Alma Mater Studiorum · Università di Bologna

facoltà di scienze matematiche, fisiche e naturali

Corso di Laurea Magistrale in Informatica

STRUCTURED P2P VIDEO STREAMING AND COLLABORATIVE FAILURE DETECTION

Tesi di Laurea in Sistemi Distribuiti

Relatore:
Chiar.mo Prof.
VITTORIO GHINI

Presentata da:
PASQUALE PUZIO

Sessione II

Anno Accademico 2011/2012

Contents

1 Introduction **9**

1.1 Benefits of P2P Video Streaming 9

1.2 Requirements for P2P Streaming 10

1.3 P2P Streaming Taxonomy 11

1.4 Performance Criteria for P2P Live Streaming 15

1.5 Challenges and Issues 16

2 Objectives **19**

3 System Architecture **21**

3.1 Hypercube Data Structure 21

3.2 Dissemination Rule 23

3.3 Adaptation of Hypercube 26

3.4 Framework for P2P Streaming 28

3.4.1 Software Entities 29

3.4.2 File Organization 31

3.4.3 Data Structures 32

3.4.4 Data Structures for Failure Detection 38

3.4.5 UDP Packet Format 41

3.4.6 Block Types 43

3.5 UDP Encapsulation/Decapsulation and Proxy-like Behavior 53

4 Population Variation **55**

4.1 Departures 55

4.2 Arrivals 58

5 Failure Detection **61**

5.1 Detection Strategy 64

5.1.1 Implementation Details 68

5.1.1.1 Piggyback 68

5.1.1.2 Functions for Failure Detection Implementation . . . 69

5.2 Recovery Strategy 74

6 Simulations and Experiments **76**

6.1 Experiments on PlanetLab 76

6.2 Experiments for Failure Detection 80

7 Summary of Work Done **82**

8 Conclusions **84**

9 Future Work **85**

Acknowledgements

I would like to sincerely thank my supervisor, Patrick Brown, for the wonderful chance to work at Orange Labs and the constant support he gave me. It was a pleasure to work with him and spend much time discussing together on several aspects of my internship. I really liked his approach to research and his friendly way to interact with me. At the same time, he also managed to provide a great guidance and strong motivations which have been fundamental for the achievement of all objectives declared at the beginning of the internship.

Thanks to this experience, I found out my taste for research, so I can surely say that it has been the most important experience I have ever had in my professional life. I will keep for ever good memories of good times I spent at Orange Labs.

I have to thank also my parents for giving me all the necessary support to make my own choices and pursue this important goal.

Last but not the least, I would like to thank also my classmates and teachers for making the Ubinet Master a great learning experience.

Abstract

Il video streaming in peer-to-peer sta diventando sempre più popolare e utilizzato. Per tali applicazioni i criteri di misurazione delle performance sono:

- startup delay: il tempo che intercorre tra la connessione e l'inizio della riproduzione dello stream (chiamato anche switching delay),
- playback delay: il tempo che intercorre tra l'invio da parte della sorgente e la riproduzione dello stream da parte di un peer,
- time lag: la differenza tra i playback delay di due diversi peer.

Tuttavia, al giorno d'oggi i sistemi P2P per il video streaming sono interessati da considerevoli ritardi, sia nella fase di startup che in quella di riproduzione [8]. Un recente studio [5] su un famoso sistema P2P per lo streaming, ha mostrato che solitamente i ritardi variano tra i 10 e i 60 secondi. Gli autori hanno osservato anche che in alcuni casi i ritardi superano i 4 minuti! Si tratta quindi di gravi inconvenienti se si vuole assistere a eventi in diretta o se si vuole fruire di applicazioni interattive.

Alcuni studi hanno mostrato che questi ritardi sono la conseguenza della natura non strutturata di molti sistemi P2P [3, 4]. Ogni stream viene suddiviso in blocchi che vengono scambiati tra i peer. A causa della diffusione non strutturata del contenuto, i peer devono continuamente scambiare informazioni con i loro vicini prima di poter inoltrare i blocchi ricevuti. Queste soluzioni sono estremamente resistenti ai cambiamenti della rete, ma comportano una perdita notevole in termini di prestazioni, rendendo complicato raggiungere l'obiettivo di un broadcast in realtime.

In questo progetto abbiamo lavorato su un sistema P2P strutturato per il video streaming che ha mostrato di poter offrire ottimi risultati con ritardi molto vicini a quelli ottimali. In un sistema P2P strutturato ogni peer conosce esattamente quale blocchi inviare e a quali peer. Siccome il numero di peer che compongono il sistema potrebbe essere elevato, ogni peer dovrebbe operare possedendo solo una

conoscenza limitata dello stato del sistema. Inoltre il sistema è in grado di gestire arrivi e partenze, anche raggruppati, richiedendo una riorganizzazione limitata della struttura.

Infine, in questo progetto abbiamo progettato e implementato una soluzione personalizzata per rilevare e sostituire i peer non più in grado di cooperare. Anche per questo aspetto, l'obiettivo è stato quello di minimizzare il numero di informazioni scambiate tra peer.

Abstract

P2P broadcasting of video streams is increasingly popular. Important performance criteria on such applications are related to:

- startup delay: delay before start of playback when connecting to the stream (also called switching delay),
- playback delay: delay between source transmission time and peer playback,
- time lag: differences in playback delays between two peers.

However today's most popular P2P live streaming systems suffer from long startup and playback delays [8]. A recent measurement study [5] over a popular P2P streaming system shows that typical startup and playback delays vary from 10 to 60 seconds. The authors also observe that, for most streams, some playback delays exceed 4 minutes! These are serious drawbacks for watching realtime events or using these applications for interactive streams.

Some studies have shown that these delays are a consequence of the mesh or unstructured nature of these P2P systems [3], [4]. A stream is divided into chunks that peers exchange. As a consequence of the unstructured dissemination, peers must constantly exchange information with their neighbors before forwarding the stream. These solutions are extremely resistant to network changes, but at the cost of losing their initial realtime broadcasting objective.

In this project we consider a structured P2P streaming system which has shown to offer great results with delays very close to the optimal ones. In a structured P2P system each peer knows exactly what block to send and to which peer. As population size may be very large, each peer should know what to do with a limited knowledge of the system. In addition, the system allows for arrivals and departures, even grouped, with limited reorganizations.

Furthermore, in this project we designed and implemented a custom solution to detect non-collaborative peers and replace them. Even for this aspect, the aim is to minimize the amount of packets exchanged between peers.

Chapter 1

Introduction

Peer-to-Peer (P2P) video streaming is a software-based solution that allows users to distribute data to a larger audience without using a single or limited set of central servers.

1.1 Benefits of P2P Video Streaming

P2P video streaming is extremely useful when the goal is to avoid system overload due to the high and unexpected number of simultaneous requests. In order to support an extremely large population, a large number of servers is needed, and in case the population increases more than we could expect, the system is not able to fulfill all the requests. Ideally, a P2P system is supposed to be able to scale and adapt its capacity to current requirements.

Furthermore, the population of a streaming system may be highly dynamic over the time, this means that if we use a centralized system, most of the time there will be a waste of resources due to the inactivity of some servers. A P2P system is much more efficient because every peer collaborates for content distribution, so there is no point in the system that allocates resources without using them at all.

We can clearly deduce that a P2P allows to dramatically reduce the cost for broadcast and make much simpler for any user to globally distribute a content

globally without any dedicated infrastructure.

More generally, P2P solutions are increasingly used by enterprises to provide high-quality and reliable services, especially for streaming, that can efficiently deal with limits of their infrastructures.

1.2 Requirements for P2P Streaming

In order to deploy a P2P system for streaming, there are some requirements that need to be fulfilled in order to make the system working in the proper way:

- **Time Constraint:** each piece of content we want to transmit, called block or chunk, has a playback deadline by which it has to be delivered to the destination peer. This means that a P2P system must meet a set of real-time requirements. In other words, if a peer receives a block after the maximum allowed time, that block can be considered lost because it is not useful anymore (not playable in case of video streaming).

- **Scalability:** potentially, a live streaming system may have a very high number of peers interested in transmitted content. A P2P system is ideally able to deal with very large populations without affecting the quality of the service achieved by connected peers.

- **Heterogeneity:** in the real world, it is very unlikely to have a set of peers with a homogeneous capacity, in terms of computational power, upload and download bandwidth, etc. A P2P system is thus supposed to be able to provide the best quality of service possible despite to the heterogeneity of peer resources.

- **Grouped Arrivals and Departures:** usually, it happens very often that a large number of peers wants to join (or leave) the system simultaneously.

This phenomena must not affect the normal working of the system. More practically, connected peers must not experience any additional delay or loss.

- **Decentralized Architecture:** any P2P streaming, according to its definition, is supposed to be resistant to multiple failures, so there must not be any possibility to have a single point failure or a bottleneck situation.

1.3 P2P Streaming Taxonomy

There are two main factors that characterize P2P systems:

- **Topology:** the structure, if any, adopted to organize the overlay network and create relations between peers.
- **Delivery strategy:** the way the system distributes pieces of content among peers. Each peer thus forwards pieces of content according to rules defined by the delivery strategy.

Nowadays, two approaches can be considered the starting point to design any other approach:

- **Tree-based or Structured:** in this approach, peers are organized according to a predefined structure and each peer has a precise position in that structure. Content is pushed from parent to child. Even relations and communication with other peers depend on the position of the peer in the structure. Generally, in this kind of systems there is an entry point which is in charge of placing a just joined peer into the structure.
- **Mesh-based or Unstructured:** in this approach, there is no predefined structure, peers establish relations and communicate by only exchanging information with each other. As we can easily imagine, a larger amount of data needs to be exchanged compared to structured P2P systems, in order to keep peers synchronized.

Generally, tree-based approaches prove to be scalable because they can easily handle very large populations, but they suffer from several factors. Indeed, they prove to be not robust in case of failures or nodes with poor resources. Indeed, the basic problem affecting tree-based approaches is the need of expensive re-structuring in case a peer stops cooperating or his upload capacity gets worse. Re-structuring may force peers to close ongoing connections with neighbors and run once again the procedure to get in the structure and change position. This could require to get aware of their new neighborhood and establish connections with new neighbors. We can state that the most important requirement for a structured P2P system is to repair the structure, in case of a peer churn, as efficiently as possible. In this case, efficient means minimizing both bandwidth and time.

Furthermore, they prove to be also not fair because a free-rider can easily make worse the quality of service provided to other peers and keep achieving a good quality for himself. However, they allow to achieve an interesting advantage: the implementation usually is much simpler.

Tree-based approaches can also adopt multiple trees. Peers belong to multiple trees at the same time but they can be internal nodes in at most one tree, in all the others they are leaves. In this kind of systems the stream is encoded in multiple sub-streams in order to ensure a minimum service quality even if some packets are lost. This helps also to handle bandwidth heterogeneity among peers. Another reason why multi-trees approach has been designed is to maximize cooperation and fairness between peers. Indeed, in a classic single-tree approach, peers in the last level (leaves) do not cooperate at all, they just receive content without collaborating for its distribution. To the best of our knowledge, there is no commercial system based on a tree-based structure.

In mesh-based approaches peers receive content by sending request packets to their neighbors. Each peer maintains a limited and random set of peers. In a pure

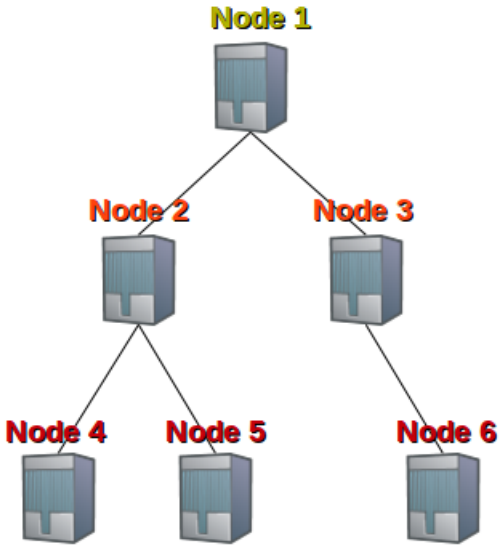

Figure 1.1: An example of tree-based P2P system

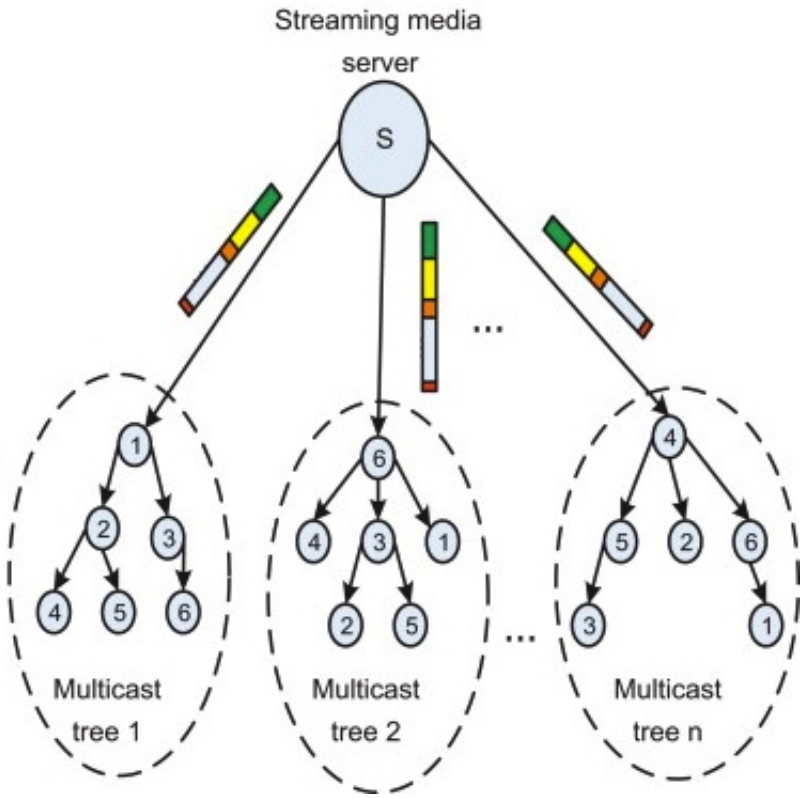

Figure 1.2: An example of multi-tree P2P system

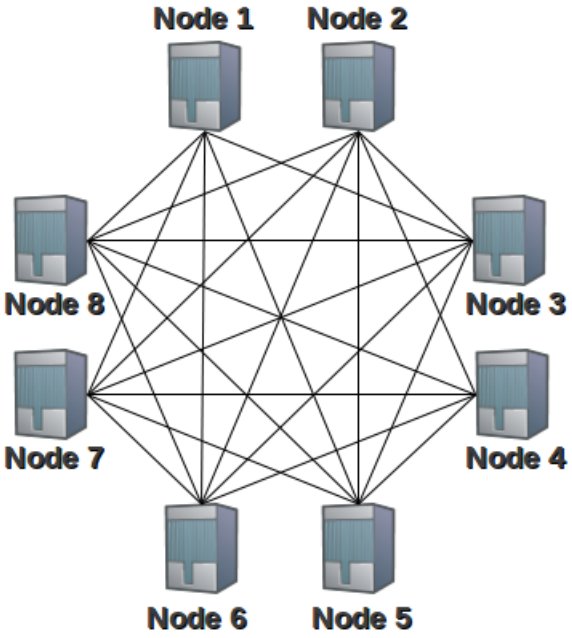

Figure 1.3: An example of mesh-based P2P system

mesh-based approach each peer selects a random set of peers and sends to them all the available blocks. Unfortunately, this approach generates a high waste of bandwidth because of duplicates. Indeed, it is very common that a peer receives a given block more than once.

Clearly, mesh-based approaches prove to be much more robust and adaptive compared to structured approaches. Differently from structured systems, a mesh-based system is completely based on cooperation, that is the reason why peers ideally have the tendency to create groups according to their capability to cooperate for global content distribution. This means that stronger peers with a high capacity (for both upload and download) will achieve a higher quality.

On the other hand, as in mesh-based P2P systems there is no structure and no specific rules for content distribution, peers need to continuously exchange synchronization packets (in some systems they are called request packets) in order to

efficiently distribute content. The most important task of a peer is to download (pull) blocks as quickly as possible from multiple neighbors. Both peer selection and piece selection has to be performed in such a kind of system.

1.4 Performance Criteria for P2P Live Streaming

Nowadays, most popular P2P live streaming systems suffer from extremely high delays. The most important performance criteria that characterize a P2P live streaming system are the following:

- **Startup Delay:** it is the time interval starting from the moment in which a peer connects to a online stream until the first piece of content has been received and is usable. In case of a video streaming system, the startup delay is considered as the interval between the connection request and the reproduction of the first video frame. In structured P2P topologies, this happens especially because when a peer asks to join the system, he needs to be assigned a position in existing structure and as soon as he gets one, he sends to other peers (neighbors) his coordinates and other information.

- **Video Switching Delay:** A peer who is already connected to a stream may want to switch to another channel. The time elapsed from the switching request until the moment in which the user can actually watch the first frame of new video stream is called video switching delay. Differently from what we could suppose, this delay could be even higher than than the startup delay. Indeed, before switching to the new video stream, the peer is asked to unjoin the previous stream, and this could take some additional time.

- **Playback Delay:** The playback delay is the exact difference between the time at which a block has been sent from the source of the stream and the time at which that frame is actually played by a peer. In structured P2P

systems, peers that are farther in structure from the source will experience increasingly long playback delays. What can happen here is that a block might be received after the maximum allowed threshold, this can cause some noises in the reproduction like small freezings or long disruptions.

- **Time Lags or Display Lags:** The difference between playback delays experiences by two peers. This could be very annoying if two users are very close from a geographical point of view, but one gets the video stream couple, or even more, of seconds before the other one. What would happen if the video stream is a live sport event?

As we said in the introduction, these delays can reach very high values. More precisely, according to measurements made by [5], typically startup and playback delays vary from 10 to 60 seconds, which is a very long time if we consider that during that time the user is forced to wait until the reproduction of the first video frame.

1.5 Challenges and Issues

As we all know, any user connected to a real wide network, especially the Internet, always experiences some issues. A P2P streaming system has to deal with these issues and minimize the impact of any kind of sudden and unpredictable event.

Generally, issues and challenges of a P2P streaming system can be summarized as follows:

- **Delays and ordering:** usually in the Internet, there is no warranty for maximum delays or ordering, so any application must take into account these issues and provide a efficient way to solve them keeping the quality of service good enough. In case of a P2P video streaming system, another critical requirement is to minimize disruptions or freezings during the playback.

- **Free riders and incentive mechanisms:** in a P2P system the cooperation between peers is fundamental in order to achieve a good quality for all peers connected to the system. If even just one peer stops cooperating, the quality can dramatically decrease for many peers. This means that a P2P system must provide incentive mechanisms, in other words, when a peer does not cooperate anymore, or keeps cooperating but below the minimum threshold, also the quality of service achieved has to be decreased or, if necessary, that peer has to be forced to unjoin the system.

- **NATs and Firewalls:** this issue, compared with the two above, is less critical. Indeed, many P2P or client-server systems just ignore this issue. However, an efficient P2P system should provide a way to bypass this kind of issues and keep providing a sufficient quality of service.

- **Node Failures or Misbehaviors:** unfortunately, it can happen that a node (peer) fails unpredictably or starts misbehaving. In case of a failure, the system has to adopt an efficient (and possibly fast) strategy for failure detection and recovery. Instead, in case of misbehavior, the system should be able to detect the misbehavior of a peer and expel him. As we can imagine, handling failures is less complex, indeed several detection algorithms have been developed and deployed. On the contrary, misbehaviors are more complex to detect and need a careful analysis. As we will se later, we designed and implemented a basic but pretty good solution for failure detection and recovery, specialized for structured P2P systems.

- **Dynamic Bandwidth and Limited Upload Capacity:** as we said above, cooperation is fundamental in a P2P system. If the cooperation of a peer goes below the minimum allowed threshold or, even worse, a peer stops cooperating, many peers are going to achieve a lower quality of service, probably lower than the minimum required to keep the service suitable. Unfortunately, as

we know, the bandwidth achieved by a peer can vary over the time, and the upload capacity generally is limited. The goal of a P2P streaming system is to deal with these limitations, maximize the quality of service achieved by each peer and keep it as stable as possible.

Chapter 2

Objectives

The primary objectives of my internship can be summarized as follows:

- **Improving stability and efficiently handling population variation:** in any P2P system, especially in Live Video Streaming, peers connected to the system can quickly vary, both in terms of number and identity. For instance, it is quite common to observe a fast increase of the population during the initial phase of the transmission, and a fast decrease during the final phase. In addition, during the transmission, it can happen very often that peers connected from a very short time, get disconnected. This phenomenon is also known as zapping.

- **Detecting and recovering failed or non-collaborative peers:** for any P2P system, it is fundamental to adopt an efficient strategy for failure detection and recovery. In addition, the system should be able to detect and fix not only peer crashes, but also other kinds of issues, for instance peers that cannot correctly cooperate anymore because their upload capacity is not large enough.

As we said above, in a structured P2P system reorganizing the structure could be tricky and cause a domino-effect for several peers. In fact, both the design and the development phase have been driven by the same goal: to minimize the

information to be exchanged in case of a change in the structure and limit the need to be aware of a change to a limited set of peers. Structure reorganization is a very common event in structured P2P systems, which can be triggered by either a new arrival or a departure. In particular, a departure is considered spontaneous when a peer correctly unjoins the system by himself, and forced when a peer is expelled or crashed.

Clearly, we can imagine that structure reorganization has to be handled as efficiently as possible. In other words, not only the amount of data and the time have to be taken into account, but also the additional delay caused by a change in the structure, which should be as close as possible to zero.

A secondary objective of this internship was to create a porting for Windows. This has been achieved by using some specific precompiling instructions in order to include or exclude some snippets of code depending on the target architecture.

Chapter 3

System Architecture

In this section we will describe the strategy and the structure adopted for dissemination. In addition, we will explore the architecture of the system from a more technical point of view and we will analyze closely the components of the system and their roles.

3.1 Hypercube Data Structure

The structure on which our system is based is hypercube [6]. Hypercube is a multidimensional structure in which every node corresponds to a vertex. For instance, if the number of dimensions is 3, the structure can host up to 8 nodes. Each node can be univocally identified through an ID, which is composed by a sequence of 0 and 1.

There are several reasons for which we can state that a hypercube is a suitable and efficient structure for P2P streaming:

- Simple enough to not require a large amount of packets to be exchanged in order to keep the structure consistent.
- Each peer knows exactly his position and his neighbors in the structure just by looking at his ID. This allows a peer to operate in the system without being

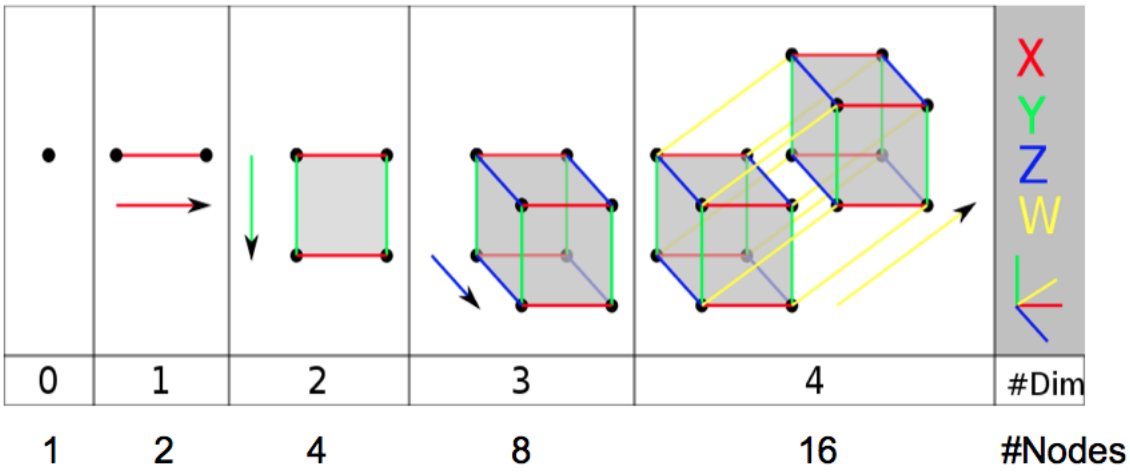

Figure 3.1: Hypercube Structure

aware of any additional information. In other words, once a peer gets the list of his neighbors and their coordinates, he can easily infer peers from whom he is going to receive blocks and peers to whom he is going to send blocks.

- Time required for global transmission is proved to be optimal: $\log_2 N$. This allows to achieve shorter dissemination delays and a faster startup delay.

- Highly scalable: number of peers is equal to 2 power number of dimensions. This means that if population size increases very fast, we can double the maximum number of peers we can host in the system, by just increasing the dimension by 1.

Before proceeding and explaining dissemination rules, we need to introduce a couple of concepts that are fundamental for a good comprehension of the working mechanism of our system:

- **Neighbor:** Peer A is a neighbor of peer B if A's ID has just one bit different than B's ID. As we said above, any peer knows his neighbors by just performing basic operations on his own ID. This means that no matter what the context is, a given peer will always have the same set of neighbors.

- **Level:** Given a peer A and his ID, the level of A is equal to the number of

1 in A's ID. Level is a fundamental property for a peer. Indeed, as we will se later, the behavior of peer and his dissemination strategy depend on his current level.

An important issue occurs if peers are structured as a hypercube: a hypercube is supposed to be complete, in other words, every vertex should correspond to a node (peer). Formally, a hypercube is complete when, given a number of dimensions n, the number of peers is 2^n. Unfortunately, it is very unlikely that the population of a system is so stable to host for all the transmission the same number of peers. As we know, users of streaming systems can get disconnected in any moment, even after just a couple of seconds from their connection.

3.2 Dissemination Rule

Our aim is to design an efficient P2P system for live streaming. In this case, efficient means that the system should disseminate content as fast as possible and with the least waste of bandwidth. Ideally, each peer should not receive the same block twice but on the other hand losses have to be minimized even in presence of continuous and massive population variations.

Fairness is another important principle which has been adopted in the design of our system. Our dissemination rules provide a fair mechanism that equally share available bandwidth among peers.

In this section we will explain in detail the dissemination rule over a complete hypercube. In the next section we will explain the strategy we adopted in order to adapt the structure when the hypercube is not complete.

Two peers can exchange blocks if and only if they are neighbors, so we assume there is a relation one-to-one between peers such that it creates edges that link peers to each other. Blocks are thus exchanged along these edges.

We adopt the following notations:

- n is the number of dimensions;
- an identifier is defined in binary notation as a sequence of bits: $b_{n-1} \dots b_1 b_0$ with $b_i \in \{0, 1\}$ for $i = 0, \dots, n-1$;
- let e_k be the n-length binary number with all zeroes except in position k modulo n. Note that $e_{k+n} = e_k$. We denote the exclusive OR (or XOR operation) on two n-length identifiers, a and b, the identifier $c = a \oplus b$, with $c_i = a_i \oplus b_i$ for all i, where $\oplus$ is the exclusive OR operator for bits. Two peers may exchange stream blocks if and only if their identifiers, a and b, differ by one bit (in binary notation);
- we denote peer $b \oplus e_k = b_{n-1} \dots \bar{b}_k \dots b_0$ as the k^{th} neighbor of peer $b = b_{n-1} \dots b_k \dots b_0$ (where $\bar{b}_k$ is NOT b_k in binary logic). In this definition k is considered modulo n so that neighbors k and $k + n$ of a given node represent the same peer.

As an example in an $n = 5$ dimension hypercube, the peer with identifier 18 has the binary representation $b = 01001$ and has the n hypercube neighbors (starting from neighbor 0 to neighbor $n - 1$): 0100**0**; 010**1**1; 01**1**01; 0**0**001; **1**1001.

The source of the stream, i.e. the node with identifier 0, transmits stream blocks the following way: 0 transmits blocks numbered k to peer $0 + e_k$. For example with $n = 5$ the source will transmit blocks 3; 8; 13; ... to peer 01000.

To recursively describe the system we must explain how blocks received by a peer are retransmitted to its neighbors. The retransmitting rule is slightly different from the block generating rule used by the source.

A peer with identifier b, receiving a block numbered k, will perform the following algorithm.

```
If:
bk = 0: then do not retransmit,
```

```
bk = 1: then retransmit chunk k to:
        all k'-th neighbors such that bi=0 for all i=k',...,k - 1
        the k-th neighbor.
```

Note that indices are always considered modulo n. Also the retransmission should start with the smallest possible index k' and continue by increasing the index until reaching k.

These rules define a hypercube network and a broadcasting algorithm over this network. We will refer to them globally as the hypercube rule. They provide an optimal dissemination algorithm in terms of number of block retransmissions before reception by peers.

As an example, we can suppose that our identifier is 9 which in binary format is 01001. We suppose also that the number of dimensions is 5. As we can see in Figure 3.2, we are going to retransmit only blocks 0 and 3. Once again, we recall that indexes of blocks are considered always modulo n, where n is the number of dimensions. In the case of block number 0, we first apply the second part of the dissemination rule, which says that since the bit in position 0 is 1, we have to forward the block to 0^{th} neighbor. To obtain the identifier of 0^{th} neighbor we have just to to invert the value of the bit in position 0. Now we try to apply the first part of the rule, which says that we have to retransmit the block to every neighbor such that in his identifier we can find a sequence of zeroes starting from any position until position $k-1$, where k is the block number modulo n. Sequences that wrap around are accepted too. In this case, we find one sequence that meets this condition, the one composed by only the bit in position 4. After applying the two rules, we know that we are going to forward that blocks to two peers: 11001 and 01000.

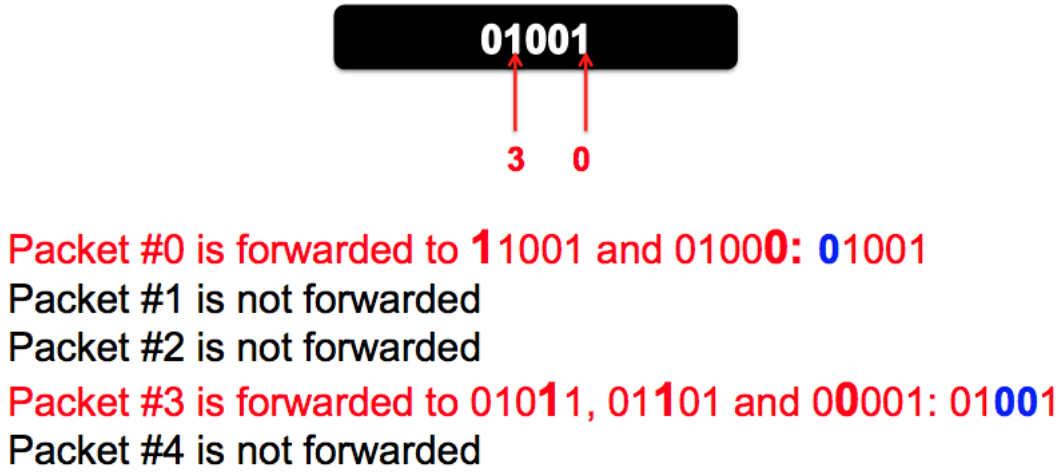

Figure 3.2: An example of application of dissemination rules

3.3 Adaptation of Hypercube

As we said in the previous section, the hypercube rule provides an optimal dissemination algorithm. However, it is very unlikely that a P2P system can manage to build a complete hypercube structure, so we need to design a solution in order to guarantee an efficient and optimal dissemination also in the presence of an incomplete hypercube.

In addition, we need also to slightly adapt dissemination rules. Indeed, if we just apply the rules defined above, it is very likely that identifiers corresponding to some neighbors are not allocated yet. This happens because the system is incomplete and some positions are still empty.

Our solution changes just a limited portion of the peer organization and keeps meeting our main design principles: fairness, limited and local knowledge and limited reorganization. This solution is based on the notion of levels, in particular, the only levels affected by the change of dissemination rules are the ones that belong to either second-last level or last level.

Previous solutions that have already been proposed to structure the diffusion for different values of N. But they are either limited to particular population sizes

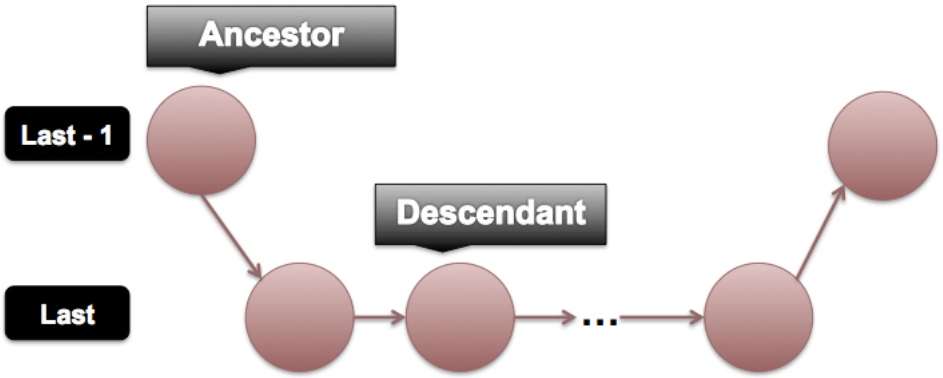

Figure 3.3: An example of descendant list in last level

[9, 10, 7] or the solution results in much longer delays [2].

Peer identifiers in level l may by classified in $\binom{n-1}{l-1}$ subsets S^j, where l is the number of levels, so l is the last level, and n is the number of dimensions. Each subset is associated to one of the $\binom{n-1}{l-1}$ identifiers, s^j for $j = 1, \ldots\ldots, \binom{n-1}{l-1}$ as defined above. We recall that $s^j_0 = 1$ and $l-1$ of the other $n-1$ bits are equal to one. Let b be the identifier of a peer in the last level l. Let k be the position of its first non zero bit starting from the right (i.e. $b_k = 1$ and $b_i = 0$ for $i = 0, \ldots, k-1$). Then b belongs to the subset S^j such that one obtains the identifier s^j by rotating b, k times to the right. Note that s^j belongs to S^j. As an example, if $l = 4$ is the last level of system with dimension $n = 8$, and if $s^j = 00011001$, then S^j is composed of identifiers:

- 00011001,
- 00110010,
- 01100100,
- 11001000.

Alternatively, the subset associated with identifier 10011001 contains only that identifier. The identifiers belonging to in any subset S^j may be arbitrarily ordered so that S^j forms a list. We assume this is the case. A possibility is to order the identifiers

by increasing value. (In that case s^j is the first identifier in the list S^j.) The routing rules used by a peer in level l are the following. When a peer receives a block k, it looks in the list S^j it belongs to, for the next peer in the list present in the system. If one is present it forwards the block to the next peer. If none is present it forwards the block k to the unique peer in the previous level $l-1$ to which peer s^l would have sent it to according to explanation above The routing rules used by a peer in level $l-1$ to retransmit a block k destined to the next level l are the following. The peer identifies the peer s^l it would send the block to according explanation above. Then it looks for the first peer present in the list S^j associated to s^l. If one is present in the system, it forwards a copy of the block to it. If none is present, it forwards a copy to its corresponding peer (for block k) in the same level $l-1$ as defined in sub-section 'complete last level'. The order in which the identifiers are attributed in the last level is not important. Peers not present in a list S^j are skipped. Peers are allowed to leave the last level without requiring the reorganization of the upper levels, the departing peer has just to notify the upcoming change to his successor and predecessor in the list.

As we said above, even if the order in which identifiers are assigned in the last level does not matter, we need to define a precise rule to build descendant lists. In other words, we need to define an order for descendant lists, which must be different than the chronological one. Also in this case, we decided to keep things simple and use the easiest possible order: peers are ordered just by their identifiers. This means that the peer associated to the smallest identifier will be the head of the descendant list, and the one associated to the biggest identifier will be the tail.

3.4 Framework for P2P Streaming

Before analyzing closely details of our implementation, it is worth giving some technical details concerning the development environment in which this system has been

developed. Moreover, the system has been developed keeping in mind that peer can have serious heterogeneity in processing power and memory. Thus efforts were put-in to make the computational and memory requirement as low as possible. The result of these efforts allowed us to run simultaneously on the same machine more than 1000 peers.

Some general details are the following:

- Programming language used is C
- IDE we used is Eclipse
- Operating system used is Linux (kubuntu distribution)
- Shell scripting is used to do get useful information from log files such as losses, delays, etc.
- UDP sockets have been used instead of TCP sockets for network connections.

The reason why we decided to use UDP connections over TCP connections, despite to reliability and congestion-aware mechanisms provided by TCP are the following:

- Generally, operating systems have constraints that limit the rate to accept new TCP connections and number of concurrent TCP connections. This means that if we adopted TCP connections, we would have restricted the size of P2P hypercube system that we can build and simulate on a single local machine.
- Another reason that supported our choice of using UDP connections is that the TCP protocol usually takes longer to establish a connection and this could generate more delay, which does not meet our main design principles.

3.4.1 Software Entities

At execution time, our system is composed by three entities: a source, a server and several clients.

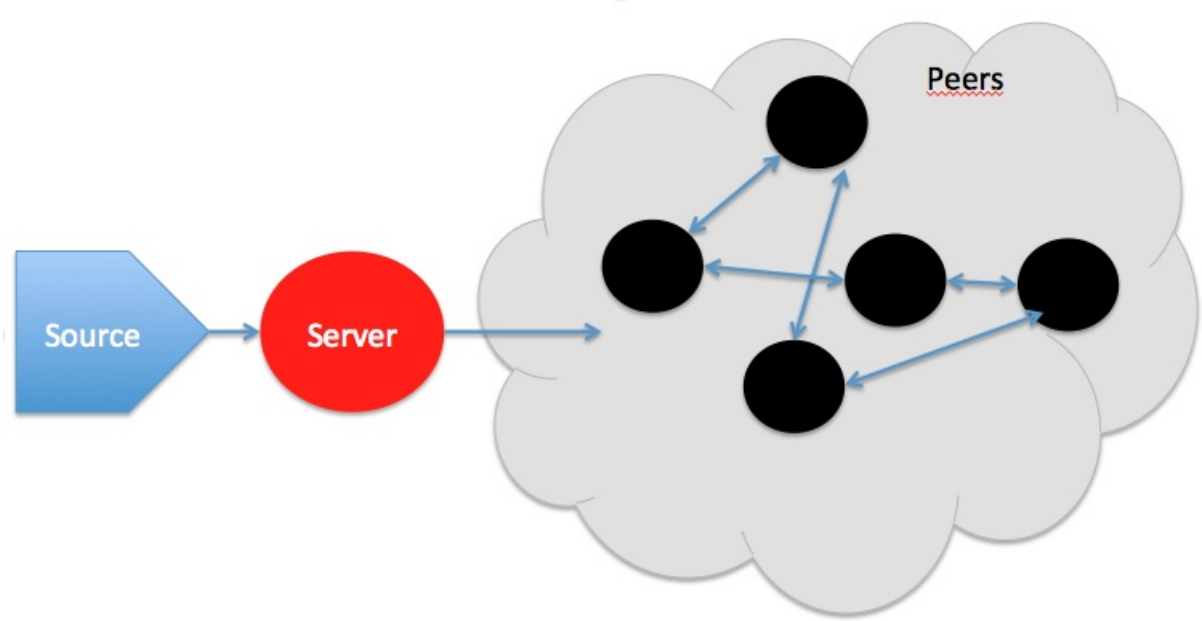

Figure 3.4: Entities of the system

Source: as his name suggests, the source is supposed to be the entity in charge of producing content and sending it to the server. Differently from the original idea, the source process does not produce any content actually. Indeed, he just reads a UDP stream incoming from a specific port, segments UDP packets into custom blocks, and forwards them to the server in order to proceed with the dissemination. In our implementation, source assigns to every block a unique identifier in order to make both server and clients able to treat blocks.

Server: the server process does not generate any content. Indeed, the role of the server is to keep track of the current status of the hypercube structure, relations between peers and coordinates of peers (IP address and port number). Furthermore, the server process is also in charge of supporting peers for some special operations that we will explain in next sections. The server is also the entry point of the system. When a new peer wants to join the system, he first has to send a request to the server in order to get a unique identifier. If the server replies by sending a positive answer, then the joining peer contacts all the peers (ancestors, neighbors, descendants, etc.) contained in the list attached to the answer block. The aim of this procedure is to

allow a joining peer to safely get in the structure and start cooperating for content dissemination.

Client: Finally, client is the name given to the entity that implements a peer. From our point of view, there is no difference between a client and a peer, so we will use these two names alternatively. Each user who wants to watch the video stream, has to run an instance of this executable. The only parameters required are coordinates of server: his IP address and the port number on which he is waiting for new requests. Once the client gets an identifier from the server, he has to perform the procedure we mentioned before in order to get in the structure and receive blocks. We recall that client is the most important entity of our system, which is quite obvious since we are describing a P2P system. Indeed, as we will see in next sections, each peer maintains several data structures which are used for storing the current status of the peer.

3.4.2 File Organization

The idea behind creation and organization of different header files and source files is to separate the network level communication from the implementation of the hypercube structure. The organization of files is as follows (see Figure 3.5).

The header files for server and client (i.e. *server.h* and *client.h*) contain all declarations of functions and variables that are required by peers to communicate with each other. This involves functions to establish and close UDP connections, read from an entity (client, server or source), write to an entity (server or source) and some particular declarations and adaptations for the target architecture in order to finally make communication possible.

The header files with name starting with diffusion (i.e. *diffusion.h* and *diffusion_ client.h*) basically contain all variables, data structures and functions necessary to create, run and manage the hypercube structure. File block.h contains the definition of data structure used for messages, that is Block. This is the basic mes-

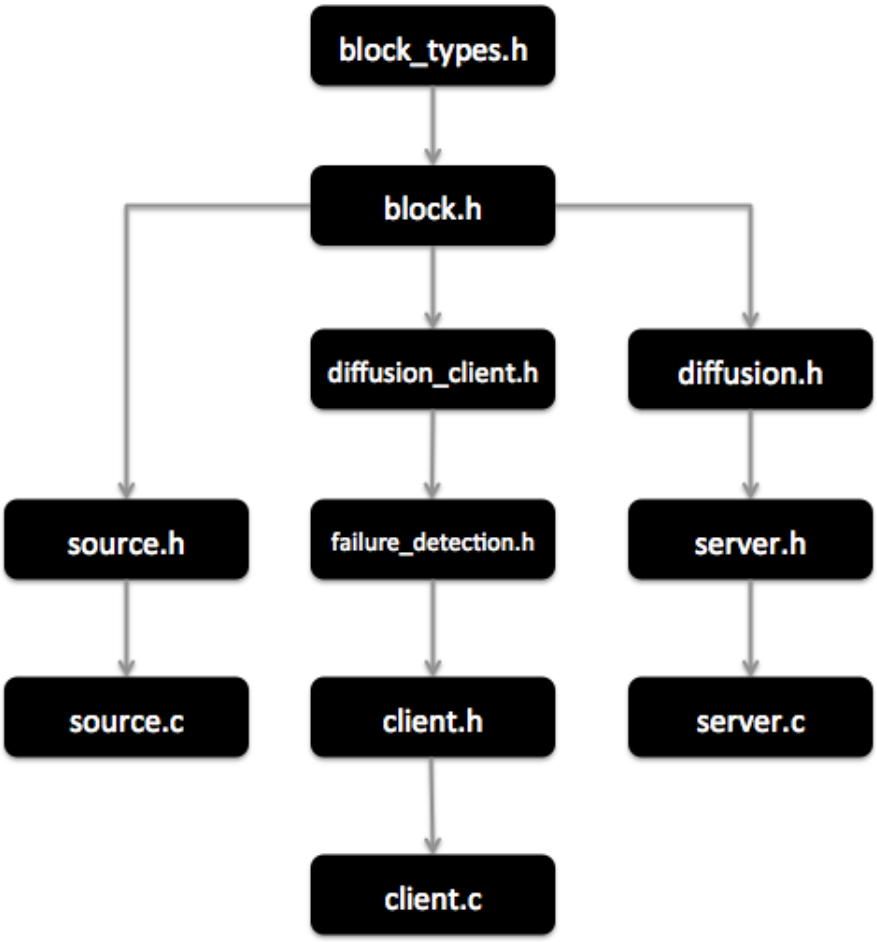

Figure 3.5: File organization

sage format that is used for data as well as control packets. File named *block.h* also contains all the functions that are used to manipulate and display a block.

File named *failure_ detection.h* contains the declaration and the implementation of all the functions and variables intended for our failure detection solution.

3.4.3 Data Structures

In this section we show the most important data structures used by a client. We thus ignore data structures used by source and server since they are not very significative for our purposes.

Data structures used by clients are the following:

- **id:** this variable contains the current ID of the peer;

- **primary_id:** this variable contains the main id of the peer. It is worth pointing out that there is an important difference between primary_id and id. Indeed, id represents the current id used by the peer during a single retransmission phase. The reason why they can be different is that just after a replacement, some previous neighbors could be not up-to-date and still send packets to the replacing peer. In order to avoid losses or disruptions due to some delays in the delivery of updating packets, the replacement peer has to keep disseminating content also as he would do in the previous position. We will clarify this aspect in next sections;

- **secondary_id:** this variable contains the previous main id of the peer. If a peer has replaced another peer in the past, secondary_id is set to his previous id;

- **dim, last, level:** this variables respectively contain values of dimension, last level and level to which the peer belongs;

- **pacsent[dim]:** it is an array of dimension 'dim' and contains information to know which data block number must be sent at each time slot. pacsent is computed by each peer by just looking at his identifier which has been assigned by the server. For example a peer with identifier 01010 will have pacsent = {1,3,3,1,1}. This means that data block sent at time slot 0 is 1, 1 at time slot 1, 3 at time slot 2, 3 at time slot 3 and 1 at time slot 4. It is worth noticing that data block number and time slot are always treated as modulo dim. as an example, if block number is 20 then it corresponds to 20 modulo 4 (i.e. dim -1) which is 0;

- **pacr[dim]:** it is an array of dimension 'dim' and allows a peer to know in advance each data block it is going to receive at each time slot. We recall

that a peer can compute this information by just looking at his identifier. For example a peer with identifier 01010 will have pacr[ ] = {4,1,2,3,0}. By checking out values of pacr[ ], a peer can realize that at time slot 6 he is going to receive data block number 8 and at time slot 9 he is going to receive block number 9;

- **times_sent[dim]:** this structure is different from pacsent and pacr, from a . times_sent gives information regarding how many times a data block is suppose to be sent by a peer. For example, a peer with an identifier 00010, will send data block 1 five times and never sends all the other block (i.e. 0, 2, 3, 4). Thus for peer 00010, times_sent[ ] = {0,0,0,5,0}. Just to explain the difference between times_sent and the other data structures, we can look at the example we showed in the description of pacsent & pacr i.e. 01010. times_sent for this peer is {0,2,0,3,0} which means that this peer sends block 1 three times, data block 3 twice and never sends the other blocks;

- **neighbour[dim]:** in this array we store identifiers, and corresponding coordinates, of all hypercube neighbors of a given peer. It is worth noticing pointing out that in this data structure we store only formal neighbors, which are those neighbors that meet the rule we defined in the section 3.1. We recall that a hypercube neighbor's identifier vary with identifier of given peer by only one bit. For peer 01010, the neighboring peer will be 01011, 01000, 01110, 00010 and 11010.

Before introducing the next data structure, a little background is required. As we mentioned before, we adopt a special strategy in order to adapt both the hypercube structure and dissemination rules in case the number of peers in the system is lower than the maximum allowed, in other words, when the hypercube is incomplete.

The only levels affected by this adaptation are the second-last and the last one. Peers in the last level are grouped in multiple descendant lists. Every descendant

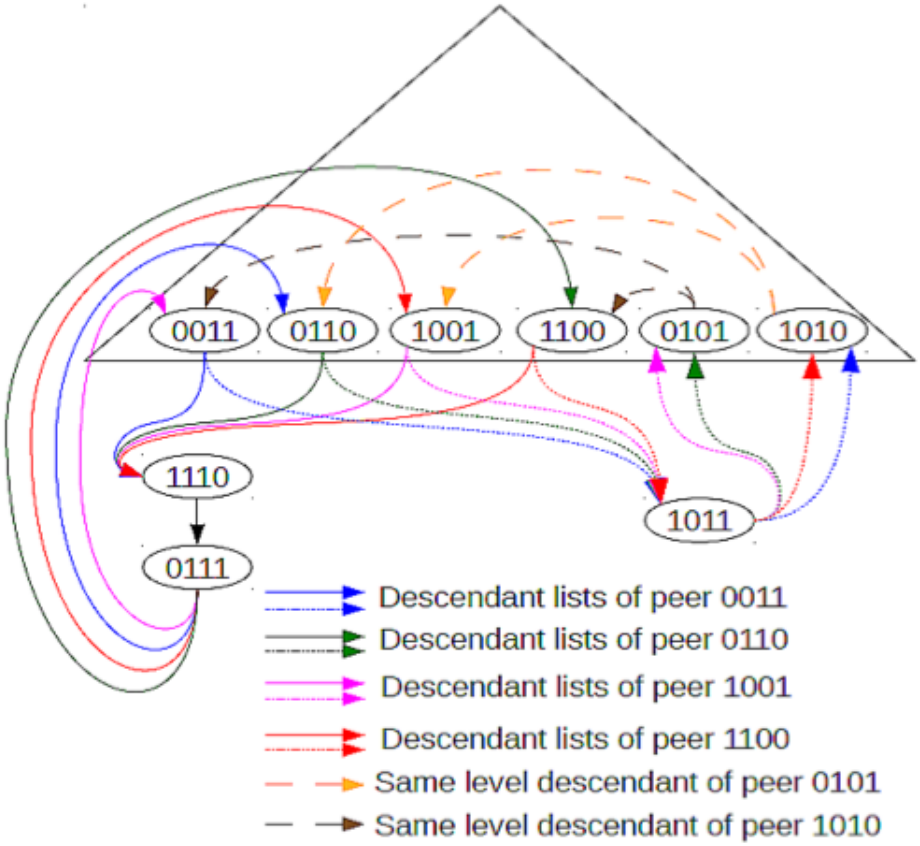

Figure 3.6: An example of descendant list

list is associated to a different ancestor, who is a peer belonging to the second-last level. Once ancestors receive a block, they forward that block to the corresponding descendant list. Indeed, also descendant lists are built according to identifiers. Each ancestor needs to keep just the identity of the head of each descendant list, which is the peer with the lowest id. It is worth saying that, for stability reasons, descendant lists are maintained also by peers that do not belong to the last or second-last level. The main reason why we decided to do so is mainly due to the very likely event in which the number of levels change during the transmission. Indeed, in that case we would need to rebuild descendant lists, this task could take some time and generate delays, resulting in worse performance.

Data structures used for managing descendant lists are the following:

- **descendants[dim][dim]:** this data structure is a two dimensional array; first

dimension holds descendant list associated with each bit position and second dimension holds details of peers in that list. As we can see in 3.6, peer 0011 has total four possible descendant lists out of which only two are not empty. Thus descendants[3][ ] and descendants[2][ ] are both empty. Instead, descendants[1][3] = details of {1110, 0111, 0110} and descendants[0][2] = details of {1011, 1010};

- **ancestors[dim]:** for peers in the last level, ancestors structure holds details of peers in second-last level that are supposed to send blocks to the head of the descendant list he belongs to. As we can see in 3.6, ancestors of peer 0111 are peer 0011, 0110, 1001 and 1100 as they send blocks to peer 1110, which is also the head of the descendant list to which he belongs;

- **same_level_ancestors[dim]:** this data structure helps peers at second-last level to hold details of the peer who are their ancestors but belong to the same level. As we can see in 3.6, same level ancestors of peer A = 0101 are 0110 and 1001 as both of them send blocks to descendant list of which peer A is the last descendant;

- **same_level_last_descendant[dim]:** this data structure contains, for each descendant list, a peer belonging to our same level, who is the next member of the ancestor chain for a given block descendant list. This structure can be used in two different ways during the transmission phase: the first one is to communicate to descendants the identity of the next ancestor to whom to retransmit when the block reaches the last peer, the second one is to send a block directly to him in case the corresponding descendant list is empty;

- **vote_list[ ], repair_list[ ]:** these data structures allow to maintain a set of peers, belonging to the last level, that could be potential replacements in case the current wants to leave the system. We will go into more detail in the section focused on management of arrival and departures;

- **send_to_desc[] (secondary_send_to_desc[]):** this data structure is maintained by every peer belonging to a level which is not the last one. Indeed, as we mentioned before, descendant lists are maintained even when it is not required, in order to be prepared in case the number of levels changes. However, this data structure is actually used by only peers in the second-last level and contains the identity of the first member of each descendant list. Instead, secondary_send_to_desc is used for retransmitting blocks received after a replacement and still pointing to our previous identifier;

- **send_to_succ (secondary_send_to_succ):** this variable contain coordinates of our successor in the descendant list. As for send_to_desc[], secondary_send_to_succ is used only for blocks still pointing to our previous identifier;

- **receive_from_succ:** this variable contains coordinates of our predecessor in the descendant list;

- **my_id_location:** by checking the value of this variable, we know if we are the last peer in our descendant list or not. Because of historical reasons, the name of this variable does not match exactly its actual meaning;

- **head_of_the_list:** by checking the value of this variable, we know if we are the first peer in our descendant list or not;

- **replacing:** this is a special variable used only when we are replacing a peer who has been expelled from the system. If it set to 1 then the departure is not spontaneous (the departing peer has crashed or whatever), otherwise the departure is spontaneous (the departing peer has autonomously decided to leave the system);

- **receive_from:** each peer in the system stores the details of peer from whom he has just received a block. Even though it is declared as an array, currently

only the first position of receiver_from is being used. This information is used for several purposes such as replying to an explicit request from a given peer, etc.

- **tableFrames[ ]:** this array keeps track of which data blocks have been received by a peer. This structure helps to:
  - ensure if all the data blocks are received by a peer or not,
  - find out the data blocks that have not been received,
  - retransmit data blocks that a peer has received but one of its related peers did not.
- **nbFrames:** this variable stores the highest block number that has been received by a peer.

3.4.4 Data Structures for Failure Detection

In this section we explain the meaning of some special data structures used for failure detection purposes.

First of all, we need to point out that our solution is to piggyback information about blocks received or lost, by attaching it to standard data blocks.

In order to do so, we defined two new data types:

```
typedef struct {
        uint32_t concerningpeer;
        uint32_t frompeer;
        uint32_t throughput[REPORT_WINDOW];
        uint32_t losses[REPORT_WINDOW];
        uint32_t nsent;
        struct timeval last_time_sent;
}
```

```
Report_History;
```

where

- **concerningpeer** and **frompeer** represent contain identifiers of two different peers,
- **nsent** is a counter where we store the number of reports received,
- **throughput[]** and **losses[]** contain a sequence of values which are used by the peer in order to decide if concerningpeer is not cooperating sufficiently. If so, the peer starts the procedure to expel concerningpeer from the system,
- **last_time_sent** store the timestamp of the last time a given report has been sent by the peer,
- **REPORT_WINDOW** is a constant which defines how many reports can be stored. Of course, reports are stored according to their chronological order, this means that if the current size of the report window exceeds REPORT_WINDOW, the oldest report is discarded,

and

```
typedef struct {
        double_t throughput;
        double_t losses;
        Client client;
        uint32_t nreports;
} Average_Report;
```

where

- **throughput** and **losses** store the averaged values of throughput and losses piggybacked with blocks,

- **client** is a data structure which contains information about the client associated to that report,
- **nreports** is a counter that keeps count of how many reports have been sent.

Data structures used for managing reports are the following:

- **leader_reports[dim][dim]:** this data structure is maintained only by those peers that are the designated leader of one or more peers. It is worth remarking that the designated leader of a given peer is always one of his hypercube neighbors. It is a bidimensional array because the first dimension is intended to individuate reports concerning a given peer, and the second dimension is intended to individuate reports sent by a given peer;
- **route_reports[dim][dim]:** this data structure is maintained by every peer. The aim of this structure is to store reports piggybacked with blocks received by a given peer. In this case, the meaning of the two dimensions is inverted: the first dimension is intended to individuate reports sent by a given peer, and the second dimension is intended to individuate reports concerning a given peer;
- **average_reports[dim][dim]:** this data structure is maintained only by leaders. The aim of this structure is to compute mean throughput and losses achieved by several peers from a given peer. This way, a leader is able to correctly detect a failed peer and expel him from the system. We recall that a peer is considered as failed when either he crashed or the level of cooperation that he is offering is not sufficient anymore.

We will go into more detail in the section focused on failure detection, where we will extensively explain every detail on our solution for failure detection and recovery.

3.4.5 UDP Packet Format

As we mentioned in previous sections, we use UDP as transport protocol. More precisely, we defined a custom UDP packet format in order to transmit both data blocks and info blocks. We recall that info blocks differ from data blocks because they do not contain any piece of content, they are used only for exchanging information required to build or maintain the hypercube structure. We will give a wide explanation of all existing block types and their use in the next section.

In order to treat our custom UDP packet format, we defined a C data structure, which is composed by the following fields:

- **uint32_t type:** a unique identifier for each block type;
- **uint32_t blknb:** a unique identifier for each data block. It is incremented by the source;
- **uint32_t trmnb:** number of times each data block has been retransmitted within the hypercube structure;
- **uint32_t id:** the identifier of the sender;
- **uint32_t dim:** current dimension of the hypercube structure. Possibly, the system should be able to dynamically vary the dimension according to current needs. Currently the system does not manage yet this kind of event but, ideally, this operation should not be very complex;
- **uint32_t last:** current last level of the hypercube;
- **uint32_t nb_neigh:** total number of peers in cl list. That list is used for several block types. Its aim is to attach to any block (not only the ones containing data) a set of peers, and their respective coordinates, which have to be used by the receiving peer;

- **uint32_t c_anc:** in origin this field was used for storing the number of ancestors in the cl list. Currently, it is used to for storing sender's id;
- **uint32_t c_desc:** in origin this field was used for storing the number of descendants in the cl list. Currently, it is used for storing receiver's id;
- **uint32_t ack:** this fields is is used by sender to store the number of the last received block. In other words, a sender uses this fields to tell the receiver the number of the last block received from him. This way, each neighbor can figure out if a packet was lost and, if necessary, resend it;
- **Client cl[MAX_DIM]:** as we mentioned before, this array contains a set of clients and their coordinates. It is used for every block type that needs to communicate to the receiver one or more coordinates of one or more peers;
- **Report_network reports[2]:** this array contains two data structures used in the context of failure detection. We will analyzing closely this data structure in the section focused on our solution for failure detection;
- **StreamPackets streamPackets:** this field is the one that actually contains data. More precisely, this fields encapsulates UDP/RTP packets sent by the stream source by wrapping them with fields we just showed. We recall that the stream source is not the same process that operates as source for our P2P system. Indeed, the stream source just generates the stream to disseminate over the system, instead the the source process is the process in charge of encapsulating the stream packets in our custom UDP block format.

Data structures used for failure detection will be studied more deeply in next sections. Instead, we now show the fields that compose StreamPackets data structure:

- **size_t sizes[1]:** this array contains the size of each UDP/RTP packet encapsulated in our custom UDP block;

- **char data[10000]:** as we its name suggests, this array of bytes is the fields that actually encapsulates packets generated by the stream source. It is defined as an array of chars because this is the best way to declare a field as a sequence of bytes. Of course, 10000 bytes is just the maximum allowed size of the encapsulated content. Indeed, the size of packet can vary depending on the size of the encapsulated content. This means that there is no waste of bandwidth.

It is worth pointing out that our UDP block format can potentially encapsulate even more than one single packet. This can be achieved by just increasing the size of sizes array and put every packet one after another in data array.

There are two important things that are worth pointing out:

- any significant property of the hypercube structure is piggybacked with data blocks;

- the total size of each data block does not exceed the recommended UDP message size, which is less than 1500 bytes. The choice of this value depends on the maximum allowed size of a IP datagram. Technically, the maximum UDP message size is 65507, but we want to avoid the unfortunate event in which the entire UDP datagram is considered lost or corrupted because of a single lost IP datagram. This means that the safest size of a UDP message is equal to the maximum IP payload size.

3.4.6 Block Types

In this section we draw up the list of all block types currently used in our system.

TYPE	SENDER	RECEIVER	PROCESSING	NOTES
1a	Any peer who wants to join the system	Server	Server returns a unique identifier for the new peer	

TYPE	SENDER	RECEIVER	PROCESSING	NOTES
1b	Server	A peer who has sent a joining request and is waiting for an identifier	The cl list attached to this block contains coordinates of neighbors, ancestors, descendants, same level descendants and same level ancestors. The new peer also uses his identifier to compute identifiers of neighbors in the hypercube and populate data structures.	Receiving peer sends block type 2 to all the peers in the cl list and block type 25 to ancestor[0] in order to join his descendant list.
11	Any peers who joins the system	Server	Server marks the id of the sender as assigned.	
2	Any peer	Any peer who is related to the sender (neighbor, descendant, ancestor, successor or predecessor in the descendant list, etc.)	Receiving peer updates coordinates corresponding to sender.	

TYPE	SENDER	RECEIVER	PROCESSING	NOTES
12	Any peer in a descendant list	Any peer in a descendant list, except the head of the list	receiving peer updates coordinates of ancestor[0].	When a peer receives an update (block type 2) from ancestor[0], he updates also all the members of his descendant list.
4	A peer, belonging to the last level, who is leaving the system	Server and predecessor in the descendant list	Server marks sender's id as not assigned anymore, so it is available. Predecessor updates his successor, if any.	For this block type, server uses the cl list in order to store coordinates of his current successor. If he is the last peer in the descendant list, the cl list is empty.
14	A peer who is the head of the list and is leaving the system or changing position (replacement).	Every ancestor of the leaving peer.	Ancestors that receive this block, update send_to_desc data structure.	

TYPE	SENDER	RECEIVER	PROCESSING	NOTES
24	A peer (not head of the list) who is leaving the system or changing position.	Successor	Receiving peer updates coordinates of his predecessor in the descendant list.	
34	A head of the list who is leaving the system or changing position.	Successor	Receiving peer becomes the new head of the list.	For this block type, cl list is used for storing coordinates of ancestors.
15	A head of the list who is leaving the system or changing position.	Ancestors	Ancestors update send_to_desc data structure.	

TYPE	SENDER	RECEIVER	PROCESSING	NOTES
25	A peer who wants to join a descendant list. He could be a peer who just joined the system or a replacement.	Ancestor[0]	Ancestor[0] retrieves the descendant list to which joining peer is supposed to belong. After that, he checks if the descendant list is empty. If so, he sends a block type 16 to the sender, otherwise he sends a block type 6.	
6	Any peer in a descendant list.	Any peer in a descendant list.	Every peer that receives this block type checks if he has a successor or if successor's id is greater or equal than joining peer's id. If so, he updates his successor, otherwise he just forwards the block.	

TYPE	SENDER	RECEIVER	PROCESSING	NOTES
16	Ancestor[0]	New head of the list.	After receiving a block type 25, ancestor[0] checks if the new peer will be the head of the list. If so, he sends to him a block type 16. We recall that peers are ordered by their identifier (from the lowest to the highest), this means that even if a descendant list is not empty, a joining peer can become head of the list.	
46	A peer in a descendant list.	A peer in the same descendant list.	A peer that just joined a descendant list, sends this block type to his successor in order to make him aware of his presence.	

TYPE	SENDER	RECEIVER	PROCESSING	NOTES
8a	A departing peer	A peer belonging to the last level who has been chosen as replacement for the departing peer.	The departing peer chooses a replacement from his repair_list and sends to him a replacement request by attaching to the block coordinates of his neighbors. We recall that neighbors mean not only hypercube neighbors but also all those peers with whom the peers communicates, such as descendants, ancestors, etc.	
8b	A replacement who has taken the place of a departing peer	Server	If a candidate for replacement has taken the place of a departing peer, he sends a special block to the server in order to make him aware of new coordinates corresponding to a given identifier.	

TYPE	SENDER	RECEIVER	PROCESSING	NOTES
18	A replacement candidate who has accepted a replacement request	Departing peer or Server	The aim of this block type is to send a positive acknowledgement to the departing peer. If a departing receives this block he can safely leave the system. In case of a non-spontaneous departure, the replacement procedure is started by the server, so it is the server that receives the acknowledgement.	This block type shows the reason why each peer maintains a repair list which is a set of peers belonging to the last level. Indeed, if a peer refuses to replace a peer who is going to depart, the departing peer resends the same request to the next peer in repair_list. If repair_list is empty or all the peers refused the replacement request, the departing peers sends a block type 88 to the server.

TYPE	SENDER	RECEIVER	PROCESSING	NOTES
28	A replacement candidate who has refused a replacement request	Departing peer or Server	The aim of this block type is to send a negative acknowledgement to the departing peer. If a departing receives this block he has to choose a new replacement before leaving the system. As for positive acknowledgements, also the server can receive this block type.	
88	A departing peer who has no available peer in his repair list	Server	Server builds a set of potential candidates for replacement by picking some peers from the last level.	
102	A leader that detects a non-collaborative peers and decides to expel him from the system	Server	Server acts as a departing peer would do. So it creates a block and attaches to it the set of neighbors of the departing peer.	The only difference between the cl list of block type 81 and the one of block type 8 is the presence of coordinates of successor peer. Since departing peer has failed, replacement has to take his place but he needs to know coordinates of his successor. Clearly, server is the only entity that can deduce this information.

TYPE	SENDER	RECEIVER	PROCESSING	NOTES
81	Server	A peer belonging to the last level who has been chosen as replacement for the departing peer.	The peer chosen as candidate for replacement acts exactly as for block type 8.	
9	Departed peer	Neighbor of departed peer	It can happen that one or more neighbors of a departed peer are not aware of his departure due to some lost blocks. The departed thus stays alive for a while in order to detect if all his neighbors are aware of his departure. If not, he sends this special update block.	
101	Server	Any peer	By receiving this block peer a peer is asked to safely leave the system.	This block type is used for simulations in order to simulate spontaneous departures.

TYPE	SENDER	RECEIVER	PROCESSING	NOTES
0	Server and peers	Any peer	Block type 0 is used only for data blocks, that are those blocks that actually contain content to be played.	

As we can see, a pretty large set of block types is currently used. This way we can easily understand the behavior of peers by just looking at block types sent and received. Indeed, this choice helped us also during the debugging and testing phase.

3.5 UDP Encapsulation/Decapsulation and Proxy-like Behavior

Most of streaming P2P systems currently in production are coupled to a specific video format or communication protocol. Instead, our system is able to deal with any format and protocol. The only assumption we make is that the communication protocol is an extension or specialization of UDP, which is an acceptable assumption since several streaming or real-time systems use UDP as protocol.

The mechanism we used in order to achieve so, is UDP encapsulation/decapsulation. The first component of the transmission chain is the source process, which can potentially receive content from any kind of multimedia source. The source process encapsulates each UDP packet generated by the multimedia source into a custom UDP packet format and transmits it over the system. Each peer can decode and reproduce the original stream by just decapsulating the UDP packet encapsulated by the source process at the beginning of the transmission. Once a peer decapsulates the original stream, he can send it to a video player.

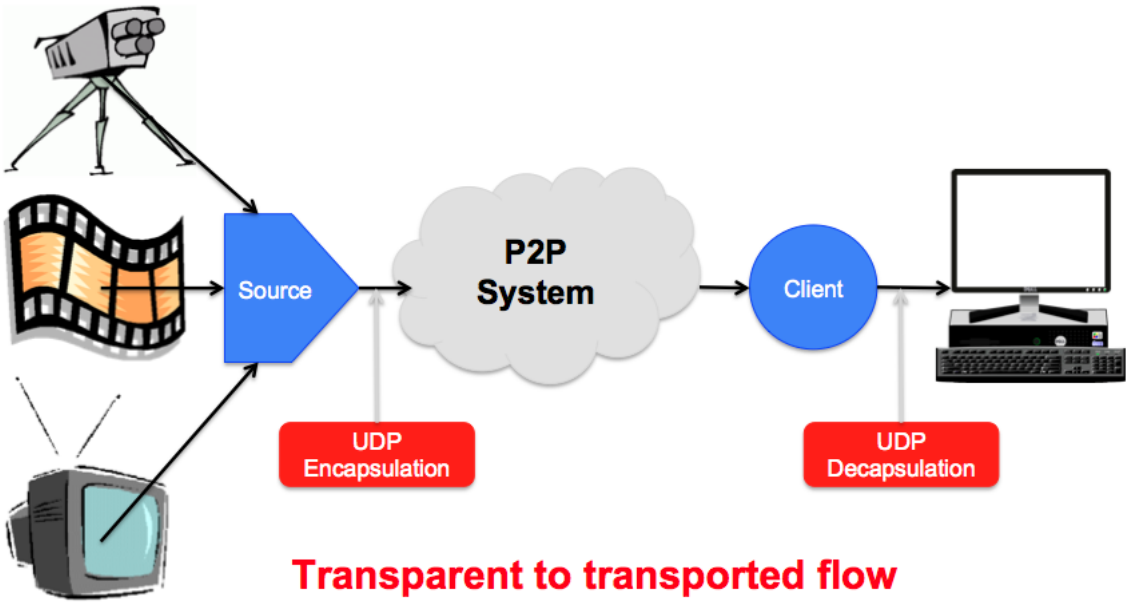

Figure 3.7: System Architecture

From this brief description we can deduce that our system works more or less like a proxy: it creates a bridge between the source of a media stream and several clients who want to reproduce that stream. Indeed, the last component of the transmission chain, in other words the software that reproduces the stream, is not aware of the dissemination strategy adopted by peers.

Chapter 4

Population Variation

In this section we will show how our system handles population variation, that is departures and arrivals. As we all know, in any system for video streaming, no matter to which taxonomy it belongs, population tends toward fast variation and high dynamicity. This means that any system has to efficiently manage departures and arrivals, even if they are grouped. Grouped means that there are several arrivals and departures simultaneously. This issue is highly amplified in a structured P2P system, where two of the most important requirement are stability and consistency.

4.1 Departures

In this section we deeply study the behavior of the system in case of a departure. In this section we assume that departures are strictly spontaneous, instead in the next section we will study how forced departures due to the presence of a non-collaborative peer are managed.

Every departure starts from a peer that decides to leave the system. As we mentioned in previous sections, before leaving the system a peer has to choose a replacement from his repair list. The replacement must be one of peers in the last level, this allows us to minimize changes required in the structure and the amount of signaling blocks to be exchanged.

Before showing in detail our departure strategy, it is worth pointing out that it is used only for non-last levels. Indeed, departing peers from last level do not need any replacement, they just have to safely unjoin their descendant list.

We now explain the departure procedure by showing an algorithm for both departing peer and replacement peer.

Departing peer

```
1: If repair list is empty then
        Ask server for a repair list by sending a 88 block
        Wait for an answer from the server and populate your
            repair list
2: Pick a replacement from the repair list and send to him a
    8 block
3: Wait for an answer from the potential replacement
4: If answer is positive then
        unjoin descendant list
   Else
        If repair list is still non-empty then
                Go to step 2
        Else
                Go to step 1
5: do not leave system as long as there are some peers that
   are not up-to-date
```

Replacement peer

```
If peer is in last level and did not accept yet any
   replacement request then
        send a 18 block (positive ack)
        unjoin descendant list
        send a 4 block to server in order to deallocate
```

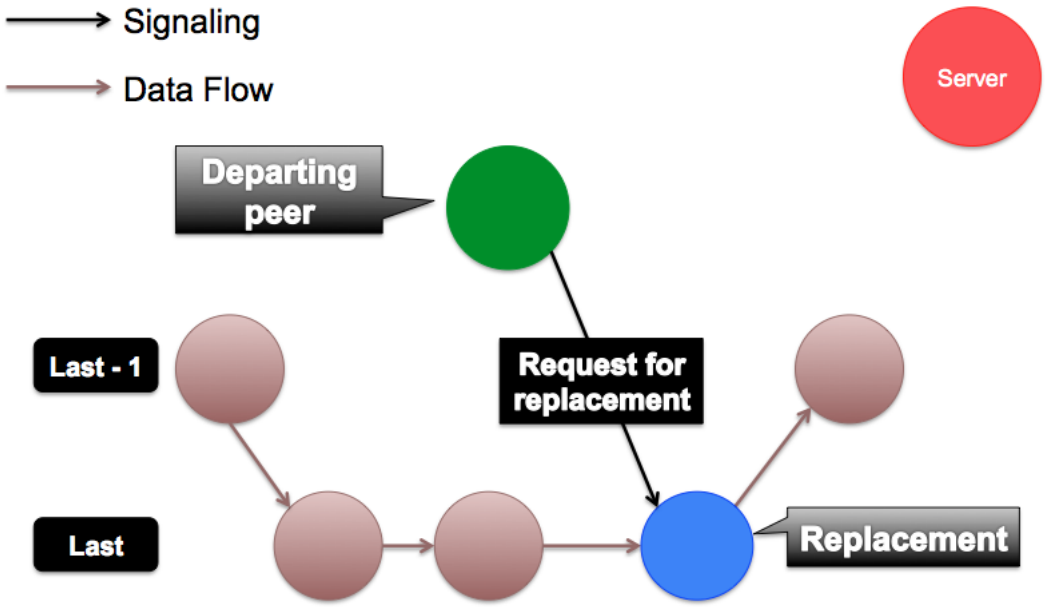

Figure 4.1: Spontaneous Departure: Phase 1

```
        previous ID
    send a 8 block to server in order to update
        coordinates
    reset and initialize all data structures
    send a block type 2 to all new neighbors
    join new descendant list
Else
    send a 28 block (negative ack)
```

As we can see our strategy to handle departure requires just a few changes in the hypercube structure. We recall that the only levels affected by a departure are the last one and the one in which departing peer is, so there is no domino-effect. This characteristic of our strategy allows to keep the number of blocks to exchange between peers as low as possible, resulting in good performance. Indeed, in a video streaming service, an user expects the system to be as stable as possible, regardless of what happens in background during the playback phase.

Another point that is worth highlighting is that a departed peer does not leave

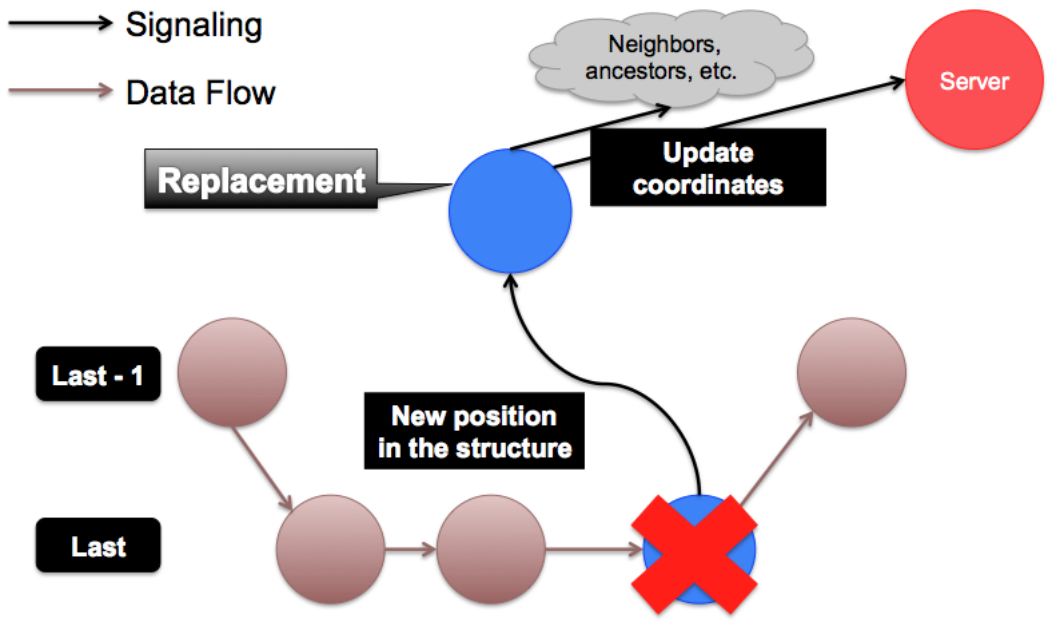

Figure 4.2: Spontaneous Departure: Phase 2

immediately the system but keeps active for a while, in order to check if every neighbor is aware of the arrival of the replacement. Indeed, if departed peer still receives some blocks after leaving the system, he sends to sender a 9 block containing coordinates of replacement. This way we can safely handle massive and grouped departures and keep the structure stable and consistent. We also avoid to lose blocks during replacement procedures since peers that are not up-to-date yet could still retransmit blocks to departed peer, in that case departed peer takes care of retransmitting received blocks.

4.2 Arrivals

Even arrivals are managed in a way that is intended to be efficient and minimize the number of changes in the hypercube structure. We recall that a new peer is always assigned an identifier belonging to last level.

The algorithm executed by a peer during the arrival phase can be summarized as follows:

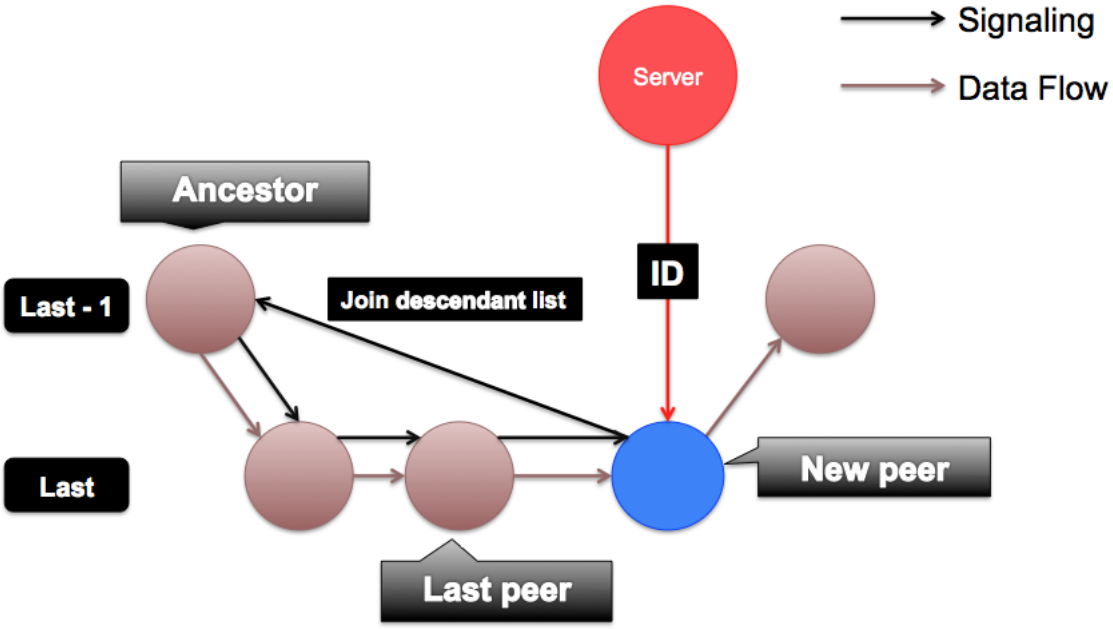

Figure 4.3: Arrival

```
1: Send a block type 1 to server and wait for an answer
2: If we receive an answer then:
        initialize data structure according to dimension and
            assigned id
        send a block type 2 to every neighbor contained in
           cl list of answer block (block type 1)
        join descendant list
        send a positive ack to server (block type 11)
   Else (timeout has expired)
        sleep for a while and resend a join request
```

At this point we have information enough to notice that the way we handle arrivals and departures is intended to favour peers that are connected to the system from longer. Indeed, it is very likely that a peer in last level will be eventually chosen as replacement for a peer in a upper level, resulting in lower delays and better performance. Moreover, users that stay connected to the system for a short time, for instance those users who switch from a channel to another very often, do

not affect quality of service achieved by users connected from longer. Generally, peers in upper levels are not aware of changes in last level. From some point of view, last level can be seen as a temporary container for just joined peers, who can be promoted to upper levels if they keep connected to the system for long enough.

Chapter 5

Failure Detection

As we mentioned in previous sections, in every peer-to-peer system there is the need to deal with sudden failures of peers. Usually a peer is considered as failed in two cases:

- he has suddenly crashed;
- he is affected by a network issue so he is completely isolated and cannot communicate with other peers.

In our case, we extended the definition of failure in order to include the case in which a peer is still active but no longer able to properly cooperate. In simpler words, peer's upload capacity is above the minimum threshold. This can happen because of several (and sometime unpredictable) reasons such as a high computational of the machine on which the client process is running, some interferences along the wireless channel, etc.

We also did some experiments in order to know exactly which is the minimum threshold for a video stream. We assume that the quality of service achieved is not sufficient anymore if during the playback, noises become very frequent. During experiments we ran two VLC processes on two different machines located in the same LAN, the first one was the source and the second one was the destination. We thus created a video stream one the first machine by using built-in features of VLC.

Instead, the VLC process running on the second machine was listening to a given port in order to reproduce the video stream. The transmission rate was varying between 300 and 400 kb per second and the encapsulation format was MPEGTS. We used the linux tool *netem* in order to inject artificial losses and delays.

The measurements obtained through these experiments can be summarized as follows:

DELAY TYPE	**DELAY RANGE**	**LOSS TYPE**	**LOSSES PER-CENT-AGE**	**QOS**
Random	20-150 Ms		No Loss	OK
Random	20-150 Ms	Random	1%	OK (Some Little Noises Affecting Mainly Audio)
Random	20-150 Ms	Random	3%	OK (Some Little Noises Affecting Mainly Audio)
Random	20-150 Ms	Random	5%	OK (Some Little Noises Affecting Mainly Audio)
Random	20-150 Ms	Random	10%	OK (More Noises, Not Suitable To Watch A Movie)
Random	20-150 Ms	Random	15%	NO (Too Many Noises)
Random	20-150 Ms	Burst	1%	OK (Some Little Noises)
Random	20-150 Ms	Burst	3%	OK (Some Little Noises)
Random	20-150 Ms	Burst	5%	OK (Some Little Noises)
Random	20-150 Ms	Burst	10%	OK (More Noises, Still Suitable To Watch A Movie)

Random	20-150 Ms	Burst	15%	OK (More Noises, Not Suitable To Watch A Movie)
Random	20-150 Ms	Burst	20%	NO (Too Many Noises)
Random	150-400 Ms	Burst	No Loss	OK
Random	150-400 Ms	Burst	1%	OK (Some Little Noises)
Random	150-400 Ms	Burst	3%	OK (Some Little Noises)
Random	150-400 Ms	Burst	5%	OK (Some Little Noises)
Random	150-400 Ms	Burst	10%	OK (More Noises, Not Suitable To Watch A Movie)
Random	150-400 Ms	Burst	20%	NO (Too Many Noises)

Rows highlighted in red correspond to those experiments in which the quality of service was not good enough.

As we can deduce from the table above, generally, a percentage of losses equal to 10% can be considered as the maximum threshold for losses. During our experiments we noticed that in some cases highly variable delays affect QoS even more than losses. In addition, we noticed also that random can have worse effect compared to consecutive losses. This can be explained by observing that packets generated by streaming protocols differ by each other, depending on the importance of the packet. Indeed, there are some kinds of packets that can be considered less crucial than the others. Clearly, consecutive losses will have a higher probability to involve also crucial packets.

5.1 Detection Strategy

We are now ready to deeply study our detection strategy. Our system model is partially inspired from the one introduced in [11]. In particular, our approach can be classified as follows (according to the taxonomy introduced in [11]):

- **Sharing information:** our approach is based on sharing information. In other words, the detection process is supported by information continuously shared between peers during transmission. This approach is supposed to be faster in terms of detection time, since the first peer that detects the failure can announce this to everyone else. Concerning announcements, our approach is slightly different than usual.

- **Negative vs Positive information:** our approach can be seen as a mix of these two approaches. Indeed, in our system we use a special data structure called *report* which can contain, depending on concerning peer, either positive or negative information.

Our approach does not adopt any keep-alive mechanism. The main reason why we made this decision is that we do not need to receive any update from our neighbors since we are supposed to constantly receive blocks from them. If we do not receive blocks enough or at all from a given neighbor, then that neighbor is marked as suspected.

In our approach, each peer is associated to a leader, who is also one of his neighbors. A leader is a special peer who is in charge of collecting reports concerning a given neighbor and deciding if he must be forced to leave the system or not. Even in this case, the leader assigned to each peer is computed by just looking at identifiers. Thus, we do not need any additional information that depends on the current state of the system.

For each peer, all his neighbors keep track of data blocks received and non-received in order to compute a report. The report is a special data structure which

contains 4 fields:

- id of *frompeer* (neighbor of concerning peer);
- id of *concerningpeer* (neighbor of leader);
- throughput;
- # losses.

Throughput and losses can both assume vales between 0 and 10. In particular, threshold shows how many blocks have been received during a given time slot, instead losses shows how many packets are missing. It is worth specifying that a block is considered lost only if a peer has received both the previous one and the next one.

When a peer receives a data block, he checks if some reports have been piggy-backed. If so, he stores the report and then decides if the concerning peer of that report must leave the system or not. As we mentioned above, a given peer is forced to unjoin the system if and only if his cooperation level is lower than the minimum threshold.

A leader receives reports concerning a given peer, from one of his neighbors, who can thus be seen as a router. The router peer is also chosen according to his identifier. In fact, if a peer F wants to send reports concerning a given neighbor C, he first computes the id of his leader L, then he retrieves the id of the only peer who can function as router R, that is the second neighbor in common between peers F and L. At each time slot, peer R checks if there are some reports to send to the destination peer. If so, he attaches to the data block the report with the highest priority, in order to deliver it to the leader. In 5.1 we show relations between peers F (frompeer), C (concerningpeer), R (router) and L (leader) and how a report is actually delivered to a leader.

We recall that neighbors' identifiers differ by only one bit and accordingly, the identifier of a neighbor of a neighbor differs by two bits. In this case, if peer F wants

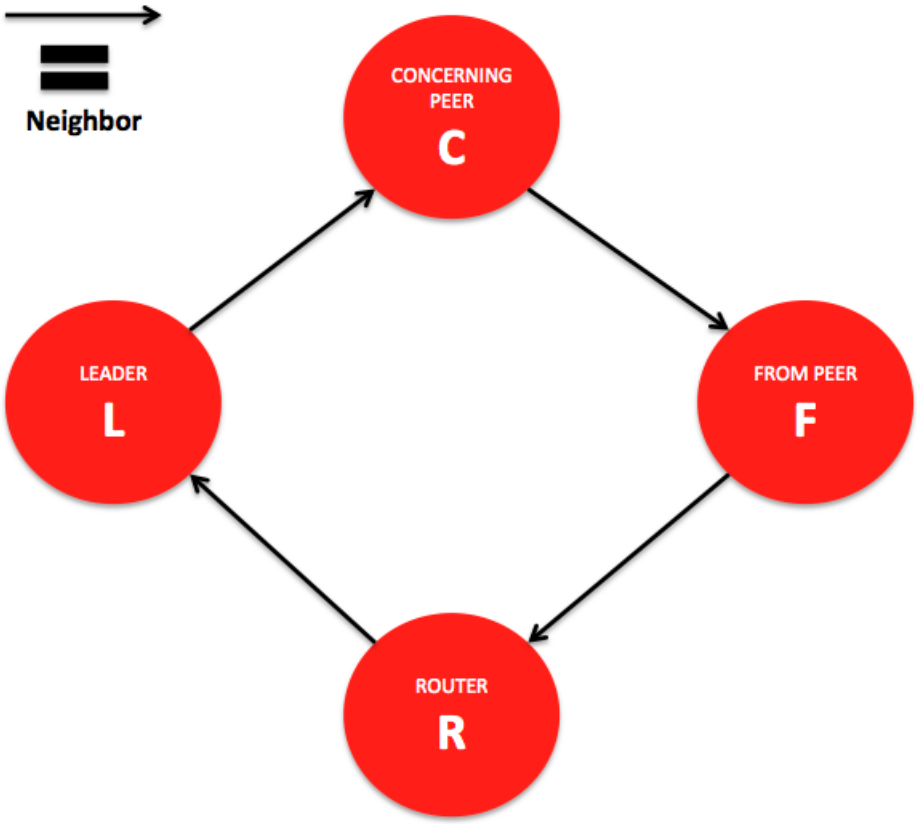

Figure 5.1: How reports are delivered to a leader

to deliver a report concerning peer C to peer L, he has to use peer R as router. Clearly, he cannot use peer C as router since peer C could fail and not to deliver reports anymore.

It is worth pointing out that in our system reports are not delivered by sending a special block type but they are piggybacked with data blocks, this avoids to add additional overhead. In addition, this choice allows to exploit the neighborhood relation between peers. In other words, if two peers are neighbors, they periodically exchange data blocks which can be used also to send further information about the current status of the transmission.

As we mentioned above, when a leader receives a report, he stores it into a special data structure, which is called *average_ reports*. Throughput and losses values of an average report are computed taking into account not only last received report but also previous reports. In order to do so, we defined the following formulas:

$$THR = \alpha \cdot (NEWTHR - THR) + THR$$

$$LOS = \alpha \cdot (NEWLOS - LOS) + LOS$$

where α is a coefficient used for assigning a weight to information just received and $NEWTHR$ and $NEWLOS$ are respectively throughput and losses values just received.

At this point, we can finally define the rule that each leader implements in order to decide if a peer has to be expelled from the system. Once a leader receives a new report, he stores it and then he checks if throughput provided by concerning peer is lower than the minimum threshold for at least two block numbers. It is worth noticing that we consider block numbers instead of single peers. The goal of this choice is to make our failure detection system resilient to false negatives. Indeed, it can happen that a peer A retransmits data blocks corresponding to a given block

number N to multiple peers, for instance peers B and C. If the source S of block number N has failed, peer A will never retransmit block number N, thus both B and C will never receive it. If we just considered peers instead of block numbers, leader of peer A would decide to expel peer A from the system since he will certainly receive negative reports by both B and C.

5.1.1 Implementation Details

We now show some technical details on the implementation of piggyback mechanism and functions specifically intended for failure detection.

5.1.1.1 Piggyback

A special function called *piggyback_Reports* is called every time we retransmit a data block. This function can be divided into two parts: the first decides how to fill the position available for reports, the second one fills the second position. The first position has to contain a report for which we function as router, so the destination is supposed to be a leader for that report. On the other hand, the second position has to contain a report for which we function as frompeer, so the destination is supposed to be a router for that report. It can happen that for a given leader, we have multiple reports that should be attached, in this case we compute the priority of each report and we thus attach the one with the highest priority.

The implementation of the function to compute report priority is the following:

```
1 static double_t get_report_priority(Report_History *rhistory)
2 {
3     struct timeval now, sub;
4
5     gettimeofday(&now, NULL);
6     timersub(&now, &(rhistory->last_time_sent), &sub);
7     double_t ms = sub.tv_sec*1000 + (sub.tv_usec/1000);
```

```
8
9       return (ms/(rhistory->nsent+1)) * (rhistory->losses[
            get_last_report_index(rhistory)] + 1);
10  }
```

where rhistory correponds to a an element in *route_reports* array.

Priority computation of those reports sent as frompeer is slightly different since information about losses and throughput are stored in another data structure and can be computed locally. The function intended to compute priority of local reports is the following:

```
1  double_t get_local_report_priority(Report_host *hreport, struct
       timeval *timestamps, uint32_t losses)
2  {
3       double_t priority;
4       struct timeval now, sub;
5
6       gettimeofday(&now, NULL);
7       timersub(&now, &(timestamps[hreport->concerningpeer]), &sub);
8       double_t ms = sub.tv_sec*1000 + (sub.tv_usec/1000);
9       priority = ms * (losses + 1);
10
11      return priority;
12  }
```

where *timestamps* is an array where we store the exact time at which a report has been sent.

5.1.1.2 Functions for Failure Detection Implementation

We now show the implementation of three very important functions for failure our failure detection system.

Leader

As we can easily deduce, this function is intended to compute the identifier of the leader associated to a given peer. As we can see, leader's id is computed by just changing the k^{th} of *peer_id*, where k is the result of the modulo operation between *peer_id* and *dim*. Once again, no additional information are required to the role of each peer.

```
static uint32_t leader(uint32_t peer_id, uint32_t block_number,
    uint32_t dim)
{
    // we compute the modulo between peer_id and dim and we
       change the value of that bit
    int modulo = peer_id % dim;
    uint32_t leader = peer_id ^ (1 << modulo);
    return leader;
}
```

Leave

Leave function is executed by a leader in order to decide if the with identifier equal to *peer_id* must be expelled or not.

```
static int leave(uint32_t leader_id, uint32_t peer_id,
    Average_Report **reports, int dim)
{
    // STRATEGY:
    // for each neighbor of peer_id (except ourselves)
    // we check if we have received reports such that throughput
       is less than 90% (5)
    // for at least two different block numbers
    int missing_blocks = 0;
    int ii, pos;
    uint32_t shifted = 1;
```

```
uint32_t xored_id;
int first_index, second_index;
int block_numbers[MAX_DIM];

memset(block_numbers, 0, sizeof(int)*MAX_DIM);

// we compute the index of peer_id
// in order to access the data structure
xored_id = leader_id ^ peer_id;
nb_bits(xored_id, dim, &first_index, &pos);

for (ii = 0; ii < dim; ii++)
{
    int blk_nmb = -1;
    // neighbor's id
    uint32_t neighbor_id = peer_id ^ shifted;
    // if the neighbor is not ourselves
    if (neighbor_id != leader_id)
    {
        second_index = ii;
        Average_Report *average_report = &(reports[
            first_index][second_index]);
        if (average_report->nreports >= MIN_REPORTS &&
            ((long int)average_report->throughput) <
                THROUGHPUT_THRESHOLD)
        {
            blk_nmb = neighbors_block_numbers(neighbor_id,
                peer_id, dim);
            block_numbers[blk_nmb]++;
        }
```

```
        }
        // next neighbor
        shifted <<= 1;
    }

    // now we check if there are at least two block numbers
    // for which the throughput is lower than the threshold
    for (ii = 0; ii < dim; ii++)
    {
        if (block_numbers[ii] > 0) missing_blocks++;
    }

    // to force a peer to leave the system, we need to meet one
       of the following conditions:
    // 1) at least two peers do not achieve a good QoS from
       peer_id
    // 2) the suspected peer has only one bit equal to 1 and
       peers that receive blocks from peer_id
    //    do not achieve a good QoS
    if (missing_blocks >= FAILURE_THRESHOLD || (missing_blocks >
       0 && number_of_ones(peer_id, dim) < FAILURE_THRESHOLD))
        return 1;
    else
        return 0;
}
```

Route

This function is intended to compute the identifier of the router peer that *our_id* is going to use in order to deliver a report to *peer_id*. As we can see, there is an additional parameter called *peer_to_skip* which is used in order to avoid that

concerning peer is used as router to deliver reports to his own leader.

```
static uint32_t route(uint32_t our_id, uint32_t peer_id, uint32_t
    peer_to_skip, uint32_t dim)
{
    // if we want to reach ourselves
    if (peer_id == our_id) return peer_id;

    // we want to find a peer such that is a neighbor of peer_id
    // in order to do so, we should know the position of the bits
        that are different
    int ii, lpos, rpos;
    uint32_t xored_id = our_id ^ peer_id;

    nb_bits(xored_id, dim, &lpos, &rpos);

    uint32_t bridge_peer;
    uint32_t shifted;

    shifted = 1 << lpos;
    bridge_peer = our_id ^ shifted;
    if (bridge_peer == peer_to_skip) {
        shifted = 1 << rpos;
        bridge_peer = our_id ^ shifted;
    }

    return bridge_peer;
}
```

5.2 Recovery Strategy

After deciding if a peer is failed or not, the next task to accomplish is to find a replacement candidate and start the procedure to safely replace him in the structure. At design time, we thought about several possible solutions, at the end we decided to keep the procedure for a forced departure as simple as possible and as similar possible to the one for spontaneous departures.

In order to achieve so, the leader sends a special request to the server (block type 102) which means that leader is asking server to create a block of type 81, which is a slight variation of block type 8. Indeed, the only difference between a 8 block and a 81 block is the first element of the cl list attached to the block. In block of type 81, server adds coordinates of future successor in the descendant list. The reason why server needs to add this information is that if replacement peer will just join the new descendant list as usual, his future predecessor will put him exactly between himself and the failed peer, instead of isolating the failed peer. On the other side, future predecessor has not a clue what coordinates of successor of failed peer are, so he just cannot give this information to replacement peer.

After receiving a block of type 102, server computes identifiers of neighbors of failed peer and fill the cl list. We recall that server can easily retrieve these information since he constantly is up-to-date on the status of the hypercube structure, especially on their coordinates. The most important advantage of our approach is that even if the block type we use has a different type, the behavior of replacement peer does not change compared to block type 8. Indeed, the behavior slightly changes only during the joining phase, in which instead of setting successor's coordinates to values attached to the cl list of block 6 or 16, replacement peer uses coordinates attached to the block received from the server.

In Figure 5.2 we can see what actually happens when a leader decides to expel a peer from the system. In Figure 5.3 we can see what replacement peer does in order to safely replace failed peer.

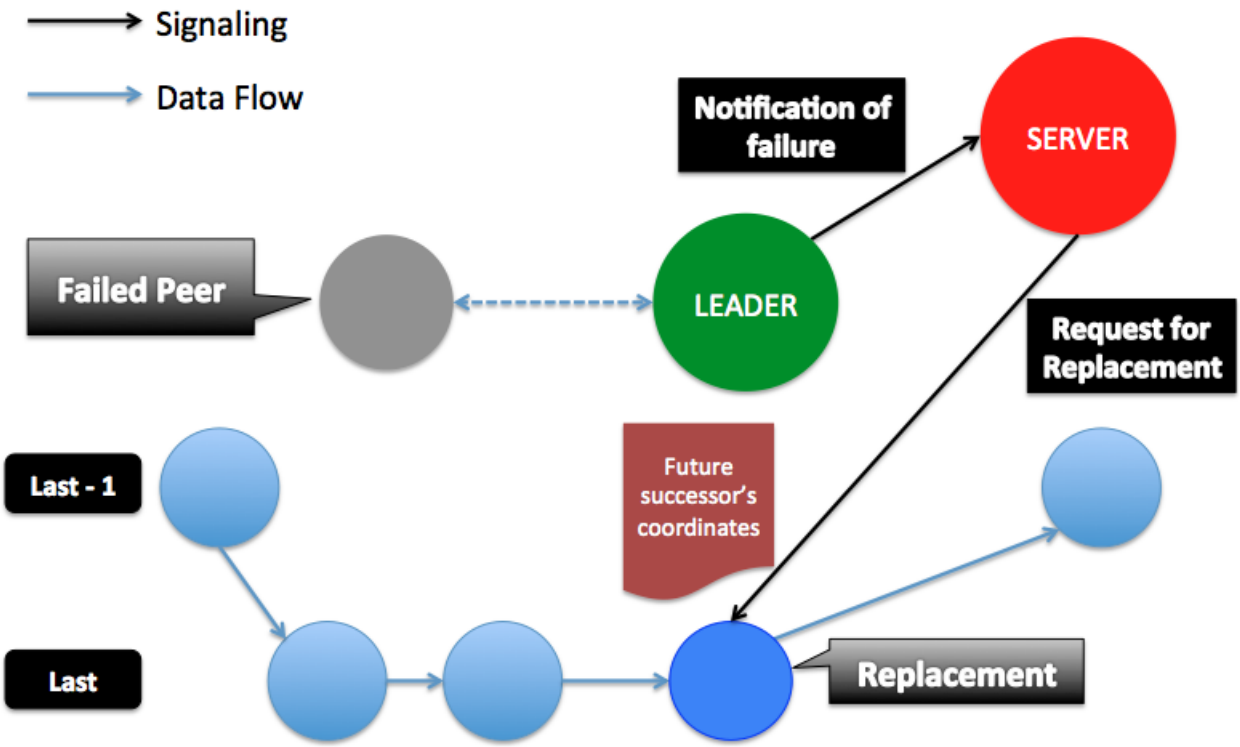

Figure 5.2: Forced departure: Phase 1

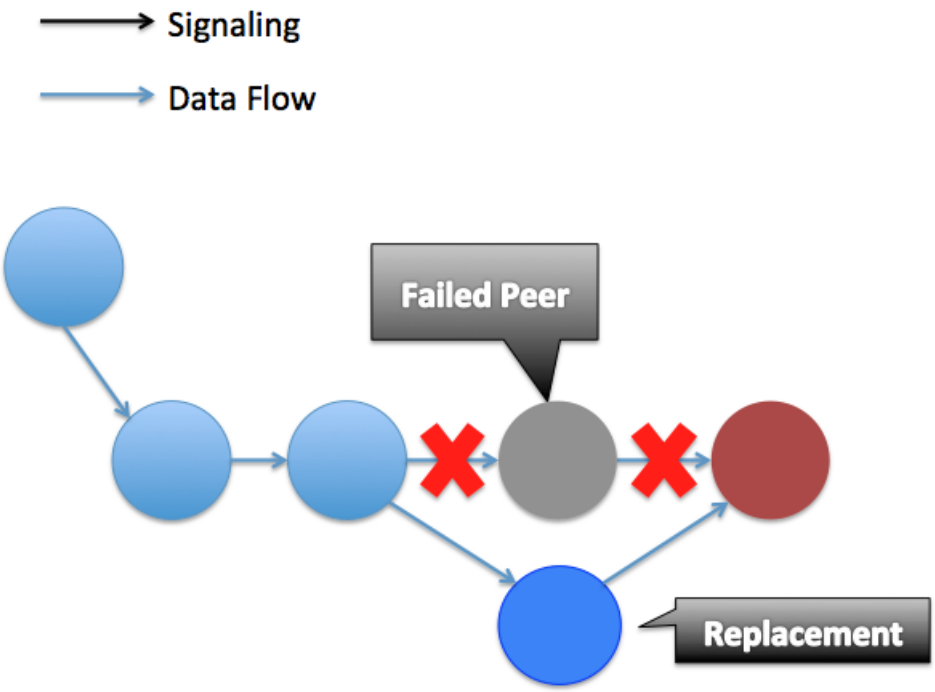

Figure 5.3: Forced departure: Phase 2

Chapter 6

Simulations and Experiments

6.1 Experiments on PlanetLab

Some of our experiments have been performed on PlanetLab. PlanetLab is a global research network that supports the development of new network services. PlanetLab can be used by researchers in order to develop new technologies for several network services including peer-to-peer systems. PlanetLab currently consists of 1128 nodes at 544 sites. Most of the machines composing PlanetLab network are hosted by research institutions, although some are located in co-location and routing centers (e.g., on Internet2's Abilene backbone). All of these machines are connected to the Internet.

The reasons why we chose PlanetLab for testing are the following:

- we can have access to a large set of geographically distributed machines.
- We can run our experiments in a realistic network substrate that experiences congestion, failures, and diverse link behaviors. As we will show in Section 6.3, in PlanetLab it is very common for a peer to experience losses and delays as well as failures.
- As PlanetLab's machines host several processes (belonging to other users) at

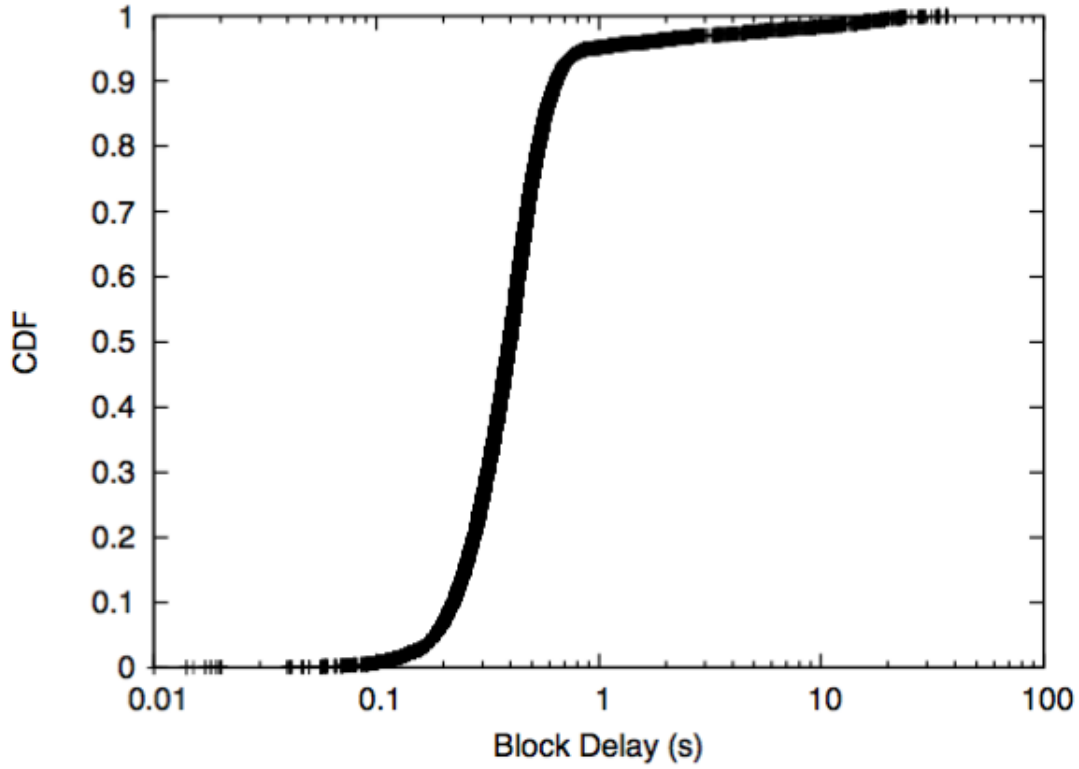

Figure 6.1: CDF of block delays

the same time, peers can experience a quite realistic and unpredictable client workload.

Thanks to all these features, PlanetLab has proved to be the best solution to allow to make realistic measurements, especially in terms of network and performance constraints. The environment provided by PlanetLab is pretty stringent and results obtained over it would be realistic enough to validate our P2P framework.

The first results are presented in 6.1. There are 480 peers connected to a hypercube structure of dimension 10. The machines hosting peer processes were distributed over North and South America, Asia, Australia, and Europe. Despite of the inherent randomness, we noticed that 97% of reception delays were under one second!

Second experiment was performed in order to test grouped arrivals while the streaming was being transmitted. 6.2 presents the peer population as function of

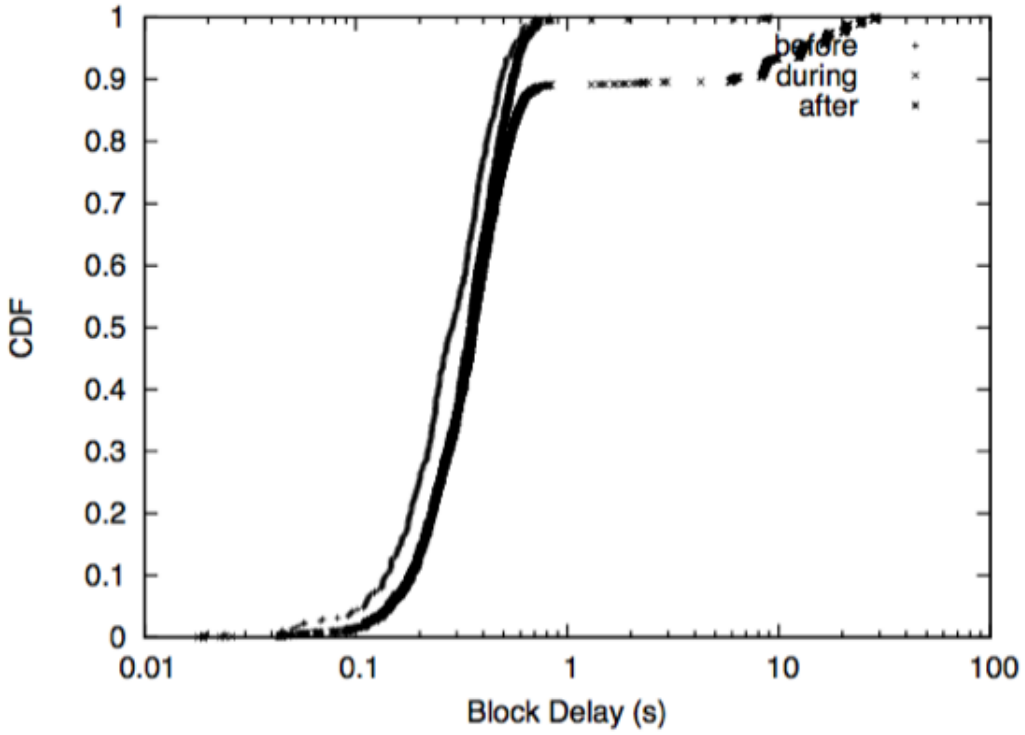

Figure 6.2: CDF of block delays before, during, and after grouped arrivals

time. The population is initially composed of 70 peers which means we created 70 peers, attributed then position in hypercube and then started source that generates stream. At time 80 sec a group of 70 more peers arrives. The population stabilizes at time 100 sec and then stays stable for the rest of the test. We present in 6.2 the distribution of delays observed before the grouped arrivals (i.e. before time 80 sec), during the arrival (i.e. time 80 sec to 100 sec), and after the arrivals. During the entire time streaming was on. We note that even during the arrival period, the reorganization results in 90% of the delays being below 1 sec. After the grouped arrivals the delays return to their previous value with a very large majority of block delays below one second.

The last experiment concerned grouped departures. A total of 130 peers were created and positioned in hypercube after streaming was started . Then 16 peers were asked to leave the system. In 6.3, we can see the delays before group departure takes off and while it is taking place. Notice that the performance are not impacted

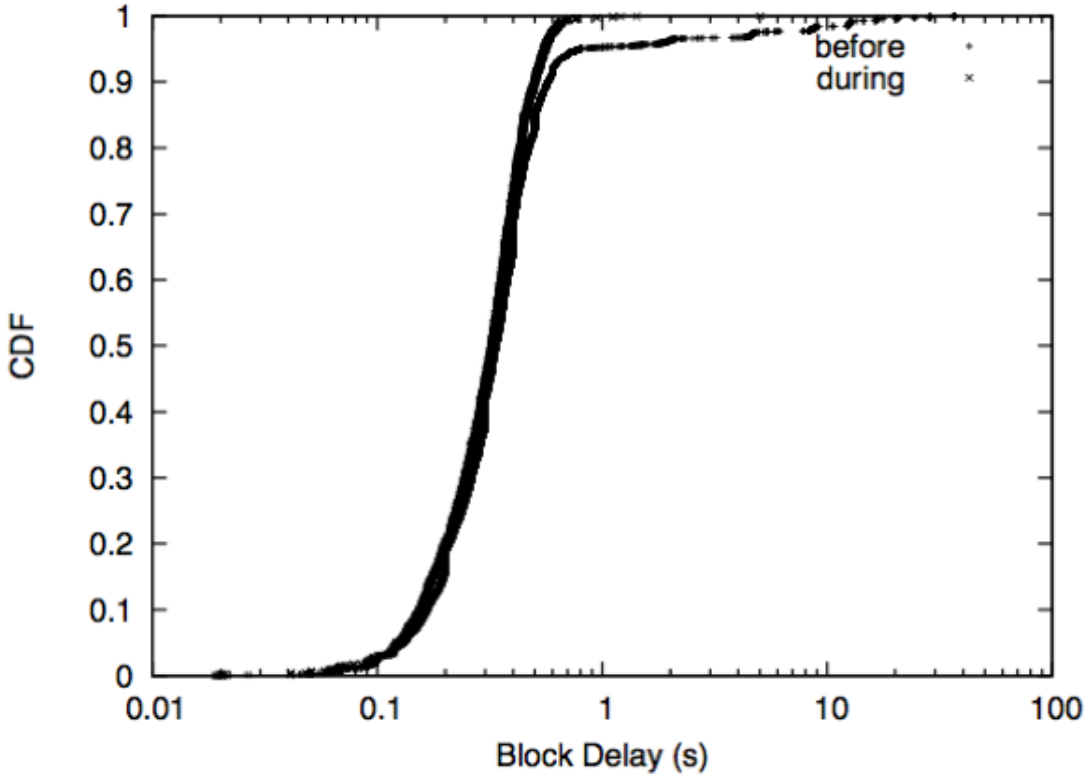

Figure 6.3: CDF of block delays before and during grouped departures

by departures.

6.2 Experiments for Failure Detection

In this section we show results concerning our solution for failure detection. In particular, we did three experiments in order to simulate the following scenarios:

1. in the first experiment we just kill one of the processes in order to simulate a sudden crash;

2. in the second experiment, we used netem in order to simulate an unstable who was experiencing a loss rate equal to 20% and highly variable delays between 1500ms and 300ms;

3. in the last experiment, we used again netem in order to simulate an unstable peer who was experiencing a loss rate equal to about 40% .

Experiment	**Loss rate**	**Delays**	**Lost Blocks**	**Sent Blocks**
1	100%	$<$ 10ms	4	
2	20%	300-1500ms	3	20
3	40%	$<$ 10ms	4	15

As we can see from the table, in all the experiments the system was able to correctly detect the failure fast enough, losing not more than 4 data blocks. This result can be considered quite good since 4 losses do not generate many noises in the playback. Moreover, in the second and the third experiment, the total number of data blocks received in the interval between the first and the last loss, is at most 20. The mean number of data blocks exchanged in one second of transmission for a stream of medium quality is about 30, this means that the system is able to correctly

detect the failure in less than one second.

Chapter 7

Summary of Work Done

The duration of the internship was five months and a half. The first part of the internship was focused on studying the state of the art in the field of P2P live streaming, performance criteria, main issued and so on. During the first part of my internship I studied also the work done by my predecessor, Anshuman Kalla. This first part of the internship took about one month. The next three months were focused on several tasks including debugging and experiments. However, the main activities were the following:

- implementing the new approach to manage descendant lists, departures and arrivals;
- implementing the new system architecture in order to make the system able to carry a RTP/UDP stream without manipulating or being aware of the content.

The last period was thus focused on studying the failure detection problem and designing, and then implementing, our solution for detection and replacement of non-collaborative peers.

Finally, I spent the last week of the internship doing several experiments on PlanetLab (see [1]).

In Figure 7.1 there is an approximate measure of the time of the internship focused on debugging and experiments, development and research.

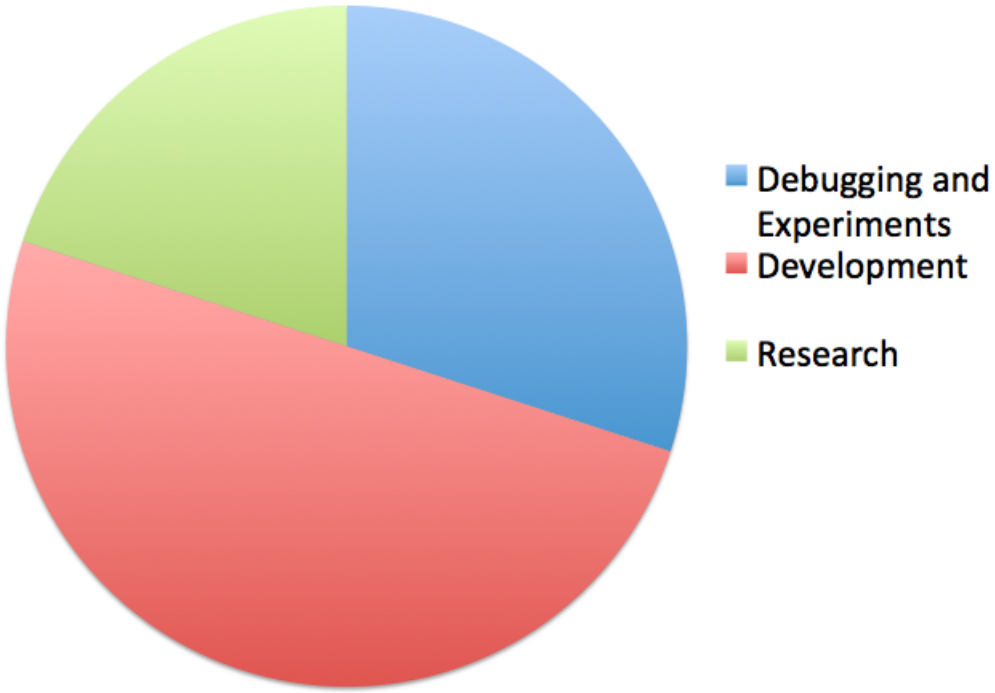

Figure 7.1: A summary of my internship

Chapter 8

Conclusions

The current status of this project is much better than it was before my internship. We indeed managed to fix plenty of bugs and dramatically improve stability and reliability of both peers and the entire system. We also managed to radically change the way peers manage descendant lists resulting in a system constantly consistent and more stable, where stable means that consistency of system is not even affected by massive departures and arrivals. Even the implementation of the mechanism to carry a RTP/UDP stream was an important part of my internship. However, the most important task of my internship was the one focused on failure detection and recovery. Indeed, that part was really interesting and challenging, especially because of the high number of issues to face and solve.

In the end, we can be very proud of progresses we did during my internship since the system has considerably improved and we managed to achieve even more than what we planned at the beginning of our work.

Chapter 9

Future Work

Before making the system actually suitable for real users, there still are a couple of improvements that need to be achieved:

- our failure detection solution efficiently deals with non-collaborative peers in intermediate levels, but it is does not work perfectly in case of non-collaborative peers belonging to last level;
- at the present time, peers inside the hypercube who do not want to cooperate (free riders) can be expelled from the system. Unfortunately, those peers can still try to rejoin the system and could result in a consistent free riding attack;
- with the change in number of peers in our P2P system, it might be necessary to decrease or increase the dimension of the hypercube structure. This means that if the number of peers grows beyond the current handling capacity of a hypercube then it is necessary to increase somehow the dimension of the structure. Alternatively, if the number of peers reduces then one might take decision to merge two existing hypercubes into single hypercube. This issue requires careful thinking and development;
- peers with better resources (especially in terms of upload capacity) should be placed at top levels of the hypercube since contribution offered by peers in top

levels is more critical. In order to achieve so, a contribution-aware policy could be used. By using such policies the system can know which peers participate actively and thus can move them to top levels of hypercube. In other words, peers with better resources and high age could be moved to top levels;

- at the present time, the system does not provide any authentication system. This means that the access to the structure, as well as the exchange of blocks is totally insecure, resulting in a high vulnerability to external attacks. Also block signing and encryption mechanism could be used in order to avoid video corruption attacks. This must be done keeping in mind time constraints of video streaming;

- finally, in order to make the system suitable for real users, a graphical user interface should be developed.

List of Figures

1.1 An example of tree-based P2P system 13
1.2 An example of multi-tree P2P system 13
1.3 An example of mesh-based P2P system 14

3.1 Hypercube Structure 22
3.2 An example of application of dissemination rules 26
3.3 An example of descendant list in last level 27
3.4 Entities of the system 30
3.5 File organization 32
3.6 An example of descendant list 35
3.7 System Architecture 54

4.1 Spontaneous Departure: Phase 1 57
4.2 Spontaneous Departure: Phase 2 58
4.3 Arrival 59

5.1 How reports are delivered to a leader 66
5.2 Forced departure: Phase 1 75
5.3 Forced departure: Phase 2 75

6.1 CDF of block delays 77
6.2 CDF of block delays before, during, and after grouped arrivals 78
6.3 CDF of block delays before and during grouped departures 79

7.1 A summary of my internship 83

Bibliography

[1] Planetlab website.

[2] S. Khuller A. L. Chow, L. Golubchik and Y. Yao. On the tradeoff between playback delay and buffer space in streaming. In *Parallel and Distributed Processing Symposium*, volume 0, pages 1–12, 2009.

[3] C. Feng and B. Li. Understanding the performance gap between pull-based mesh streaming protocols and fundamental limits. *INFOCOM 2009, IEEE*, pages 891 –899, 2009.

[4] N. Hegde, F. Mathieu, and D. Perino. Size does matter (in p2p live streaming). In *CoRR*, volume abs/0909.1713, 2009.

[5] X. Hei, Y. Liu, and K. Ross. Inferring network-wide quality in p2p live streaming systems. *Selected Areas in Communications, IEEE Journal*, 25(9):1640 –1654, December 2007.

[6] S.L. Johnsson. Optimum broadcasting and personalized communication in hypercubes. *Computers, IEEE Transactions*, 38(9):1249–1268, Sep 1989.

[7] H. Katseff. Incomplete hypercubes. In IEEE Transactions, editor, *Computers*, volume 37, pages 604–608, 1988.

[8] Y. Liu. On the minimum delay peer-to-peer video streaming: how realtime can it be? In *MULTIMEDIA 07*, Proceedings of the 15th international conference on Multimedia, pages 127–136. ACM, 2007.

[9] Jenn-Yang Tien, Ching-Tien Ho, and Wei-Pang Yang. Broadcasting on incomplete hypercubes. *IEEE Transactions on Computers*, 42(0018-9340):1393–1398, 1993.

[10] N.-F. Tzeng and H. Kumar. Traffic analysis and simulation performance of incomplete hypercubes. In IEEE Transactions, editor, *Parallel and Distributed Systems*, volume 7, pages 740–754, jul 1996.

[11] Shelley Q. Zhuang, Dennis Geels, Ion Stoica, and Randy H. Katz. On failure detection algorithms in overlay networks. In *IN IEEE INFOCOM*, 2003.