible of limiting the cost incurred in the VN maintenance: the larger $N_{\text{max}}$ is and the larger is the overhead imposed by the view-maintenance algorithm. However, at the same time, the larger the difference between $N_{\text{min}}$ and $N_{\text{max}}$ is and the higher the probability of fixing the erosion introduced by the out-churn on the VN only waiting for new joins will be. This is an important aspect, as every time a virtual node size falls below the $N_{\text{min}}$ threshold the OMP spends a large amount of messages for moving nodes from the children VNs to the father. In some sense the gap between $N_{\text{min}}$ and $N_{\text{max}}$ represents a buffer which prevents premature attraction procedures to happen.

![Figure 5.6: Impact of $N_{\text{max}}$ on overhead generated by the Virtual Tree OMP.](image)

Figure 5.6 reports the message overhead per process versus churn for different $N_{\text{max}}$ values and $N_{\text{min}} = 10$.

The message overhead, expressed as average number of help messages produced per nodes and per round, increases with churn as a consequence of the large process mobility among virtual nodes needed to keep non leaf virtual node populations above the $N_{\text{min}}$ threshold. The growth is larger for configurations with lower $N_{\text{max}} - N_{\text{min}}$ gaps. This behaviour is justified by the fact that the smaller is the delta, the larger is the probability to have
a VN with size smaller than $N_{min}$. The extreme case is represented by the configuration $vn(10, 10)$ that forwards all process joins toward the leaf virtual nodes and, for every leave, attracts a node from a children.

![Figure 5.7: View Update frequency: best of static vs dynamic.](image)

Figure 5.7 shows the effect of the dynamic update frequency for the local view-maintenance algorithm. For each level of churn included in the plot, we first evaluated the lower static update frequency needed to avoid disconnections\(^4\). Then we run the same test with the dynamic frequency mechanism.

The solution based on the dynamic frequency mechanism exhibits worse performance if compared with the best static setting with the only exception of churn rate $c = 10^{-1}$. This is due to the fact that the dynamic frequency grows linearly and decreases exponentially and thus assumes values that are often bit larger the perfect one (represented by the best static frequency). The trend is inverted in the case of churn rate $c = 10^{-1}$ because, for this leave/join rate churn is less uniform on different VN as a consequence of shorter update periods. In this case the dynamic update frequency mechanism is able, for example, to impose a larger update period $f = 2$ in VNs experiencing mild churn, while using higher frequencies in VNs where churn is stronger.

### 5.5.4 Performance comparison

Figure 5.8 reports an evaluation of the ability of different overlay networks to support aggregate query answering algorithms. This comparison provides the reader with a qualitative analysis that substantiate the motivation presented in the Introduction. We compared an overlay network generated by Cyclon

\(^4\)Note that this optimal frequency can be calculated only \textit{a posteriori}.\n
5.5. PERFORMANCE EVALUATION

Figure 5.8: Percentage of interval valid queries in a scenario with 1000 processes and low churn rate for different overlay networks.

[92], an overlay network designed as a forest of spanning trees, a random graph and a virtual tree overlay network.

**Cyclon** is a gossip-based protocols that maintains local views that closely represents uniform random samples of the system population. Graphs built by Cyclon show strong connectivity and low diameters even when perturbed with strong churn. These characteristics are provided by means of a periodic shuffle of local views that tends to evict dangling links caused by out churn. As a query answering protocol we implemented a simple flooding-based techniques that naively propagate the query and the corresponding answer through all the reachable processes.

**Forest of spanning trees**\(^5\) (multiple tree) where the query is propagated in a parallel way (all the trees are exploited at the same time for query propagation and answers collection). This setup resembles the Virtual Tree query answering protocol with the only difference being represented by the lack of a OMP protocol able to repair damages occurring to trees due to churn.

**Random graph** is a topology known for its strong connectivity and low diameter. Strong connectivity lets random graphs sustain massive node failures with extremely low probabilities of disconnection. However, without specific algorithms, they are not able to repair damages caused by churn. In this case we employed the same query answering algorithm considered in the Cyclon case.

The topologies considered in Figure 5.8 are composed by 1000 processes. In each run we have injected churn and we tried to execute 100 queries. All the topologies have been rebuilt from scratch before any single query execution.

\(^5\)note that each DHT [89],[81] can be represented by a forest of spanning trees
CHAPTER 5. OBTAINING A VALID QUERY BY USING VIRTUAL TREE

Concerning Cyclon, we let the protocol execute 50 view shuffles before executing the query to let it reach a steady state. For the multiple trees case we built a forest formed by 10 spanning trees. In all the three cases we configured the graphs such that, on average, nodes always had the same degree.

Figure 5.8 reports the percentage of queries that ended with an interval valid answer for the three solutions in a scenario with $c = 0.005$.

The overlay network generated by Cyclon completes a very few Interval Valid query answering. Such bad performance is caused by the shuffling techniques employed by Cyclon to refresh local views. This shuffle mechanism continuously changes the paths between two processes on the overlay networks making the paths extremely unstable. Therefore, the probability that a querying node gets all the answers from nodes that remain in the system for the whole query duration is extremely low and close to 0.

The solution based on multiple trees overlay network is able to provide interval valid query answers with low probability. This performance is caused by the absence of any mechanism able to counteract the disruption of trees due to churn. Therefore this overlay network disconnects quickly. On the other hand if a tree, by chance, is not disconnected by churn during the time of the query, then the returned answer is interval valid.

The random graph-based solution exhibits good performance: it achieves Interval valid querying answers with high probability. This is due to the robustness of the random graph structure with respect to churn and to the huge number of redundant paths between any two processes. This provide a good degree of stability to paths during the time of the query. Only tests performed with larger populations (up to 10,000) show that also the performance of the Random Graph get worse. This result stem from the fact that the larger is the population and the larger is the query execution time: with larger query execution time the churn has more time to disconnect the network corrupting the query result.

Figure 5.9 compares the cost of executing interval valid queries. To the best of our knowledge WildFire [20] is the only other protocol able to ensure valid results. More precisely WildFire was introduced by Bawa et al. as an algorithm able to provide single-site valid results. Note that single site validity semantics is weaker than interval validity as it only guarantees that the result of the query contains at least contributions from all the processes that were connected through a stable path toward the querying node for the query duration. WildFire consists of two phases: the broadcast phase and the convergecast phase. In the broadcast phase the query is forwarded inside the network using a broadcast primitive; the broadcast propagation creates multiple spanning trees that are used during the convergecast phase. Each process, receiving the broadcast, starts its convergecast phase sending back its contribution toward the root of the spanning trees. In our tests we reproduced
within a VT graph the same settings proposed in [20]: variable number of processes \(N = [2^{11}, 2^{12}, 2^{13}, 2^{14}]\), and continuous churn (\(c = 10^{-2}\)).

The result shows that a query performed in Virtual Tree uses, on average, \(1/4\) of the messages of a WildFire query. This difference stems directly from the approaches adopted by the two different solutions: while WildFire builds the structure needed to disseminate the query and collects its results \textit{on-demand}, i.e. when a query is started, our solution continuously maintains the VT graph and thus incurs a reduced overhead during the query execution. The strategy adopted by WildFire clearly pays back in all those situations where queries are executed not so frequently.

The last comparison that is proposed is related to the message overhead imposed to keep the overlay connected. In this test Virtual Tree is compared with other famous solution like Cyclon [92] and Chord [57, 89] (the most famous DHT implementation) that are already introduced in this section. As remarked in the previous chapter Virtual Tree exploit the reliability of the virtual node in order to meet the connectivity exhibited by the unstructured overlay (such as Cyclon) and the facilities achievable by a structured solution. This complex virtual overlay shows management costs higher than other solution, this is due to the management of the multiple local views that Virtual Tree has to handle; the dynamic adaptation proposed in 5.4 and analysed in 5.5.3 is able to sensibly reduce the number of the messages used to maintain the overlay, in fact it consumes just the double of the messages of a DHT but it allows to maintain the connectivity for churn rates order of magnitude higher than a DHT [79].

Figure 5.10 shows the number of the messages used to maintain the connectivity in the structure in the case of \(c = 0.001\). In order to have a fair
comparison we chose the highest churn rate admissible by Chord DHT that
do not compromise the performance. For coherence with the previous graphs
in the figure the DHT is called “Multiple Tree”. The overhead imposed by
Virtual Tree is mush higher than Cyclon, this is quite predictable, in fact
unstructured topologies are characterized by low maintenance costs and high
connectivity, the update frequency requested to a Cyclon is in fact very lim-
ited if compared with the structured solutions. Structured solution has an
high management costs due to their complex structures but on the other hand
they offer facilities that structured topology cannot provide. The tests put in
evidence that Virtual Tree is not a convenient for low churn rate, so if there
are no needs to have valid query it is preferable to use other solutions such as
a DHT, at the opposite for higher churn rate Virtual Tree is a very good - and
probably the only - way for having the facilities of a structured network.
Chapter 6

Large scale distributed computations with volunteer computing

“Computer time-sharing technology might result in a future in which computing power and even specific applications could be sold through the utility business model, like water or electricity”

Prof. John McCarthy @ MIT Centennial Celebration (1961)

In the previous chapters it is shown the way to obtain a valid query in a dynamic environment by using a virtual tree structure. In this chapter we detail a practical application of the concept previously introduced in the field of cloud/grid computing. The possible applications of a querying system for a distributed environment are very large: every time we need to synchronize some replicas, collecting aggregated statistic about the system or looking for a specific resource, are just few examples of operations that require a system-wide query. The more accurate and faster is the query, the more efficient will be the whole system. The previous chapter extensively details the performance of Virtual Tree also comparing its performance to other protocols. This chapter shows how Virtual Tree can be used as a fundamental block of another system
and why it should be preferred to other protocols.

In the last five years a relevant part of the research in distributed computing moves toward this direction, MapReduce is “a programming model and an associated implementation for processing and generating large datasets that is amenable to a broad variety of real-world tasks”. MapReduce paradigm consists of two steps (also called functions in [36]): the map and the reduce. In the map step the data are adapted in a standard key-value form, then in the reduce step the data are elaborated and aggregated in order to compute the final result. The computation is orchestrated by an autonomous sub-system that parallelizes the computation in a large cluster.

The research effort related to MapReduce encourages the researchers to deal with a new challenge: moving a MapReduce-like framework toward a dynamic distributed environment. The challenge is motivated by several reasons, mainly: the cost of a centralized managed approach and the energy requirement [49]. Huge service providers such Google or Facebook are looking for geographic scale solutions with the aim of reducing the costs and the energy consumption of their clusters [88]. The extreme solution of geo-scaling is represented by a distributed dynamic network; the first example of that was SETI@Home project [7] that historically represents the proof of the potential of this kind of solutions [8]. MapReduce is designed to run in a centralized cluster with a single orchestrator (i.e. the job scheduler); it is able to deal with runtime-faults it is not suitable for a dynamic distributed system due to:

- concurrency: in a distributed dynamic environment there isn’t a centralized scheduler, it is in contrast with the concept of distributed, then every machine potentially represents a job scheduler. MapReduce does not allow multiple schedulers;

- resource sharing: a MapReduce instance does not allow to share resources with another one. In a distributed environment the resources are shared among all of the job schedulers and they have to negotiate them;

- dynamism: a dynamic distributed system is always affected by the churn phenomena, the available resource amount continuously evolves\(^1\). The MapReduce scheduler deals with a well-known resource set and not with a variable one, this represents a great issue because it forces to re-design the entire scheduler architecture that is the core of MapReduce;

- heterogeneity: the resources managed by MapReduce (CPU, memory, disk capability and so on) are quite homogeneous in the cluster; in a

\(^1\)in terms of CPUs, memory, and network capabilities
In order to achieve the goal of having a parallel computing instance running on a distributed dynamic system the ideas proposed by J. Dean et al. in [36] should be re-engineered. In 2006 O. Babaoglu et al. [11] sketch the driving directions for a cloud-computing system able to run over a multi-tenant\textsuperscript{2} distributed dynamic system. Starting from 2008 several researchers are moving in this direction proposing architectures and possible solutions [94] [78] [12] [34].

In this chapter we introduce Cloud to Peer: a MapReduce [36] like parallel computing system. Cloud to Peer follows the idea introduced by MapReduce of proposing a fixed schema for the data and the computation but it differs from it because it offers more flexibility in the data schema (it can change for every computation). Goal of Cloud to Peer is to perform all of the operations on-line by avoiding (as much as possible) the usage of the disk. This could be possible by exploiting the large number of nodes that compose the system; moreover the choice of data schema is free: the programmer must select the best one for its own algorithms optimizing them for saving disk space and memory.

Cloud to Peer exploits the capabilities of Virtual Tree in order to address the issues of concurrency and dynamism proposing a new architecture for distributed computing in a decentralized environment.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figures/figure6_1.png}
\caption{Number of collected resources}
\end{figure}

\textsuperscript{2}previous solutions, like SETI@home, do not allow multi-tenancy
Virtual Tree exhibits good performance in churn management and query handling, these two characteristics are perfectly suitable for a distributed computing platform because they (i) prevent the system from disconnection ensuring the reachability of each resource and (ii) the query answering system provides a powerful and easy way to map continuously the system resources. Moreover, the capability of offering a valid query in presence of churn represents an important enhancement for the system. In order to run a fast computation the scheduler has to look at the best resources suitable for the specific computation. Having a valid query the scheduler can look for them assuming that result gathered by the query surely includes all of the resources currently present in the network (due to interval validity), so it can compute a finer resource selection. To have an idea about the order of magnitude of the resources missed by a non valid query we show in figure 6.1 the number of results that are collected by a valid query in Virtual Tree and a non valid query. The non valid query are performed in Cyclon and in a multiple-tree architecture (such as a dht), protocols already used in section 5.5.4. The tests run with a population of 8.192 nodes in presence of churn \(c = 10^{-2}\) and are tuned in order to have in Cyclon and in multiple-tree the same number of messages used for a valid query in Virtual Tree (see figure 5.9 for further details). The query algorithm used in Cyclon is the flooding and in the multiple tree is the simple broadcast-convergecast (see 3.1). In the presented setting the flooding implemented in Cyclon loss about the 13% of the resources and the broadcast/convergecast implemented in the multiple-tree loss about the 25% of them. The valid queries in Virtual Tree introduce a sensible difference in the gathered results and for this reason they are preferable for this application.

The chapter presents the architecture and the implementation of Cloud to Peer, providing also a comparative evaluation between Cloud to Peer and MapReduce using some famous benchmark tests like Grep and String Sort.

6.1 Architecture

A cloud computing environment is defined as a set of technologies deployed to enable the access to a shared pool of computing resources (networks, servers, storage, application services) that can be provisioned and released with minimal management effort or service provider interaction [73]. In a dynamic environment the set of available resources changes during the time due to the assigned resources and the churn; then, the cloud system must be able to: (i) receive new resources from the in-churn, (ii) remove the resources no longer available due to out-churn, (iii) assign the available resources to the client.
applications, (iv) substitute the faulty\textsuperscript{3} resources at runtime.

The fundamental services of a cloud computing system designed for dynamic environment are\cite{11}:

- Slicing: the slicing service is responsible for assigning the right proportion of resources (so called subclouds) to applications, taking care of other possible requirements about the nodes that are offering the resources. The slicing service accomplishes the functions (iii) and (iv).

- Bootstrapping \cite{59}: it is responsible for starting up an application from scratch, for example a new node that wants to share its resources must use this service in order to get connection to the network. The bootstrap service accomplishes the function (i).

- Churn handling: it is responsible for assisting the application in handling out-churn and failures. It accomplishes the function (ii).

The core of a cloud system is represented by the Slicing service; the Bootstrapping and the Churn Handling services support the Slicing in order to keep up to date the available resource set. A large part of distributed protocols already address the problem of continuous join (bootstrap service) and leave (churn handling) of nodes from the system, each protocol designed to run in a dynamic environment must be able to provide the two services; the research in peer to peer network provides a huge amount of examples of these protocols, they are usually called overlay networks. From a general point of view a distributed dynamic cloud system can be seen as a Slicing Service mounted on top of an overlay network; to this end the facilities exposed by the overlay network play a key role in the implementation of the slicing service. The Slicing service is independent by the architecture implemented by the overlay network; in the cloud terminology it is possible to state that the overlay network is seen from the slicing as a cloud system in turn. The Slicing requires a unique service from the overlay: the possibility of collecting the available resources. Once the resource set is continuously available and updated the slicing service can infer the fair association of them to the applications. Each node of the network represents a resource of the cloud system; the overlay network should be able to provide the whole list of the available resources specifying their the capabilities in terms of CPUs, memory, storage and network connection. We name this list the Remote Resource Catalogue [RRC].

Once the RRC is available the Slicing can provide the resources to the client of the cloud system. The application scenario that we want to achieve is the following:

\textsuperscript{3}due to out-churn or real faults
1. the user wants to run an operation in a given dataset (the user is both
   the owner of the program and the dataset);

2. the user contacts a node connected to the cloud system (in the following
   access point) in order to execute the operation in the cloud. The user
   has to specify the size $n$ of the required sub-cloud;

3. the access point, receiving the request of the user, accesses the Remote
   Resource Catalogue and selects the best node set of size $n$ accordingly
   to the program and the dataset constraints. It contacts the nodes in the
   set one-by-one asking about their availability to join to its sub-cloud.
   If a node (or more) refuses then the access point substitutes this node
   with a new one available within the RRC (a node can refuse the requests
   because it is already involved in another computation)

4. once the access point has collected $n$ nodes (in the following slaves) it
   distributes the dataset and the code among them

5. the slaves run the code on the assigned piece of the dataset and then
   they send back the outcome to the access point

6. the access point has to (i) substitute the faulty slave(s) during the exe-
   cution and (ii) collect and aggregate the slaves results

7. once the final result is available at the access point it is given back to
   the user

Figure 6.2 sketches the described application scenario. In a distributed
environment every node potentially represents an access point, then, every
node must be able to handle the slicing service.

The slicing service have to: (i) build up the sub-cloud, (ii) distribute the
input data to the slave nodes, (iii) distribute the code to the slave nodes, (iv)
verify the liveness of the slaves, (v) substitute the faulty slaves, (vi) collect the
final result. The five operations identify two separate components of the ar-
chitecture of the slicing service: the first one is responsible of the construction
of the sub-cloud and of the distribution/retrieval of the data and the results;
the latter is responsible of monitoring the evolution of the computation. We
call this two components the Job Manager and the Job Tracker with reference
to the MapReduce name style. The resultant architecture is shown in figure
6.3.
6.2 Job Manager and Job Tracker

Having the RRC service the design of the Job Manager and the Job Tracker is an easy work. In order to show that in this section we will assume that the RRC is available, then in the next section we present how the RRC can be implemented.

In this section we present the algorithms implementing the functionalities shown in table 6.1.

<table>
<thead>
<tr>
<th>Job Manager</th>
<th>Job Tracker</th>
</tr>
</thead>
<tbody>
<tr>
<td>build up the sub-cloud</td>
<td>verify the liveness of the slaves</td>
</tr>
<tr>
<td>distribute the input data to the slave nodes</td>
<td>substitute the faulty slaves</td>
</tr>
<tr>
<td>distribute the code to the slave nodes</td>
<td></td>
</tr>
<tr>
<td>collect the final result</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Job Manager and Job Tracker functionalities
The first block that we analyze is the Job Manager. Job Manager receives the code and the dataset as a user input; accordingly to the user request it starts to build up the sub-cloud, then it distributes the data and the code to the slaves in the sub-cloud. In the end, receiving the partial results from the slaves, the Job Manager aggregates them in order to obtain the final result.

The sub-cloud building is performed by collecting a set of $n$ nodes from the RRC. The parameter $n$ is defined by the user. Nodes in the RRC can be selected accordingly to some specific criteria (e.g.: prefering nodes with a large memory size) or at random if no constraints are required. In this work we want to put the focus on the architecture, to this end the details of the selection criteria will be not discussed; in the algorithm we refer to a generic function `selectFromRRC` that will abstract the selection policies. The nodes are contacted one-by-one by using the `joinSubCloudRequest` function. A node, receiving the request, evaluates its current workload replying with an ACK or NACK message. In [65] the authors allow to use queues for the jobs to be run; a dynamic distributed environment can be composed by hundred of thousands nodes, the probability of having no nodes free of charge is very limited; we suggest to always look for the nodes that can immediately run the job avoiding the queues. The only exception is represented by data locality policies (mainly in the case of large input) because data locality severely affects the execution time [95]. The procedure conclude in the case of $|ACK| = n$. Algorithm 1 detail the pseudo-code.

The reader is invited to note that the sub-cloud building procedure solves...
Algorithm 1: Sub-cloud building procedure

**Data:** the size of the desired sub-cloud \( n \)

**Result:** a collection of nodes \( sc \) representing the sub-cloud

1. \( sc = \emptyset \);
2. while \( |sc| < n \) do
   3. node = `selectFromRRC`();
   4. send(`JOINSubCloudRequest`, node);
   5. \( t = `setTimeout`(); \)
   6. wait until: `JOINSubCloudReply` != received and \( t > 0 \);
   7. if `JOINSubCloudReply` == `ack` then
      8. \( sc = sc \cup node \);
3. return \( sc \);

concurrency issues; in a centralized approach (like MapReduce) the scheduler is responsible for the resource tracking and it has to manage the correct assignment avoiding node overload. In this case the nodes autonomously decide to join or not in the sub-cloud. This approach completely solves the problem of concurrency among several schedulers: to each request is assigned a new sub-cloud exploiting the large availability of the nodes that usually characterizes the distributed dynamic systems.

The data/code distribution in the sub-cloud is performed by copying them into each node. This is a standard network based data transfer operation with the only trick of implementing an optional partitioning of the input. The aim of the distributed computing is to parallelize the code execution among the machines, to this end a partial chunk of data is assigned to each machine; the result of this operation is that not the whole dataset is copied to each node but only the assigned chunk in order to reduce the network transfer cost. The divide operation is defined “optional” because sometimes, in order to compute the partial results, the entire dataset is necessary. The possibility of partitioning must be declared by the user. The code of the data/code distribution is detailed in algorithm 2.

The last operation performed by the Job Manager is the collection of the partial results and the computation of the final result. In order to implement this operation the job manager waits until all the partial results are received, then it starts to compute the final result. The computation of the final result
CHAPTER 6. LARGE SCALE DISTRIBUTED COMPUTATIONS WITH VOLUNTEER COMPUTING

Algorithm 2: Data/Code distribution procedure

Data: the dataset DATA, the algorithm to perform CODE, the sub-cloud sc, a flag partitionable that states if the DATA can be divided in chunks

1. foreach node\_i \in sc do
   2. if partitionable then
      3. data\_chunk = get\_Chunk(DATA, i);
      4. send(data\_chunk, node\_i);
   5. else
      6. send(DATA, node\_i);
   7. send(CODE, node\_i);

is function of the way the partial results are computed, for this reason the code for computing the final result must be provided by the user\(^4\); we call it aggregation code. The code for this operation is shown in algorithm 3.

Algorithm 3: Computation of the final result procedure

Data: the aggregation code agCODE, the partial result set pRes, the sub-cloud sc

1. wait until: |pRes| < |sc|;
2. return exec(agCODE, pRes);

In MapReduce the data transfer is hidden by HDFS [46] [85]: a distributed file system introduced by Google that connects all the machines of the cloud system. In a distributed environment it is impossible to use a distributed file system abstraction due to the size and the churn of the network. The fault rate (equivalent to the out-churn) in a managed cluster is about 2-3 nodes every 1000 per day [86], a dynamic environment exhibits the same out-churn per second. Moreover, the size of the network can be much higher than hundreds of thousands of node, these numbers prevent the application of HDFS (or others) due to its limits in terms of scalability and overhead of the name

\(^4\)it is equivalent to the reduce function of MapReduce
space of the system [86]. In a dynamic environment the HDFS abstraction is substituted by algorithms 2 and 3 that move the data during the computation. The impossibility of using an HDFS-like system clearly imposes a limitations; for example a distributed file system introduces the possibility of replication with great benefits for deterministic data locality: in a distributed dynamic environment replication is not impossible (e.g. [45]) but it is probabilistic due to the dynamism.

**Job Tracker**

The Job Tracker is the responsible of the liveness of the computation. The aim of this component is to ensure that eventually the final result will be carried out. In order to accomplish this goal, the Job Tracker has to verify both the status of the computation and the node. During the computation the Job Tracker continuously monitors the nodes in the sub-cloud asking about the state of completion of their computation. The Job Tracker can decide to substitute the node(s) in three cases:

1. the node does not reply to heartbeat messages
2. the computation state does not evolve since the last heartbeat message
3. the computation evolution is drastically slower than the other nodes in the sub-clouds

in the cases 1 and 2 the node is substituted with a new one; in case 3 the node can be substituted with the fastest one in the sub-cloud at the end of its computation or, as an alternative, with a new one. If the computation is at the beginning the node will be substituted with a new one, otherwise with the fastest one in the sub-cloud. A fair evaluation about the computation evolution can be very hard. For example a too severe evaluation about the speed of the nodes (see the third bullet) can be very risky: it exposes the system to continuous substitution of the nodes in the sub-cloud with negative impact on the global performance. If the designer has a low degree of confidence with the environment in which the cloud is developed, we suggest to prefer a conservative approach or not to consider the third substitution case at all. Note that with computation we mean both the data transfer time and the elaboration time.

The algorithm for implementing the Job Tracker can be seen as fault detector [27] [25] that takes care about the health of the nodes; in case of bad health it provides to the substitution of them. The Job Tracker procedure is triggered at the end of the Sub-cloud building procedure (see Algorithm 1), the algorithm stores the computation evolution of each node within a status array:
for each node it sends a statusRequest, once the statusReply message is received (or the timeout is elapsed), the algorithm applies the three policies previously described. In the code proposed in Algorithm 4, the policies 2-3 are evaluated by the function evaluateStatus while the first policy is explicitly managed. The evaluateStatus function proposes as return value true if and only if both policies 2-3 are satisfied. Once the node is candidate to be substituted with a new one, the Job Tracker procedure executes the selection and the data/code distribution operations. These operations are already analysed in algorithms 1 and 2.

6.3 Implementing the cloud with Virtual Tree

In section 6.1 we show the algorithm for building a basic cloud computing system over an overlay network able to provide the abstraction of the Remote Resource Catalogue. RRC plays a fundamental role in the implementation of the system because it is the repository of the available resources. The RRC has to be accessible by each node and it has to reflect the current status of the system with negligible delays. Section 6.5 will show that in a managed environment the RRC-like services are always implemented in a centralized manner; at the opposite, the systems designed for unmanaged environment emulate the RRC service with some lacks. Moreover, the performance exhibited by these protocols are affected by the presence of the churn: in the case of structured overlay networks it is possible to map the available resources only in the case of low churn rate; at the opposite the solutions that use of unstructured overlays are resilient to the negative effects introduced by the churn, but they don’t provide the same capability of resource mapping of the others. The optimal solution is represented by the resilience of the unstructured overlay coupled with the possibility of having a resource mapping useful for performing the job assignment.

In the managed environment all the machines of the cloud continuously notify their resource availability to the centralized RRC service, then the RRC service exposes an interface toward the scheduler. In order to implement a remote resource catalogue in a dynamic distributed system a structured network is mandatory: having a structure it is possible to design an aggregation/distribution schema in order to collect, aggregate and continuously distribute the resource list. The only way to do that in an unstructured environment is represented by the broadcast, practically infeasible for a large network. An aggregation/distribution schema is also a key feature for reducing the update delay in the RRCs; for example, in a DHT it is possible to deliver the updated catalogue in less than O(log n) hops, on the contrary, even neglecting the costs, the broadcast distribution delay is (i) unpredictable.
Algorithm 4: Job Tracker procedure

Data: $DATA$, $partitionable$, $CODE$, the sub-cloud $sc$, the computation status array $statusArray = [0\%...0\%]

1. forever: foreach $node_i \in sc$ do
   2. send($statusRequest$, $node_i$); $t$ = set.Timeout;
   3. wait until: $statusReply != received$ and $t > 0$;
   4. if $statusReply == received$ and evaluateStatus($statusReply$, $status_i$) then
      5. $status_i = statusReply$;
   6. else
      7. $n = |sc|$; $sc = sc/node_i$;
      8. while $|sc| < n$ do
         9. newNode = selectFromRRC();
         10. send($joinSubCloudRequest$, newNode);
         11. $t$ = set.Timeout;
         12. wait until: $joinSubCloudReply != received$ and $t > 0$;
         13. if $joinSubCloudReply == ack$ then
            14. $sc = sc \cup newNode$;
      15. if $partitionable$ then
         16. dataChunk = getChunk($DATA$, $i$);
         17. send(dataChunk, newNode);
      18. else
         19. send($DATA$, newNode);
      20. send($CODE$, newNode);

and (ii) it could not ensure the delivery of the catalogue to every node (see section 5.5.4).
CHAPTER 6. LARGE SCALE DISTRIBUTED COMPUTATIONS WITH VOLUNTEER COMPUTING

74

Figure 6.4: The RRC aggregation-distribution schema in Virtual Tree

Virtual Tree perfectly matches all of the requirements for implementing the RRC:

- it exhibits a churn resistance close to the unstructured overlays
- it offers a tree-shaped ready-to-use topology for implementing the aggregation/distribution schema and, in the case of balanced tree, the distance root - leaves is shorter than the one experienced in a DHT. In general Virtual Tree proposes paths shorter then a tree composed by physical node\(^5\) because they are aggregated in the vn: the height of a tree of \(n\) nodes is \(\log(n)\), the heigh of a Virtual Tree is \(\log(n/|vn|) \leq \log(n)\). This feature provides an aggregation/distribution schema even faster than a DHT. The reader is also invited to note that if the update delay does not satisfy the requirements, it is always possible to rise up the branch factor of the VT in order to sensibly reduce the path length.

The RRC can be implemented in a Virtual Tree with very few modifications in the querying algorithm. The modifications consist in the introduction of a special query named RRC\textsc{distribution}. The query does not contain any request: the content of the query is simply represented by the updated RRC. The nodes, receiving the query, store its content as a new catalogue and sends back their current resource status as a reply to a normal query. Figure 6.4 sketches the continuous RRC distribution-aggregation. The root of the Virtual Tree, receiving the aggregated results, computes the new catalogue and starts again to distribute it into the whole system through the RRC\textsc{distribution} message, triggering this way a new aggregation procedure. At the first start-up of the system the bootstrap service forces the distribution of an empty catalogue that starts the lifelong aggregation-distribution of the RRC. Thanks to the introduction of the RRC\textsc{distribution} query Virtual Tree ensures the availability of the RRC into all of the nodes with low update delay.

\(^5\) a DHT can be always seen as a forest of trees
The rest of the system is implemented by following the algorithm presented in section 6.1. The following subsection details some implementation improvements that we have implemented with the aims of limiting the overhead imposed by the RRC distribution and improving the performance of the sub-clouds.

6.3.1 Improvements

In this section we detail four possible modifications that can be applied to the methodology introduced in the previous sections with the aim of improving the performance of the overall cloud distributed dynamic system. The three improvements are summarized in the following bullets:

- RRC differential update in place of full replacement
- distributed aggregation of the final result in place of centralized aggregation at the access point
- non-uniform DATA partitioning among the sub-cloud nodes

The first improvement consists of the implementation of the differential update of the RRC. The RRC is composed by a large list of nodes with their available resources. The RRC is continuously distributed in the system; the procedure presented in the main section suggests to substitute the entire list with the new one. This approach forces to substitute (and then moves in the network a larger amount of data) the entire RRC even if small or absent changes have to be performed. In order to reduce the amount of data that the system must aggregate and distribute, the node replies to the RRC distribution query with only the real resource changes. If no changes occur from the last update then the node will reply with a special no-change message. Due to this modification the RRC distribution message will only contain: (i) information about the new nodes in the system, (ii) information about nodes to be removed and (iii) information about the resource changes. If a node is not listed in the RRC distribution message then it has not to be modified in the RRC. In the case of joins, during the entering procedure the nodes will be equipped with a new full RRC.

The second improvement can strongly enhance the performance of the entire system. In the proposed aggregation schema (see Algorithm 3) the data are sent to the nodes of the sub-cloud and then the partial results are given back to the access point that provides the computation of the final result. The aggregation of the final result can be heavy if computed by a single node such as the access point. For this reason we introduce the possibility to aggregate the final result in parallel per steps by using the nodes in the sub-clouds
Figure 6.5: Parallel aggregation of the partial results in Cloud to Peer Architecture

that has completed their computation. Figure 6.5 shows how the parallel aggregation works: node $b$ sends its partial result to node $a$ that merges it with its own one; the outcome is sent to the access point. Node $a$ is called partial aggregator. Nodes $c$ and $d$ do the same. The merged partial results are successively sent to the access point that provides to compute the final result. The aggregation topology followed by the partial results is decided at the beginning of the computation. The Job Tracker must be aware about the aggregation architecture in order to ensure the correct substitution of nodes in case of faults.

The third improvement impacts the implementation of the Algorithm 2 for the data distribution. The nodes composing the sub-cloud are chosen with common criteria from the RRC; during the computation the performance of some nodes can vary, then it can be useful to modulate on-line the load associated to each node. In the proposed implementation the data that need to be elaborated are uniformly divided in chunks by the access point; this strategy imposes to all the nodes the same load without the possibility of changes. In order to modulate on-line the load (i) the data must be divided in $k$ chunks such that $k \gg |sub-cloud|$ and (ii) the responsibility of the association of the chunks must be moved from the access point toward the nodes of the sub-cloud: every node asks for a new chunk to elaborate immediately after the completion of the previous one. This improvement allows the faster nodes to accomplish the work of the nodes that experience an unpredictable loss of performance. Once the chunk is elaborated, it is immediately sent to its partial aggregator that will do the rest.
6.4 Performance evaluation

In this section we evaluate the performance of Cloud to Peer with respect to Hadoop [56]: a standard implementation of MapReduce. Aim of the comparison is to evaluate if the large availability of nodes of a distributed dynamic environment can compensate the performance of a centralized cluster running a standard cloud system for managed environments such MapReduce.

The developed prototype of Cloud to Peer used for the comparison follows the algorithm previously presented and also implements the first two improvements proposed in section 6.3.1.

The comparison is performed as follows: the system is stressed with some common benchmark operations (grep, word count and string sort) with different characterizations. The performance of MapReduce are measured only in the case of centralized cluster\(^6\), at the opposite the performance of Cloud to Peer are evaluated in three separate test beds: (i) the same centralized cluster used for MapReduce, (ii) a set of office machines and laptops connected with a standard Gigabit Ethernet and (iii) an internet cluster emulated through PlanetLab. For further details please refer to table 6.2. In all of the tests the only metric used to evaluate the performance is the throughput expressed in MB/s.

An important point to remark is that test bed number (ii) is developed inside our department. Cloud to Peer was installed inside the machines of professors, researchers, students and administrative staff, that run their normal activity during the execution of the tests. This represents a real application scenario for a dynamic distributed cloud environment, the results obtained in this test bed are of particular interest.

<table>
<thead>
<tr>
<th>#</th>
<th>test bed</th>
<th>OS</th>
<th>Connection</th>
<th>Core</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Centralized Cluster</td>
<td>Linux</td>
<td>10 Gigabit</td>
<td>4</td>
<td>2 Gb</td>
</tr>
<tr>
<td>2.</td>
<td>Office Network</td>
<td>Windows, Linux, OS X</td>
<td>1 Gigabit</td>
<td>1 to 4</td>
<td>1 to 4 Gb</td>
</tr>
<tr>
<td>3.</td>
<td>Planetlab</td>
<td>Linux</td>
<td>Internet</td>
<td>1 to 8</td>
<td>1 to 16 Gb</td>
</tr>
</tbody>
</table>

Table 6.2: Cloud test beds

The benchmark operations used are: Grep, Word Count, String Sort. The characterization of the three benchmarks is reported in table 6.3.

Excluding the String Sort, the benchmarks can be executed on-line (i.e. the computation of the partial result starts and goes on together with the

---

\(^6\)the cluster is the same used in [9]
Table 6.3: Cloud Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time Complexity</th>
<th>input size</th>
<th>output size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grep</td>
<td>$O(n)$</td>
<td>large</td>
<td>very small?</td>
</tr>
<tr>
<td>Word Count</td>
<td>$O(n \log n)$</td>
<td>large</td>
<td>medium</td>
</tr>
<tr>
<td>String Sort</td>
<td>$O(n \log n)$</td>
<td>large</td>
<td>large</td>
</tr>
</tbody>
</table>

reception of the data). The main difference among the benchmarks relies on the output size: since the main bottleneck of a cloud system is represented by the network costs [95], it results that the more efficient is the network management, the better will be the performance. Scope of these tests is not to evaluate the algorithms, we want to evaluate the cloud engines; this is why we use very common and easy algorithms.

The benchmark input is represented by a text file of 40 Gb obtained by a composition of several Free eBooks by Project Gutenberg [77].

The presented results are the average of ten independent runs with:

- no fault in the case of the MapReduce Cluster
- at most one fault in the case of office machines
- at most 2 faults in the case of internet cluster

The imposed fault limits are evaluated in function of real churn levels experienced in the three networks (managed environment, LAN and Internet) in a time window equivalent to the test duration.

Grep test

The first test we run is represented by the Grep. Grep is an utility for searching plain-text data sets for lines matching a regular expression. The parallel implementation is very easy: the dataset is divided into $n$ chunks where $n$ is the size of the sub-cloud. A chunk of data is assigned to each node in the sub-cloud: the nodes scan the assigned chunks line by line looking for matching sub-strings. The outcome of is represented by a file containing all of the matching strings. At the end of the computation the partial results are sent to the access point that provides to merge them together computing the final result. MapReduce, by its architecture, must first translate the dataset into an equivalent key-value storage and then it computes the aggregation of the partial results. This operation imposes to write in the partial results a string for each line of the document: at each line in the input file (keys) is associated
6.4. PERFORMANCE EVALUATION

A boolean variable (value) sets to 1 if the line matches the regular expression, otherwise it is sets to zero. In the reduce phase only the values with an associated key set to one are written into the final result. The reader can imagine that both the lines not matching the regular expression and the lines that succeed the operation are written into the partial results. This imposes a large amount of useless I/O. The computational complexity of the two algorithms is the same but the "brute force" I/O imposed by MapReduce limits the throughput with the respect of the implementation for Cloud to Peer. Due to this reason we expected that Cloud to Peer should have better performance in the case of same test bed and approximate equivalent performance in the case of the others. The results confirm and overcome our expectation: Cloud to Peer exhibits better performance than MapReduce, too better performance in all of the test bed! The mapping phase that translates the dataset to a key-value storage in MapReduce is a very limiting operation for the grep test. For this reason we prefer not to consider this test because it seems to be not so fair.

**Word Count test**

The second test is represented by the word count. Word count operation has the aim of counting the number of occurrences of the words inside a document. The algorithm is implemented for Cloud to Peer similarly to the Grep: the dataset is divided in chunks that are assigned to the sub-cloud nodes. Each node produces the list of the words contained by its chunk with the related number of occurrences. Once the partial result is computed, it is sent to the access point through the partial aggregator; thus we use the parallel aggregation technique. In order to aggregate the partial results, it is sufficient to sum the number of occurrences of the same words. The MapReduce implementation works the same way: the dataset is translated in the key-value form where the keys are the words and the values are the occurrences. The only differences are that (i) MapReduce does not use parallel aggregation and (ii) it starts to count the words after receiving the chunk. Cloud to Peer does not wait for receiving the entire chunk: once the data are available, even just a line of the chunk, the computation starts conclude immediately after the last line is received; it is a real on-line computation. The Parallel aggregation and the on-line processing should favour Cloud to Peer in the case of centralized cluster test bed. In the rest of the test beds we expect worse performance due to network latency.

The results shown in figure 6.6 confirm the expectation: Cloud to Peer exhibits better performance in the case of the cluster and worse in the case of LAN and internet. The results evidence a constant gap between Cloud to Peer
and MapReduce of about 20 MB/s partially reduced by the increment of the sub-cloud size. In the remaining test beds Cloud to Peer reaches the upper bounds of 25 MB/s @ LAN and 15 MB/s @ internet. Test shows that the large availability of nodes in the case of Word Count does not compensate the performance of a cluster due to network delays. In the graph is also plotted the throughput of a single machine on the same input as reference point.

**String Sort test**

The third test is the String Sort. Also in this case the implementation follows the standard pattern: dataset divided in chunks to be assigned in the sub-cloud. The ordering operation is computed as in the merge-sort: the partial results consist of a set of ordered arrays that have to be merged in a single ordered one. The ordering of the chunks is performed by using the quick-sort algorithm. The key-value map of MapReduce is represented by number (value) and position inside the array (key). In this algorithm the ordering of the chunks cannot be performed on-line; the procedure can only starts after data transfer completion. The parallel aggregation is still possible.

The String Sort is characterized by an output size equivalent to the input, then, the amount of data that is moved among the network is higher than the previous tests; due to this reason, the results will be severely biased by the network latency. The performance of Cloud to Peer should still remain good but slightly worse than MapReduce (because its strong optimization of data transfer among the nodes).

Figure 6.7 confirms the expectation: MapReduce beats Cloud to Peer but the gap dividing the two solutions in the centralized cluster is smaller than the
one experienced in the previous test (in which Cloud to Peer beats MapReduce). This time the performance obtained in the LAN and the internet are very bad; the tests performed in the internet on the Planetlab network suffer of continuous disconnections due to bandwidth ban-out, so they are not representative and we prefer to do not show the results.

Remarks

The three performed tests prove the effectiveness of Cloud to Peer. Thanks to the presence of Virtual Tree as fundamental building block of the architecture, it is possible to use the entire framework in a centralized or in a distributed dynamic environment without any modification. The update delay of the RRC results always negligible if compared with the jobs duration (from tens of minutes to sometime hours). The system is evaluated with a limited number of nodes (at most 128 nodes in the case of Planetlab); in order to have a precise idea about realistic update delays a large network is required. Anyway, the tests executed in Chapter 5, even in the case of high churn, show that the query (i.e. the update of the RRC) is always completed in less than 30 time units\(^8\) in a network of 100,000 nodes.

Grep, Word Count and String Sort, are all I/O bounded operations; their performance are strongly related to the network capability. To have an idea about the capability of performance scaling of the Cloud to Peer we have to move toward CPU-bounded applications. A typical example of a strong CPU-bound operation is the brute-force attack in reverse hashing; to this end, we

---

\(^8\)a time unit can be assumed equivalent to one second
provide a Cloud to Peer implementation of reverse hashing of 4 digits. The algorithm takes as input an hash code generated by 4 digits in SHA1 [39], then it tries to guess them by using a brute-force attack. The algorithm works this way: the 4-digit key space is partitioned and the partitions are assigned to the nodes of a sub-cloud. Each node checks if the source four digits are included in its key space by trying all of the possible combinations. The outcome of the partial result is represented by the four digits that generate the hash code or an empty file. The final result is computed by merging the non-empty files. Parallel aggregation is disabled for this application. The test beds used for this application are represented by the centralized cluster and Planetlab.

Figure 6.8 shows the throughput experienced by the brute-force attack in the two test beds. The results prove that in the case of CPU-bounded task the performance are independent from the test bed. The throughput of the centralized cluster is similar to the one of Planetlab. Once the network is no longer the limit, the performance of the centralized cluster can be easily overcome by increasing the sub-cloud size. The performance scale quite linearly with the size of the sub-cloud without experiencing upper bound on the throughput.

6.5 Related works

This section deals with the related works. Cloud computing and grid computing collect a large number of protocols and implementations with the aim of providing a scalable service. The protocols can be grouped in two large families: managed and unmanaged. The current state of the art is characterized
by a large availability of managed protocols like MapReduce [36], Amazon EC2 [91], Microsoft Azure [28], Google App Engine [40], Eucalyptus [41], and GoGrid [48] and an emergent research interest in unmanaged cloud systems [12] [63] [64]. The proposal described in this chapter belongs to the second family, anyway - as described in the introduction - it is clearly inspired by the managed protocols like MapReduce. The main difference of our proposal with respect to the managed protocol is the absence of centralized modules with strong reference to the scheduler. Resource monitoring, fault detection and other services are distributed in all of the nodes that compose the system. The main differences with the family of the unmanaged cloud protocol are the absence of the RRC abstraction and the underlying structure used for performing the matchmaking of the job that clearly cannot be Virtual Tree.

Protocols for managed environments

The three Amazon Web Services (AWS) [91], Elastic Load Balancer, Auto Scaling and CloudWatch together expose functionalities which are required for provisioning of application services on Amazon EC2. Elastic Load Balancer service automatically provisions incoming application workload across available Amazon EC2 instances. It also pays close attention to health conditions of instances and based on that it performs traffic re-routing from faulty instances to healthy ones. Auto Scaling service can be used to dynamically scale-in or scale-out the number of Amazon EC2 instances for handling changes in service demand patterns. Finally the CloudWatch service can be integrated with an application provisioner for collecting real-time information related to application services and VMs. The data monitored by CloudWatch service are required by other AWS services including Elastic Load Balancer and Auto Scaling services. Although AWS services are published and hosted separately, all functions of these services are supported via WSDL APIs, hence enabling the simple integration of services.

Eucalyptus [41] is an open source Cloud computing platform. The system is composed of three controllers that manage the virtualization environment based on centralized and hierarchical network and service structure. The three controllers are Node Controller, Cluster Controller and Cloud Controller, respectively being in charge of managing physical resources for virtual machines, coordinating Node Controllers within the same availability zone, processing connections from external clients and administrators. Among the three controllers, the Cluster Controller is a key component that undertakes activities related to application service provisioning and load balancing. Each Cluster Controller is hosted on the head node of a cluster to enable an availability zone, while inter-connecting outer public networks and inner private networks
together. By monitoring the state information of instances in the pool of server controllers, the Cluster Controller can select the available service/server for provisioning incoming requests. However, as compared to AWS, Eucalyptus still lacks of some of the critical functionalities, such as auto scaling, live migration and support for built-in provisioner.

Azure [28] Fabric Controller aims to be a highly redundant and fault-tolerant service designed for monitoring, maintaining and provisioning Cloud servers that hosts applications. Fundamentally, Windows Azure Fabric has a weave-like structure, which is composed of servers, load balancers, and edges (Ethernet and serial communications). The Fabric Controller manages the servers in the Windows Azure Fabric differently depending on various factors such as service types. If a server is a hardware load balancer then it is managed through a custom driver interface which is implemented from an Azure supported driver model for compatibility purpose. If a Cloud is marked as a service node, then a built-in service, named Azure Fabric Controller Agent that runs in the background and tracks the current state and the goal state of the server, and reports these metrics to the Azure Fabric Controller. If a fault state is reported by the Agent, the Fabric Controller can manage a reboot of the server or undertake re-provisioning of running application services from the current server to other healthy servers. Besides managing servers and load balancers, the Fabric Controller is also in charge of service provisioning by supporting a declarative service model. Declarative service specifications is encoded in every application service, which is used by the Fabric Controller for matching the services/VMs that meet required demands of CPU, bandwidth, operating system, redundancy tolerance and etc.

GoGrid [48] Cloud Hosting offers developers up to three F5 Load Balancers for each account for distributing application service traffic across servers, as long as IPs and specific ports of these servers are attached into the load balancers. The load balancer implements two algorithms for routing application service requests. Round Robin algorithm distributes the incoming traffic to servers in sequence, one after another by taking turns in a distributed fashion. And Least Connect algorithm keeps routing incoming messages to the server that maintains least connection/request sessions. If the load balancer detects that a server crash has happened then all future requests will be bypassed from the crashed server, and will be redirected to other available servers. GoGrid Cloud Hosting only gives developers a programmatic API to implement their custom auto-scaling service. This is in contrast with the Amazon EC2 and Azure Cloud platforms that offer fully functional application scaling and load-balancing services. GoGrid Developers have to write a piece of code to collect usage data, and run/stop servers or migrate up to other servers based on collected data themselves.

Unlike other Cloud platforms, Google App Engine [40] offers developers a
6.5. RELATED WORKS

scalable platform in which applications can run, rather than providing access
directly to a customized virtual machine. Therefore, access to the underly-
ing operating system is restricted in App Engine. Load-balancing strategies,
service provisioning and auto scaling are all auto-magically managed by the
system behind the scenes.

All leading Cloud computing platforms are able to perform load balanced
provisioning and auto scaling to some degree. However, the techniques im-
plemented in each of these Clouds vary: some use a centralized approach,
some use a hierarchical approach. Some of these platforms apply distributed
state replication of critical services for achieving fault-tolerant behavior. Some
platforms can be managed automatically; some offer mixed management, i.e.
both manual and automated; some enables complete automatic scaling such
as Google App Engine, trading off flexibility and architecture constraint; while
some enforce developers to take on the responsibility to plan, design and im-
plement scaling and resource allocation tasks manually.

No single Cloud infrastructure provider has their data centers at all pos-
sible locations throughout the world. As a result Cloud application service
(SaaS) providers will have difficulty in meeting QoS expectations for all their
users. Hence, they would like to logically construct hybrid Cloud infrastruc-
tures (mixing multiple public and private Clouds) to provide better support
for their specific user needs. This kind of requirements often arises in enter-
prises with global operations and applications such as Internet service, media
hosting, and Web 2.0 applications. This necessitates building technologies and
algorithms for seamless integration of Cloud infrastructure service providers
for provisioning of services across different Cloud providers.

Protocols for unmanaged environments

Section 6.1 deals with the possibility of building up a cloud service in an un-
managed environment. The proposed architecture put in evidence that one
of the fundamental building block of the system is represented by the overlay
network used. Every overlay network is potentially suitable for building a dis-
tributed dynamic cloud, both structured and unstructured architecture can be
used. Kim et al. in [63] [64] and Babaoglu et al. in [12] are two representative
examples of grid-computing systems designed on top of a structured and an
unstructured overlay.

The grid proposed by Kim et al. in [63] [64] works as follow: the Job
Injection Node (i.e. the access point) generates a unique id so called GUID
for the submitted job using an underlying DHT mechanism, and initiates the
insertion of the job into the network. The responsibility for the job will be
assigned to the node whose GUID is closest in a clockwise direction around the
Chord [89] ring to the job GUID, via the routing mechanism. Since the Owner Node of a job is determined on the randomly generated GUIDs, some initial amount of load balancing is automatically achieved by spreading ownership of the jobs somewhat evenly across the nodes in the system.

To implement the matchmaking algorithm, the system leverages a tree-shape overlay network called RN-Tree that will represents the sub-cloud assigned to the job. The RN-Tree is an implicit tree built on top of the network, and consists of all currently participating nodes. Each node can determine its RN-Tree parent node based on only local information, which enables building the tree in a completely decentralized manner. The maintenance of the RN-Tree is performed through periodic update evaluated on the Chord ring status.

The matchmaking process correctly works only in the case of low churn rate, in fact, if the network is very unstable (with many new node joins or node departures), the RN-Tree also becomes unstable and this results in having outdated aggregated resource information or invalid parent node pointers. In such situations, some nodes may not even be able to find their correct RN-Tree parent nodes, so that matchmaking requests cannot be forwarded to higher level nodes in the RN-Tree. However, as soon as the underlying DHT mechanisms stabilize the overall network, the RN-Tree periodic mechanism for updating aggregated resource information will enable the matchmaking process to make progress. This is a not negligible limitation in fact DHTs are severely affected by the influence of the churn as remarked in [79].

This work differs form the one proposed in this chapter because the authors do not implement a real Remote Resource Catalogue; the most similar abstraction is represented by the RN-Tree that must relays on the DHT structure. The absence of a clear RRC is remarked by the fact that if two nodes perfectly match the job constraints but they are so far in the key space of the DHT, they will never be in the same sub-cloud beside the case of a sub-cloud which includes the entire network. Another difference is represented by the decoupling introduced between the access point and the node responsible of the computation. The decoupling imposes to move the data from the user to the access point and from the access point to the node responsible of the computation with a potential negative impact on the performance of the data intensive job.

In [12] Babaoglu et al. present a cloud-system architecture built on top of an unstructured P2P network. Figure 6.9 shows the layered architecture of the system.

At the base of the architecture there is a Peer Sampling Service [PSS] obtained by an unstructured architecture [61]. The unstructured overlay network built with a local-view exchange technique a-la-Cyclon [92] exhibits very strong resistance to the churn that allows the Peer Sampling Service to be
always available. The PSS and the related unstructured architecture use the Bootstrapping Service to gather an initial set of nodes to start the message exchange.

The Slicing Service (SS) is used to rank the nodes according to one or more attributes. This service is used to request slices of the whole Cloud according to some user-defined criteria, e.g., a fraction of 5% of the total number of nodes, the top 1% fastest nodes, and so on. When a user requests the allocation of some nodes according to a specific metric (multi-attribute metrics can be supported as well), the SS ranks the nodes according to that metric, and returns the set of resources matching the query. To achieving the ranking based on a specific attribute the SS uses the following gossip protocol: every node generates a random number from the interval [0..1] and then look for neighbour peers to swap its random value. The swapping between two peers takes place only if the order of random values doesn’t reflect the order of the attribute values over the nodes, e.g., if a node has random number 0.7 and attributes value 10 and a neighbour peer has random number 0.3 and attributes value 20, then a swapping between their random values is performed. The candidates peer for the swapping are discovered using the constantly changing set of neighbours provided by PSS.

The Aggregation Service (AS) is used to compute global measures using local message exchanges. The AS allows each peer to know system-wide parameters without the need to access a global registry. Examples include the network size (number of nodes in the Cloud), average load, number of active partitions (subclouds) and so on. The AS works by using a standard gossip aggregation.

T-Man [60] is a gossip-based protocol for building an overlay network with a given topology (tree, ring, mesh or other structures). The system uses T-Man to bind together the nodes belonging to the same slice, by linking all peers of the same slice with a separate ring overlay (which is different from
the random overlay maintained by the PSS). Suppose that a user requests the creation of a subcloud (slice) with 3 nodes: the system selects the requested number of nodes and creates a ring overlay. Each node of the slice has a direct link to its predecessor and successor. Thanks to the T-Man protocol, the ring overlay is maintained even if nodes in the slice fail: the failed nodes are removed from the ring, and links are rearranged to connect the surviving peers. Multiple slices can be active at the same time.

The Dispatcher is responsible for handling the requests submitted by the user through the high level user interface, and translate them into the appropriate low level gossip protocol commands which are sent to the other nodes. Also in this case the main difference is represented by the absence of a Remote Resource Catalogue. In this case the sub-clouds are created by using the Slicing Service that continuously ranks the nodes with several metrics. The SS periodically update the data by using a gossip protocol. Gossip is a best-effort approach then it does not reflect the actual resource availability of the dynamic system; this is far from the concept introduced by the remote resource catalogue. Moreover to use system-wide queries in order to retrieve real-time results matching the user requirements in a unstructured network is very hard.

6.6 Remarks and future works

In this chapter we introduced a general architecture for a dynamic distributed cloud system. The architecture is inspired by the driving directions exposed in [11] and by MapReduce framework [36]. The chapter deals with the importance of the Remote Resource Catalogue as fundamental building block of a dynamic cloud system. Virtual Tree is perfectly suitable for implementing an efficient Remote Resource Catalogue, in fact, together to a slightly modified version of the querying algorithm presented in 5, it represents a ready-to-use RRC for cloud systems.

The chapter is voted to the description of Cloud to Peer, an implementation of a dynamic distributed cloud based on the algorithm presented in 6.2 and the Virtual Tree based RRC. Cloud to peer implementation is validated with respect to MapReduce. The exhibited performance can be even better than MapReduce, but in general they strictly depends from the network capability. Common networks like a LAN or the internet have a limited amount of bandwidth and are not suitable for I/O bound applications; at the opposite, they represent very good solutions for CPU bound task thanks to their large availability of resources (i.e. nodes). The upper bounds experienced in the LAN and in the internet have not to represent a limit for the diffusion of cloud designed for distributed dynamic system: even if they are not able
to substitute a centralized cluster, they are much faster than a single node; thus, they can represent an interesting zero-cost solution for all of the organizations that already own a large number of machines and a LAN and sporadic needs of fast data analysis (companies, universities, departments are perfect candidate). Moreover, the large availability of the nodes present in this settings allow to balance the load among them with the aim of using a limited amount of resources. The main benefit introduced by this remark is that the machine can be used during the office hours for the cloud computation with no-observable delays on the application concurrently used by the employees. A proof of that is represented by the tests performed in our department (LAN test bed): we have always run the tests during the office hours and no one\footnote{we select a set of employees not aware about our experiment, after some test days we interviewed them and they confirm that they didn’t denote any delays in their machine} has experienced delays.

Dynamic distributed clouds are an interesting research topic also for the green-computing because they solve the issues experienced in the data-centers: power supplying, heating and disaster recovery. As remarked in this chapter a dynamic distributed cloud can run on the same machine already in use by the people; if the cloud is in use it implies a larger consumption, otherwise ii would not consume at all. This remark solves all of the possible power management issues imposed by a private cluster that remains active even in the case of zero load. The analysis related to the consumption economy will be focussed in future works.
Chapter 7

Conclusion

This thesis explores the issues of having valid query results in dynamic network. It is easy to have a valid query result in a managed environment such as a database, because it is possible to temporary stop the dynamism (thanks to the transaction abstraction) till the end of the query evaluation by using lock operations on the involved resources; a dynamic environment is characterized by independent and autonomous entities on which it is not possible to invoke the lock operation. The query has to be evaluated on line during the continuous evolution of the state of the resources and the resources themselves. The dynamism exhibited by a dynamic network denies to obtain valid results like in a database: all of the protocols designed to issue a query in a dynamic system always offer best effort semantics. In 2004 and successively in 2007 with a journal publication, Bawa et al. [20] introduce the problem of having a valid query in a dynamic network proving that (i) it is impossible to obtain a snapshot of the network and (ii) it is impossible to consider all of the contributions of the nodes composing the network if the dynamism is unbounded. The proposed assumption is far from the reality: in the real systems the dynamism is usually bounded. Exploiting this remark, this thesis investigates on the possibility of ensuring some validity semantics in a dynamic system in which the dynamism has a bound.

The thesis proves that in a dynamic system it is possible to obtain a valid result by considering all of the contributions of the nodes connected to the network. The achievement complies with the definition of Interval Validity provided in [20] that is impossible to be achieved in unbounded dynamic network.

The result is achieved introducing the abstraction of virtual node that allows it to ensure the stability of the paths of the network; moreover the
thesis proposes a tree-shaped organization of the virtual nodes, called Virtual Tree, that provides a ready-to-use structure for issuing the query and it allows to design a simple query algorithm with a clear termination condition. The thesis provides both formal and experimental proofs about the correctness of the proposed approach. Another important result is that the Interval Validity is not matter of bounds, in fact a query issued on the Virtual Tree by using the proposed query algorithm is always Interval Valid if the structure does not experience disconnection during its execution.

The last contribution of the thesis is represented by an application scenario of the Virtual Tree and the valid queries: a dynamic cloud system called Cloud to Peer. Thanks to the reliability of the Virtual Tree and the validity of the results obtained by the querying algorithm it is possible to easily implement the abstraction of remote resource catalog that is a fundamental block of a cloud/grid computing system because it allows to correctly list the resources available in the network for a distributed parallel execution of the tasks.

In this document we mainly focus the attention on the validity of each query, but it is not always strictly required. Sometimes “fast” result could be better than a “valid” one (assuming the impossibility to have a “valid and fast” result). The Resource Location [4] in cloud computing represents a relevant task in which fast is sometimes preferable to valid: if the resource look-up query is valid, it is possible to calculate the best solution fitting the requirements; anyway, even a sub-optimal solution calculated on a non-IV result can be sufficient. A tree is a very fast architecture for query routing, relaxing the validity property, the Virtual Tree can be easily lighten by the fact that many messages and many structures are introduced in the code with the only goal to ensure the interval validity, then the protocol can become even more (i) scalable and “light” in terms of overhead.

Concluding the contributions of the thesis can be summarized in the following bullets:

• a provable strategy for having valid query results
• a novel abstraction for dynamic environment (virtual node)
• an architecture for a volunteer computing system based on the valid queries and on the virtual nodes

The future works based on the thesis can be various; in fact the three contributions can be the basis of three main future directions of the work:

• querying system in dynamic network are always assumed to be best-effort, the proved possibility of having valid results can be exploited for enhancing the performance of the protocol that are query-dependent, [45] and Cloud to Peer are two clear examples
• virtual nodes can represent the basis of a new family of “virtual protocol” implementing complex structure for dynamic networks, moreover several well-known architectures such as the DHTs can be re-implemented exploiting the reliability of the abstraction allowing to use them in environment with strong dynamism

• the architecture and the implementation of the Cloud to Peer architecture represents one of the first solution for the private cloud computing platform, Cloud to Peer allows to exploit the wasted resources of the PCs of an organization for implementing a zero-cost cloud, reducing the cost and improving the energy efficiency
Bibliography


95


of the 15th annual ACM symposium on Principles of distributed computing (PODC), 1996. 48


[33] Oracle Database. www.oracle.com/. 2


[65] Jik-Soo Kim, Beomseok Nam, Michael A. Marsh, Peter J. Keleher, Bobby Bhattacharjee, Derek Richardson, Dennis Wellnitz, and Alan Sussman. Creating a robust desktop grid using peer-to-peer services. In International Parallel and Distributed Processing Symposium (IPDPS), pages 1–7, 2007. 68


[77] Gutenberg Project. www.gutenberg.org/. 78


[87] Skype. www.skype.com. 4


[94] Lamia Youseff, Maria Butrico, and Dilma Da Silva. Toward a unified ontology of cloud computing. Grid Computing Environments Workshop (GCE08), 2008. 63