September 2006

The Pulse system
A new p2p prototype for live streaming

Diego Perino

France Telecom's supervisor: Fabio Pianese

Eurecom's supervisor: Ernst Biersack

Politecnico di Torino's supervisor: Paolo Giaccone

Not confidential thesis

Institut Eurecom Politecnico di Torino France Telecom R&D

Abstract

With the diffusion of broadband connections, at relatively low prices, Internet is no more considered as before. There is a new point of view, not anymore a simple channel to distribute information, but a wide range of services and applications with high bandwidth requirements, now supported. The p2p revolution permitted to overcome the last bandwidth bottlenecks represented by the limited server capacities, favouring the diffusion of even more bandwidth demanding applications.

Some of these applications were developed for the live media streaming. In this work we firstly present an overview of the existing p2p approaches to live streaming, highlighting their features, with all their advantages and drawbacks. Then, we introduce Pulse, an unstructured p2p system for live streaming defined as flexible, scalable and robust.

We list the aspects that were not addressed in the Pulse definition in order to realize a real world prototype of the system. We describe the researches done about the issues we have to deal with in a real implementation, like membership management, peers' synchronization and coding mechanism. We continue with the protocols and algorithms defined to solve these problems. We also present the modifications brought to the core algorithms in order to make them working in a real world environment.

At the end, we describe Pulse's performances, analyzing the results of the experiments we run with the prototype. We also evidence the Pulse ability to accommodate heterogeneous peers advantaging more generous ones. Moreover, we show that Pulse is able to exploit the resources available in the system by distributing the available bandwidth among peers. We also validate the algorithms and protocols we defined in order to realize the Pulse prototype.

Résumé

La diffusion des accès haut débit à Internet a révolutionné son utilisation. Il n'est plus seulement un moyen pour la diffusion des informations mais il offre aussi des services et des outils qui n'étaient pas disponibles avant. La révolution du pair-à-pair a permit de dépasser la limite de bande passante imposée par les serveurs. La diffusion d'outils, qui demandent encore plus de bande passante, est alors devenue possible. Entre autres, beaucoup d'applications pour le streaming live ont été réalisés.

Dans ce travail nous présentons d'abord une étude des approches p2p existantes pour le streaming live. Nous proposons une comparaison entre elles en remarquant les avantages et les problèmes de chacune. Ensuite nous présenterons Pulse, qui est un système pair-à-pair non structuré pour le streaming live. Il a été conçu pour être flexible, robuste et capable de passer à l'échelle.

Nous remarquons les aspects qui n'étaient pas pris en compte dans la définition de Pulse pour réaliser un vrai prototype. Nous présentons une étude aux problématiques comme "membership management", "peers' synchronization" et "coding mechanim". Ensuite nous proposons les protocoles et les algorithmes étudiés pour résoudre ces problèmes dans la réalisation du prototype. Nous proposons aussi les modifications apportées aux algorithmes principaux pour les adapter à une véritable implémentation.

Enfin nous présenterons les performances de Pulse en analysant les résultas obtenus par les tests réalisés avec notre prototype. Nous montrons la capacité de Pulse de passer a l'échelle en suivant une loi logarithmique. Nous remarquons la propriété de Pulse de servir des pairs hétérogènes en favorisant les plus généreux. Pulse est capable de bien distribuer la bande disponible dans le système entre les pairs en exploitant leurs ressources. Nous proposons aussi une validation des algorithmes et protocoles que nous avons introduits pour la réalisation du prototype.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the nineties Internet was mainly a channel to spread information, services and contents under the control of industry. At the beginning Internet access capacities where very limited and not so spread: everybody remembers the 56 Kbps modem plugged to the traditional telephone line [1]. Since 2000, thanks to the development of new connection technologies, the bit rate of Internet access capacities started to grow. Moreover the monopolies of National Telecommunication providers disappeared in different countries favoring the birth of different private Telecommunication providers. Thus broadband Internet access became accessible to more and more people. This has completely changed the way to look at Internet together with the IT technical improvements. Today Internet is no more a simple channel to spread information, but it offers a new range of services no more under industrial control.

It is important to mention the peer-to-peer revolution, a radical changing of the the Internet concept. In this manner it is possible to retrieve information and services by a simple exchange of data and a share of resources among users, without,anymore, the control of the enterprises. The first, and also the today most popular, p2p applications are the ones for file sharing. However lot of other p2p solutions have been proposed for different purposes.

In the last years two main phenomena have taken more and more relevance: VOIP telephony and media streaming.

The possibility to transport a voice call upon the IP network has changed the traditional view of telephonic communications. Previously, voice calls needed a traditional telephone line to be carried on; on the contrary now there is the possibility to have a voice communication with a simple Internet connection. This opened new research and business fields. This technology is not very recently but it has known an explosion after the recent development of lot of VOIP systems; among them the first one was Skype. With its ten millions users it allows to communicate all around the world for free by simply having an Internet connection.

Another phenomenon, even more recent, is the media streaming. More and more radio and television broadcasters provide on-line (often real-time) access to their programs. This mean that now it is possible to access through Internet to services that were previously possible only with TV or radio devices.

In the last year another interesting trend came out: spread personal and original videos worldwide. This was possible thanks to companies like YouTube or DailyMotion which permit sharing videos

[1] Actually the author have still such connection in its parents' house in Susa (Turin)

of everybody with everybody. With its 70 millions videos watched everyday on its site, YouTube attests the relevance this phenomenon is acquiring on the Internet stage.

The idea behind Pulse perfectly fits the current trends: to give to everybody the possibility to show its video..."live"! For example users would like to share a party with their friends which are on the other side of the world, or somebody would like to share with its family an important event of its life. Pulse has been studied exactly for this purpose: to give to everybody the possibility to have its own TV channel where he can live broadcast his videos. A user simply needs a web-cam, an Internet connection and the Pulse application and then it can live broadcast what he wants. This would offer a new way to share videos and interesting economical implications.

1.1 Multimedia content distribution

The simplest way to transfer a multimedia file is the *bulk file transfer*. The content is simply a common file of known size available at a source. Every client has first to download the complete file and then it can reproduce its multimedia content. There are several ways to get the file. For example, the file could be stored at a server and a client has just to download it as in the traditional client-server model. Otherwise the content could be spread through a p2p file sharing application and so on.

Differently, the multimedia content could be delivered in the form of a *media stream*. The content is no more distributed as a simple file, but it is delivered as a continuous ordered flow of data from a source. A user doesn't need to get the complete content but just to reproduce the flow of information as it is received.

There are two ways to distribute a media stream: *on-demand* or *live*.

On-demand streaming consists in the delivering of a multimedia content of a known size, that is available at a source and that can be get in any moment. When a user wants to reproduce the content, it contacts the source that starts the stream. The most important challenge is the stat-up delay, that is to say the time a user has to wait after a request in order to start reproducing the content. However, this requirement is not a strict real-time requirement since users can wait some instants before to start the content play-out.

Live streaming consists in the delivering of a multimedia content that has to be reproduced as soon as it has been created or few moments later. So we are dealing with the distribution of a file of unknown and unpredictable length in which the data are only available for a small period of time. In this case the most important challenge is the play-out delay, that is to say the time elapsing between the content production and its play-out. It is clear that live streaming presents strict real-time requirements since the volatility of the content.

1.2 Live streaming

The traditional client-server model is suitable for live streaming, but it is impracticable to a big audience because of scalability problems. In fact, a server has a limit bandwidth and cannot serve more than a limited number of clients. Since the nature of the media content, it is very probable that lot of users want to see the stream at the same time and so the server cannot support them.

The best way to distribute a content from a source to a group of hosts at the same time is the IP multicast. However the deployment of IP multicast has been limited due to a variety of technical

and non technical reasons. First of all it requires changes in the network machines increasing the complexity and the overhead at the routers. Thus not all Internet equipments are able to support IP multicast. But, above all, it presents commercial problems since lot ISPs disable it. This because they don't want to carry multicast traffic of other ISPS users without gaining any money.

Since network level Multics doesn't offer an applicable solution, the idea of implementing multicast functionality at application level came out. The packets are no longer replicated at routers inside the network but at the application level or at the end hosts. This solves the problem of changing the network infrastructure, overcomes the USP and the traditional client-server model limitations. This solution is known as Application Level Multics and lot of systems have been developed for this purpose.

Another possibility is to use peer-to-peer networks for the live streaming. The stream receivers act as clients and servers at the same time replicating packets they receive. P2p live streaming systems organize nodes in an overlay in order to address the following issues:

- *Decentralization.* There is not a central authority that performs operations or that coordinates peers, but all the tasks have to be done locally. This may lead to sub-optimal performances but it permits to obtain a scalable and reliable system. In fact there is not the problem of the limited resources of a central entity, and a node failure doesn't kill the whole system.
- *Quality of the delivery data path.* Since the data replication is done at the end-hosts, in these systems the paths from the source to the receivers are different from the unicast paths between them. An important challenge is thus to obtain data paths that differ as less as possible from the unicast paths between source and receivers. Another interesting aspect regards the node degree that indicates the number of peers an host serves at the same time.
- *Robustness of the overlay.* End-hosts are less stable than routers or network equipments and so they can join, leave or simply fail. These systems have to limit the effects of peer arrivals/departures.
- *Control overhead.* The overlay has not a static structure; on the contrary it continuously evolves during time. There is the need of control messages to permit this evolution and to keep the system working. An important challenge is to limit the volume of the messages exchanged.
- *Adaptativeness.* This point is related to the heterogeneity of peers. Hosts may differ one from another in the amount of resources they have: for example they can have different processing powers, different access capacities, etc. These systems have to maximally exploit all the available resources and in particular the available bandwidth.
- *Fairness.* There is not a unique way to define fairness because its meaning changes according to contest where it is used. A p2p system has to address the problem of selfish peers that try to download the content without serving other nodes. These peers are called free-riders and the system must be able to penalize them. Ideally a peer should only get as much as it contributes. However, this is not compatible with a live streaming system where the download rate should be constant and equals to the stream rate. A live streaming system could try to favor more generous peers offering them advantages in term of reliability and latency.
- *Content nature.*The live streaming content has to be reproduced at the receiver either at the reception or shortly after, and it is available at source only for a limited period of time. Such a content is thus very sensible to delays and losses.

The next chapter presents the results of a research that we have done about live streaming problem and about already implemented p2p streaming systems. This research allowed us to understand the most important choices done in the Pulse's design and to propose possible improvements to the system. The research's results could be considered a good overview of the different existing p2p approaches to the live streaming problem.

Chapter 2

Related work

The first attempts in using p2p systems for distributing a streaming content date back to 1997 when a first p2p solution for on-demand data streaming had been proposed. Few years later (2000) some systems for live streaming, working on two-tired infrastructure (networks where only the core is a p2p overlay, normally of powerful nodes, and the end-users are just receivers), appeared. From 2001, the p2p live streaming subject became very popular and researchers started to focus great attention to it.

Some p2p live streaming systems have been proposed. They can be classified in three main categories:

- Structured. In these systems peers are hierarchically organized in a tree or in multiple-tree overlays and data is forwarded from the source to peers upon these structures.
- Unstructured. The content is divided into pieces that are spread in the network by the source without follow a pre-defined structure. Peers have then to established relationships among them to retrieve their missing pieces.
- Others. These systems use an hybrid approach that exploit both previous techniques.

In the following sections we are going to describe these three categories by presenting an example for each of them, and by discussing their properties and performances.

2.1 Structured p2p live streaming systems

In these systems peers are organized in a fairly static structure upon which the stream is spread. This structure is a hierarchical (single or multiple) tree , with the source as root, where every node receives the whole data content from its parents and transmit it to its children. Differences between systems belonging to this category mainly concern the way peers are organized and the algorithms used to create, to maintain and to repair the tree structure.

These systems focus their attention on the quality of the data paths from the source to the receivers. In particular, they try to minimize the differences between the unicast paths linking the source to the receivers and the paths actually followed by the data. There are three main parameters to evaluate the quality of data paths:

- *length of the data path*
- *link stress* It quantifies the load on the network. It is normally computed by counting the number of packets that cross the same link.
- *node stress* It quantifies the load at nodes. It is normally computed by counting the number of packets that a node receives.

The goal of such systems is obviously to minimize as much as possible these three parameters.

The data paths can be modeled through mathematical analysis since they are built following some determined rules. It is thus possible to study the structure properties. For example, it is possible to estimate the time needed by a peer to receive a chunk or the number of peers a node serves at the same time (node's degree) etc.

To determine the play-out time of a stream is thus not a problem, since it is possible to estimate the time required by a peer to receive data from the source. The data always follows the same route and thus the delay is almost constant.

These systems has to focus also great attention to the placement of peers in the structure. It is necessary to place the peers with more resources close to the source. By doing this, they could receive the most recent data and distribute them among a wide range of peers. On the contrary, if peers with scare resources are placed close to the source, they will not have enough resources to serve a wide range of peers slowing down or interrupting the stream distribution.

Structured systems suffer the peers' transiency; this because if one node covering a key role in the structure fails, lot of peers will lose data. For example if a node close to the source fails, all the peers belonging to its sub-tree will lose data until the failure will be repaired.

In these systems the control overhead is mainly devote to the structure building and maintenance.

In the next sections we are going to present single tree and multiple-tree structured systems. In the last section an improvement to single tree solutions will be presented.

2.1.1 Single-tree

Single-tree p2p live streaming systems are today the most populars since they represent the most intuitive way to reproduce the IP multicast structure at application level across many tunnelled unicast connections. In these systems peers are hierarchically organized in a tree structure where the root is the stream source. The content is spread as a continuous flow of information from the source down to the tree. Systems belonging to this category mainly differ in the algorithms used to create, maintain and repair the tree structure.

Since these systems are very close to IP multicast, trying to emulate its tree structure, they are able to achieve data paths that don't differ too much from IP multicast paths. In fact these systems are able to achieve link stress and data path lengths that are 1.5-2 times bigger than the ones of IP multicast.

The stream reception delay of a peer is bounded to the number of hops required to reach the peer from the source. In such systems the number of hops only depend on the position the peer occupies in the tree. This position is fairly static and thus the delay is almost constant.

In single-tree systems all the load is supported by the interior nodes of the tree while the leafs are just receiving data. Thus, if an interior node has not the required computational or bandwidth

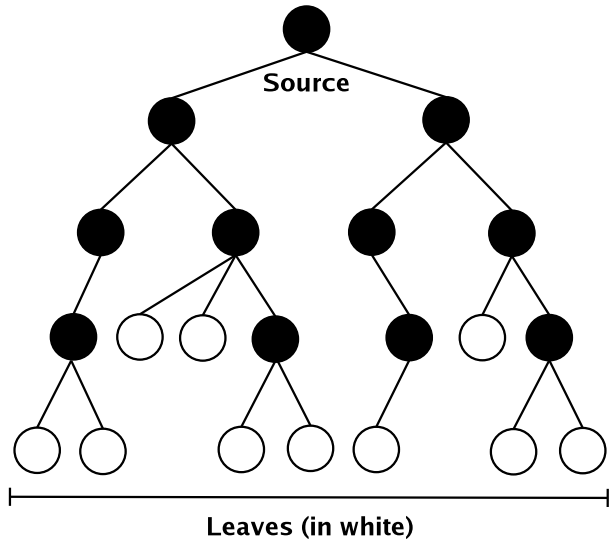


Figure 2.1: Single-tree model

resources to serve all its children, peers in its sub-tree will suffer of high delays in data reception or will never receive the stream. These systems don't seem to exploit very well all the available peers' resources and in particular the available bandwidth; in fact only few peers are in charge of the data forwarding while the others are just receiving data.

Single-tree systems suffer nodes' transiency. If a leaf of the tree fails or leaves, the system will not suffer. On the contrary, if an interior node fails, peers on its sub-tree will lose data until the tree structure will be repaired. The amount of data lost varies from one system to another and depends on the repairing mechanism adopted. These systems implement data recovery mechanisms, such us source coding and so on, in order to face this problem. The data play-out of different systems is thus affecting in different degrees by peers' transiency, according to the data recovery mechanism adopted by each system.

These systems present lack of control about peers' behavior; there is not a mechanism to penalize free-riders and to incentive peer cooperation.

The most popular system using a single tree approach is Nice.

2.1.1.1 Nice

NICE [23] is an acronym that stands for "NICE the Internet Cooperative Environment". It is a project of the university of Maryland aiming to create a group of collaborative applications and to prove that such applications can achieve better performances than ones that don't collaborate. People working on this project proposed in 2002 the Nice application-level multicast protocol. This protocol has been studied to support applications with large receiver sets, low bandwidth and soft real-time data stream. It can also be extended to applications with high bandwidth requirements.

In order to build the tree overlay, NICE assigns hosts to different levels that are sequentially numbered from the lowest. Members of each level are organized in clusters which size is bounded between k and 3k-1 (k is a constant). Every cluster has a leader that is chosen to be the center (with respect to distances) between all its members.

The mapping of peers to levels is done as follow:

- All hosts are part of the lowest level where they are partitioned in different clusters.
- Cluster leaders of the lowest level (level 0) are also part of level 1 where they are newly organized in clusters.
- Cluster leaders of level L_i are also part of layer L_{i+1} where they are partitioned in clusters.

This procedure is done recursively until there is only one peer in a level (Figure 2.2[1]).

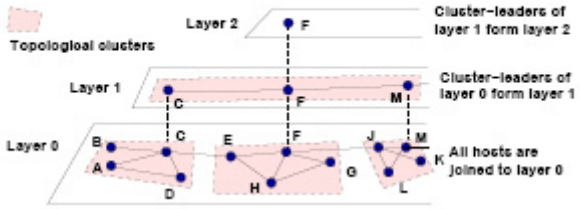


Figure 2.2: Nice structure

By implementing this mapping it is possible to achieve the three following properties:

- An host belongs to only one cluster at each level.
- If an host is present at level L_i, it is also present at every level L_j where j<i, and it is the cluster leader for these levels.
- There are at most $log_k N$ levels, where N is the number of peers and k is a constant.

The peer access to the system is provided by a Rendez-vous point that gives to the new peer a list of peers of the upper levels. The new peer contacts them and it is inserted in the tree by coming down in the hierarchy. At every level the peer is inserted in the cluster whose leader is the closest (distance-wise) to the peer. Mechanisms for cluster maintenance, that foresee heartbeat messages among peers of the same cluster, and mechanisms for cluster refinement, that foresee merge or split of clusters as needed, have been defined.

The data delivery path is a tree and more specifically a source-specific tree. The forwarding mechanism is described in Figure 2.3.

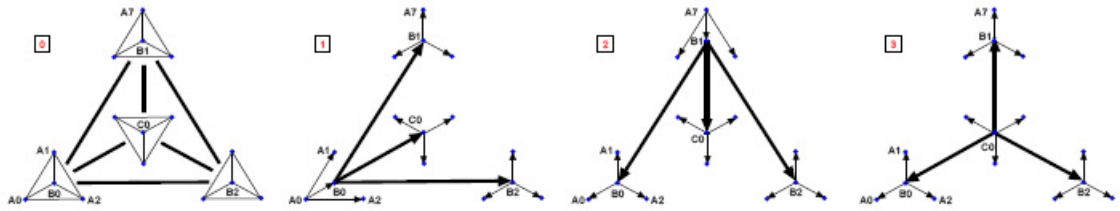


Figure 2.3: Nice forwarding mechanism

[1]Figure 2.2 and 2.3are taken from [23]. Figure 2.1 and 2.5 are taken from [1].

The source forwards the packet to all member of its cluster at level L_0. Each cluster leader is then responsible to forward the data to all members of clusters for which it is the leader, and to all the member of its upper level cluster.

Some simulations and real-world experience results are reported in [23], and show a low link stress, short path lengths, and limited control overhead even in presence of node departures. However, some weaknesses are shown in the amount of data lost in presence of node failure. In this case the 20% of peers don't receive a part of the stream. Another problem is represented by a cluster leader departure; in fact the amount of control overhead generated in this situation is quite high (O(log N) where N is the number of peers).

2.1.1.2 Other existing systems

Other live streaming systems, implementing a single-tree structure, have been proposed.

Two systems have been defined by the university of Standford. A first system, called SpreadIt, has been proposed in 2002. It is a single-rooted tree with bounded fan out, which limits the nodes' bandwidth usage while making the maximum propagation delay optimal. Its goal is to provide an unreliable multicast infrastructure. A second system have been proposed by the same group and it is built upon the previous one; it is called PeerCast. This system is a developed application mostly used for Internet radio broadcast.

A system has been proposed by the CS department of Carnegie Mellon University and it is called End System Multicast (ESM). It aims more to overcome practical issues of a real system implementation than to propose new models or algorithms. Its system structure is in fact really similar to one of SpreadIt and PeerCast.

2.1.1.3 An improvement to the single-tree approach: ZigZag

There are some systems that, even if belong to the single-tree family, introduce some tricks to face the majors drawback of a single-tree solution.

We are going to describe ZigZag [24] that has been proposed by the University of Central Florida and that improves the NICE protocol.

The tree organization is very close to the one proposed by Nice. Peers are organized in levels and clusters in the same way as Nice does. The difference is that each cluster has not only a leader (here called cluster head) but also an associate head. The algorithms for structure building and maintenance are quite similar to Nice and all the Nice structure's properties are still valid.

The main difference from Nice is however the data delivery path. While in Nice everything if forwarded by the cluster leader, here the responsible for data forwarding is the associate head. It recovers data from a non-head member of the above level and sends it to all its cluster members.

The most interesting improvements with respect to Nice are :

- The worst case node degree is bounded to 6k-3 (where k is a constant). This offer an higher level of scalability than Nice..

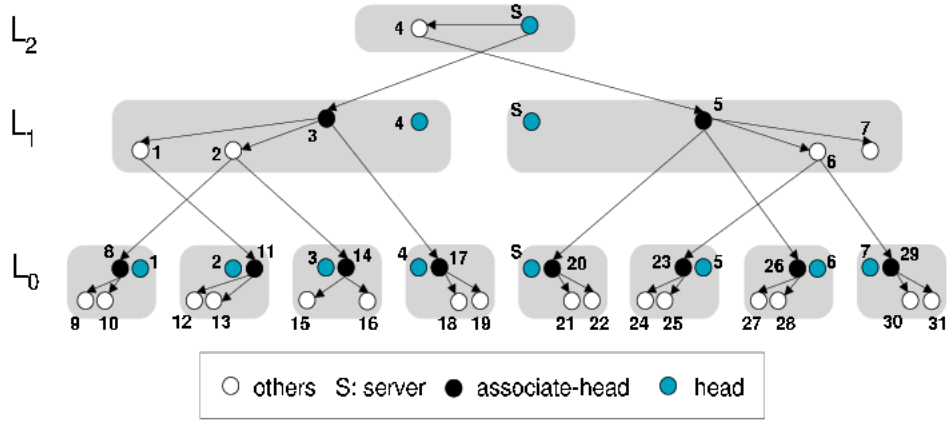


Figure 2.4: ZigZag structure

- The recovery in case of head or associate head departure is easier than the recovery after the cluster leader departure in Nice. This is possible thanks to the existence of two head roles. In fact, if the head fails, the associate-head will immediately take the role of head and select a new associate-head. The same happens in case of associate-head failure: the head can immediately select a new associate-head. Problems arise in case of contemporary failure of head and associate-head. This solution offer a structure that is thus more robust than the Nice structure.

This version of ZigZag is called I-ZigZag ("I" stands for indirect) in order to differentiate it from a second version called D-ZigZag ("D" stands for Direct).

In D-ZigZag members of a cluster directly download data from a non-head member of the above level removing the role of the associate head. This second version is more suitable for low latency applications but it is less robust and has higher bandwidth requirements for member nodes.

2.1.2 Multiple-tree

One of the main issues of single-tree approach is that the load burdens only few peers in the network. As seen in the previous section, this aspect also leads to resiliency problems.

In order to address these issues a multiple-tree approach has been proposed. The idea is to built N different trees, sharing the same source, among peers. The stream content is divided in N complementary stripes and every stripe is spread upon a different tree. It is important to place a node at different levels in the different trees. In particular, the key challenge of these systems is to build a forest of interior-node-disjoint trees. This mean that these systems try make a node an interior node for only one tree and a leaf for all the others.

Since all peers are involved in the data distribution, the load is spread among all nodes. This also leads to reduce the link stress and the node stress (number of packet that are sent by a node) with respect to a single-tree approach.

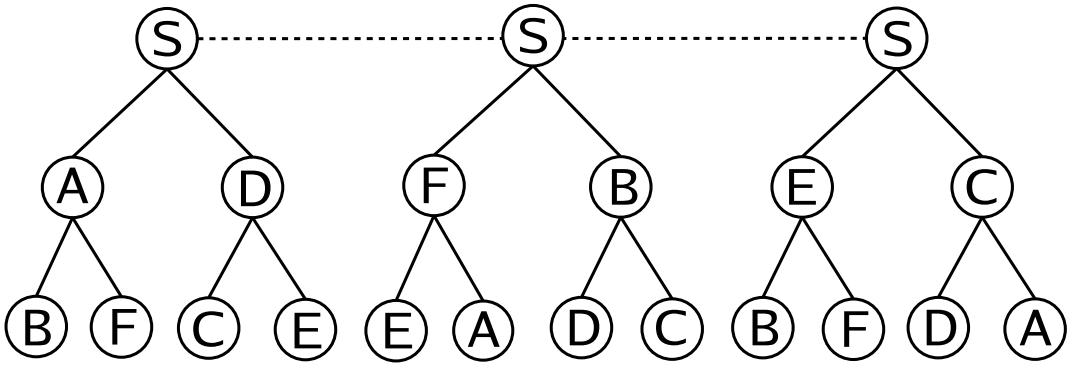


Figure 2.5: Multiple-tree model

Another important improvement,with respect to single-tree approach, is the robustness. In fact a node failure causes losses on only one stripe. Peers thus loose only a small amount of data until the failure is repaired.

A drawback of these systems is the bigger control overhead with respect to the single-tree approach. This because there are many trees to build and maintain.

This approach is relatively new and is very fashion now. One of the first proposal that uses this approach is SplitStream; we are going to describe it in the next section.

2.1.2.1 SplitStream

SplitStream [12] system has been proposed in 2003 by the Microsoft Research center. It is able to build interior-node-disjoint trees that accommodates peers with different bandwidth capacity limiting the overhead due to tree construction and maintenance.

SplitStream is built upon Scribe ([26]) which is in turn based on Pastry ([27]). In particular, Scribe builds a multicast structure by the union of all the Pastry routes from the receivers to the node responsible for the chosen multicast ID. The Pastry routes have the interesting properties that every node belonging to the route shares at least the first digit of its ID with the destination node ID. Thus the multicast tree built by Scribe involve nodes whose IDs have in common at least the first digit.

SplitStream is able to build interior-node-disjoint trees exploiting this properties. It builds different trees exploiting the functionality of Scribe. The tricks consists in chose one different group ID for each stripe and these IDs have to differ for at least the most significant digit. By doing this one node can be an interior node for only one stripe.

Moreover SplitStream allows nodes to declare the number of stripes they would like to receive/transmit and it has techniques to accommodate these requirements without loosing the previous property.

2.1.2.2 Other existing systems

P2PCast ([30]) is a system developed at CS department of New York University. It is largely based on the SplitStream architecture. It aims to improve SplitStream performances by removing the dependencies from the DHT overlay and by better allocate the spare bandwidth among nodes.

PrefixStream [28] is a system developed at INRIA of Rocquencourt. It is built upon a De Bruijn graph where the graph's node are replaced by disjoint clusters of nodes.

CrossFlux [15] is a system developed at the Computer Science department of the Neuchatel University. This system, beyond the multicast tree links, introduces some backup links for fast failure recovery. Moreover, ClossFLux implements a mechanism to optimize the overlay based on the nodes' available bandwidth.

2.2 Unstructured p2p live streaming systems

In these systems peers are no more organized in a hierarchical structure where the stream is forwarded as a continuous flow of data. Here, the source splits the stream in a series of pieces (often called chunks) and distributes them to different peers. The relations established among nodes are "data driven" in the sense that a peer establishes relations with potential providers in order to get its missing pieces.

In these systems there is not a real data path since every chunk follows a different route to arrive at peers. It is not possible to model these systems through mathematical analysis since the unpredictability of data exchanges. Some papers presents mathematical models where they can assure a bound for the number of hops a chunk does before to arrive at peers. However these mathematical models are only valid under great assumptions that are not often realistic.

As a consequence, it is not possible establish a precise time bound for packet reception. It is thus necessary to adjust the stream play out time according to the download rate. This is not a trivial task since the download rate can fluctuate during time.

Unstructured systems offer good performances in term of robustness. Since there is not an overlay structure a node failure doesn't affect the system. Peers that cannot get data from a failed node, will download their missing pieces from another provider without the need of repairing mechanisms.

In an unstructured system, where peers are not organized in a fairly static structure, it is not necessary to place peers with more resources close to the source in order to achieve better performances. Here the peers naturally adjust their position in the overlay according to the network changes. This means that peers with lot of resources are naturally closest to the source and ,if the network conditions change, peers will naturally move far/close to the source according to the change happened. By doing this, these systems can accommodate peers with different resources and in particular with different access capacity.

These systems try to minimize the control overhead; however it is not possible to verify if the overhead generated by these systems is lower or bigger than the overhead generated by a structured system.

In these systems it is possible to introduce the concept of fairness. It is in fact possible to evaluate the behavior of the other peers and to penalize or advantage them according to it.

The most famous system that implements such approach is CoolStreaming/DONET and we are going to present it in the next section.

2.2.1 CoolStreaming/DONET

DONET is a data driven overlay network for media streaming presented in [2]. Every peer of the systems has to focus its attention on three main aspects: the knowledge of other nodes in the network , the selection of peers to exchange data with and the chunk scheduling.

In DONET the first problem is solved thanks to the exchange of knowledge messages using a gossip membership protocol to distribute them. When a node receives a membership message, it will update/create the entry in the membership list. Then it forwards the message to another randomly selected peer until the message is spread to all nodes. When an entry is not update for a certain time is discarded.

A node keeps M partners with who exchange chunks among the peers present in the membership list. It periodically randomly selects new partners and gives them a score. The score is calculated for each peers as the maximum value between the number of chunks sent and received from it. After exploring new partners the one with the lowest score among the new and the old ones is discarded.

The scheduler uses a Rarest first policy between the chunks owned by the partners to select data to require to each of them. The sender simply sends the required chunks to its partners. Information about the chunks a peers own is spread among partners using a bitmap; the chunk number of the first chunk of the bitmap is also sent with it. Receiving these 2 information a peer can select what chunks download and from what node.

The playback delay in DONET is semi-synchronized.

2.3 Other approaches

There exists some systems that cannot be easily classified in the previously presented approaches because they utilize hybrid solutions. We are going to present Bullet that is probably the most significant example among them.

These system try to exploit the positive features of both structured and unstructured approaches mitigating their drawbacks. In particular they are able to well exploit the resources present in the network achieving an high total bandwidth. However they could probably have an high overhead due to both tree management and missing pieces recovery.

2.3.1 Bullet

Bullet [13] is data distribution system targeting large-file transfer or real-time multimedia streaming to a large number of receivers.

This system is to utilize both a tree structure and a mesh overlay to distribute the content among peers.

The data content is divided into chunks and they are sent in different points of the network. Peers receive data from their parents of the tree but they have to find other peers to download missing chunks from.

To summarize there are 3 main strengths to solve for a system like this: an efficient tree building and maintenance, a way to spread nodes' information in the network (membership management), an efficient peer selection strategy.

As concern the first aspect Bullet is designed to work with different kind of tree overlays. The authors assume that the slowest overlay link is the one that determine the throughput of the entire tree. So the best possible tree overlay to use would be the one with the maximum bottleneck link (OMBT). Build this kind of overlay is a NP-hard problem and the authors have proposed an offline greedy OMBT algorithm. It works attaching an entry node to the node, already in the tree, with the higher throughput overlay link to it.

As concern membership management Bullet uses RanSub ([14]) a scalable approach to distribute uniform random subset of global state to all nodes of an overlay tree. This is done by collet and distribute messages. Collect messages start from the leaves of the tree and go up to the source leaving state information at each node. Distribute messages start at the source and propagate down to the tree distributing random subset of participants to all nodes exploiting information of collect messages. This operations are done at every constant period of time (called epoch) and the messages propagated among nodes contain information about peers and packets they have. So receiving these messages a node is able to find and locate interesting peers from which download data.

Every node periodically evaluates its sender and receivers peers. Providers are evaluated by counting the number of duplicated packets they sent to the local node with respect to the total number of packet received. If this ratio is bigger than a threshold a new possible provider will be tested. If this new peer offers better performances, it will replace the old one. In the same way a peer evaluates its receiver peers by counting the number of useful (not duplicated) packet the peer is sending to the receiver. Periodically every peer drops the worst receiver and inserts a new one.

Bullet can achieve a total bandwidth that is twice with respect to a simple tree solution and 60% more than a simple gossip protocol. Bullet well performs in lossy network and in presence of transient peers.

2.4 Conclusions

In the previous sections we have presented different approaches for p2p live streaming systems.

It is not easy to make a comparison among them since the evaluation can vary according to the assumptions we make on the application environment.

In Table 2.1 we made a comparison between different systems supposing as application environment the today's Internet. Thus we are dealing with a large set of heterogeneous hosts that can join/leave the system in an unpredictable way. Hosts mostly differ in their access capacity and present an unpredictable behavior.

Approach	Quality of data paths	Robustness	Adaptiveness	Fairness	Management complexity	Control overhead
Structured	Very good	Bad	Medium	-	Medium	Low-Medium
Unstructured	-	Very good	Very good	Good	Medium-High	Medium-High
Hybrid	Good	Good	Good	-	High	High

Table 2.1: Comparison of p2p approaches to live streaming

In such environment unstructured systems seems to better exploit peers' heterogeneity. Moreover they show a better resilience to transient peers. However the quality of the data path is unpredictable since the unforeseeability of the data exchanges.

Structured approaches are able to achieve better performances in term of quality of the data paths but they suffer the peers' transiency.

Hybrid solutions well performs in exploiting heterogeneity and responding to peers' transiency. However they present an high overhead and high management complexity since they have to maintain both a tree structure and a mesh overlay.

There is not a "best" approach; every solution has its advantages and its drawback. An approach may thus be suitable or not according to the goals a system want to achieve and the environment where it will be used.

Chapter 3

PULSE

During my internship I worked on Pulse which is an unstructured p2p system for live streaming. It has been defined to support different kinds of multimedia content, also with high bandwidth requirements, and to distribute it to a large set of heterogeneous receivers.

The Pulse system has been proposed by a PhD student, Fabio Pianese,who had already defined all the principles and the main ideas of the system. He has written a paper [3] describing them and he has developed a complete event-driven simulator to evaluate the performances of the core algorithms.

My role in the project was to create a prototype of a Pulse's node in order to evaluate the performances of the system in a real world environment. This required to face some problems that hadn't been taken into account in the simulator. The main issues I had to study during my internship were the following:

- the design and the implementation of a protocol for the exchange of peer knowledge and data
- the design and the implementation of a mechanism for peer discovery
- the implementation of a mechanism to synchronize the peers' clocks
- the implementation of a data coding mechanism in order to support data losses
- the implementation of a mechanism to interface the Pulse system to a data source/player in order to create/reproduce the stream
- the refinement of existing algorithms in order to solve problems that eventually arise dealing with real world condition

All these aspects have been studied, fixed and tested on real world testbeds (Grid5000 and PlanetLab) in order to evaluate the performances.

This chapter will first introduces the Pulse system as of its original design, then it will describe my contributions to the system and the modifications made to the core algorithms in order to fit the given requirements.

3.1 Pulse - general concepts

Pulse aims to distribute a live content among a large set of peers. This goal introduces two main problems. First of all the system must be scalable since it targets to spread the stream among a large set of users. Second it has to take into account the characteristics of this group of peers. This in order to exploit as better as possible their resources and to give them the best possible service. The natural environment where a streaming system like Pulse can be used is Internet. Thus the large set of peers where the stream will be spread are the Internet hosts. The system must thus be really scalable in order to support such a large number of peers. Furthermore these peers are very heterogeneous; they have different resources and in particular they have very different access capacities. Some hosts have very high speed Internet access while others have scarcely a sufficient bandwidth to support the stream rate.

The main issue for Pulse is to exploit all the available resources at each host in order to provide the stream to each peer. In particular it should try to use all the bandwidth available at each host in order to maximize the total capacity of the system and to gather enough bandwidth to distribute the content to all users.

The motivation for the choice of an unstructured approach comes from these reasons. As we have seen in the previous chapter, a good way to face heterogeneity and transiency is the use of an unstructured approach. With the choice of a structured solution, it would be necessary to place the "best" peers in the most "important" positions of the structure. By doing this they could receive the most recent data and distribute them among a wide range of peers. On the contrary, if peers with scare resources are placed close to the source, they will not have enough resources to serve a wide range of peers, slowing down or interrupting the stream distribution. By choosing an unstructured solution peers are able to move in the system according to their resources and they can quickly react to the network changes. This means that peers with lot of resources are naturally closest to the source and, if the network conditions change, peers will naturally move far/close to the source according to the change happened.

Pulse has been designed to be flexible, scalable and robust. It is data-driven and receiver based; the data exchange mechanism is studied to incentive node cooperation, allowing improvement in the quality of the mesh and reducing the average stream reception delay.

All Pulse nodes are identical and all the associations among them are result of independent decisions based on the local information available at each node. It is not easy to analytically describe the structure of the overlay network that is created among Pulse nodes: it mostly depends on the characteristics of the underlying network, on the resources available at each node and on the chunk distribution/retrieval algorithms executed by each peer.

3.1.1 The stream

The stream is a continuous flow of data generated by a source. It is characterized its bit-rate representing the amount of data required to encode a second of video. This bit-rate is normally constant but it may change during the video transmission. In Pulse, the stream is not transmitted as a continuous flow of data, but it is split into a series of pieces called chunks. Chunks are generated with a constant rate and, since the stream bit-rate may not be constant, they can have different size. However they present a constant average size.

In order to achieve better resilience to chunk loss, the source should apply to the data an error correction code such as FEC.

Chunks are pushed by the source to few peers and then exchanged among them. Each chunk is marked with the original encoding time at the source which is called "media clock".

The "lag" of a chunk is defined as the difference between its media timestamp and the timestamp of the chunk the source is encoding at that moment. Each peer has a buffer where it stores the received chunks. Figure 3.1 shows the structure of the Pulse's buffer and its notations. All the notations are expressed referring to the chunks' lag.

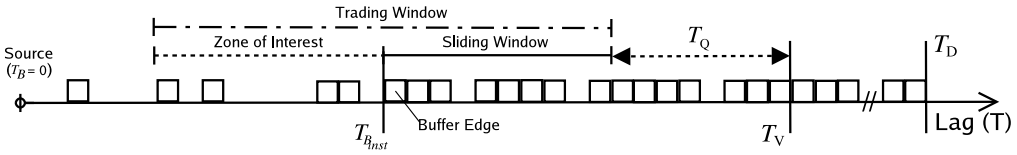


Figure 3.1: Pulse buffer

Parameter	Description
$T_{b_{ist}}$	Average lag of the chunks the peer is requiring. It is placed in the middle of the trading window and it quickly fluctuates.
$T_{b_{avg}}$	Average value for a series of $T_{b_{ist}}$.
T_k	Lag difference between the chunk at the end of the trading window and the chunk at the end of the peer's buffer.
T_d	Lag of the chunk at the end of the peer's buffer.
T_q	Lag of the chunk at the end of the trading window.
Trading window	In includes the chunks the peers is trying to obtain from the other peers. It's size is double than the sliding window's one.
Sliding window	It is the oldest part of the trading windows. It contains the chunks, having a lag greater than the chunk at $T_{b_{ist}}$, the peer is trying to retrive. It will left shift, allowing all the trading window to slide, when it contains a sufficient number of chunks.
Zone of interest	It is the newest part of the trading window. It contains the chunks, having a lag smaller than the chunk at $T_{b_{ist}}$, the peer is traying to retrive.

Table 3.1: Buffer's parameters

This choice has been done in order to simplify the buffer's representation.

When the system is in a steady state we can suppose that peers present a constant average reception delay. The value of $T_{b_{avg}}$ is almost constant and thus also the "lag" values contained in the trading

window don't change too much. Therefore this buffer's representation can eliminate the continuous exchange of information about the IDs of the chunks a peer is trying to retrieve. If the system is not in a steady state, these values will fluctuate. However the information about chunks' lag is still less volatile than chunk numbers.

Moreover, since the play-out rate is constant, the T_v value is also constant during time.

3.1.2 Knowledge management

There are 3 possible levels of knowledge about another peer in the system:

- *"white" knowledge.* This is the basic form of knowledge. It simply consists in knowing the existence of a peer. At this level the only available information about a remote peer are its IP address, its TCP and UDP listening ports. It is not possible to directly establish a relation with a peer for which there is only white knowledge. It is before necessary to contact it or to be contacted by it through messages, in order to retrieve more detailed information about its state. The white knowledge doesn't need to be refreshed and doesn't expire after a timeout. It only expires if the peer is not reachable when the local peers tries to contact it.

- *"blue" knowledge.* In addition to the white knowledge, at this level basic information about the buffer's state at the remote peer is also known: in particular the $T_{b_{avg}}$ and the T_d values of the peer's buffer. Peers for which there is blue knowledge can be selected as recipients of more detailed information about the local peer buffer's state. Blue knowledge will expire after a certain amount of time if it is not updated. When this timeout expires the peer is demoted to a white knowledge state.

- *"red" knowledge.* In addition to the blue knowledge, more detailed information about the buffer's state of the remote peer is available at this level, such as the chunk ID of the peer's buffer edge, the bitmap of the peer's trading window, the measure RTT between the local and the remote peer. One of the parameter of the red knowledge is the history score: it records as a numeric value the quality of the previous interactions with that peer. The red knowledge has very limited validity and the timeout is shorter than the blue knowledge timeout. When it expires the peer is demoted to a blue knowledge state. Peers belonging to the red knowledge group are the only ones that can be chosen as partner for data exchange.

3.1.3 Data exchange

Pulse is a data driven system. In such systems the chunks are not forwarded on a pre-built structure, since there are not prescribed node roles like father/children, internal/external, etc. A node simply sends pieces to others which look for them. It is the availability of data that "drives" the chunks flow. In fact, in order to get its missing chunks, a peer establishes relations with possible providers for these chunks.

Every constant period of time, called *epoch,* the local peer selects a new set of peers to which it will send data. In particular, it chooses two groups of peers [1] :

[1] There exists a third group of peers to which the local peer will send data: the *new peers.* These peers are not selected but they are the ones which sent data to the local peer and that are not in the missing or forward group. They will be served by the local peer before to serve the forward peers if there is excess bandwidth.

- *missing set:* these are peers having a trading window that overlaps with the local peer's trading window. If both sides have chunks the other one needs, exchanges with these partners can be mutual and they should convey the bigger amount of data to the local peer. A configurable quota of missing peers is selected using a tit-for-tat policy among peers which give data to the local peer in the last epoch. Peers are sorted by decreasing amount of data and the first ones are chosen. The remaining missing slots are filled with peers having an overlapping trading window, sorted by decreasing overlap size. If there are still free missing slots, they are filled by selecting peers at random.
- *forward set:* these are peers whose trading window is not overlapping with the local peer's. These peers will be served only if the local peer has excess bandwidth. They are chosen by ordering the known peers with a non-overlapping trading window by their *history score,* and then by selecting the ones with the highest score. In case of ties, random chooses are done.

3.1.4 Chunk selection

Structured systems don't require a chunk selection mechanism since the stream is spread upon the pre-built structure as it is generated by the source. On the contrary, in a data-driven system, it is necessary to define chunk selection strategies in order to assure that every peer receive a chunk before the chunk's playback deadline. A chunk selection strategy can be split in three phases: a first one where the local peers selects the chunks it needs, a second phase where it orders these chunks according to a certain priority and selects the prior pieces, and a third one where the local peers selects the providers for the prior chunks.

Pulse implements two strategies for chunk exchange. A peer can explicitly requests some chunks from the other peers. They are selected by using a local Rarest-first/ Random policy. The missing chunks of the trading window are ordered by the number of copies present among the peers of the red knowledge group and then the rarest ones are selected. Ties are broken choosing a random chunk between those with the same rarity. For every selected chunk a random provider to request the piece is chosen.

When a peer wants to send a chunk to another peer, it checks if there are queued requests from that peer. If so, it will give one of the requested chunks to it by selecting the Least sent; ties are broken choosing one chunk at random. On the other hand, if the selected peer has not requested pieces yet, the local peer will send it a chunk using a Least-sent First/ Random strategy. This strategy consists in ordering the available chunks by the number of times they have been sent by the local peer, and then in selecting the one that has been sent the least so far. If there are chunks that have been sent the same number of times, one among them is selected at random.

3.2 Contributions

My role in the Pulse's project was the realization of a prototype of a Pulse's node. When my internship started the main ideas of the Pulse system, described in the previous section, had already been defined. There was a paper [3] describing the Pulse's design and its most important features. It helps me to understand the global functioning of the system and the design's choices. At the beginning of the internship there was also a complete event-driven simulator designed in order to test the core algorithms of the system. It provided me an useful starting point for my deep understanding of these algorithms. Moreover the simulator gathered me some interesting results about the system.

However, it soon became clear that some aspects of a real-world implementation of the node were not addressed in the simulator.

In order to run the core algorithms, every peer needs to know other peers in the system. In the simulator this knowledge is not a problem. Since the simulator has to simplify the network model, in order to reduce the complexity and the time required for a run, a peer knows all the others in the system without the need of discover them. Therefore a mechanism for membership management had to be designed and implemented. I looked to different existing solutions for this problem and then I designed and implemented a solution that can fit as much as possible the requirements of the Pulse system.

Another aspect, related to the membership management and not addressed in the simulator, concerns the way a peer can join a Pulse session. In the simulator, since a peer knows all the others, it is naturally inside the system. On the contrary, in my prototype, I need to define a way for peer entrance. I had implemented a solution according to the membership management protocol I had previously designed.

Alone the membership management is not enough to support the core algorithms; in fact the knowledge of the existence of other peers is not sufficient to undertake the data exchange. In order to work correctly the algorithms need more information. In the simulator there is not explicit message exchange in order to get detailed information about the peers' state. It assumes that a peer knows this information for peers that has been selected as recipient of data. Moreover also these receiver peers know detailed information about the sender peer. In a real-world implementation the message exchange is obviously required. Thus I had to design and implement a complete Pulse's protocol for message exchange. I had designed and implemented this protocol in order to provide to the core algorithms all the information they require.

Since all the information required by these algorithms is related to time, the peers' clocks have to be synchronized. The simulator runs on a single machine and the peers' clocks are obviously synchronized. This is not a realistic assumption: in the real world, every peer runs on a different machine and thus synchronization among clocks is required. This was another issue I had to solve. I investigated some possible ideas from the existing literature and then I designed a mechanism which could be integrated with the original Pulse protocol.

One of the last problem I had to face is the generation and the play-out of the stream. In the simulator this is not an issue because its goal is to evaluate the algorithms' performances and no data are actually exchanged. In order to implement a real system I had to interface Pulse to a video source/player and to define a mechanism for data encoding.

Once all the limits of the simulator were addressed, I had also had to modify some algorithms in order to better adopt them to a real-world environment. A few more problems that are not considered in the simulator's algorithms had to be faced.

3.2.1 Membership management

Every Pulse's node has to know a subset of the other peers in the system; this is a common necessity for all p2p systems. The way a peer acquires and updates this set is called membership management.

Different solutions to this problem have been proposed by existing p2p applications. Some of them are centralized, requiring the presence of a fixed starting point to enter the system. Others leverage decentralized nature of the p2p network itself to provide the joining node with the initial contact

information. The problem of finding the entry point to the system, be it the address of the tracker or of some already joined node, is outside the scope of our research.

In BitTorrent , for example, the tracker gives to a new coming peer a list of addresses of peers already in the system. This group of peers composes the first peer set of the new node. The information contained in the peer set is updated thanks to the messages exchanged during the permanence in the system. Moreover, always during the permanence in the system, a peer can meet unknown peers that enter in the system later than it and receive its address from the tracker. If a known node becomes unreachable because it leaves the system, it is discarded.

Solutions implementing a completely distributed approach, which doesn't need of a central entity, have also been implemented. Gnutella (version 0.4) is a good example for these systems. The Gnutella membership management protocol is completely decentralized and based on a flooding approach. A new coming peer has to contact a peer already in the system; this peer forwards the discovery message (called "ping") to all its neighbors. Each node receiving the message will do the same until the maximum number of allowed forwarding hops has been reached. Peers receiving the ping message reply to the new comer with a "pong" message, thus growing its peer's set. In order to keep this peer set updated and to maintain its size constant, these ping messages have periodically to be sent also by peers already in the system. Also the messages for the search phase (called "query", "query it" and "push") contribute to the grown of the peers' set of a peer.

The solution proposed by BitTorrent could probably be suitable for a system like Pulse. However I want to maintain a completely distributed solution without introducing a central entity like a tracker. This because the central entity constitutes a single point of failure. In particular the BitTorrent tracker shows problems with big flash crowds. Thus I have decided to design and implement a completely distributed solution. In Pulse the messages spread among peers by the membership management protocol are called Blue messages. As the name suggest, they carry information to spread and update the "blue knowledge".

They need to reach the maximum number of peers and have to be sent at a low frequency. This because, in a steady state, the blue information would not change too much during the peer's permanence in the system. Blue messages are not the only source of information for peer discovery: the peers in the system can also be discovery by the exchange of the other messages which are sent at higher rate with respect to the Blue messages.

A flooding mechanism would not be a suitable solution because of the high overhead that it would introduce. In fact the overhead grows exponentially with the number of hops a message can do.

A possibility is to use a gossip-based protocol. In a gossip based protocol a peer forwards a message with a certain probability to a random subset of nodes chosen among the group of known peers of the system. These protocols offer good scalability and reliability proprieties. It can be shown that the load on each node increases logarithmically with the size of the member group and thus they are scalable. The use of a redundant messages, assured by the nature itself of the gossip protocols, provides reliability in case of node failures and high loss rate in the network.

Gossip-based protocols are studied to work with the knowledge of all participants nodes; thus the choice of the subset of peers to which forward a received message, is done among all peers of the system. Some techniques have been studied to implement membership management protocols providing a subset of peers upon which a gossip-protocol can work without losing their proprieties.

In particular I looked at the SCAMP protocol (SCAlable Membership Protocol) presented in [5]. This protocol is decentralized and doesn't require complete knowledge of the system. Its main idea is to give each peer a "partial view" of the system, that is to say a subset of the other peers present

in the system. This set is used to forward the messages received by other nodes. If a new joining node sends a subscription request to a member chosen uniform at random from nodes already in the system, the partial view size is bounded to $(c+1)logN$. N represents the number of nodes in the system while c is a parameters of the SCAMP protocol. It has been proved that with a partial view size of $logN$ a gossip message can reach every node of the system. This result is valid supposing that there are no losses. In case of losses, with a link loss probability of ϵ, the number of required peers in a a node's partial view is $\frac{logN}{1+\epsilon}$.

I found an interesting comparison between different gossip-based protocols. SCAMP seems to perform well in terms of load at nodes. However, I could find out that there are other gossip protocols that perform better. In my opinion, a protocol similar to SCAMP is enough for our purposes since it has the advantages of being simple, computationally light and of not introducing an high overhead.

The protocol I have implemented uses an approach similar to SCAMP. The main goal of this protocol is to provide to each peer a list of peers to which send blue messages (as the PartialView of SCAMP) and another list of peers from which to receive blue messages (as the InView of SCAMP). These two lists are built thanks to a probabilistic propagation of subscription messages among peers already in the system. The sizes of PartialView and InView are bounded to $logN$ as in SCAMP, since the propagation mechanism adopted is quite similar to the one proposed by it.

The gossip routing proposed by SCAMP is not able to satisfy the requirements of our blue message routing. In fact, by sending blue messages to a random subset of node of the PartialView, and by allowing to forward the same message more than once, the peers have a too limited knowledge about other peers in the system. I have performed some tests using this routing mechanism that have confirmed my hypothesis. I have thus implemented another simple but efficient routing mechanism. Every peer sends, at a constant rate, a blue message to all members of its PartialView. Each peer that receives a blue message from another peer, forwards it once to all peer of its PartialView.

By doing this, every peer has sufficient knowledge about other peers in the system also with very low rate for blue message sending (1 message every 10 minutes). The amount of traffic generated is more or less the same generated by SCAMP. In my protocol every message is only forwarded once to all peers of the PartialView. SCAMP forwards a message to few peers but more than once; it thus sends a total amount of packets equals to the amount of packets sent by my routing mechanism.

A detailed description of the algorithms used by our protocol can be found in Section 3.3.1.

3.2.2 Access to the system

A peer which wants to join an unstructured p2p system needs an entry point to the network. It is possible to allow the access from just one single entry-point, from multiple entry points or through an external entity.

In Pulse it could be possible to allow the access to the system from only one peer already inside it, whether the stream source or a common peer. This solution would probably create a very high load at the entry node making the size of different peers' PartialViews not balanced. Another consequence would be the not balanced load due to the blue message forwarding; in fact it could be higher at the entry node and at its neighbors than at the other peers in the system. In [5] it is proposed an indirection mechanism in order to solve the previous problems. However, this mechanism is quite complicated and demanding in term of computational resources. We have defined a lighter implementation of this indirection mechanism. When a node joins the network it contacts the entry

node that answers with a set of nodes chosen at random from its PartialView. The new node uses as entry point one of the received nodes. However, this mechanism doesn't completely solve the problems of a single entry point but just mitigate their effects.

We have thus decide to allow the access to the system from every node already inside it. This can homogeneously distribute the load among all nodes and make the size of different peers' PartialViews balanced. In Pulse, we suppose at the moment that the IP address/port of at least one node belonging to the system are embedded in a ".pulse" file (see Appendix B). Other mechanisms are however possible and could be used in a future version.

The algorithm for the access to the system is described in section 3.3.1.

3.2.3 Synchronization mechanism

A streaming system like Pulse has strict time requirements. In fact, in a live streaming system, the multimedia content is a continuous flow of information that continuously evolve during time. The data are valid for a limited period of time and then they become useless.

In Pulse there are lot of parameters that are strictly related to time. In order to allow their correct estimation it is necessary that all the peers are synchronized among them. Therefore we have defined a mechanism in order to maintain a loose synchronization between different nodes' clocks.

The mechanism implemented is inspired to the NTP protocol [21]. In our system the synchronization has to be done with respect to the source clock. This because all the system's notations are based on the concept of "lag", i.e. the delay of a chunk, with respect to the chunk the source is encoding. A peer, knowing the initial time and the stream rate, can compute the chunk number of the chunk the source is encoding and thus it can compute all its system's parameters.

The accuracy required in the synchronization has to be sufficient to correctly estimate the chunk ID of the chunk the source is encoding and the RTT between two peers. As it concerns the chunk ID estimation, it is related to the rate of chunk generation. Supposing a stream rate of 512 Kbit/s and a chunk size of 4 KB (which is a very small value for a chunk size) the number of chunks generated in one second is 16. This mean one chunk every 0.0625 seconds. The accuracy required has therefore to be in the order of almost 10^{-2} s. I think that this accuracy is also enough to support higher stream rate. For example, doubling the stream rate and keeping constant the chunk size, a new chunk is generated every 0.03 second and this accuracy is still enough.

As concerns the RTT estimation between two peers, I think that the previously proposed accuracy is also sufficient to correctly estimate it.

The NTP protocol is able to provide a better accuracy than what is needed by our application. Moreover NTP organizes hosts in a hierarchical structure where machines with more precise clocks are on the top. These hosts are continuously synchronized to the national reference clocks and continuously exchange synchronization messages among them. Our application doesn't need such structured organization.

What I get from the NTP protocol is the way a client synchronizes itself with a reference server. In my application I cannot distinguish between clients and servers. However I can define as clients the peer who needs to synchronize itself, and as server a peer that is already synchronized with the source clock.

The mechanism proposed by the NTP protocol allows to compute the RTT and the clock offset between a client and a server. Using "my" mechanism a peer can compute its offset with respect to

a reference peer. A peer can compute its offset with respect to the source by knowing the offset of the reference peer with respect to the source and the offset between itself and the reference peer.

The offset value computed by this synchronization mechanism is not valid forever. In fact, the natural clock drift of each peer's clock can affect peers' synchronization on the long time. It is thus necessary to periodically refresh the clock offset repeating the synchronization operations.

A detailed description of the synchronization algorithm can be found in section 3.3.2.

3.2.4 FEC

In order to achieve better resilience to chunk losses, it could be useful to introduce in Pulse a mechanism for error recovery.

In unicast protocols, solutions like ARQ (Automatic Retransmission reQuest) can be used; however they are not suitable in a multicast environment. If all the receivers that don't receive a packet send a retransmission request to the source, the network will be flooded with requests and retransmitted packets. The situation will be even worse if the receivers use an ack based approach. In this case the number of ack generated is greater than the number of retransmission requests of an ARQ scheme, and the number of retransmitted packets is the same. Some solutions have been proposed to reduce the number of retransmission requests and of retransmitted packets in order to adopt ARQ scheme to multicast.

In multicast environments erasure code solutions may be chosen. In a typical erasure code, the sender encodes redundant packets before to send both original and redundant packets to the receiver. The receiver can reconstruct the original packets receiving a fraction of the total packets sent. In order to do that, the receiver must know the exact positions of the losses or of the corrupted packets. In a real multicast environment, the multicast source applies the code to the original content. Every receiver has to reconstruct the original content from the received packets.

Examples of standard erasure codes are the (N,K) Red Solomon erasure codes. They take K original packets and encode (N-K) redundant packets resulting in a total of N packets to send. The receiver can reconstruct the complete sequence of original packets by simply receiving K packets.

Network coding represents an alternative approach to the classical source coding scheme. With the network coding the source don't apply any form of codification to the the original packet but it simply spreads them in the network. Every intermediate nodes may send out new packets that are linear combinations of previously received information.

In a p2p content distribution environment the Network coding could help in different ways:

1. It could reduce the download time.
2. It could be more robust in case of peer departures or failures.

In Pulse network coding is not a suitable solution mainly for the following reasons[2]:

[2]It could also be possible to introduce the Network coding in the Pulse system. This solution has not really been studied for the reasons explained in this section. However I want to chose an erasure code that could be compatible with a network coding solution. In particular I have to select an erasure code that works with encoding/decoding matrix which can also be used by a Network coding solution. In [16] I have found a Reed-Solomon code implemented using Vandermonde matrix. These matrix have all the proprieties required for a Network Coding approach.

- The multimedia content is a stream that continuously change during time. The data are valid only for a small fraction of time then they became old. When a peer has a sufficient number of packets on which apply network coding, they will be close to their play out time. Thus producing extra packet is vain considering that they become old in few moments.
- It is not possible to apply an integrity control on the received packets because they have not been coded by the source.
- Network coding would introduce a big overhead for the stream reconstruction. This overhead would consume lot of CPU cycles probably affecting too much a system with strict time constraints like Pulse.

The most suitable solution for a system like Pulse is thus a source coding approach. Without a coding scheme the receiver must receive all the packets generated by the source in order to reconstruct the stream. This solution is not flexible in a system where some packets would not be received for log time, or would never be received, blocking the sliding window. A coding scheme allows the sliding window to advance even in case of some missing packets. Of course this solution introduces extra packets in the system. A possible alternative could be to don't introduce a coding scheme and to allow the sliding windows to admit some missing packets. In this case we are forced to reduce the quality of the stream reconstructed.

I have decided to introduce a Reed-Solomon erasure code implemented used Vandermonde matrix. This coding scheme is presented in [16].

The source, when producing chunks, splits the media content in a sequence of pieces of the same length. When it has produced K pieces it can encode the (N-K) remaining pieces using the Reed Solomon code. It then inserts the resulting encoded chunks in the source buffer in order to make them available to the other peers. For our purposes the N value correspond to the length of the sliding window. The value $p = 1 - \frac{K}{N}$ [3]represents the percentage of chunks that can miss in the sliding window allowing it to slide anyway.

3.3 Algorithms

In this section all of the Pulse's algorithms are detailed and discussed. Some of them were already present in the simulator and have not been changed during the prototype's implementation. Others were included in the simulator but had to be changed in the prototype in order to get them working in a real world environment. Finally some other algorithms were not present in the simulator and have been written on purpose for the prototype. For each algorithm we will insert its original version, when available, describe all the changes performed to it and show its final version. The reasons for the changes will also be indicated. A suggestion on the final structure for all the protocol messages will be described in Appendix A.

3.3.1 Access to the system and Membership Management

We are going to describe the access mechanism together with the Membership Management protocol since they are strictly related. The algorithm was not present in the simulator where this mechanism is not present, as basic contact information for all peers is supposed known by all nodes. It has

[3]Equivalent to *1-coding rate*.

been designed and implemented on purpose for the prototype. The algorithm's definition and its main ideas have been described in section 3.2.1 and in section 3.2.2.

In our membership management protocol, each node can act as entry point, be it the stream source or a common peer. This node already in the system is called *contact* node.

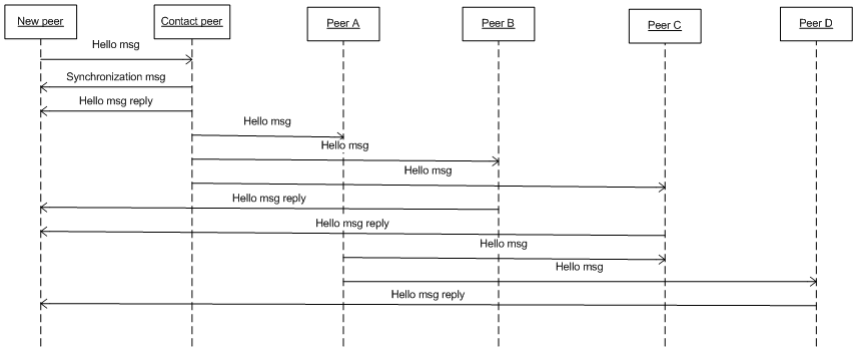


Figure 3.2: Sequence diagram of the access to the system

The new peer starts with a PartialView simply containing the contact peer. The new coming node sends an Hello message to its contact setting the contact field of the message to 1. When the contact peer receives the message, adds the new peer to its InView, sets the contact field of the message to 0, and it forwards the Hello message to all the members of its PartialView. Moreover, it sends back to the new peer a synchronization message; we will better explain later its purpose.

When a peer, whether it is the contact for the new comer or not, receives an Hello message, it keeps the message and it sends back an Hello message reply with a probability inversely proportional to its current PartialView size. If it doesn't keep the message it will forward it to a random peer of its PartialView without answering to the Hello message sender peer. See the pseudo-code for Hello message reception (algorithm 1).

The new peer adds to its InView all peers who replied with an Hello message reply to its Hello message.

Now, every peer has a set of peers to which it will send/forward the blue messages and a set of peers from which it will receive the blue messages. These sets are respectively the PartialView and the InView. In order to face peer departures or failures, entries in the PartialView /InView expire after a timeout. Each node has to periodically resubscribe to the system in order to be kept in the lists of the other peers. The re-subscription mechanism uses the same technique of the access to the system, but the Hello message is sent to one peer randomly selected from the PartialView. This re-subscription mechanism has an important property: it helps to re-balance the size of the peers' PartialView, since re-subscriptions are sent to one random peer of the PartialView of the resubscribing nodes while the access to the system of all peers are done through few entry points.

The forwarding of ordinary blue messages is really simple. Each peer periodically sends a blue message to all peers of its PartialView. Each peer receiving a blue message verifies whether it

Algorithm 1 Reception of an hello msg

```
if sender_peer in not present in peer_list:
    insert sender_peer in peer_list
if local_peer is the contact
    forward hello_msg to all peers of partial_view
    send synchro_msg to sender_peer
    with probability prob=1/(1+length( partial_view))
        add sender_peer to partial_view
        send hello_msg_reply to sender_peer
else
    with probability prob=1/(1+length( partial_view))
        add sender_peer to partial_view
        send hello_msg_reply to sender_peer
    with probability prob=1-1/(1+length( partial_view))
        forward hello_msg to a random peer of partial_view
```

comes from a peer of its InView. If so, the message will be forwarded once to all the peers of its PartialView. The record of the blue message sender is updated if there is not any more detailed or fresher information available about it. If the message doesn't come from a peers of its InView or if the message has already been forwarded, it is discarded.

The mechanism proposed by SCAMP for graceful peer departure has also been implemented in our protocol. A peer that wants to leave the system sends a Bye message to all peers in its InView indicating in the message one random peer of its PartialVew for each of them. A peer receiving a Bye message replaces the entry of the leaving node in its PartialView with the peer indicated in the message. By doing this the the PartialView/InViews' size of the involved peers don't change and all the operations are done locally.

As explained in the previous section every peer can discover other peers in the system and insert them into its peer list through the reception of every Pulse message. However the level of knowledge acquired by the reception of different messages reception is different:

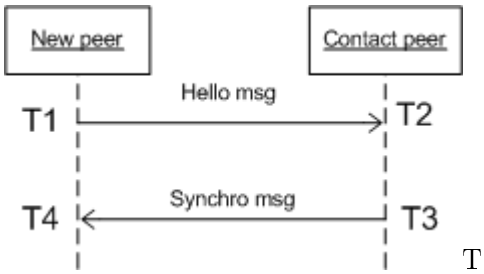
- A blue message contains information to acquire "blue" knowledge
- An Hello contains information to acquire "white" knowledge
- A Red message (usage described in section 3.3.3) contains information to acquire "red" knowledge.

Hello messages, Hello message reply and Blue messages are sent using a UDP connection between involved peers.

3.3.2 Synchronization mechanism

Since in the simulator all peers run on the same machine, their clocks are synchronized. This algorithm has thus been written on purpose for the prototype. Its design is described in section 3.2.3.

As we have seen in the previous section, when a peer joins the Pulse system and sends an Hello message, the contact node sends a Synchronization message back to the new comer (Figure 3.3).



T1: Hello message origination timestamp
T2: Hello message reception timestamp
T3: Synchronization message transmission timestamp
T4: Synchronization message reception timestamp

Figure 3.3: Sequence diagram of the synchronization with respect to source clock

By all the information contained in this message the new peer can compute its offset with respect to the source clock .

In fact, as explained in [21], it is possible to compute the clock offset and the RTT between two nodes as follow:

$$\text{RTT} = \frac{(T4-T1)-(T3-T2)}{2} \qquad \text{Clock offset} = \frac{T2-T1+T3-T4}{2}$$

By knowing the clock offset between the two peers it is simple to compute the clock offset with respect to the source. In fact, we have simply to add to this value the contact peer clock offset with respect to the source in order to obtain the clock offset of the new peer with respect to the source. This parameter is contained in the Synchronization message. The complete formula to compute the new peer clock offset is the following:

Clock offset$= \frac{T2-T1+T3-T4}{2}$ + contact peer clock offset

Of course if the contact peer is the source, the contact peer clock offset is zero.

The clock offset obtained by this computation is not valid forever because of the natural clock drift of each host. So these synchronization operations have periodically to be done. I decided to couple these periodical synchronizations with the re-subscription mechanism for blue message routing (see section 3.3.1). In fact they are done at a sufficient rate to maintain clocks synchronized.

Synchronization messages are sent upon UDP connections between involved peers.

3.3.3 Peer selection for Red messages

In the Pulse system, a peer selects the recipients of its data among the group of peers for which it has a red knowledge. In the simulator, in order to simplify things, it is assumed that a peer have detailed information about the peers it has selected as data recipient. Moreover it owns information about peers which select it as their data recipient. Therefore,there is not a message exchange to spread detailed information about peers' state. In the simulator, the red knowledge group of a peer is thus composed by:

- peers of its missing set that thus have a trading window that overlaps its trading window

- peers of its forward set that thus have a window non-overlapping its trading window
- peers that are sending it data but are not in the missing of forward set. These group, since the greater amount of data is downloaded from peers with a trading window overlapping its trading window, is mainly composed by peer with a trading window that overlaps its trading window.

To summarize, the bigger amount of members of the red knowledge group have a trading window that overlaps the trading window of the local peer while a small group of members have a non-overlapping window with the local peer.

In the real prototype it is necessary to exchange messages in order to spread detailed information about peers' state. I defined a specific message, the "red message", whose main function is to spread these information. As previously said, the knowledge acquired by the reception of such a message is of "red" type. This message has to be sent at high frequency and to an "interesting" group of peers. This group of peer has to follow the same characteristics of the peer's red knowledge group of the simulator. Every peer has thus to select recipients of red messages as follow:

- by selecting a group of peers with a trading window that overlaps its trading window
- by selection a smaller group of peers that have a trading window non-overlapping its trading window

The solution I adopted fort this selection is remotely inspired to [11]. The system presented in the paper has a different goal with respect to Pulse, but it presents an interesting approach for partner selection. Peers are organized into concentric rings and a certain number of peers of each ring are selected to be partners. In our system we can do the same dividing peers into rings delimited by distance from the local peer $T_{b_{avg}}$. For each known peer the difference between its $T_{b_{avg}}$ and the $T_{b_{avg}}$ of the local peer is computed, and the peer is placed in the correct ring. A certain number of peers of each group is then selected to be recipient of red messages (see Figure 3.4). In particular I need to select a bigger number of peers with $T_{b_{avg}}$ closer to the local peer $T_{b_{avg}}$ and a smaller number of other peers. So I decided to uses the following formula to determine the number of peers that has to be selected for each group: $max\ number\ of\ recipients/2^{\mathbf{i}}$ where i is the group level number (the levels are numbered starting with 0 for the group closer to the local peer $T_{b_{avg}}$). I decided to create 3 rings: a first one where the peers have an overlapping trading window with the local peer, a second ring where peers don't have an overlapping trading window but their distance from local peer's $T_{b_{avg}}$ is bounded to *3*trading window size* and a third ring that contains peers with a non-overlapping window which have a $T_{b_{avg}}$ far from the local peer $T_{b_{avg}}$.

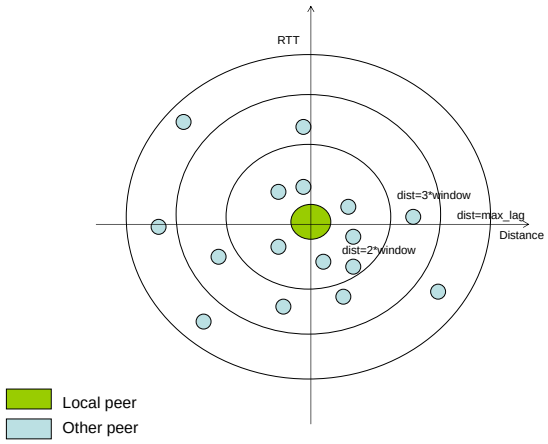


Figure 3.4: Peer division for red peer selection

The pseudo-code of the algorithm used to select the recipient peers for red messages is reported in algorithm 2. The red messages are sent at a constant rate to the peers selected by this algorithm.

Algorithm 2 Red peers selection

Remove all the outdated peers from the *red_list* and insert them in the *blue_list*
list=*blue_list*
for **i**=0,1:
  dist=(i+2)*w
  temp_list= select from *list* peers that |peer.$T_{b_{avg}}$ -*local_peer*.T_b|< **dist**
    insert in *red_list* MAX_NUMBER_OF_RED_PEERS/$2^{\mathbf{i}}$ peers randomly selected from *temp_list*
  remove from *list* all peers present in *temp_list*
temp_list= select from *list* peers that |peer.$T_{b_{avg}}$ -*local_peer*.T_b|> **dist**
insert in *red_list* MAX_NUMBER_OF_RED_PEERS/$2^{\mathbf{i}}$ peers randomly selected from *temp_list*

3.3.4 Chunk exchange

One important requirement of an unstructured system like Pulse is that the nodes receive chunks before their playback time. A possibility could be to realize a receiver request mechanism where every peer requests the older chunks first. This in order to allow the trading window to rapidly slide and to receive the chunks before their playback deadline. However this solution is not suitable mainly because the great part of peers would require the older chunks while only few peers would try to get new pieces reducing the piece diversity.

A possible alternative can be a sender selection strategy where every chunk provider selects the piece to send among the receiver missing chunks at random. However this solution is not suitable because more than one provider could select the same chunk for the same receiver introducing lot of duplicates and thus wasting lot of resources.

Probably the most suitable solution is to use a Rarest first strategy as used in Bit Torrent. In this case all the pieces have more or less the same amount of copies in the system and this increases the

probability that chunks are received on time for their playback. Moreover, thanks to the high piece diversity the trading window of all peers can properly slide because all pieces are available.

As explained in section 3.1.4, in the simulator there are two mechanisms for chunk selection: explicit receiver requests and sender choice.

The former is indeed based on a Rarest first policy. A peer counts the number of copies of every needed chunk available in the group of peers for which it has red knowledge. Then the rarest chunks are selected and requested to one random provider. The peer can select a maximum number of requests for each other peer until a total number of requests is reached. The total number of requests and the number of requests per peer can be set together with the maximum number of peers to which send requests. When a peers want to send a chunk to another peer it checks if there are explicit requests received from it. If so, the peer selects, among the requested chunks, the least sent and sends it. If there are not explicit recipient requests a second chunk selection mechanism is used. The local peer selects the chunk to send using a Least sent/ random first strategy. It orders the chunks needed by the remote peer by the number of times they have already been sent. Then it randomly select one chunk between the least sent ones and send it.

The combined practice of sending and requesting policies can improve the performances because the information about one of the two parties is increased by the information of the other party, increasing the probability to always find a suitable piece to send. Moreover information about rarity of pieces are implicitly forwarded with the requests.

In the prototype I tried to implement the two techniques. I decided to insert the requests in the red messages. So, once the requests have been computed, they are inserted in the red message for the provider peer. Each request will be deleted when the corresponding chunk is received or when a timeout expires and the request has not been satisfied by the selected provider.

However the duplicate chunks problem come out when there are not explicit receiver requests. In fact, even if I am not using a random policy, it is very probable that more than one peer decide to send the same chunk to the same peer at the same time. Some tests have been done and the duplicate chunks problem clearly came out. This problem, in the simulator is solved thanks to a function that checks the provider choices avoiding duplicates. Of course this is not possible in a real prototype where I implemented the following solution.

The sender peer orders the chunks by the number of time they have already been sent and it makes a shuffle between the chunks that have been sent the same number of times. Then it sends the ordered list to the remote peer and it waits for a red message as reply. The receiver peer chooses the first chunk of the list that has not been asked to another peer and has not yet been received. Then it prepares a red message with the request of this chunk and it sends the red message to the sender peer. Moreover the recipient peer remembers the requested chunk in order to avoid duplicate requests. Also this kind of requests have a timeout that expires if the chunk is not received. If there are not interesting proposed chunks the receiver peer sends a red message with an invalid request number; by doing this the sender peer knows that there are not interesting chunks between its proposals. On the contrary, receiving a red message with a request, the sender peer now has a requested piece to send to the recipient peer.

A drawback of the use of the list messages is the high overhead introduced by them. Each time a provider wants to send a chunk and it has not queued requests, it sends a message with a list of chunks to the receiver. It is thus necessary to reduce the number of list messages sent. It could be possible to send list messages at a constant rate. The receiver of a list message could store the chunk proposals and, if has not explicit requests for that provider, it will put in its red messages one piece selected from the last received list of chunks. The selection mechanism among the received

proposals could be the same as before. But probably it wouldn't work well because the Least missing policy doesn't work well if the information is not completely fresh. A possible alternative could be to select, from this list of proposals, the rarest chunks.

A probably better solution to the high overhead could be to eliminate the provider selection mechanism only serving the explicit receiver requests. This is the final solution I have adopted for the prototype. In this case it is necessary to increase the total number of possible requests. Moreover it is necessary to always maintain some pending requests for peers that are serving the local peers. This in order to always provide them a request to serve. It is possible to obtain that by setting the number of requests per peer to a value bigger than 1 and by computing again a new request for a provider as soon as the local peer has received a piece from it. Of course, once this new request has been computed, the local peer will send a red message to the provider peer.

To chose the number of requests per peer is not trivial. In fact if I chose a too big value (i.e. 4) and the provider peer doesn't contain the local peer in its data exchange sets, the local peer will have too many pieces that it will never get because waited from a provider that is not serving it. On the contrary, the choice of a too small value (i.e. 2) could probably be not sufficient to always provide a pending requests for the provider peer.

Another important parameter to correctly set is the request timeout. A too big timeout could block pieces, that will not be sent by the selected provider, for a too long time blocking the receiver's sliding window. On the contrary, with a too small timeout value, a provider could not have the time to serve the peer before the timeout expiration. A probably good idea could be to not set a fix value for the timeout but to adjust it according to the RTT between two peers.

Also the number of peers to which send request could not be easily set. With the choice of a too big value, the local peer could propose requests to lot of other peers that are not serving the local peer. On the contrary, a too small value could not provide to all peers which want to serve the local peer, requests to satisfy. This problem could be mitigated, but not completely solved, with a smart request allocation. I defined an algorithm that first allocates requests to peers that recently serve the local peer and then distributes the remaining missing piece requests to the other peers. However a correct choice of the parameters is really important to obtain good performances from the system.

Algorithm 4 Chunk to send scheduling

#This function is called at a constant rate: each time it tries to send a chunk to a different peer. The peer selection is a simple round robin.
if **local_peer** is NOT the source and **remote_peer** has some requests:
    ordered_list= sort the **remote_peer**.*requests* by number of times a chunk has been sent (least sent-rarest first)
    chunks=select from *ordered_list* the Least sent chunks
    if length(*chunks*)==1
        chunk=*chunks[0]*
        send **chunk** to **remote_peer**
    else
        chunk=select one random piece from *chunks*
        sent **chunk** to **remote_peer**

Differently from the other common peers, the stream source adopts a push technique to distribute chunks. In fact it simply sends to the peers of its missing list the least sent chunks without looking

Algorithm 3 Send Chunk Requests (local 'rarest random' policy)

Determine missing chunks in the local buffer's trading window, store their IDs in *chunks_needed*
chunk_requested stores the ID of the chunks already requested

```
for peer in RED_KNOWLEDGE_List:
    for chunk in chunks_needed
        if peer can offer chunk
            add peer to chunk.providers

for chunk in chunk_requested
    if chunk has been received
        remove chunk from chunk_requested
    elif chunk.request_timeout is expired
        remove chunk from chunk_requested
```

Remove from the *chunk_needed* the chunks present in *chunk_requested* vector

Order *chunk_needed* by chunk availability into a table, with copies_available as the index:
ordered_same_rarity←(copies_available, [chunk ID, [*providers*]])

Sort *ordered_same_rarity*'s lines from the less available set of chunks (rarest first)

```
provider_load=0
for each row of ordered_same_rarity extract the vector [chunk ID, [providers]], called chunks:
    while chunks is NOT empty
        selected_chunk=random.choice(chunks)
        providers=providers for the selected_chunk
# first round to select only providers that are serving the local peer
        while(providers is NOT empty)
            selected_provider=random.choice(providers)
                    if (selected_provider.num_requests<MAX_REQUESTS_PER_PEER) and
(actual_time-selected_provider.last_received_chunk_time)<serving_threshold
                add selected_chunk request to selected_provider.requests
                increment num_requests
                if num_requests>=MAX_CHUNKS_TO_REQUEST
                    return
            remove selected_provider from providers
# There are not providers for this chunk that are serving the local peer
#Second round to select one providers
        providers=providers for the selected_chunk
        while(providers is NOT empty)
            if (selected_provider.num_requests<MAX_REQUESTS_PER_PEER)
                add selected_chunk request to selected_provider.requests
                increment num_requests
                if num_requests>=MAX_CHUNKS_TO_REQUEST
                    return
            remove selected_provider from providers
        remove selected_chunk from chunks
```

at the receivers' buffer and without considering the peers' requests. This because sending to the peers their missing chunks some of the newest pieces won't be injected in the system. Some peers could be blocked waiting pieces that will never be injected in the system. We have performed some trials with the source sending the chunks that miss to the peers; some of the newest pieces are not distributed and thus the lag of all peers grows linearly until they reach a disconnection threshold.

3.3.5 Peer selection algorithms

These algorithms have been written for the simulator and have not been changed in the prototype.

The peer selection algorithm is executed once at each epoch by every peer in order to select the peers to which send data. As explained before two groups of peers are chosen: the *missing* and the *forward* peers (see section 3.1.3 for a description). The missing peers are selected among the peers with a trading window overlapping the trading window of the local peer (algorithm 5). The selection strategy first selects peers which give more data to the local peers in the last epoch (algorithm 6). If there are available slots, it will then select the peers with a $T_{b_{avg}}$ closer to $T_{b_{avg}}$ of the local peer (algorithm 7). Finally, the remaining slots are filled by randomly select peers in order to discover some possible new interesting providers (algorithm 8).

The forwards peers are selected among the peers with a trading window non overlapping the trading window of the local peer. Peers are ordered by history score and the ones with the highest values are selected. See pseudo-code in algorithm 9.

Algorithm 5 Peer Selection Algorithm

```
#Decrease History score for nodes we served through FORWARD exchanges
Update_History(FORWARD_List, -1)
#Increase History score for nodes which served us through unexpected exchanges
Update_History(NEW_List, +1)

#Initialize peer exchange lists
Old_Contributors=[MISSING_List, FORWARD_List, NEW_List]
MISSING_List=[]    #contains MISSING exchange partners
FORWARD_List=[]    #contains FORWARD exchange partners
NEW_List=[]    #collects the peers which send data/requests during the epoch

#generate the MISSING neighbor list for the next epoch
Add_MISSING_TFT()    #first, add the top-contributing neighbors
Add_MISSING_Nearby()    #then, add peers with the most overlapping interests
Add_MISSING_Random()    #finally, may add random nodes to the list

#generate the FORWARD list based on History scores
Add_FORWARD_History()
```

Algorithm 8 Add_MISSING_Random()

#All_Peers is the list that contains all the local information about the active peers in the network
#(we assume that the size of All_Peers >> MAX_MISSING_SLOTS)

while length of MISSING_List < NEARBY_RESERVED_SLOTS: #we still leave some free slots!
  peak an entry from All_Peers with a uniform random distribution, called **peer**
  if **peer** is NOT already contained in MISSING_List:
    add **peer** to MISSING_List
    Update_History(**peer**, -1)

Algorithm 6 Add_MISSING_TFT()

TFT=[] #an empty list
Sort Old_Contributors by decreasing *data_contribution*

for each **contributor** in Old_contributors:
  if **contributor**'s *data_contribution* > MIN_TFT_CONTRIB:
    add **contributor** to TFT

while TFT is NOT empty AND length of MISSING_List < TFT_RESERVED_SLOTS:
  take first element out of TFT, called **peer**
  add **peer** to MISSING_List

Algorithm 7 Add_MISSING_Nearby()

#All_Peers is the list that contains all the local information about the active peers in the network
copy from All_Peers to Known_Peers the records that have a valid $T_{B_{avg}}$

for each **peer** in Known_Peers:
  compute **peer**'s ΔT_B=$T_{B_{avg}}(local) - T_{B_{avg}}(peer)$ and *distance*=$|\Delta T_B|$

sort Known_Peers by increasing *distance*
while Known_Peers is NOT empty AND length of MISSING_List < NEARBY_RESERVED_SLOTS:
  take first element out of Known_Peers, called **node**
  if **node**'s ΔT_B<0: #choose only peers that are behind
    discard **node**, continue
  if **node** is NOT already contained in MISSING_List:
    add **node** to MISSING_List

Algorithm 9 Add_FORWARD_History()

#All_Peers is the list that contains all the local information about the active peers in the network
copy from All_Peers to History_Peers the records that have valid History score H and $T_{B_{avg}}$
sort History_Peers by decreasing H

while Known_Peers is not empty AND length of FORWARD_List < MAX_FORWARD_SLOTS:
  take first element out of Known_Peers, called **node**
  compute **node**'s *chunk_ distance*=$|T_{B_{avg}}(local) - T_{B_{avg}}(peer)|$
  if **node**'s *chunk_ distance* < NEARBY_THRESHOLD:
    discard **node**, continue
  if **node** is NOT already contained in MISSING_list NOR in FORWARD_List:
    add **node** to FORWARD_List

The peer selection algorithm of the source has to be different from the algorithm run by a common peer. In fact it doesn't need to retrieve pieces and so all the policy based on pieces received cannot be used. Moreover it doesn't receive feedback about the behavior of the peers in the system and so can be exploited by peers that try to retrieve all chunks directly. So the source selects at random its missing peers between the ones with a value of $T_{b_{avg}}$ smaller than a threshold. The pseudo-code is reported in algorithm 10. Moreover the source has not a forward list. This because its role is to push the newest pieces in the system. This could only be done by sending pieces to peers with low values of $T_{b_{avg}}$ and thus having only a missing list.

Algorithm 10 Source: add nodes to MISSING_List

#All_Peers is the list that contains all the local information about the active peers in the network
copy from All_Peers to Known_Peers the records that have a valid $T_{B_{avg}}$
TMP=[] #empty list

for each **peer** in Known_Peers:
  if **peer**'s $T_{B_{avg}} < 2W$:
    add **peer** to TMP

while TMP is NOT empty AND length of MISSING_List < NEARBY_RESERVED_SLOTS:
  take random element out of TMP, called **node**
  if **node** is NOT already contained in MISSING_List:
    add **node** to MISSING_List

Add_MISSING_Random()

3.3.6 Peer bootstrap's algorithms

Also these algorithms for peer bootstrap have been written for the simulator and have not been changed in the prototype.

As concern peer selection strategy, described in algorithm 11, the new coming peer selects random peers from all its peer set in order to fill its missing list. This because it has not a valid $T_{b_{avg}}$ value and so cannot apply the previous described selection algorithms. For the same reason a new coming peer has not a forward list.

Algorithm 11 Peer bootstrap: add nodes to MISSING list

set $T_{B_{avg}} = \infty$
fill Known_Peers with random peers
while Known_Peers is not empty AND length of MISSING_List < MAX_MISSING_SLOTS:
    take first element out of Known_Peers, called **node**
    if **node**'s $T_{B_{avg}} > T_{B_{INIT}} + W$:
        discard **node**, continue
    add **node** to MISSING_List

Algorithm 12 Peer bootstrap: set initial $T_{B_{ist}}$

compute the current chunk at source C_{SRC}

how_many is the number of chunks from the local buffer contained in the window $[T_{B_{INIT}} - W,\ T_{B_{INIT}}]$
if **how_many** $> W/2$: #1/2 of the first half is somewhat full..
    new_tb_ist=$T_{B_{INIT}}$ #settle on the virtual initial $T_{B_{INIT}}$
else:
    new_tb_ist=$C_{SRC} - C_{LOCAL}(0)$ #use first packet in buffer to compute working $T_{B_{ist}}$
delta_tb=$T_{B_{INIT}}$-new_tb_ist

if delta_tb >= BOOTSTRAP_MAX_DELTA_TB: #unsuccessful connection attempt
    start over connection process, $T_{B_{ist}} = \infty$, $T_{B_{avg}} = \infty$
    remove outdated chunks from buffer (outside the $[0,\ T_{B_{INIT}} + W]$ interval)
    return
else: #allow $T_{B_{ist}}$ to float
    $T_{B_{ist}}$=new_tb_ist

Algorithm 13 Peer bootstrap: set first $T_{B_{avg}}$

if the node is still NOT CONNECTED:
    tb_avg=tb_bootstrap
    if local peer has chunks in the buffer:
        →set initial $T_{B_{ist}}$
    else: #no chunks: either a first join or a reconnect
        $T_{B_{ist}} = \infty$, $T_{B_{avg}} = \infty$

    if can_start_sliding():
        the local peer is now CONNECTED

Function *can_start_sliding*():

for $T_{B_{TMP}}$ in $[T_{B_{INIT}} - W,\ T_{B_{INIT}} + W]$:
    A is the number of chunks contained in the window $[T_{B_{TMP}} - W,\ T_{B_{TMP}}]$
    B is the number of chunks contained in the window $[T_{B_{TMP}} - W,\ T_{B_{TMP}} + W]$
    if $A \geq W/2$ AND $B \geq W$:
        $T_{B_{ist}} = T_{B_{TMP}}$
        $T_{B_{avg}} = T_{B_{TMP}}$

Chapter 4

Implementation

In this chapter we are going to describe the development of the Pulse's node prototype. When we decided to start with the Pulse's implementation we had firstly to choice the programming language to use. Once it had been chosen we had looked at some frameworks that could help us in the development of some Pulse's componenents.

In the first section we are going to present the programming language we have chosen for the implementation, explaining our motivations. In the second section we are going to present a framework used in the Pulse's development. In the last section of the chapter we are going to describe the structure used for the node's implementation.

4.1 The Python Programming Language

The first important step in the development of an application is the choice of the programming language to use. We have decided to use the Python programming language because we retain it the most suitable for our purposes.

Python is an high level programming language that can be used for many kinds of software development. It has been chosen for the development of many popular applications: one among all, the Mainline version of BitTorrent [31]. Also the applications developed by our team at France Telecom R&D were usually written using the Python programming language. One of the applications conceived and implemented by our team is Solipsis [32]. Solipsis is a shared virtual reality system based on a network of peers. Following a p2p scheme, peers collaborate to build up a common virtual world. It is written in Python with the support of the Twisted framework (see section 4.2).

The Python interpreter is distributed under an OSI-approved open source license and can thus be used for free, even for commercial applications.

Python is an interpreted language and so its programs require no compilation or linking. Moreover, it offers an high level of portability.In fact, it is platform independent and can run on different operating systems such as Windows, Linux, Mac OS, and others. This can be particularly interesting in case of a future release of the prototype.

Python is completely object oriented and this is very useful for our purposes. It allows us to logically divide the different algorithms and the different tasks of the Pulse system in different modules and classes. This offers a better maintainability to the system. For example if somebody wants to modify one algorithm in order to test new policies, it will be easy to find out the class to change.

Since Pulse has strict time requirements, we need a prototype that doesn't consume too much CPU cycles and that is able to run fast. Beyond a careful development, we need a programming language offering good speed performances. The Python programming language offers better performances with respect to other object-oriented programming languages, such as Java, and with respect to other interpreted languages, such as the shells, awk and perl and it .

Python offers other interesting features that can help the programmer in the development of an application. For example, it offers automatic memory management mechanism and it gives the possibility to embed python code in other programs.

.

4.2 Twisted framework

An important problem to solve in the development of an application like Pulse, is the network interface programming. Normally this aspect takes lot of time because of all the problems related to the message parsing and socket management. Since we have decided to use the Python programming language, there exist a network framework written in Python, called Twisted, that can reduce our network programming problems.

Twisted is an event-driven networking framework written in Python that supports TCP, UDP, SSL/TLS, multicast, Unix sockets and a large number of protocols. This framework is widely used for the development of applications written using the Python programming language. For example, the most recent versions of the BitTorrent Mainline program are coded using the twisted framework. It proves very useful to the implementation of the Pulse application for different reasons.

First of all, it simplifies the network programming. It offers some modules that manage sockets in a transparent way for the developer. The programmer has just to specify what actions the application has to take when it receives packets, when a connection is closed, etc., without dealing with socket declaration and management. Nevertheless, it is not necessary to parse the messages: in fact, network modules give back to the application the message as it was sent from the other side of the connection. This modules are called Protocols and we have used them for all Pulse's network interfaces.

The framework can also be useful in order to write concurrent code. Twisted uses non-blocking calls to support concurrency in a program. It introduces the concept of *callback* and *callback chain*.

The typical asynchronous model for alerting an application that some data is ready for it is known as *callback*. The application calls a function to request some data passing a *callback function* to it. The callback function will then be called when the data is ready with the data as argument. It then performs whatever task it was that the application needed data for.

The Twisted object that manages the callback chains is called *deferred* . The application links a series of functions to the deferred. These functions will be called (in order) when the results of the asynchronous request are available. The deferred is also responsible for the *errback chain* management. These chains are similar to the callback chains, but the errback functions will be called if an error arises in the asynchronous request. The availability of deferred and callback/errback is useful for our implementation because it offers the possibility to organize the system differently from the traditional way.

The Twisted reactor, an object that is used to recursively call a function, to listen on sockets, etc., is implemented using these chains. We are mainly going to use one functionality of the Twisted reactor:

the CallLater function. It allows the programmer to schedule a future event such us function calls. This is very useful for functions that need to be called every constant period of time.

4.3 System design

The Pulse's system is based on a group of algorithms that have to run together in order to make the whole system work.We have decided to structure the system in different modules, each one responsible to run some algorithms and to perform some particular tasks. This allow us to program each module independently from the others, focusing our attention on its particular function without thinking to the global system. Moreover, if we wanted to change one algorithm, for example to test a new policy, it would be possible without changing the complete application.

Figure 4.1 summarizes the modular division of the Pulse prototype node. The arrows indicates the relations among the different modules.

We first need a module that manages the others and that coordinates the different tasks. In our prototype we have assigned these functions to the *System Management* module.

The stream content is exchanged among peers in the form of a series of chunks. We clearly need a module for their storage and management. We have called it *Buffer Manager.* Moreover it is also responsible for the management of system's variables because they are tightly related to the chunks reception.

The management of the knowledge about other peers in the systems and the selection of subsets among them have to be performed by a separated module. In our prototype these tasks are performed by the *Peer Manager* module.

We have decided to separate the message sending from the message reception. The control and data message sending have to be done every constant period of time and thus we need a module responsible of their scheduling. We have called this module *Scheduler.* Moreover control messages can be sent as a consequence of a particular event and thus this module has to provide a mechanism for that. On the contrary, message reception is an involuntary event and thus we have separated it from the message sending. We called the module responsible of the reception of all messages coming from the network *Network Manager.*

In the following pages all the modules will be presented in detail.

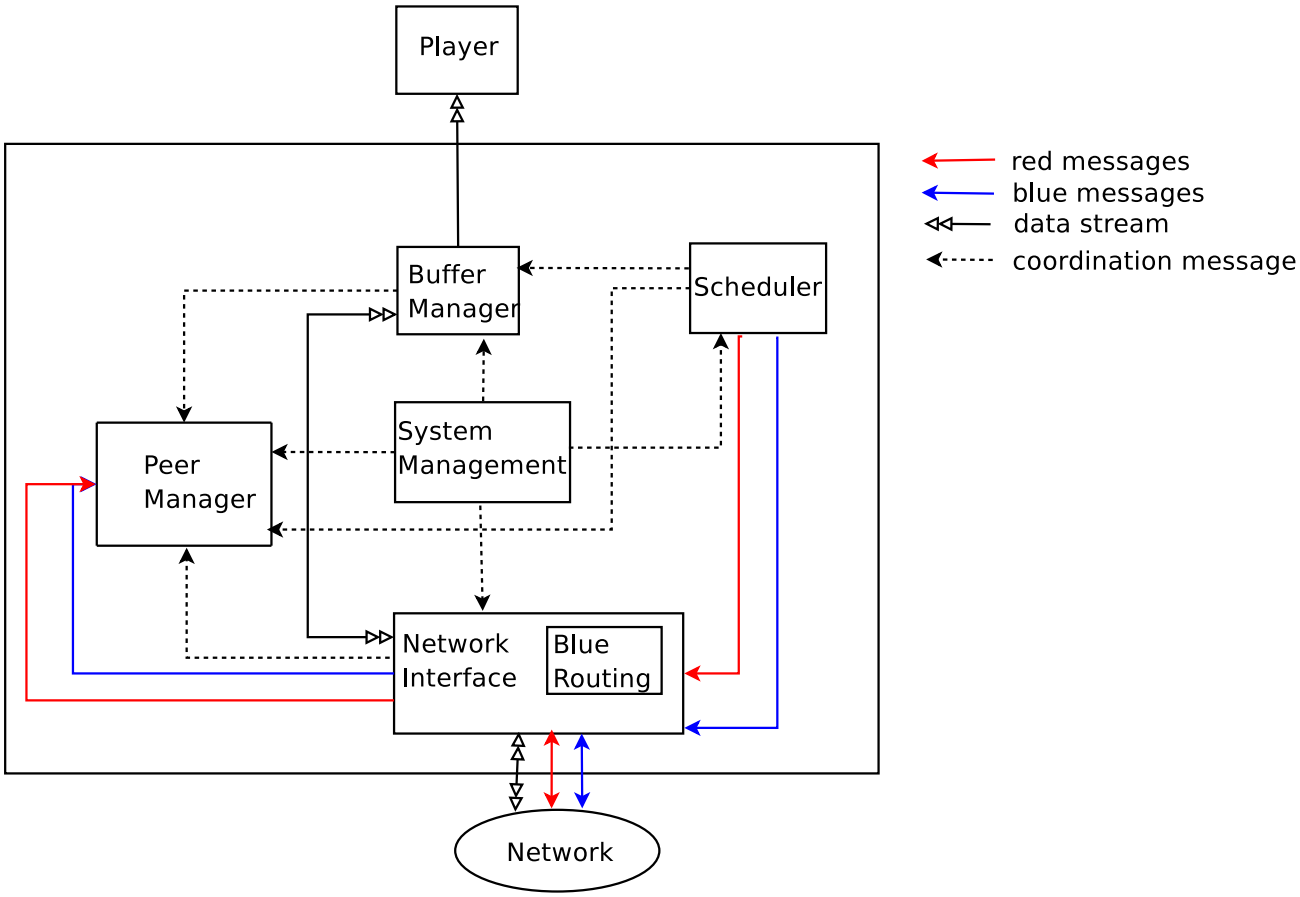


Figure 4.1: System structure

4.3.1 System Management

Since each module is responsible for its particular task, we need a module that can act as a glue between the others in order to make the system working. We have called this module System Management and it is responsible for the management and the coordination of all the other modules.

A first important task assigned to this module is the parameters' configuration. There are two categories of parameters: those regarding the stream and those regarding the node's configuration.

As their name suggest, the stream parameters describe the characteristics of the multimedia content such as stream rate, file format, etc. In this category there are also parameters related to the Pulse's session of that particular stream such as size of the sliding window, the initial time of the stream, etc.

The node's parameters are related to the configuration of the local node i.g. max upload bandwidth, max download bandwidth, max number of missing peers etc.

We have thus decided to define two files containing the two type of parameters: a "pulse" file and a "configuration" file. The former contains the information related to the stream (it can act as the "torrent" file in BitTorrent) while the latter contains the configuration's parameters of the node.

The configuration file is stored in the local node's disk while the Pulse's file is generated by the source when it starts the stream. Every client which wants to join the stream has to retrieve this file.

The System management module is responsible to read the two files, to configure the parameters and to create the instances of the other modules. Finally it can start the Pulse node.

It could be necessary to send some commands to the system while it is running. In particular, if a version of the prototype will be distributed, a user could decide to stop the stream, to restart it or to do similar actions. We have thus to provide a run-time command interpreter where a user can send commands to the application. The system management module is the most suitable to contain it. Since our prototype is an experimental release and, in our simulations, we do not use to send run-time commands to the system, we didn't implement this interpreter.

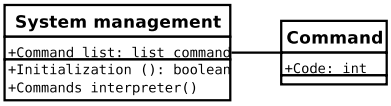


Figure 4.2: UML class diagram of the System Management module

As the above UML diagram shows (Figure 4.2) we have decided to create two classes for the System Management module .

The Twisted reactor, where all the callback functions of the others modules are bound, is also contained in the System management class.

4.3.2 Buffer Manager

The multimedia content is spread among peers in form of a series of chunks. We have thus to provide a module to contain and reorder the chunks received. Moreover, all the variables related to the them have to be computed in order to store up-to-date information about the local peer's state. We have called this module Buffer Manager.

This module must contain an ordered list where the received chunks are stored. In order to know the chunks contained in that list, without going through it all the times, we have defined another list called "bitmap". All the other buffer's variables (see section 3.1.1) are computed on this bitmap and stored by this module. The bitmap is simply a list of bits where each entry corresponds to a precise value of chunk's lag. For every received chunk, the position corresponding to its lag is set to 1.If the chunk has not been received, the corresponding bitmap entry will be set to 0. The bitmap contains information about chunks with a lag in the range [0 T_d].

The origin of the chunks contained in the chunk list is different in the stream source and in common peer.

If the peer is the source, this module will receive the stream directly from an external entity. This entity can be an encoder, that encodes the stream received from a video source, or can be a file or can be the video source itself. The module then divides the stream into chunks and then encodes them with a FEC code (presented in section). These chunks are sequentially inserted into the list,

the bitmap positions corresponding to their lags are set to one and the other buffer's parameters are updated.

If the peer is not the stream source, it will download chunks from its neighboring peers. When a chunk is received, it is inserted in the buffer, the bitmap position corresponding to its lag is set to 1 and the other buffer's parameters are updated. Moreover, in order to reproduce the stream, it is necessary to rebuild the original sequence of data from the received chunks. We need to define a lag value in order to represent the chunk a node should reproduce in every instant of time; this because a node must always know what group of chunks it has to decode and to convert into a stream. This lag value is represented by the T_V parameter. Chunks with lag values larger than T_V, are decoded by this module using the FEC decoding mechanism. Once the stream has been rebuilt, this module pass it to a player for the playback.

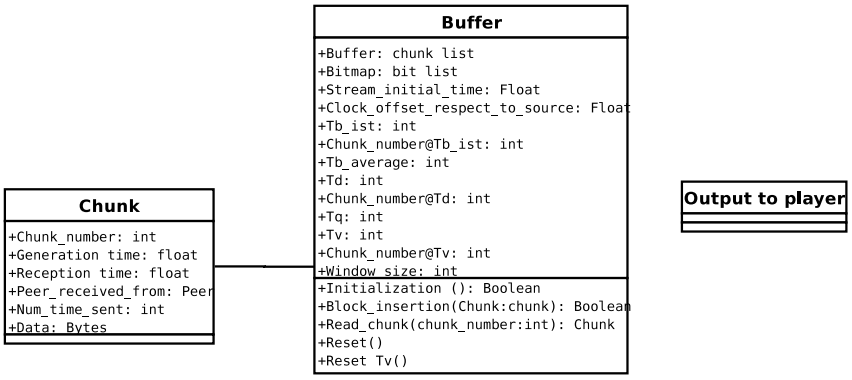


Figure 4.3: UML class diagram of BufferManager module

We decided to create 3 classes for this module (see Figure 4.3):

- Buffer:
  It contains the chunks' list and the bitmap. All buffer notations are managed by this class.
- Chunk:
  We decided to create a specific class for chunk representation. This class has to contain at least the chunk ID and the data bytes. Additional information is also stored ,such as the number of times the chunk as been sent, the peer the chunk has been received from, etc.
- External interface:
  We decided to create a dedicated class to interface the module to an external entity. In this class the multimedia content is treated as a stream, while in the rest of the system the content is divided in chunks. In the stream source, this class receives the stream from a video source or an encoder ,then it divides the stream into chunks and it encodes them with a FEC code. On the other hand, in a common peer, this class rebuilds the original stream from the received chunks and passes the rebuilt stream to a player at a roughly constant pace.

A separate thread has been dedicated to the output to player. In fact this class has strict time constraints since it has to receive/send chunks to stream from/to a player.

In the other two classes, the previously mentioned CallLater function of the Twisted reactor has been used. The calls to this functions are bound to the reactor of the System management class.

4.3.3 Peer Manager

In Pulse, like in all p2p systems, the knowledge of other peers and the management of the information about them are really important. We thus need a dedicate module performing these tasks. In our prototype it is called Peer Manager.

The first important problem to face is to define the way in which we store peer's information. In Pulse, each peer can be classified according to different criteria:

1. according to the level of knowledge the local peer has about it
2. according to the data exchange list it belongs
3. according to its lag distance

Since a peer can be part of these different lists we have decide to don't keep a series of lists and insert peers into them. Instead we maintain a single list of peers and each peer record stores the names of the lists to which the peer belongs. A specific function returns the required list by selecting peers, from the global peer list, that belong to the required list.

In Pulse, the information about peers is not static but they can quickly change. Lots of events can cause the modification of peer's parameters; for example they need to be updated after each message reception or after the peer selection etc. In order to update the internal record of a peer, this module must provide one function for each event bringing fresh information.

Since there are lot of parameters that have a limited time validity, this module must check their freshness. For example, the knowledge about a peer expires after a timeout ,and this module has to demote the peer's knowledge state.

We decided to implement inside this module also the peer selection's algorithms for data and red messages exchange. They have to be run at a constant rate and after their execution the peer records must be updated.

The computation of the "rarest chunks" has to be done in order to prepare the requests for another peer, before sending a red message. This computation is performed among the peers for which the local peer has a red knowledge. This function could also have been inserted into the Scheduler module that, as we will see in the next section, is responsible for message scheduling. However, since the system knowledge is stored in this module, and thus also the information about the pieces owned by each peer, we have decided to put the computation of the rarest chunks inside the Peer Manager module. This function is then called by the Scheduler when needed.

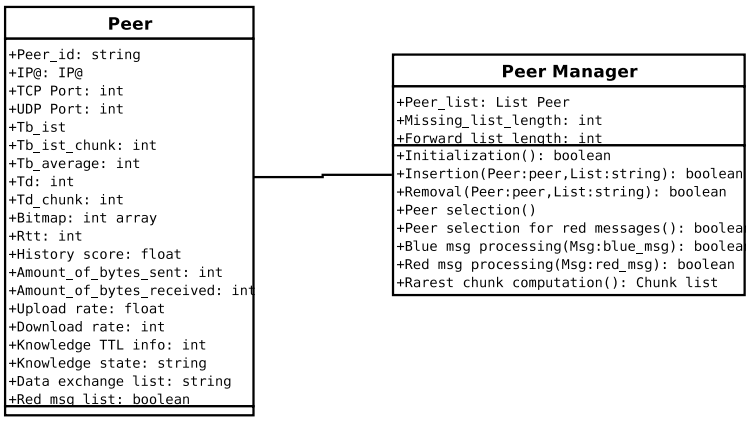


Figure 4.4: UML class diagram of the Peer Manager module

For this module we decided to create 2 classes (see Figure 4.4):

- Peer:
  We decided to create a single dedicated class to store the information owned by the local peer about another remote peer.
- Peer Manager:
  This class is responsible for all the tasks previously described. It contains the peer list. It runs the peer selection algorithms (for data and red message exchange) and the algorithm for rarest chunk computation. The Twisted callLater function is widely used inside this class in order to periodically call functions. In particular it is used to call the peer selection algorithm functions. When called, this class it is bound to the Twisted reactor contained in the System Management module.

4.3.4 Scheduler

In the prototype we decided to split in two modules the message management. The Scheduler module is devoted to the sending messages while the Network Interface module, described in the next section, is responsible for the receiving messages. We opted for this choice because the message reception is an event completely unpredictable and not controlled by the local peer. On the contrary, the message sending occurs at constant rate or as a consequence of a determinate event. Thus, the sending messages requires a module to schedule their occurrences.

Furthermore, before sending a message, a certain number of operations must be performed. In fact it is necessary to select a specific peer to which send the message and compute the information to be inserted into the message fields.

Since in Pulse there are three main kind of messages (blue, red and data) we decided to implement three classes, one for each type (see Figure 4.5).

The algorithms used to send messages, described in section 3.3.4, are implemented inside these classes. Since the messages are sent at a constant rate, in these classes the callLater function of the Twisted reactor is widely used. However the messages can also be sent as a consequence of external

event and not only at a constant rate. Thus, the classes must provide a function that allows other modules to send messages. An example is the reception of a data message; after this event a red message with the new requests has to be sent to the peer which gave the chunk to the local peer.

The Data scheduler behave in a different way whether the peer is the stream source or a common peer. The two algorithms are implemented in the same class and a flag is used to run the correct one. One specific thread has been used for the data scheduler since it is very important to send chunks at the maximum possible rate without the interference of other modules' tasks.

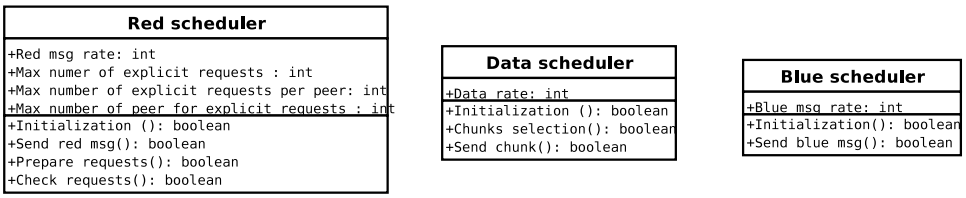


Figure 4.5: UML class diagram of the Scheduler module

4.3.5 Network interface

In this section we are going to describe the Network interface module, whose main role is the reception of all the messages coming from the network.

We have decided to send/receive the control messages on UDP connections while the data messages are transferred using TCP connections. We need to safely transfer chunks from one peer to another avoiding complete packet retransmissions at application level. We thus decided to choose TCP in order to exchange data messages because it offers a reliable packets' transfer. Moreover it offers a very efficient congestion control mechanism dispensing us to think about this aspect. On the contrary, since control messages are sent at a regular rate, we can admit some losses or corruptions without retransmitting them. Moreover the targets of control messages change very often. So the overhead introduced by the establishment and the management of TCP connections will be too high. This module is thus responsible to manage two sockets (one for TCP and one for UDP) listening on two different ports, for both kind of message reception. These ports can be defined in the node configuration's file.

For this reason we have decided to create one class for TCP connections and one class for UDP connections (see Figure 4.6).Both are implemented using the Protocol classes of the Twisted framework (see section 4.2). Thanks to them, we need just to define what actions the application has to take when it receives a message. We programmed the receiving function so that the message is parsed using the Pulse's message header and, according to the message ID, the corresponding function of the responsible module is called. The information contained in the message will be processed by the function called.

We decided to use again a separate thread for the two server classes. This because the message reception could potentially use up lot of resources, and we don't want it interfere with the rest of the system.

We defined a dedicated class for the messages. This class is responsible for the message set up. It creates the common header and then, according to the request, it returns the message demanded.

We decided to also implement the routing for the membership management messages (see section 3.3.1) into this module. This choice has been done considering that this routing is largely independent from the state of the system. Therefore, it is better to implement it into this module, avoiding to unnecessarily surcharge the others. Also, the PartialView and InView lists, used for the membership management routing, are managed by this module and not by the Peer Manager module.

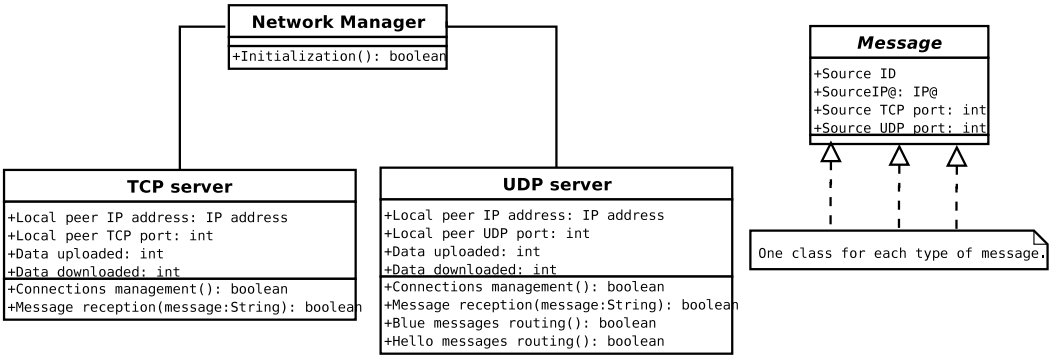


Figure 4.6: UML class diagram of the Network interface module

Chapter 5

Experiment results

A first part of the experimental phase has been devoted to the validation of our Pulse implementation. These tests allowed us to find out some coding bugs and to check if the choices done to address a real implementation were correct. During this phase we continuously improved the protocols and mechanisms defined to provide to the core algorithms all the elements they need to correctly run in a real world environment.

Once we relied in our implementation, we started tests whose goals were to validate the core algorithms and the simulator implementation.

5.1 Experiment techniques

We have run experiments on two testbeds: *Grid5000* and *Planet-Lab.*

The former is a platform distributed over 9 sites in France. It has been developed by the Grid5000 project that aims to provide 5000 CPUs for grid computing and network application testing. The project is in its development phase and nowadays every site provides from 50 to 350 machines for a total of 1200 machines. The different sites are interconnected by the Renateur Education and Research network that provides links at Gbit/s.

Grid5000 is a very useful instrument for testing the Pulse system because everything is under the control of the user and there are not external factors that can alter the results. A user reserves, for a certain period of time, a certain number of machines where it is the only person allowed to run tasks. By doing this, the user can be sure that results won't be affected by high CPU load. The high speed links assure that there are not bottlenecks between the different nodes.

Planet-Lab is a platform composed by machines spread worldwide. It has been developed by a consortium of universities and research institutes in order to provide a testbed for network application testing in a real world environment. Every associated member provides a certain number of machines and a user can run experiments on all the active machines. The interconnections are not dedicated links but the traffic among Planet-Lab nodes pass through normal Internet links. The machines are not under the control of a particular user but lot of people can perform tests using the same machines at the same time.

Since the nature of the two testbed we decided to run two different kind of experiments on the two platforms.

On Grid5000, where we have the complete control of the machines and of the network, we decided to set up artificial scenarios. In every scenario we divided peers in different groups each one with a different upload bandwidth limitation. This allow us to verify the prototype's performances under determined conditions of peers' heterogeneity and system's available bandwidth.

On Planet-lab we decided to perform tests without limiting the peers' bandwidth, since the limits are naturally provided by the interconnections among nodes. By doing this, we can verify the Pulse's performances in a real world environment where there is not a priori knowledge of the hosts' conditions.

5.2 Analysis metrics

Pulse is an unstructured system and it is thus not possible to analyze it using the traditional metrics for structured system analysis.

First of all it is impossible to define a mathematical model of the system. In fact, the nodes are not organized in a tree or multiple-tree structure following strict rules. The relations among peers are results of decisions taken according to the data availability and it it not possible to forecast it. This make impossible any kind of a prior analysis in order to foresee delay bounds, etc.

Since there is not a unique data path the stream follows from the source to the nodes, it is also impossible to utilize all the metrics related to the quality of the data path. These metrics are very useful to analyze a structured system. However, in a system like Pulse it has no sense to speak of node stress or link stress since the route continuously changes and it is different for each chunk. Moreover, metrics like average reception delay and data loss rate cannot be used in their traditional definitions.

We have thus defined some metrics according to the nature of the Pulse system. These metrics are described in the following sections.

5.2.1 Average node reception delay

Since it is not possible to utilize the traditional reception delay definition, we need a metric to express the delay of a node with respect to the stream generation. As we have seen in section 3.1.1, there are two buffer's parameters representing the average age of the chunks a peer is trying to retrieve ($T_{b_{avg}}$) and the stream play-out delay (T_v). The T_v value is set at a certain distance from the $T_{b_{avg}}$when a peer is connected to the system. This is done in order to allow some fluctuations of the $T_{b_{avg}}$value without destroying the stream play-out.

We have decided to utilize the $T_{b_{avg}}$value as indicator of the average reception delay of a node.

First of all, we analyze the evolution of the $T_{b_{avg}}$ during the experiment time. Since in our experiments peers are organized in different classes according to their upload capacity, we provide an average value evolution for each class. We also provide the variance of this average value in order to show the size of its fluctuations.

We use another interesting plot in order to show the distribution of peers of the same class with respect to the $T_{b_{avg}}$. This figure is an histogram plot that indicates the number of peers of a class located at each $T_{b_{avg}}$value in a particular instant of time.

5.2.2 Bandwidth use

Peers need to retrieve chunks at an appropriate rate in order to correctly playback the stream. An important elements to monitor is thus the peers' incoming bandwidth.

Also the upload rate of peers has to be monitored. In fact peers should contribute as much as possible to the system in order to increase its total bandwidth and to permit a correct stream delivering to all nodes.

Since in our experiments peers are divided in different classes, we have decided to analyze the average outgoing and incoming bandwidth of each class.

5.2.3 Chunk rarity

Peers need to receive every chunk before its play-out time. In order to achieve this goal Pulse uses a "rarest first" policy as chunk selection strategy.

We want to monitor if this policy effectively works in our prototype implementation and if there are a sufficient number of copy of every chunk in the system. By doing this, we can observe either peers always have new chunks to exchange among them of not. For this purpose we realize a plot that indicates the number of copy of each chunk present in the system.

5.2.4 Chunk losses

An important aspect to take into account is the quality of the stream play-out. The goal of the system is to provide to all peers the multimedia content at the same quality as it is generated by the source. In order to achieve this goal a peer should lose a percentage of packets smaller than the one recoverable with the FEC used.

We evaluate the percentage of packets lost at the T_v value. By doing this we can evaluate the quality of the stream received at the moment a peer is reproducing it.

5.2.5 Control overhead

One important goal of p2p system is to limit the control overhead with respect to the amount of content data transferred. We have not defined a particular metric for the monitoring of the control overhead. This because in our system, differently from a structured approach, the number of control messages can be tuned by design. Thus we know a priori the number of control messages every peer injects in the system.

5.3 Results

We are going to present the results obtained the preliminary tests performed with the Pulse prototype.

First, we run some scenarios on Grid5000 in order to validate our prototype and to check either it correctly works in normal bandwidth conditions or not.

Then we run experiments on Planet-Lab without bandwidth limits in order to validate our prototype with real network conditions.

Finally we tried to reproduce the scenarios for which tests have also been performed using the Pulse simulator. We have compared the obtained results with the simulator results.

In tables 5.1 and 5.2 are reported the parameters used for all the experiments.

Parameter	Value
Stream rate	512 Kbit/s
Chunk size	4 KB
Chunk rate	16 chunks/s
Sliding window size	32 chunks
% of packet loss admitted	20%

Table 5.1: Stream parameters

Parameter	Value
Number of "missing" slots	4
Number of "forward" slots	8
Number of requests per peer	2
Total number of requests	38
Request timeout	0.25 s
Epoch duration	2.00 s
Blue message rate	0.1 msg/s
Red message rate	8 msg/s

Table 5.2: Peer parameters

5.3.1 Scenarios on Grid5000

In order to validate our prototype we run tests on Grid5000 creating scenarios where peers have limited excess serving capacity. By doing this, we are sure that the results are not affected by a lack of bandwidth in the system. These experiments allow us to modify and improve the algorithms and mechanisms developed in the prototype.

We also tried different peer arrival patterns in order to see how to system react to them.

We tested the scalability of our prototype implementation performing experiments with a growing number of peers.

Symmetric High bandwidth scenario

Table 5.3 reports the peers' configuration for this scenario. All the peers arrive at the beginning of the experiment and leave at the end. The number of peers used for this experiment is 100.

Class	% of peers	Upload bandwidth
RICH peers	100	4 * SBR[1]

Table 5.3: Scenario's description

By looking at Figure 5.1 (b) it is possible to see that peers maintain a constant average reception delay. The average delay value is around 20 chunks that corresponds to 1.2 seconds of stream, since

the source generates 16 chunks every second. The variance of this average delay is of about 10 chunks. This small variance value indicates that the fluctuations of the reception delay are bounded to less than one second. This is a real important result because the distance set between $T_{b_{avg}}$ and T_v is enough to absorb these fluctuations without problems for the stream play-out.

Figure 5.1 (b) shows a greater variance of the average reception delay in the beginning of the experiment. This trend is due to the peers' arrival in the system. There are many peers that have an empty buffer and that need to get a great number of chunks. So the time required to get chunks can be higher with respect to the time required when the system is in steady state. The few peers already in the system have to serve a big number of new comers in order to allow them to connect to the system.

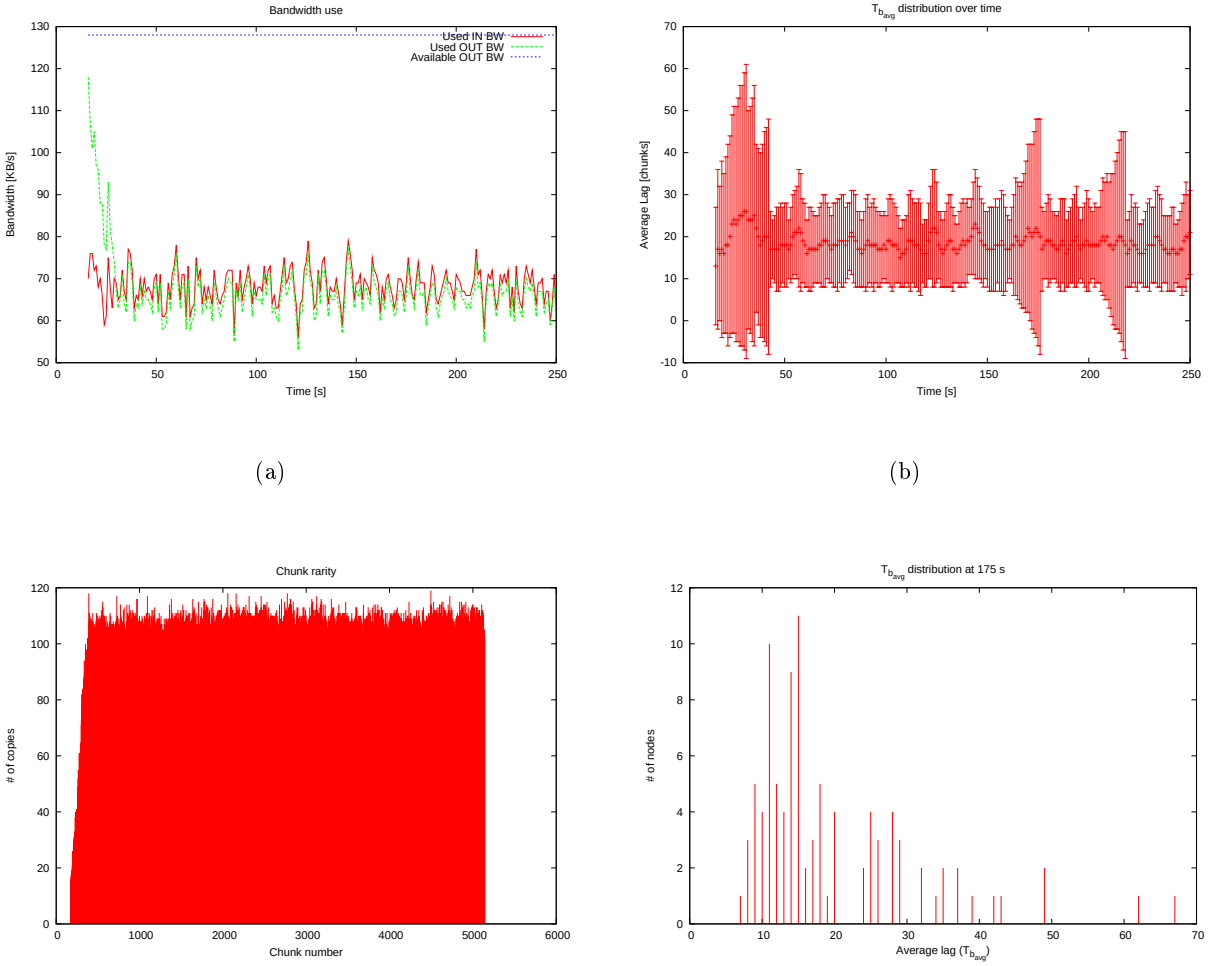


Figure 5.1: Symmetric

This trend is also confirmed by the bandwidth use plot; in fact there is a peak of the upload

bandwidth in the beginning of the experiment due to an higher bandwidth consumption of the peers already in the system

Picture 5.1 (b) also shows some peaks of variance around 175 and 220 second. In these cases, the fluctuations become higher than in the rest of the experiment and reach 25 chunks (1.5 second). This is a normal behavior of the system; in fact, as it is possible to see in Figure 5.1 (d), some peers can have for few moments an higher $T_{b_{avg}}$value because they don't manage to get some chunks immediately. However, they stay connected to the system and once again the distance between $T_{b_{avg}}$ and T_v is able to absorb the $T_{b_{avg}}$ variations without problems for the stream play-out.

In Figure 5.1 (a) it is possible to see that peers upload/download at a rate a bit higher with respect to the stream rate. This indicates the presence of some duplicate chunks. In Figure5.1 (c) we can quantify to 10% the amount of duplicate chunks. These duplicates are present because some requests cannot immediately be satisfied and the provider sends the chunk after the request timeout expiration. However, the receiver peer has already requested the chunk to another provider. As a consequence, more than one copy of the requested chunk are received.

The percentage of packet loses at T_v is on average 3.5%. We apply at the stream a FEC code that is able to support 18% of chunk loses. In this case every node is able to reconstruct the original stream without losing video quality.

Symmetric High bandwidth scenario with spike arrivals

From this experiment we want to see the reaction of the system to spike arrivals of peers.

Table 5.4 reports the peers' configuration for this scenario. The 25% of the peers arrive at the beginning of the experiment while the others 75% arrive after 120 seconds. At the end all the peers sequentially leave. The total number of peers present in the experiment is 100.

Class	% of peers	Upload Bandwidth
RICH peers	100	2*SBR

Table 5.4: Scenario's description

Pulse seems to react well to spike arrivals. As it is possible to see in Figure 5.2, at 120 seconds there is a peak of upload bandwidth as a consequence of new peer arrivals. New peers have an empty buffer and the peers already in the system serve them at high rate in order to provide all their missing chunks.

In the same figure it is possible to see a peak of variance of the average reception delay during the spike period. Figure 5.2 reports a comparison between peers' $T_{b_{avg}}$ distribution before and during the spike. During the arrivals it is possible to notice that some peers present an higher value of $T_{b_{avg}}$ because of the high data demand during this period. However, all peers stay connected to the system and the distance between $T_{b_{avg}}$ and T_v is able to absorb the average reception delay fluctuations without influencing the stream play-out.

It is interesting to notice that the average $T_{b_{avg}}$ value increases during the spike until it remains constant around 20 chunks. This because the number of peers grows from 25 to 100 during the spike.

The percentage of packet loses is around 3.7 %. This value is a bit higher than in the previous scenario as a consequence of the chunk loses during the spike period. Once again the FEC code we use is able to recover the missing chunks allowing the stream play-out without losing video quality.

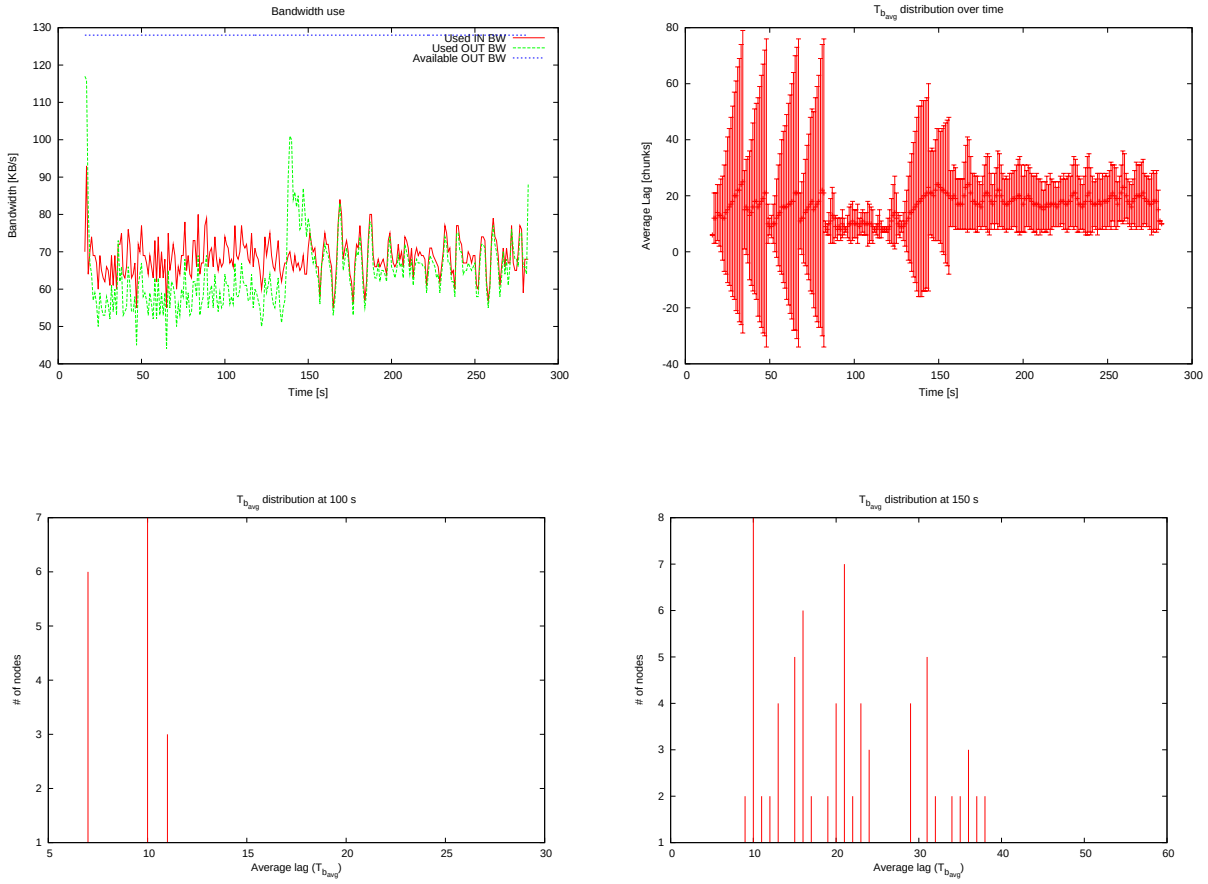


Figure 5.2: Symmetric spike arrivals

Increasing the number of peers

We have performed a series of experiments increasing the number of peers in the system in order to evaluate the scalability of our prototype implementation.

The scenario we used is the same of the previous tests where all peers have 1 Mbit/s upload bandwidth. The tests have been run with 10, 20, 30, 50, 70, 100 and 200 peers. We didn't manage to obtain more nodes from the Grid5000 testbed because of the high number of users during the days we performed the tests.

Figure 5.3 reports the relation between the $T_{b_{avg}}$ value and the number of peers in the system. The x axis represents the number of peers while the y axis represents the $T_{b_{avg}}$ value. For the x axis we use a logarithmic scale. It is possible to notice the logarithmic relation between the two variables. In fact the shape of the curve is almost a straight line that indicates a logarithmic increase of the $T_{b_{avg}}$ value in response to the increasing number of peers.

This is a real important results for our system. In fact, structured tree systems show the same kind of response to the increasing number of peers. We can thus conclude that the performances of our prototype, in term of scalability, are not lower than a structured tree system.

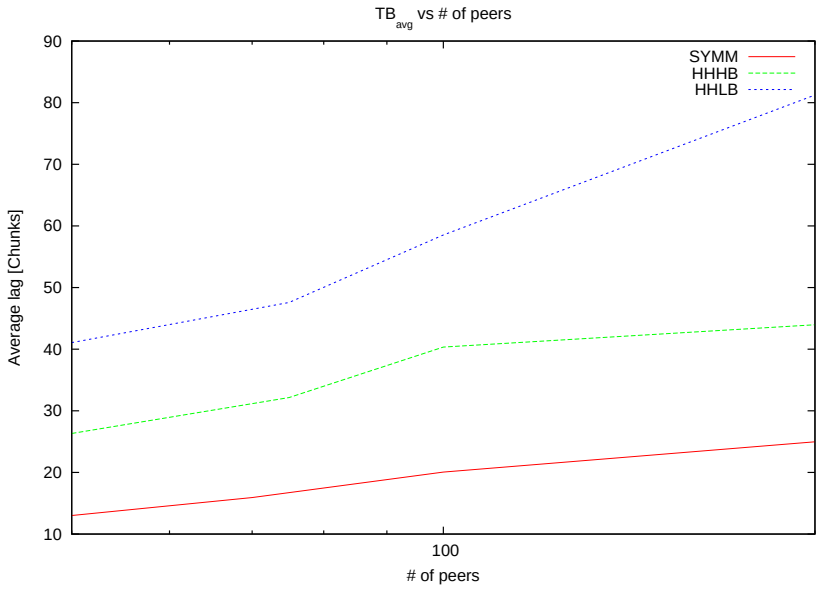


Figure 5.3: $T_{b_{avg}}$ vs number of peers

5.3.2 Planet-Lab

We performed an experiment on Planet-Lab in order to validate the functioning of our prototype with real network conditions.

Differently from the other experiences we decided to reduce the stream rate to 128 Kbit/s. This because there are many Planet-Lab nodes that present very strong limitation in the upload bandwidth. The number of peers involved in this experiment is 100. All peers arrive at the beginning at leave at the end of the run.

Since the unpredictable state (bandwidth limitations, CPU load etc.) of the Planet-Lab nodes we used other two plots to analyze the results. A first plot (Figure 5.4 (c)) shows the median reception delay of nodes using the 80-percentile to represent its fluctuations . This plot is used to clean the results from very poor node statistics. The second plot (Figure 5.4 (c)) is a representation of the average value of the reception delay without its variance. .

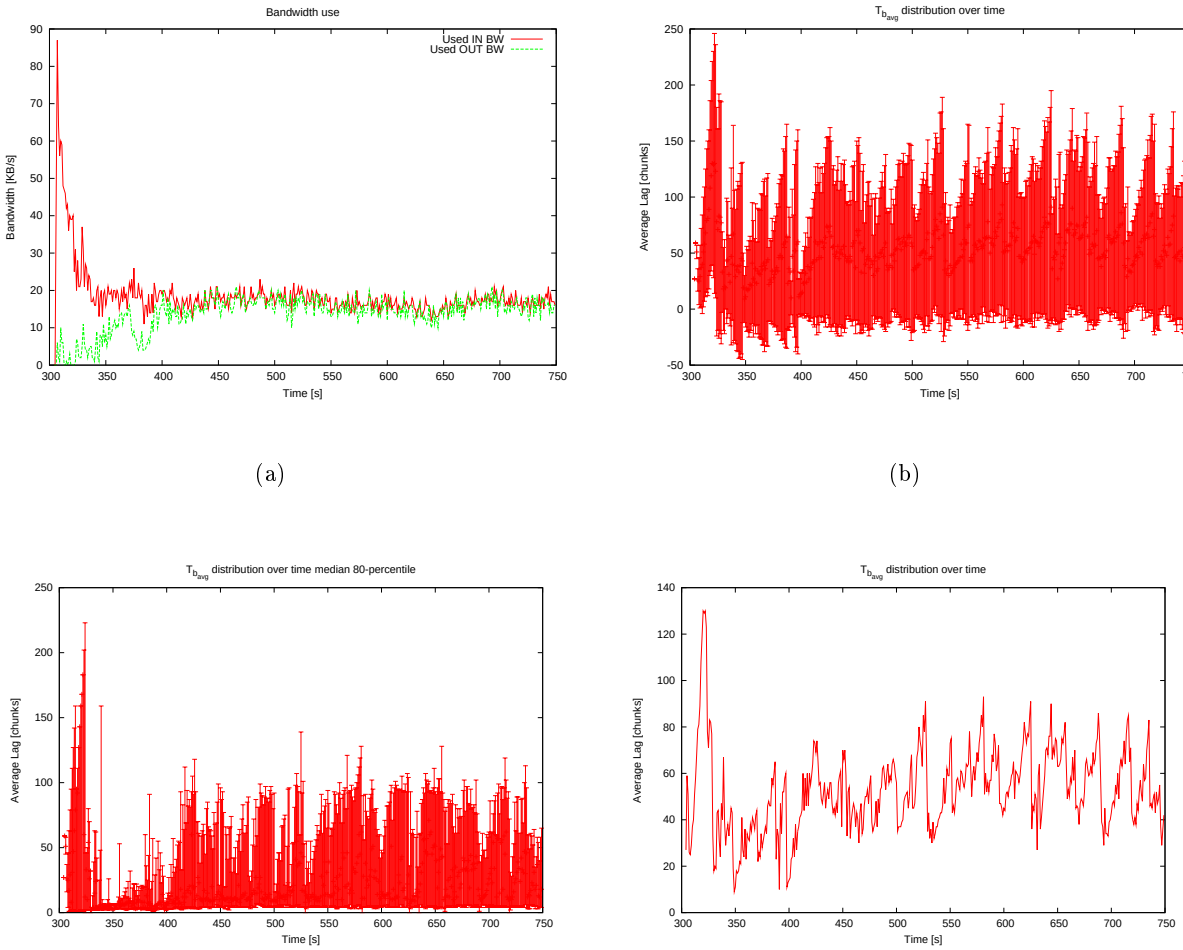


Figure 5.4: Planet-Lab

The results demonstrate that our prototype works in real network condition. The performances obtained are clearly not the same of Grid5000 tests since the presence of significant delays among nodes.

The peers maintain an average delay that is less stable with respect to the Grid5000 runs. This instability is probably due to the presences of some nodes with very low bandwidth capacities that are not able to maintain a constant delay value. As it is possible to see from the figure, peers maintain an average value fluctuating around 50 chunks while its variance is about 50 chunks. However almost all peers are able to stay connected to the system for all the experiment time. In fact only two nodes continuously connect/disconnect from the system.

By looking at Figure 5.4 (c) it is possible to see that the median delay value is about 25 chunks and that the 80% of peers present delay fluctuations around 25 chunks. This mean that if we don't consider the very poor nodes, the performances of the prototype with real network conditions are not so much different from the ones obtained in the Grid5000 runs.

The bandwidth use plots shows a number of duplicate chunks higher with respect of the Grid5000

experiments; in fact they are around 25% . This can be explained by the choice of the parameters. In fact we used the same values we set in the Grid5000 experiences but they should be tuned in order to obtain better performances.

By reducing the number of duplicates, we could probably obtain better performances also in term of delay.

5.3.3 Simulator scenarios and comparison

It is interesting to reproduce the same scenarios tested in the simulator in a real world testbed like Grid5000. By doing this we can compare the results and verify the simulator and core algorithms behaviors.

We decided to use Grid5000 in order to reproduce the simulator scenarios because we are sure that there are not external factors, such as high CPU load or insufficient upload bandwidth, that influences the results.

Low Heterogeneity High Bandwidth

Table 5.5 describes the configuration used for this scenario. The total number of clients used for this experiment is 100.

In this scenario the bigger amount of peers can upload at a rate equals to the stream rate while the 20% of peers have an upload rate equals to four times the stream rate. The total amount of excess bandwidth is around 60%. In this scenario it is possible to test the functionality of missing exchanges, but also the importance of forward exchanges should emerge. Peers of the VERY RICH class should provide to the peers of the NORMAL class a great amount of data, since their high upload bandwidth.

We have performed three runs with this scenario. A first test where all peers arrive at the beginning of the experiment and leave at the end. A second run where the 25% of peers arrive at the beginning of the experiment and the other 75% arrive after 120 seconds. This second test has been done in order to see how the system react to spike arrivals of peers with this bandwidth conditions. Finally we have performed a test where peer arrivals are uniformly distributed over the first 100 seconds of the experiment.

Class	% of peers	Upload Bandwidth
VERY RICH peers	20	4*SBR
NORMAL peers	80	SBR

Table 5.5: Scenario's description

The results obtained by our prototype are very similar to the results provided by the simulator. As it is possible to see in Figure 5.5 peers of both classes show an average reception delay that is constant. In the simulator both classes obtain the same average value that is around 20 chunks. In the prototype only peers belonging to the VERY RICH class are able to achieve the same performances while peers belonging to the POOR class have an average reception delay of 35 chunks. Moreover in the prototype experiments the variance of the reception delay is higher than the simulator one.

These differences are understandable because the control information is exchange via messages and not naturally obtained by peer selection. The decisions are taken by peers according to the

knowledge spread by these messages and they are thus less efficient than the decisions taken in the simulator. Moreover the time required to exchange control and data messages is not negligible as in the simulator. The performances should not be exactly the same.

However the peers that are advantaged by the prototype are the ones that give more to the system. This trend validates the tit-for-tat policy adopted by some of the Pulse algorithms. This policy has been defined in order to advantage peers that give more resources to the system. This is exactly what it is possible to see in this experiment.

Peers of both classes never disconnect from the system. They are able to absorb the fluctuations of their average reception delays and thus the stream play-out is not affected.

Very rich peers present 2.5 % of chunk loses while poor peers present 3.8 % of chunk loses. Thus both classes are able to playback the stream without losing video quality since the FEC we apply is able to recover up to 18% of losses.

It is possible to notice that in the prototype experiment there is a 10% of duplicate chunks that are not present in the simulator. Moreover peers of the VERY RICH class could probably serve more chunks to the NORMAL class in order to reduce its average reception delays. In fact VERY RICH peers don't exploit all their available upload bandwidth.

We have also performed an experiment with spike arrivals after 120 seconds. The 25% of the peers arrive at the beginning of the experiment while the other 75% arrive after 120 seconds. All clients quit the system at the end of the experiment. The total number of clients involved is still 100.

It is possible to observe that the system well react to the spike arrivals. In Figure 5.6 it is possible to see that, at the moment of the spike, the average reception delay of both classes increases until it reaches the values of the previous experiment. This is due to the growth of the number of peers in the system that move from 25 to 100. The variance of the average delay of NORMAL peers is not affected by the spike while the VERY RICH class shows a peak of variance during the new peers entrance. However there are no peers that disconnect from the system while in the simulator some peers' $T_{b_{avg}}$ reaches the disconnection threshold.

Figure 5.7 reports the results for the scenario where peer arrivals are uniformly distributed over the first 100 seconds of the experiment. There are not interesting trends emerging from this run.

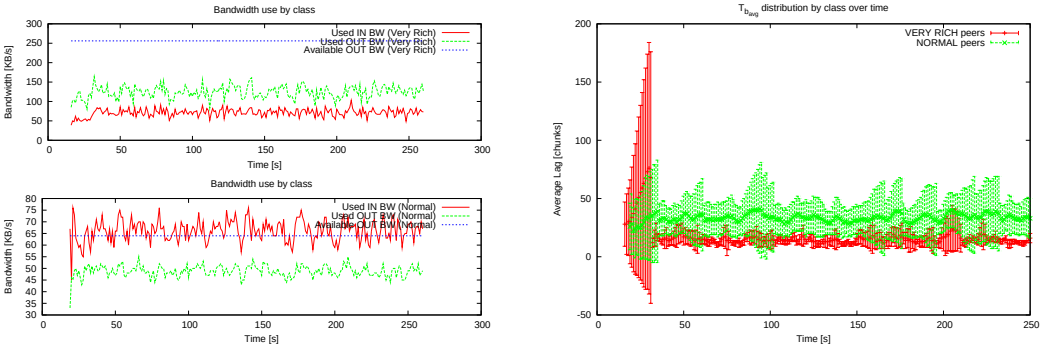


Figure 5.5: Low Heterogeneity High Bandwidth

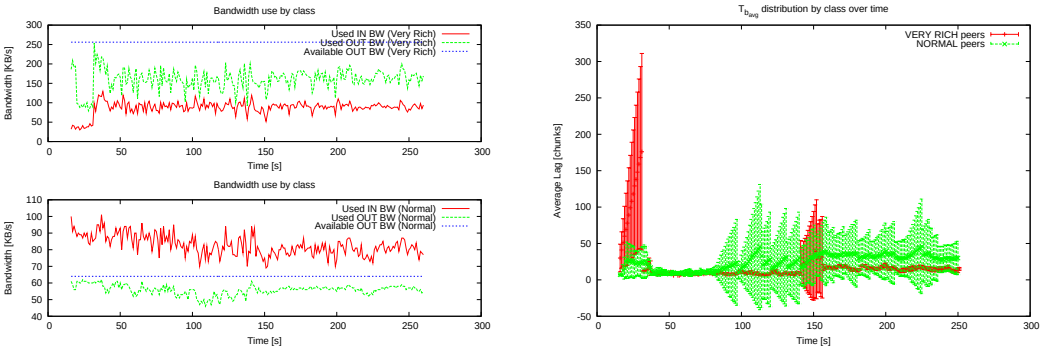


Figure 5.6: Low Heterogeneity High Bandwidth spike arrivals

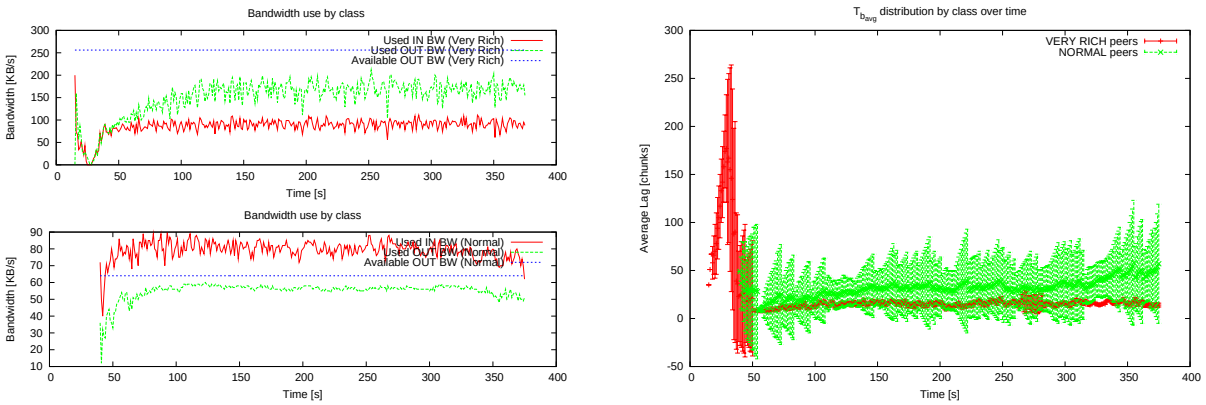


Figure 5.7: Low Heterogeneity High Bandwidth uniform arrivals

Low Heterogeneity Low Bandwidth

Table 5.6 describes the configuration used for this scenario. All peers arrive at the beginning of the experiment and leave at the end. The total number of clients used for this experiment is 100.

The total amount of excess bandwidth is around 20%. This scenario highlights the functionality of missing exchanges while the forward exchanges would probably be very limited because of the small bandwidth difference between the two classes.

Class	% of peers	Upload Bandwidth
RICH peers	20	2*SBR
NORMAL peers	80	SBR

Table 5.6: Scenario's description

The trends emerging from this experiment are quite similar to the ones of the previous scenario.

Here the very limited excess bandwidth increases the differences between the two classes penalizing even more the POOR peers. However, also peers of the RICH class seem to be a bit affected by the bandwidth scarcity since they present a bigger value of the average delays' variance with respect to the previous experiment (see Figure 5.8).

By looking at the bandwidth plot it is possible to notice that, even if there is less available bandwidth with respect to the previous experiment, peers of both classes are not able to completely exploit it. This is quite strange. In fact we are not in a content bottleneck state since the chunks rarity plot (Figure 5.9) shows the presence of many copies of the same chunks in the system.

The two holes present in the plot are due to the disconnection of 10 peers. In fact these peers don't receive a sufficient amount of data in order to maintain their $T_{b_{avg}}$ value under the disconnection threshold. These peers have thus problems in the play-out of the stream for few seconds. On the contrary, the other peers are able to always stay connected to the system without losing any second of the stream playback.

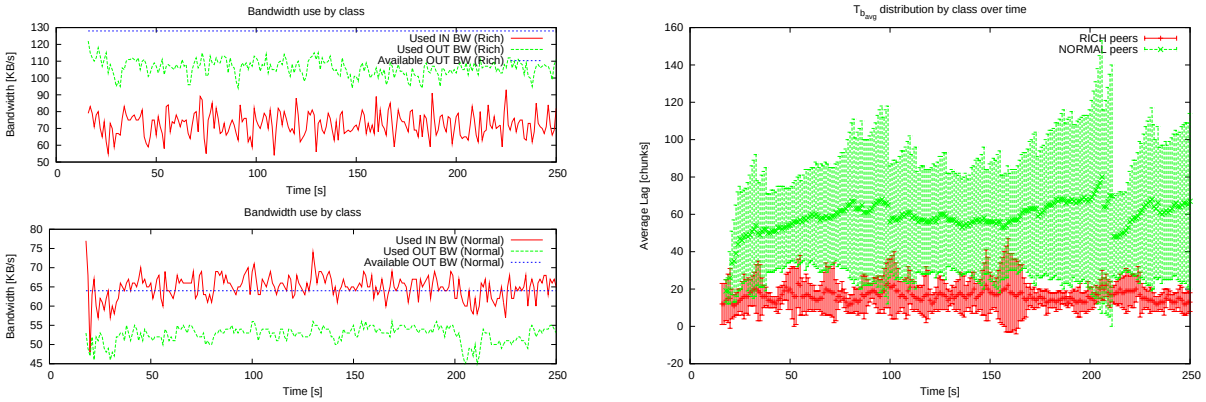


Figure 5.8: Low Heterogeneity Low Bandwidth

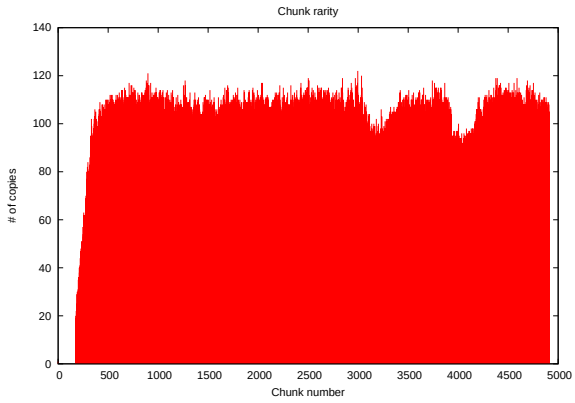


Figure 5.9: Chunk rarity

High Heterogeneity Low Bandwidth

Table 5.7 describes the configuration used for this scenario. The total number of clients used for this experiment is 100. This scenario is a worst case scenario, where the greater part of peers contribute less than the stream bit rate. The total amount of excess bandwidth in the system is of only 4%.

This scenario aims to test the functionality of forward and missing exchanges. In particular the former are really important because the richer classes should contribute their excess bandwidth to the poorer classes.

We have performed two runs for this scenario. In the first experiment all peers arrive at the beginning and leave at the end of the simulation. In the second run the 25% peers arrive at the beginning of the experiment while the others 75% arrive after 120 seconds.

Class	% of peers	Upload bandwidth
VERY RICH peers	4	4*SBR
RICH peers	20	2*SBR
NORMAL peers	21	SBR
POOR peeers	55	SBR/2

Table 5.7: Scenario's description

As it is possible to see in Figure 5.10 peers of the POOR class are not able to stabilize their average reception delay to a constant value. In fact, once they are connected to the system, their delay grows until it reaches the disconnection threshold. Indeed, peers of the other classes show a constant average reception delay. The Pulse algorithms seem to discriminate peers according to their class. In fact peers of the VERY RICH class have the lowest delay value followed by the peers belonging to the RICH class and then by NORMAL peers.

Peers belonging to these three classes can absorb the variation of their average delays thanks to the distance between $T_{b_{avg}}$ and T_v values. Moreover they are able to playback the stream without losing video quality because the percentage of chunks they lose is under the 18% threshold recoverable by the FEC mechanism.

If we look at the bandwidth consumption it is possible to notice that peers of the VERY RICH and NORMAL classes are not able to completely utilize their available bandwidth. They can serve more pieces but they don't manage in doing that. However, if we look at the chunk rarity plot, it is possible to observe that we are not in a content bottleneck state since there are many copies of the same pieces in the system. There is thus something that limits the bandwidth exploitation in our prototype implementation.

The holes present in Figure 5.12 are due to the disconnection around 25-30 peers of the POOR class. As previously said, they are not able to stabilize their reception delay, and they continuously connect/disconnect from the system.

We have also run the same scenario with spike arrivals. The 25% of the peers arrive at the beginning of the simulation while the other 75% arrive after 120 s. By this experiment it is possible to observe that the system well reach to the spike arrivals. In fact, after a period of instability due to the new arrivals, the system become newly stable as before the spike.

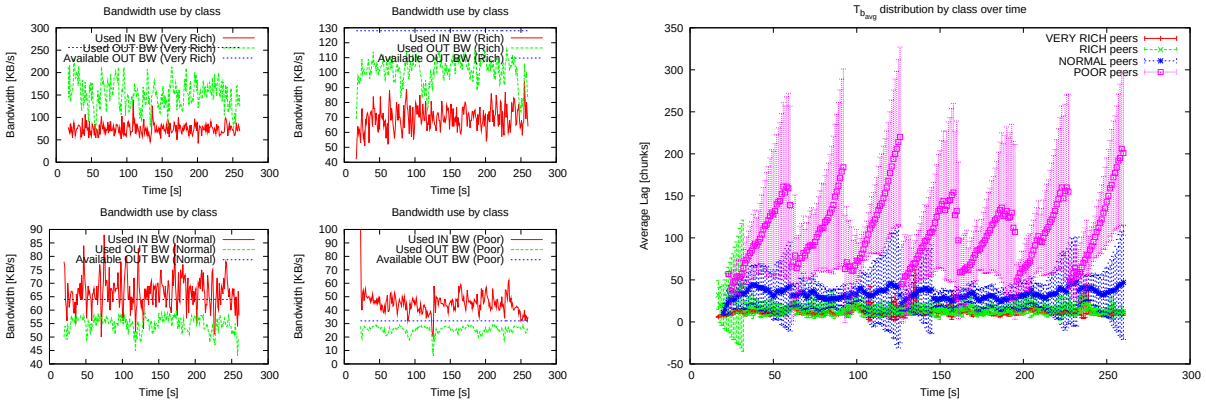


Figure 5.10: High Heterogeneity Low Bandwidth

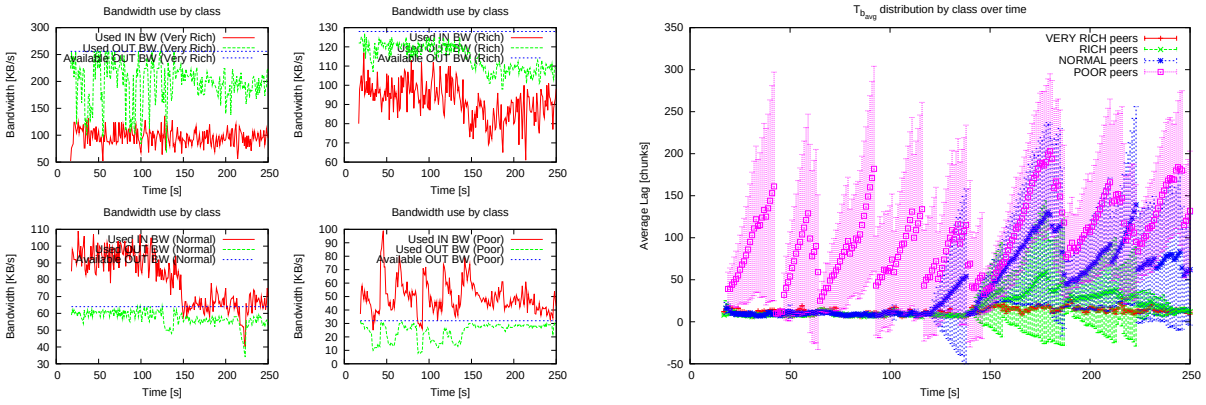


Figure 5.11: High Heterogeneity Low Bandwidth spike arrivals

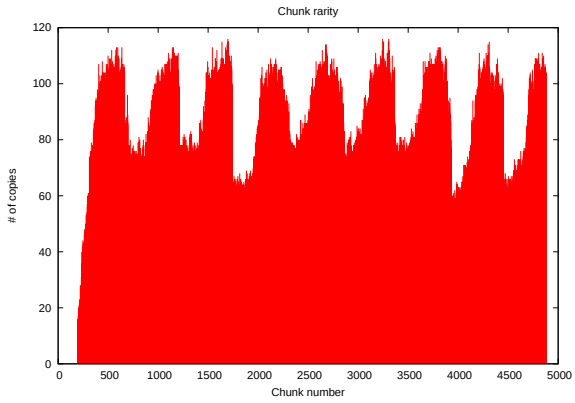


Figure 5.12: Chunk rarity

5.3.4 Conclusions

From the results obtained by the preliminary prototype tests we can validate the correct functioning of the core algorithms in a real world environment.

The *chunk selection mechanism* assures the piece diversity in the system. All pieces are replicated more ore less the same number of times allowing peers to receive every chunk before its play-out delay.

The *peer selection algorithms*, with forward and missing exchanges, well distribute chunks among all peers. In particular the tit-for-tat policy, implemented in the missing set selection, discriminates peers' behaviors advantaging the more generous peers. The forward exchanges provide to poorer peers the chunks they need to playback the stream content.

We cannot completely validate the *bandwidth allocation mechanism* since our prototype doesn't completely exploit all the available bandwidth of nodes. This seems not the fault of the mechanism itself but a problem in the prototype implementation. In fact, if we look at the bandwidth effectively used, it is possible to see that it is well distributed among the different classes of peers. Moreover this mechanism provide the stream content to peers on time for its play-out.

As concern the *simulator* we can conclude that it well predicts the behavior of the algorithms in a real world environment. In fact the results of the simulations are really similar to the ones of the prototype. The main differences are mostly due to the the not complete bandwidth exploitation of the prototype.

Finally we can validate the prototype implementation.

The *membership management protocol* we introduced, provides to all peers a blue knowledge peer set sufficiently large for a correct algorithms functioning.

The *FEC code* we decided to use is able to recover the peer's missing chunks. Of course it cannot work where peers continuously connect and disconnect from the system since the very limited number of chunks received.

The *synchronization mechanism* we implemented is able to always maintain peers synchronized with the source clock allowing a correct estimation of the chunk the source is transmitting. Moreover the RTT computed between hosts seems to correctly estimate the real RTT.

The *peer selection mechanism for red messages* seems to work pretty well since all nodes have a sufficient number of red knowledge's peers. Thus the peer selection and chunk selection algorithms can correctly work having such information.

The prototype has mainly two problems: the incomplete bandwidth exploitation by nodes and the presence of 10% of chunk duplicated.

The presence of duplicate chunks is related to the *data exchange mechanism*. In fact, if a provider cannot immediately satisfy a requests, it will send the requested chunk after the request timeout expiration. As a consequence the receiver peer asks for the chunk to another provider that newly send the same piece to it. This could probably be solved by a more careful parameters tuning. As explained in section 3.3.4, this tuning is not an easy problem to solve. It requires more experiences in order to find out the best settings.

However the main problem to solve in the prototype implementation is the exploitation of the available bandwidth. We have not clearly identified the real cause but we have some hypothesis.

The limitation could be related to a problem in the function responsible for the chunk distribution. Probably the way in which we have implemented this function doesn't permit a complete exploitation of the resources. In fact, in our experiments, we have artificially limited the number of chunks a peers can send in one second. The way in which we limit the rate could reduce even more the threshold we fix.

The problem could be related to the data exchange mechanism. The way in which we manage the chunk requests could limit the number of packets sent by every peer. Or, once again, the parameters we choose for our experiments are not well tuned in order to completely exploit the bandwidth resources.

Otherwise the problem could be due the number of FORWARD slots. We used the same value chosen for the simulation. Probably in a real world environment is necessary to change it in order to improve performances.

Chapter 6

Conclusions

In this this work we presented the implementation of a prototype of Pulse, an unstructured p2p system for live steaming. We also analyzed the Pulse performances in real network environments.

The bigger amount of the time has been devoted to the implementation of the Pulse prototype. We first pointed out what were the issues not addressed in the Pulse design in order to realize a real prototype. We made a research about existing solutions to problems like membership management, peer's synchronization and coding mechanism. Then we designed algorithms and protocols to face these issues in a system like Pulse. Moreover we modified some of the Pulse core algorithms in order to adapt them to a real world environment.

Finally, we presented some preliminary results about the Pulse performances analyzing the results obtained by our experiments. The Pulse prototype performs well in different conditions. The peers distributed the overall bandwidth of the system among them in order to provide to all nodes a sufficient download rate for the playback of the stream. The nodes maintain a small average delay from the source and correctly playback the stream. The system well scales to a big number of nodes showing a logarithmic trend in response to the increasing number of peers.

6.1 Open issues

There are two main problems that emerged from the analysis of the prototype's performances: the duplicate chunks and the incomplete exploitation of the available bandwidth.

In order to face duplicates problem more tests should be performed in order to better tune all the parameters related to Pulse algorithms. By doing this we hope to reduce the percentage of duplicates to less than 5%.

As concerns the bandwidth limitation we are going to better analyze the problem in order to find out its real cause and to fix it.

6.2 Future work

First of all we are going to fix the problems emerged in the experiments. Once they will be solved, other experiments will be performed in order to analyze the prototype in its final version. In

particular more experiments reproducing the simulator scenarios and using the Planet-Lab testbed will be run.

We are also going to implement the last features that are not present yet in our prototype in order to have a complete real application. In particular we are going to implement an interface to a player that nowadays has only be designed. Once a complete version will be ready, we hope a public version of Pulse will be released.

Bibliography

[1] Fabio Pianese , "P2P Live Media Streamig: Delivering Data Streams to Massive Audiences within Strict Timing Constraints", master thesis, September 2004

[2] Xinyan Zhang, Jiangchuan Liu, Bo Li, Tak-Shing Peter Yum, "CoolStreaming/DONet: A Data-driven Overlay Network for Efficient Peer-to-Peer Live Media Streaming", in Proceedings of IEEE Infocom, 2005

[3] Fabio Pianese, Joaquin Keller, Ernst Biersack, "Pulse, a flexible P2P Live Streaming System",Proceedings IEEE Infocom, April 2006

[4] Xiaofei Liao, Hai Jin, Yunhao Liu, Lionel M. Ni, Dafu Deng, "AnySee: Peer-to-Peer Live streaming",in Proceedings of IEEE Infocom, April 2006

[5] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, Laurent Massoulie, "Peer-to-Peer Membership Management for Gossip-Based Protocols", in IEEE Trans. on Computers 52(2), 2003, pp. 139-149

[6] Vivek Vishnumurthy, Paul Francis, "On etherogeneous Overlay construction and Random Node Selection in Unstructured P2P Networks", Proceedings IEEE Infocom, April 2006

[7] Reza Rejaie, Shad Stafford, "A Framework for Architecting Peer-to-Peer Receiver-driven Overlays", Cork, Ireland, June 2004

[8] Nazanin Magharei, Amir H. Rasti, Daniel Stutzbach,"Peer-to-Peer Receiver-driven Mesh-based streaming", Proceeding of the ACM SIGCOMM, Poster Session, August 2005

[9] Reza Rejaie, Antonio Ortega, "PALS: Peer-to-Peer Adaptative Layered Streaming", Monterey, California, June 2004

[10] Nazanin Magharei,Reza Rejaie,"Understanding Mesh-based Peer-to-Peer Streaming", Technical Report CIS-TR-06-02, University of Oregon, February 2006

[11] Bernard Wong, Aleksandris Slivkins, Emin Gun Sirer, "Meridian: A Lightweight Framework for Network Positioning without Virtual Coordinates",In Proceedings of SIGCOMM Conference, Philadelphia, August 2005

[12] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, Atun Singh, “SplitStream:High-Bandwidth Multicast in Cooperative Environments”,19th ACM Symposium on Operating Systems Principles, 2003

[13] Dejan Kostic, Adolfo Rodriguez, Jeannies Albrecht, Admin Vahdat, “Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh”,in Proc. of ACM SOSP, 2003

[14] Dejan Kostic, Adolfo Rodriguez, Jeannies Albrecht, Admin Vahdat, "Using Random Subset to Build Scalable Network Services", Proc. 4th USENIX Symposium on Internet Technologies and Systems (USITS),March 2003

[15] Marc Schiely, Pascal Felber, "CROSSFLUX: A Self-Adaptive System for Peer-to-Peer Media Streaming", under submission

[16] Luigi Rizzo, "Effective Erasure Codes for Reliable Computer Communication Protocols", Computer Communication Review, 27(2):24–36, April 1997.

[17] James S. Plank, Ying Ding "Note:Corection to the 1997 Tutorial on Reed Solomon Coding", Technical Report UT-CS-03-504, University of Tennessee, April, 2003.

[18] Christos Gkantsidis, Pablo Rodriguez Rodriguez, "Network coding for Large Scale Content Distribution",IEEE Infocom 2005

[19] Thinh Nguyen, Avideh Zakhor, "Distributed Video Streaming with Forward Error Correction", in Proc. Packet Video Workshop, Pittsburgh, USA, April 2002

[20] Miguel Castro, Manuel Costa, Antony Rowstron, "Debunking some myths about structured and unstructured overlays", NSDI'05, Boston, MA, USA, May 2005.

[21] D.L.Mills, "Network Time Protocol", RFC 958 September 1985

[22] Reza Rejaie, Shad Stafford, "A Framework for Architecting Peer-to-Peer Receiver-driven Overlays", in Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video, Cork, Ireland, June 2004

[23] Suman Banerjee, Bobby Bhattacharjee, Christopher Kommareddy, "Scalable Application Layer Multicast", Technical report, UMIACS TR-2002.

[24] Duc A. Tran, Kien A. Hua,Tai T. Do "A Peer-to-Peer Architecture for Media Streaming", IEEE Journal on Selected Areas in Communications, Special Issue on Service Overlay Networks, January 2004

[25] Christina Fragouli, Jean-Yves Le Boudec, Jorg Widmer, "Network Coding: An Instant Primer", LCA-REPORT-2005

[26] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In Proc. of the 3rd Intl. COST264 Workshop on Networked Group Communication (NGC 2001), London, UK, 2001.

[27] Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for largescale peer-to-peer systems. In: IFIP/ACM Intl. Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany (2001)

[28] Anh-Thuan Gai, "Structuration en graphe de de Bruijn ou par incitation dans les reseauix pair à pair", These de doctorat, Juin 2006

[29] Suman Banerjee, Bobby Bhattacharjee, "A Comparative Study of Application Layer Multicast Protocols", Under submission

[30] A. Nicolosi and S. Annapureddy. P2PCAST: A peer-to-peer multicast scheme for streaming data. 1st IRIS Student Workshop (ISW'03)

[31] BitTorrent, *http://www.bittorrent.com/*

[32] Solipsis, *http://solipsis.netofpeers.net/*

Appendix A

Protocol message format

Pulse header

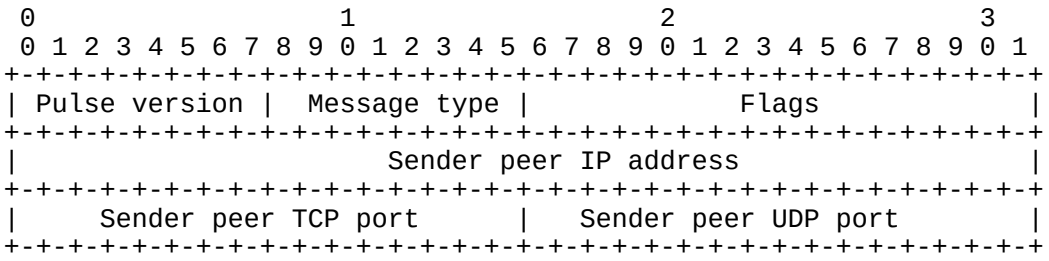


Figure A.1: Pulse message header

Pulse version = this field indicates the Pulse protocol version used by the sender peer.

Message type = this field is used to distinguish the different messages of the Pulse protocol.

Flags. There are two flags that are used in our prototype implementation: the *connected* flag and the *contact* flag. Their functions are explained in the messages where they are used.

Sender peer IP address: this is a 32-bit field used to indicates the IP address of the peer who sent the message.

Sender peer TCP port: this is a 16-bit field used to indicates the port used for incoming TCP connections by the sender peer.

Sender peer UDP port: this is a 16-bit field used to indicates the port used for incoming UDP connections by the sender peer.

Hello message

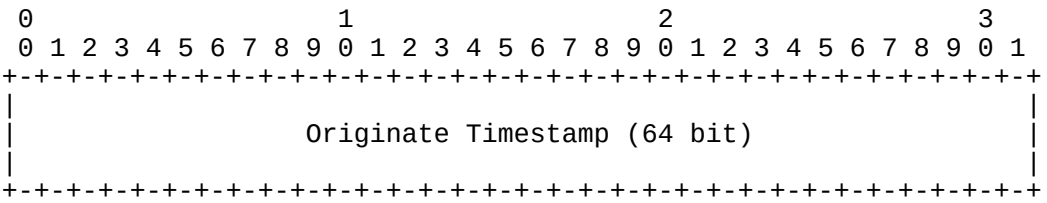


Figure A.2: Hello message

Originate timestamp: this is a 64-bit timestamp that indicates the time at which the message has been sent by the sender peer.
The *contact* flag is used in this message to indicate if the receiver peer is the contact for the sender peer.

Blue message

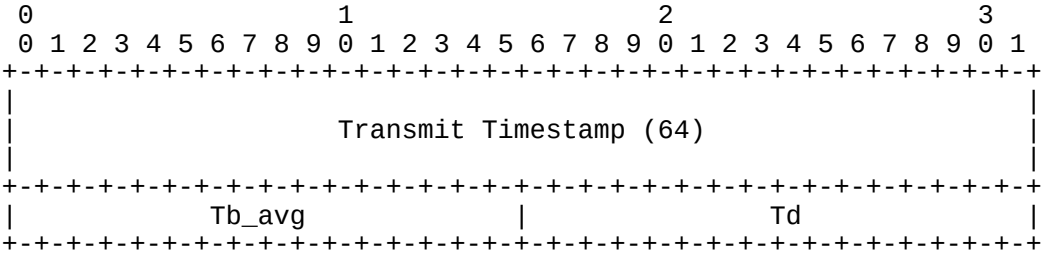


Figure A.3: Blue message

Transmit timestamp: this is a 64-bit timestamp that indicates the time at which the blue message is sent by the local peer.
$T_{b_{avg}}$: this is a 16-bit integer that indicates the $T_{b_{avg}}$of the sender peer.
T_d: this is a 16-bit integer that indicates the T_d value of the sender peer.

Bye message

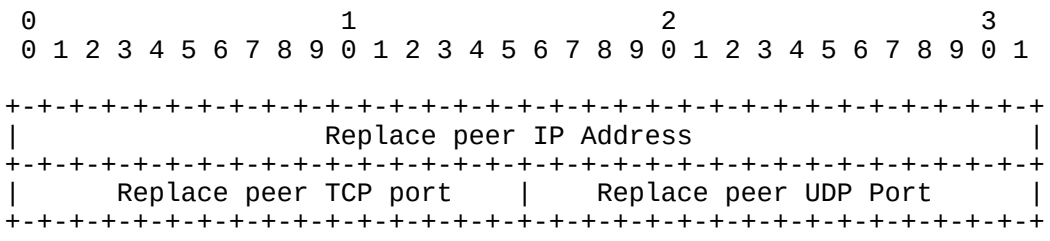


Figure A.4: Bye message

Replace peer IP address: this is a 32-bit field used to indicate the address of the peer that will sobstitute the sender peer in the receiver peer partial view.
Replace peer TCP port: this is a 16-bit field used to indicate the incoming TCP connection port of the peer that will sobstitute the sender peer in the receiver peer PartialView.
Replace peer UDP port: this is a 16-bit field used to indicate the incoming UDP connection port of the peer that will sobstitute the sender peer in the receiver peer PartialView.

Syncronization message

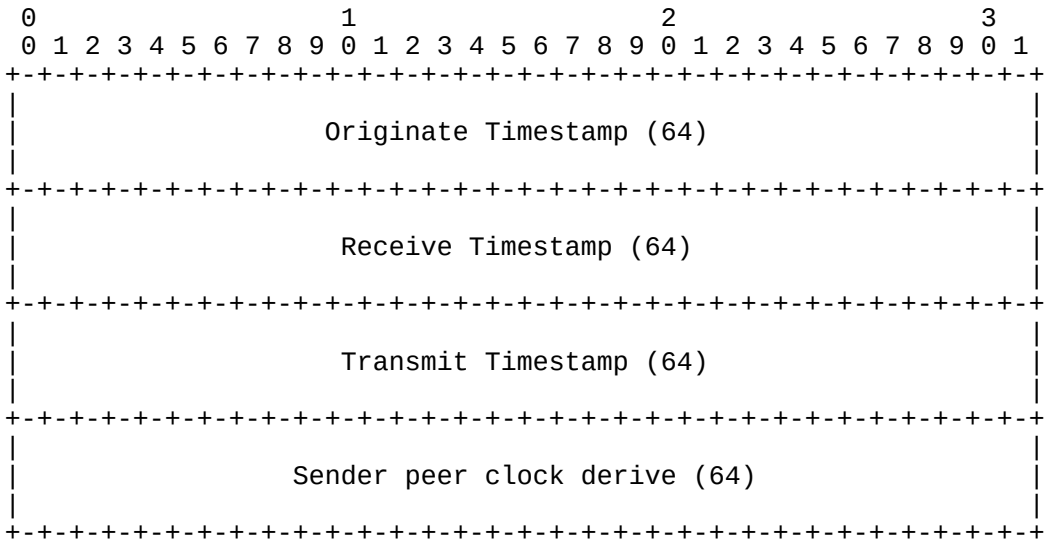


Figure A.5: Synchronization message

Originate timestamp: this is a 64-bit timestamp that indicates the time at which the hello message has been sent by the new peer.
Receive timestamp: this is a 64-bit timestamp that indicates the time at which the hello message has been received by the local peer.
Transmit timestamp: this is a 64-bit timestamp that indicates the time at which the synchronization message is sent by the local peer.
Sender peer clock derive: this is a 64-bit timestamp that indicates the derive of the local peer clock as respect to the source clock.

Red message

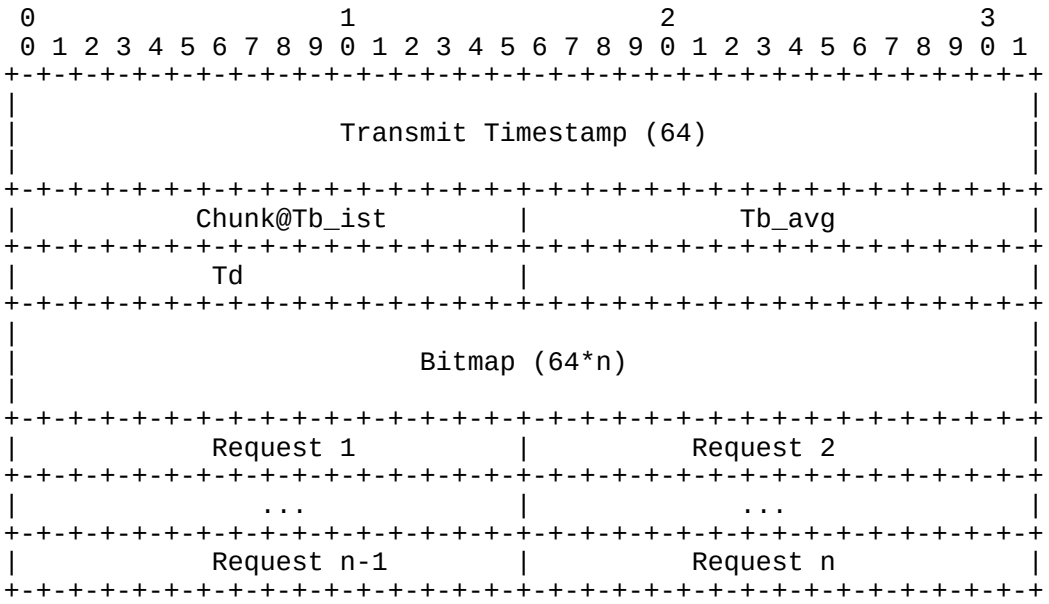


Figure A.6: Red message

Transmit timestamp: this is a 64-bit timestamp that indicates the time at which the red message is sent by the local peer.
Chunk@$\mathcal{T}_{b_{ist}}$: this is a 16-bit integer that indicates the chunk number of the chunk with a lag equal to the $T_{b_{ist}}$value of the sender peer.
$\mathcal{T}_{b_{avg}}$*:* this is a 16-bit integer that indicates the $T_{b_{avg}}$ value of the sender peer.
T_d :this is a 16-bit integer that indicates the T_d value of the sender peer.
Bitmap: this is a sequence of bits representing the trading window bitmap of the sender peer. The bits can assume 0 or 1 value. If 1 the bit indicates that the sender peer owns the chunk in the corrisponding position of the trading window. The length of the bitmap field can vary according of the length of the trading window. This length is written in the pulse file.
Request i: this is a 16-bit integer that indicates one of the chunks requested by the sender peer to the receiver peer. The maximum number of chunk to request is written in the pulse file. If there are not enough requests the empty fields are filled with 0 values or padding.
The *connected* flag is used in this message to indicate whether the sender peer has a sufficient number of chunks in its buffer to be considered connected to the system or not.

Data message

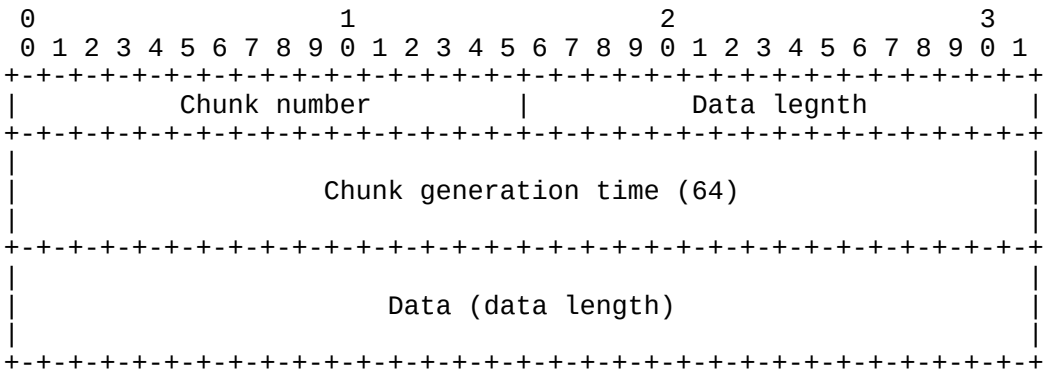


Figure A.7: Data message

Chunk number: this is a 16-bit integer indicating the chunk number of the chunk that is transmitted in this packet.
Data length: this is a 16-bit integer indicating the number of bytes of the data content of the chunk that is transmitted in the message.
Chunk generation time: this is a 64-bit timestamp indicating the time at which the chunk transmitted in this packet has been generated by the source.
Data: this field contain the data content of the chunk. Its length is defined in the pulse file and it also expressed in the data length.

Appendix B

Pulse files

Example of ".pulse" file

Stream_initial_time 1155815656.75

Chunk_size 8000

Stream_rate 128000

Sliding_window_size 32

Tk 100

Admitted_packet_lost 6

Value_of _Tb_at_bootstrapt 56

Max_value_for_Tb_at_bootstrap 64

Reconnect_threshold 394

Interval_of_time_used_to_compute_Tb_avg 3

Contact_IP 172.24.11.33

Contact_TCP_port 2222

Contact_UDP_port 2224

Example of node configuration's file

Red_message_rate 16

Blue_message_rate 0.01

Max_upload_rate 0

Max_download_rate 0

TCP_port 2225

UDP_port 2226

Max_number_of_peers_in_red_message_list 16

Max_number_of_missing_peer 4# first round select providers that are serving the local peer

Max_number_of_missing_peer_selected_by_tft 3

Max_number_of_forward_peers 8

Epoch_duration 2

History_score_initial_value 10

Total_max_requests 8

Max_peer_to_send_requests 4

Max_requests_to_send_to_each_peer 2

Request_timeout 0.25

Inteval_of_time_used_to_average_the_rate 2.0

Max_number_of_times_to_forward_a_Blue_msg 1

Max_number_of_time_to_forward_an_Hello_msg 2

Heartbeat_timeout_blue 1000

Reconnection_timeout 50000

Red_knowledge_timeout 1.0

Blue_knowledge_timeout 1000.0