

# **RELIABLE PEER-TO-PEER MULTICAST STREAMING**

by

Sushant Gautam

A thesis submitted to

the Faculty of Business and Information Technology

In conformity with the requirements for the degree of MSc in Computer Science

University of Ontario Institute of Technology

Oshawa, Ontario, Canada

(January 14, 2013)

Copyright ©Sushant Gautam, 2013

## **Abstract**

P2P is increasingly gaining its popularity for streaming multimedia contents. The architecture of streaming has shifted from traditional client server architecture to P2P architecture. Although it is scalable and robust it faces its own challenges and problems such as churn. In tree topology frequent joining and leaving of users in search for better quality and reliable streaming makes the P2P network instable. This thesis provides an effective approach to achieve a resilient network for streaming. Relying on a single tree to receive data from single parent may leave the user deprived of getting the data if any of its ancestors leaves the network. Therefore we present an ideal solution to this problem by introducing a backup tree for the existing base tree. The backup tree is constructed based on parameter such as bandwidth and delay. In case of failure of a node, its children along the tree receive the data from the nodes of backup tree. We present an efficient algorithm for the construction of base tree as well as the backup tree which are based on normalization of two entities of nodes: bandwidth and delay. Through mathematical formulation and experimental setups we show that introducing a backup tree for an existing base tree can help provide resilience to the network.

## **Acknowledgements**

I would like to take this opportunity to express my sincere appreciation to my thesis supervisor, Dr Ying Zhu for her expert guidance and valuable feedback throughout the research. I deeply thank my parents, brothers and friends for their continual encouragement and support. Finally, many thanks go to all of those who helped me in any respect during the completion of this thesis.

# Table of Contents

Abstract.....	ii
Acknowledgements.....	iii
Chapter 1 Introduction.....	1
1.1 Introduction:.....	1
1.2. Background.....	4
1.2.1 P2P Overview.....	4
1.2.2 Live Streaming Architectures in Current Schemes.....	4
1.2.2.1 Client- Server Architecture:.....	5
1.2.2.2 Limitations of Client-Server Architecture:.....	7
1.2.2.3 P2P Architecture:.....	7
1.2.2.4. Challenges of P2P Architecture.....	11
1.3. Motivation and Objective.....	14
1.4. Tools Used.....	15
1.4.1 Python/Twisted:.....	15
1.4.2 Networkx (Network Visualizer):.....	18
1.4.3 Tkinter (UI Application):.....	19
1.5. Thesis Outline.....	21
Chapter 2 Related Works.....	23
2.1 Live Streaming Applications.....	23
CoolStreaming:.....	23
PPLive:.....	25
2.2 Multicast Tree and Reliability.....	27
Chapter 3 Reliable Multicast.....	31
3.1 Definition (Reliability):.....	31
3.2 Example:.....	33
3.3 Proposition:.....	37
3.4 Proof:.....	37
Chapter 4 Implementation.....	41
4.1 P2P Implementation in Twisted (Server and Node).....	41
4.1.1 Server:.....	41
4.1.2 Node:.....	44
4.2 Input Conversion:.....	47

4.2.1 BRITE Nodes:.....	48
4.2.2 Input Conversion Algorithm: .....	50
4.2.3 P2P Input Nodes: .....	52
4.3 Topology Formation: .....	53
4.3.1 Node Arrival (Joining of Peers):.....	55
4.3.2 Node Departure (Leaving of Peers) .....	58
4.4 Base Tree Construction Algorithm: .....	59
4.5 Backup Tree Construction Algorithm:.....	62
4.6 Results.....	65
4.6.1 Comparison (Random vs. Algorithm).....	65
4.6.2 Bandwidth Analysis .....	67
4.6.3 Delay Analysis .....	68
Chapter 5 Conclusion.....	71
Bibliography (or References).....	72

## List of Figures

**Figure 1.1 Client-Server Architecture**

**Figure 1.2. Mesh Topology**

**Figure 1.3 Single Trees vs. Multiple Trees**

**Figure 1.4. A simple Networkx graph**

**Figure 1.5. A simple Tkinter UI Application**

**Figure 2.1 System Diagram for DONet Node**

**Figure 2.2 The system architecture of PPLive P2P-VoD**

**Figure 3.1 Two Alternate Multicast Tree**

**Figure 3.3. An optimal Tree**

**Figure 3.4 A complete Multicast Tree T'**

**Figure 3.5 Swapping the sub tree  $W_1$  and  $W_j$ .**

**Figure 4.1 Content Producer Class**

**Figure 4.2 Server Protocol Class**

**Figure 4.3 Server Factory Class**

**Figure 4.4 A P2P Server Class**

**Figure 4.5 Node Protocol Class**

**Figure 4.6 Node Factory Class**

**Figure 4.7 A Complete Node Class**

**Figure 4.8 Input Conversion Process**

**Figure 4.9 Nodes generated from BRITE**

**Figure 4.10 Fields and its meaning of BRITE outputs fields**

**Figure 4.11 Input Conversion Algorithm**

**Figure 4.12 P2P Input Nodes**

**Figure 4.13 Data structure for storing nodes**

**Figure 4.14 Server Log**

**Figure 4.15 A base and its backup tree**

**Figure 4.16 Normalization Algorithm**

**Figure 4.17 Node Selection Algorithm**

**Figure 4.18 Backup Tree Algorithm**

**Figure 4.19 Average Node Failure**

**Figure 4.20 Average Bandwidth**

**Figure 4.21 Average Delay**





## **List of Tables**

**Table 3.1 All binary states for 3 nodes**

**Table 4.1 Average Node Failure (Algorithm vs Random Selection)**

**Table 4.2 BandWidth Analysis**

**Table 4.3 Delay Analysis**



## **List of Abbreviations**

<b>CS</b>	<b>Client Server</b>
<b>P2P</b>	<b>peer To Peer</b>
<b>PCs</b>	<b>Personal Computers</b>
<b>IP</b>	<b>Internet Protocol</b>
<b>VOD</b>	<b>Video on Demand</b>
<b>DRM</b>	<b>Digital Right Management</b>
<b>QoS</b>	<b>Quality of Service</b>
<b>VHO</b>	<b>Video Hub Offices</b>
<b>SHO</b>	<b>Super Hub Offices</b>
<b>GUIs</b>	<b>Graphics User Interfaces</b>
<b>POP3</b>	<b>Post Office Protocol 3</b>
<b>IMAP</b>	<b>Internet Message Access Protocol</b>
<b>HTTP</b>	<b>Hyper Text Transfer Protocol</b>
<b>SSL</b>	<b>Secure Socket Layer</b>
<b>DNS</b>	<b>Domain Name Server</b>
<b>SSH</b>	<b>Secure SHell</b>
<b>ISPs</b>	<b>Internet Service Providers</b>

**MDC      Multiple Description Coding**

# Chapter 1

## Introduction

### 1.1 Introduction:

There was a time when news used to take days to travel from one place to another, entertainment were regional either they were movies, talk shows or game shows but due to the advancement in technology all these information can travel from one to place to another in a lightning speed. Basically, with the rise of internet we have been able to come closer and share information in no time. Internet has become a new paradigm; one cannot imagine a day without it. Internet has become a part of our daily routine. Also with the increase in usage of handy and portable media capturing devices like cell phones, iPods, iPhones, notebook, PCs which are integrated with the high definition cameras have attracted and facilitate people to capture all moments of their life and share them with other people through social networking websites and media sharing websites like 'YouTube'. The availability of high bandwidth, faster video encoding techniques, and free software tools has also encouraged lot of people to capture and share their videos online. These technological gadgets equipped with high speed internet have changed the face of entertainment industries such as movie, talk show, news and games. People are indulging themselves with these means of entertainment more and more. Sharing and accessing these information has become much easier and cheaper through streaming.

Streaming is usually done in traditional client server architecture implementing IP multicast. This approach of streaming seemed limited in many ways: poor hierarchical routing, poor scalability, difficulty in deployment, high maintenance and lack of authentication. So, to overcome these drawbacks peer to peer architecture was introduced for data streaming. P2P architecture has attracted most of the service providers, developers and other internet community. Most of the multimedia applications available online are somehow based on P2P architecture. P2P applications are based on second and third type of P2P structure. It is a different approach of live streaming than usual IP multicast as it is developed based on P2P architecture. In this approach, each individual node is a client and a server. Each node periodically gains the inadequate data from other node and gives the available data to other nodes. This method is proved to be efficient and effective.

P2P started with the concept of file sharing. After the popularity of P2P for file sharing this architecture was further used for media streaming. These days it is being rapidly used for VOD (video on demand) as well as for Live streaming of media. The different type of applications based on P2P for streaming based on content distribution can be distinguished in three different ways [6]. First - Ongoing download is elastic which means it can be delayed depending on the availability of bandwidth. Once the download is complete then only user can access that file for its own purpose. For second case let's take an example of video file in which the playback starts as soon as the downloading application has downloaded enough data in the buffer. In this case application can detect

and pause in between for rebuffered, if it has enough data in the buffer to continue playback without depletion in quality. The third case is of real time streaming or live streaming. In which the delay of more than few seconds will be considered as an issue.

P2P Streaming in itself consists of few problems but it is the most cost effective way of sharing information among each other. Some of the problems faced by the P2P Streaming are: Churn (dynamic nodes), DRM (digital right management), pollution attack (mixing bogus chunks for the content) etc. One of such problems that bring instability in the network is churn (frequent leaving and joining of nodes). Churn can cause different problem for the users such as playback issues and bad video quality, this problem in return can cause user to leave the network. To avoid such problem, stabilizing the network or dealing with churn is an important issue. To overcome this problem we have proposed a concept to increase resilience of the topology.

To achieve resilience in the network we construct a backup tree for the existing base tree. The base tree is the major tree which streams the data from the source to the nodes in a tree topology hierarchy. Backup tree is the supporting tree, which is an alternate tree structure for the base tree. In case of failure of node due to any reason, its descendant's children will receive the data from the backup tree. This proposed concept increases the resilience of the tree in case of failure of any nodes.

## **1.2. Background**

### **1.2.1 P2P Overview**

P2p literally means no server. To describe this precisely I have to say P2P is a concept in which every node acts as a server and a client. Each node periodically exchanges data and their status information with other nodes. Each node extracts unavailable data from other nodes and provides available data to other nodes. On a broad concept P2P can be classified into three types [9]: Centralized Peer-to-Peer, Pure Peer-to-Peer, and Hybrid Peer-to-Peer. Centralized P2P means there is a central server which helps form the network of the particular application. This server doesn't contain any of the data but it keeps track of the clients available in the network. This server is known as a tracking server but not a data server. Examples of centralized p2p: Napster, BitTorrent. Pure P2P contains no central server not even the tracking server. Whenever a client requests data it is flooded to every other node and nodes having the data reply to the client. Examples of Pure P2p: Gnutella 0.4, Hybrid P2P is a mixed concept of Pure P2P and centralized P2P which consists of both the central tracking server and the nodes exchange their data by flooding. Examples of Hybrid: Gnutella 0.6, Kaaza. These were the early P2P applications which were built on the concept of P2P file sharing.

### **1.2.2 Live Streaming Architectures in Current Schemes**

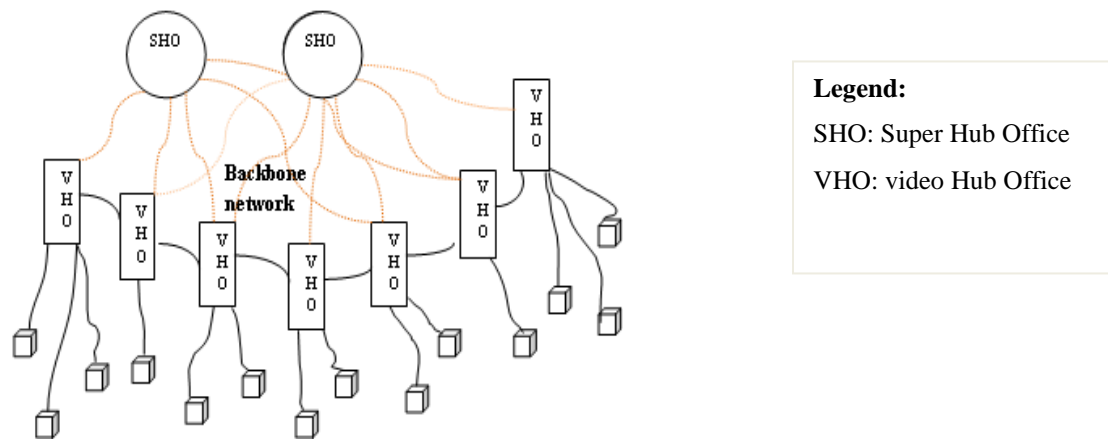
P2P architecture can be basically categorized into two types on the basis of their architecture as (CS) Client-Server architecture and (P2P) Peer-to-Peer architecture. CS architecture consists of server and client, server explicitly forwards the data and client



receives the data whereas in the P2P architecture the client not only receives the data but also forward the data to the other clients in the network. Existing P2P applications mostly follows the traditional client-server architecture consisting of regional server to serve their users. Client-Server architecture faces lots of challenges like improper resources utilization, high cost for equipment, single point node failure etc. Prone to these problems has made P2P architecture rise in commercial as well as in academic sector. Researchers and commercial sectors are these days more attracted to P2P architecture since it has brought new concepts in cost sharing. P2P architecture tends to resolve these problems but this architecture itself faces practical challenges like dynamics nodes, QoS, DRM etc.

#### 1.2.2.1 Client- Server Architecture:

Currently most Live Streaming architecture is based on server-client hierarchy. As described [11] client server architecture consists of 3 major components: SHO (Super hub offices), VHO (video hub offices) and clients.



**Fig. 1.1 Client-Server Architecture**

SHO are the datacenters which holds maximum amount of data. They break up the video frames into IP packets and deliver it to the VHOs; VHO then supplies the requested data to the users. The links between the SHO and VHO consists of high speed bandwidth therefore; accessing the SHO for data is not a critical issue. However the greater distance between VHO and SHO may lead to some delay in data arrival. The client posses a set up box for video decoder which reassembles the IP packets to data stream and the video is then displayed to the users. This approach of client-server faces challenges such as single-point of failure, cost sharing and server overload.

The architecture of “iCloud” [6] is an enhanced approach of the traditional Client-Server architecture. The components in this methodology are the same as in Fig 1.1 but the key difference that makes this architecture efficient is that the regional servers collaboratively work to serve the users. A user is served by multiple stations implementing the concept of “iCloud”. This approach basically forms a group of servers to work in collaboration to serve the users’ request. It depends upon available bandwidth and this bandwidth decreases as the number of hops between the servers’ increases. So, the server selection to form a group to work collaboratively depends upon the proximity. The servers that are destined to serve a user request must be within a definite range or within certain hops. Also this approach implements the request dispatch protocol which helps to determine the servers to form a collaborative group to serve a user.

The collaboration of server eliminates the problem of single-point-failure, if one server accidently goes down then the users can still access the data from other servers. Since a

particular user can access data from multiple servers this helps to reduce the work load from a particular server and also in case if the data requested by the user is not available than the regional server can share their data so the regional server doesn't have to access the data from the SHO. But still this faces the problem as face by above architecture as the number of user increases.

#### 1.2.2.2 Limitations of Client-Server Architecture:

Based on the above architectures it can be resolved that Client-Server architecture is good only if there is only limited number of users to be served. As the number of internet users is growing day by day the client-server architecture fails to meet the requirement for these increasing users. This was one of the main reasons why P2P prevailed over Client-Server Architecture. Mainly Client-Server faces following setbacks: (a) Single Point of Failure (b) No Resource Sharing (c) High Cost. These limitations have made P2P more viable solution for streaming.

#### 1.2.2.3 P2P Architecture:

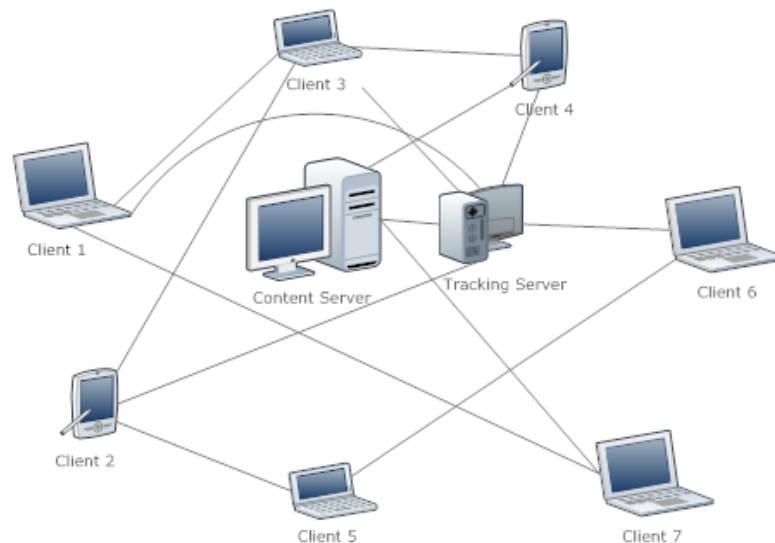
Most of the current video streaming are done by multicasting. In multicasting, the data are forwarded to a desired host in a single transmission through routers or any other networking devices. But this operation is expensive since it requires those routers which can support multicast. Many solutions and approach were purposed to solve this problem like tree structure algorithms, due to the dynamic nature (nodes which can join and leave

at any time) of the nodes this effort was also not so effective. Structures like mesh and forest were suggested but they were too complex to implement.

The network overlay for P2P streaming can be broadly classified into two different topologies [17].

#### 1.2.2.3.1 Mesh Topology

Many P2P applications such as “CoolStreaming” [8] and “Prime” [16] are built on mesh topology formation. In mesh topology there is no specific topology formation. In mesh topology a node comes and joins the network requesting the tracking server to get the appropriate peer. Mesh topology has no defined hierarchy, a single node gets the stream data from multiple peer and share data to multiple nodes.



**Figure 1.2. Mesh Topology**

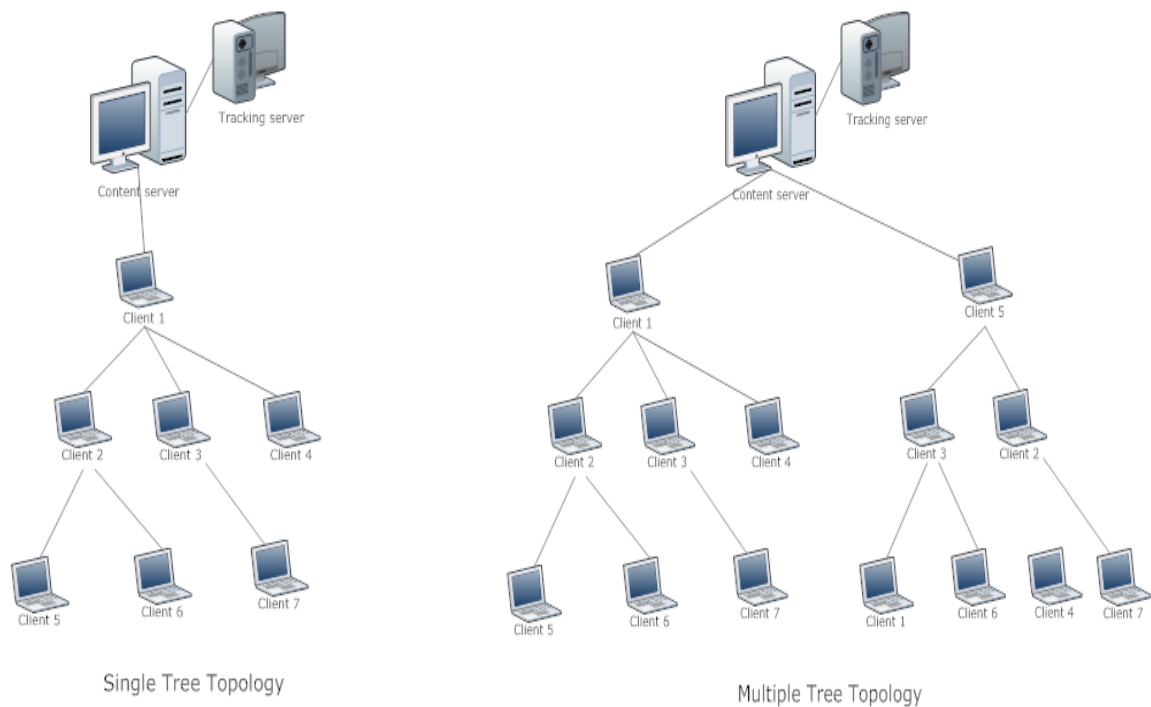
As in concept of P2P file sharing, the streaming P2P applications work in the same manner. A new node connects to the network. It goes to the tracking server to get the

information of other available nodes. The tracking server sends the information of other available node to the new node. The new node begins exchanging data with its peers. A single peer in this case can receive the data from multiple parents. To deal with the frequent arrival and departure each individual node keeps other nodes information. A node can get the peer information either by asking the tracking server for the new peer information or exchanging the peer list information among each others. An individual node may leave the network gracefully, in this case the leaving node informs the tracking server about its departure so the tracking server will remove the node from the list as well as inform other nodes about its departure. A node may also leave the network due to computer crashes or some other reason, in this case neighboring node will know about its departure if the neighbors did not receive the keep-alive message from the departing node. Keep-Alive is a kind of “hello” message which is exchanged between the peers so as to know the live status of other neighboring nodes.

#### 1.2.2.3.2 Tree Topology

P2P Streaming which forms tree topology as an overlay network has a distinct hierarchy of nodes. The structure of a tree topology is defined in depth in “Narada” [25], “ChunkySpread” [26], and “SplitStream” [27]. In a tree topology each node has a single parent and multiple children. A node joins the network and asks the server for parent, the node is assigned parent at a certain level of tree which depends upon the bandwidth, delay and the distribution of tree. A tree has two attributes depth and fan-out (width). The delay increases as the depth of the tree increases. Since, the data flows from top to bottom

the node lying at the bottom of the tree receive the data at last. The bandwidth of the tree decreases as the width of the tree increases. Increase of width means the number of children of a particular node increases and the node has to share the bandwidth among all its children. One of the problems of single tree topology is that the leaf node (nodes that have no children) does not share the bandwidth.



**Figure 1.3 Single Trees vs. Multiple Trees**

As the leaf node in single tree topology doesn't share the resources, in order to utilize their resources an approach of multiple tree was proposed [18]. In case of multi tree topology there are multiple trees for a single stream of data. The content server divides the stream into multiple streams and sends each divided stream into individual sub-tree. Each individual node is present in the sub tree so they receive each stream through

different parents. Each node may have different positions in different sub-trees in different levels. As the data flows from top of the tree to the bottom each node eventually receives the stream. This approach helps to utilize the resources of each node in the single tree either they are the interior nodes or the leaf nodes. But it still faces the problem of reliability, if a node leaves the network than all the sub-trees have to be reconstructed or repaired which may interrupt the stream and if it is a live streaming than a small delay can ruin the interest of viewer from viewing the content.

#### 1.2.2.4. Challenges of P2P Architecture

P2P solves most of the problem faced by the client server architecture but yet it faces its own challenges. Each Architecture and design has their own advantages and disadvantages. Below are some of the major challenges faced by P2P applications. Some of the P2P applications have solved these challenges to some extent and some of them have not. Considering the general architecture and their analysis here are some of the challenges faced by today's P2P streaming applications. Some of those major challenges are discussed below:

##### i) Churn

In a P2P multicast system the nodes are dynamic in nature. Peers are joining and leaving at very high rate. They can come and leave at any point during the exchange of messages. We cannot rely upon single node to deliver the data to its child nodes as most of our P2P algorithm implements tree architecture. There has to be a solution for the instability of nodes in a session to make that process more reliable and algorithm more robust. One of

the solutions is to implement the mesh architecture but it also increases the problem of scalability and overhead in the network. It also increases the network congestion. In such case we think implementing a many-to-many architecture would be a good option and solution to most of our problems. Those P2P applications which form an overlay tree topology face this problem. A solution to this problem was proposed in paper [12] by determining dynamics users. This paper categorizes users into three types: Silent users, Long-term users, Short-term users. If silent or long-term users can be determined then they can be placed on the top of the tree structure which helps to generate stability to the tree structure. By following this approach the problem of dynamics users can be solved to some extent, but the problem in this approach lies how to determine the silent and long-term users.

ii) Network Congestion

Congestion refers to the amount of traffic in the network. It increases if there are lots of requests to the server which the server cannot handle. If a server sends data at faster rate than that the receiver can receive, this also causes congestion. Congestion not only affects single users but also affects lots of users which are following the same path. Analogy of this problem can be given as the traffics in the road. Heavy traffic in a particular path not only affects a single vehicle but the entire vehicle which follows the same path. Congestion affects the quality of service of the system and leads to queuing delay; packet loss etc. Proper congestion control mechanisms should be employed to avoid such circumstances. Data rate control mechanism helps in the congestion control as congestion



occurs when one peer is sending packets at faster rate but the peer on the receiving end is at low bandwidth and is not able to accept packets at the same speed sender is sending packets. Video transcoder tools can be used to control the congestion in the networks. [13].

### iii) Pollution Attack

It is a kind of stream pollution in the P2P architecture where the attacker mixes the bogus chunks into the stream while sending the packets to receiver and these chunks degrade the quality of received media at the receiver's end [15]. In the P2P applications such as PPlive and PPstream, a polluter can introduce the corrupted chunks into the video stream. During an ongoing video streaming session to attract more peers, an attacker can advertise that it has large no. of video chunks available to upload on high bandwidth (As it can be checked these using clients like BitTorrents & Vuze). When peers sends request to this client for downloading these video chunks, it then sends bogus chunks along with the file requested. This receivers further shares this video stream with other peers and very quickly these polluted chunks spread all over the network and the video quality degrades. This attack will force peers to leave the network and the member count will decrease immediately.

To prevent these jammer attacks BACKLISTING defense mechanism is suggested which try to put those kinds of peers on backlist which advertise an unusual large no. of chunks to attract down loaders. Another way to create a backlist is to identify the polluted chunks by comparing the characteristics of received chunk and this can be done using audio

video processing techniques. Further sharing those results with all the peers in the network will prevent attacker from pollution the video stream. Apart from blacklisting methods other methods which can be used to prevent attacker from polluting the video stream are: Traffic encryption and Hash verification.

#### iv) Security Attacks

Security attacks are same as the pollution attacks. P2P application such as Napster (in 2001) and CoolStreaming (in June 10 2005) were shutdown due to legacy issue. Sharing of unlicensed chunks by a single peer can duplicate the chunks to other peer and later can be circulated to the whole network. Initially a peer shares an unlicensed chunk to other peers. Other peers who are relying on this peer to get data extract the data along with the unlicensed chunk shared the initial peer. Similarly in this fashion the chunk is shared to all the peers in the network. This is one of the key challenges that a P2P application faces. Few solution to this problem have been suggested such as blacklisting, hash encryption etc [14]. In case of blacklisting when one the client discovers that one of the peer in the network is sharing an unlicensed chunk he will flood this information to the whole network and other peers who are extracting data from this unlicensed peer will stop fetching the data. In this was the client sharing the unlicensed chunk will be isolated and no will share data with this peer.

### **1.3. Motivation and Objective**

P2P applications have evolved from the limitations of client-server architecture, which started as file sharing and later gained its popularity in streaming media. The streaming

media can broadly be categorized as VOD (video on demand) and Live Streaming. One of the most emphasized problems of streaming is Churn. Churn is caused by users leaving and joining the network frequently. In case of VOD a stream of data is streamed over the network which resides on other node's hard drive. While for Live Streaming the streaming data is not saved in any storage device, it is just played as the data is streamed from one node to another. A few seconds of delay is tolerable in case of VOD as the user will be able to view the stream after a certain pause (interruption) but in case of Live Streaming there can be no delay at all. If delay occurs frequently in Live streaming user will lose his/her interest and leave the network. So, in the case of Live Streaming, the issue of reliability is very significant. This decreases the reliability of the network. So, to overcome this problem construction of backup network topology to the existing topology is proposed.

The objective of this thesis is to:

- i) Construct a resilient tree to increase the reliability of the network.
- ii) Maximize the performance of the network utilizing parameters such as bandwidth and delay.

## **1.4. Tools Used**

### **1.4.1 Python/Twisted:**

Python is an object-oriented scripting language. It has been commonly used for scientific and research purpose for last decade. It is considered to be an easy and efficient tool to develop and test small prototype models. It has a rich libraries which can fit multipurpose

(can be used for UIs, web interfaces and writing server, clients and more). As python is an open source language, people from different communities are contributing to develop APIs for python libraries which increase the applicability of this language in various fields. It can be used for all sorts of development such as: system programming, GUIs, internet scripting, integrating components, database programming, gaming and more. Python being an interpreter language its execution speed may not be as fast as compared to other compiled language such as C, C++. This is due to the fact that python translates its source code to intermediate form known as byte code, which provides a great leverage over other programming language that is platform - independent. Since python doesn't compile their source code all the way down to binary code, some programs written in python may execute slowly in comparison to C, C++. Following are some of the well known company using python [2]:

- Google uses python for its web search and employs Python's creator.
- Google's App Engine uses python for application development.
- YouTube video sharing is written in python.
- NSA uses python for cryptography and intelligence analysis.
- Maya (3D modelling & animation software) provides a python API.

These are the few areas where python has been used. Among lots of libraries, Twisted is one of the frameworks which are gaining its popularity among the community. Twisted is a networking framework built on python.

As mentioned in [1], [3] Twisted is an asynchronous, event-driven networking framework written in python. Twisted doesn't have to deal with complexity of threading and also unlike synchronous frameworks it can process events from multiple network connections without making the application unresponsive. It also means that a single thread controls the whole execution of the program (multiple tasks) but an individual task may release the control over to other task while it is waiting for any other events, resource etc. Since twisted is written in python it's a cross platform, object oriented, interpreted language. It takes the network input and converts that to a process call. Due to this nature of Twisted, it is considered to be efficient network programming framework.

Twisted contains web servers, chat clients, chat servers, mail servers and more. It also supports numerous protocols such as POP3, IMAP, HTTP, SSL, and others. It's a great tool with lots of functionalities such as mail, web, news, chat, DNS, SSH, Telnet RPC and more. Being a cross platform language application built on twisted can run on Linux, Unix, and Mac OSX. Using twisted has lots of advantages such as [3]: security, stability and it's easy to write servers and clients.

- Asynchronous/Event-driven:
- Flexibility: one can start off quickly as twisted provides high level classes not just that one can also implement their ideas through the scratch.

- Open Source: It's a great advantage that twisted is open source. So, that one can modify the classes or built a package as per their wish on top of the existing classes.
- Community-backed: Twisted has a strong community of developers who are willing to help new comers and also support when one runs into a trouble.

#### **1.4.2 Networkx (Network Visualizer):**

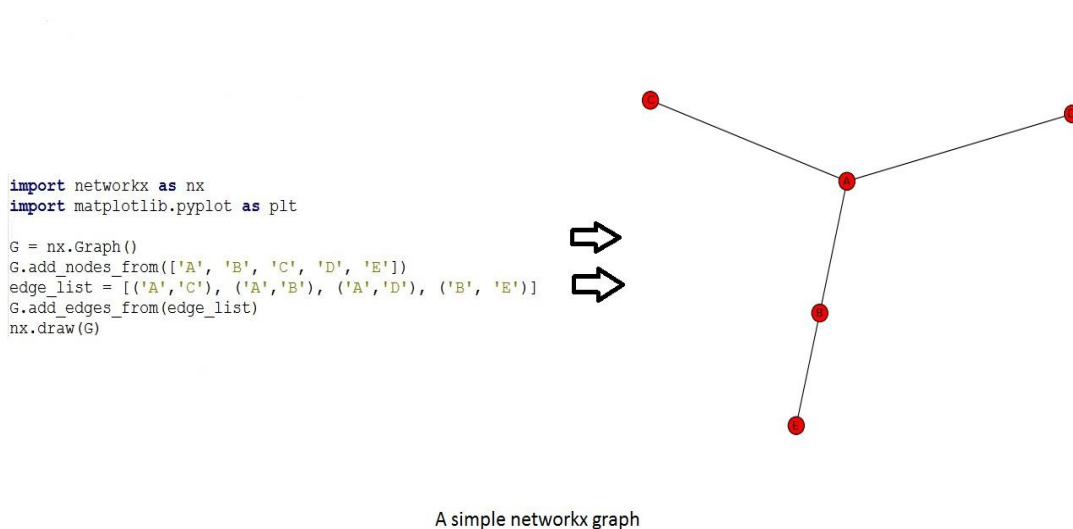
As mentioned in [4], [5]: "NetworkX is a Python language software package for the creation, manipulation, and study of the structure, dynamics, and function of complex networks." Networkx helps to visualize large real world graphs. Since network is completely written in python it is highly scalable, efficient tool for visualization purpose.

Some of the key features of networkx are as follows:

- It contains data structures for representing huge networks and graphs
- It has enough graphs to suit structure and dynamics of social biological and infrastructure networks.
- Graphs can be converted to and from any formats.
- Any graphs attributes such as adjacency, degree, diameter etc. can easily be explored.
- Any portion of the entire graph (i.e. sub graph) can be explored in detail.
- It is written in python, so any operating system can support the library.

Networkx has been used in my thesis to visualize the nodes and edges. Nodes represent an individual client connected to the network while an edge represents their connection. The edge also displays the amount of data flowing through that connection as well the delay between the two nodes that are connected to each other. The topology used for the visualization of this network is a tree topology.

A simple example of networkx is given below.



**Figure 1.4 A simple Networkx graph**

### **1.4.3 Tkinter (UI Application):**

Tkinter is UI interface written in python for development of GUI applications. There are other GUI applications for python such as wxPython, Qt etc. but Tkinter is considered to be the standard UI toolkit. Tkinter provides a fast and easy way to develop UI application in python. As other UI framework Tkinter also provides various controls such as buttons, labels, text box etc. There are altogether 15 controls which are known as widgets.

Some of these frequently used widgets are as follows:

- *Button*: It provides button the applications.
- *Label*: used mostly for displaying texts.
- *Message*: used mostly for writing multi line text.
- *Canvas*: used mostly to draw different shapes.
- *CheckButton*: used to select multiple options as checkboxes.
- *Entry*: used to accept values from the user.
- *Frame*: used to hold or organize other widgets.
- *Listbox*: used to provide multiple choices to the user.
- *Menubutton*: used to provide menus in the application.
- *Menu*: used as the menu options (values) in the Menubutton widget.
- *Radiobutton*: used to provide multiple options to user but only one can be selected.
- *Scale*: used to provide a slider widget.
- *Scrollbar*: used to provide scroll options to other widget.
- *TkMessageBox*: used to display prompt message to the user.

These widgets have their own attributes which helps to provide good look and feel to the applications. Some of the common attributes of these widgets are: dimensions, colors, fonts etc.

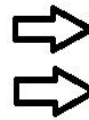
Tkinter in my thesis has been used for displaying the events of the whole network. It can also be considered as a log for the server. It displays all the events happening in the



network in term of texts. Events such as joining of new node, bandwidth update of the node, leaving of the node, and reassigning node to the new parents will be displayed in Tkinter window as they happen in the network.

A simple Tkinter application is displayed below:

```
from Tkinter import *  
  
root = Tk()  
  
tk_label=Label(root, text="Hello Tkinter!")  
tk_label.pack()  
  
root.mainloop()
```



A simple Tkinter Application

**Figure 1.5 A simple Tkinter UI Application**

## 1.5. Thesis Outline

This thesis is organized in five chapters.

Chapter 1 is an introduction and background of P2P Streaming. Chapter 2 describes the related works done in the field of P2P Streaming. The chapter gives information about the architecture of the some P2P application as well as development and research performed in the field of reliability of the P2P Streaming.

In Chapter 3, we address the definition of the problem, explain the problem with diagrammatic examples and provide a mathematical solution to the problem. Chapter 4 presents an implementation of the experimental setup alongside with the results obtained.

Chapter 5 summarizes what have been done in this research, and discuss some future works that can be applied to this research.

## Chapter 2

### Related Works

Most of the internet traffic today is dominated by video. Based on internet traffic index provided by Cisco [19] one third of internet in 2009 was dominated by video traffic and is expected to reach around 57% by 2014. This increase of internet traffic suggests that video streaming is becoming a popular fashion in recent days. Streaming application such as PPLive [7], CoolStreaming [8], and Cabernet [9] is hype in current scenario of P2P usage. Even though P2P streaming is gaining its fame it has its own unique problems which are being tackled by researcher and academic professionals. Some of the problems such as topology analysis, handling dynamic nodes, streaming architecture and DRM are active areas of research in current scenarios.

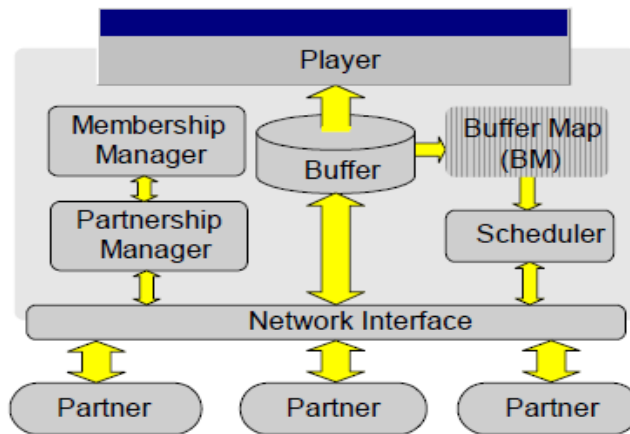
#### 2.1 Live Streaming Applications

Some of the applications which has gained fame in research areas due to its efficiency and versatile architecture are briefly described below

##### **CoolStreaming:**

CoolStreaming [8], [20], [21] doesn't maintain any complex structure that governs the flow of the data. The direction or path of the data is determined at the real time. Also the partner within a node keeps periodically changing which enhances the video quality. The information of the data is periodically exchanged between partners. Results from the planet lab shows that the performance of the CoolStreaming is directly related with the

availability of data and partners. According to the authors Cool-Streaming doesn't have any global or complex structure, data forwarding is directly related to the data availability rather than a specific direction and it is robust and resilient. CoolStreaming basically consists of three components namely membership manager, partnership manager and scheduler.



**Figure 2.1 System Diagram for DONet Node**

*Membership Manager*-Each CoolStreaming node has unique key as an IP and also a membership cache which keeps a partial track of active nodes. Every new node contacts the source node which redirects the new node to a random deputy node, the new node then retrieves list of partner from the deputy node. Every new node contacts the source node because this node exists throughout the streaming. According to the author the major issue faced was maintaining and updating the membership cache. Every node in the CoolStreaming periodically generates a message in certain time which contains four values (seq\_num, id, num\_partner, ttl) and every other node which receives these

message updates their membership cache with the above values for that particular node.

The membership information from one node to another is exchanged by gossiping.

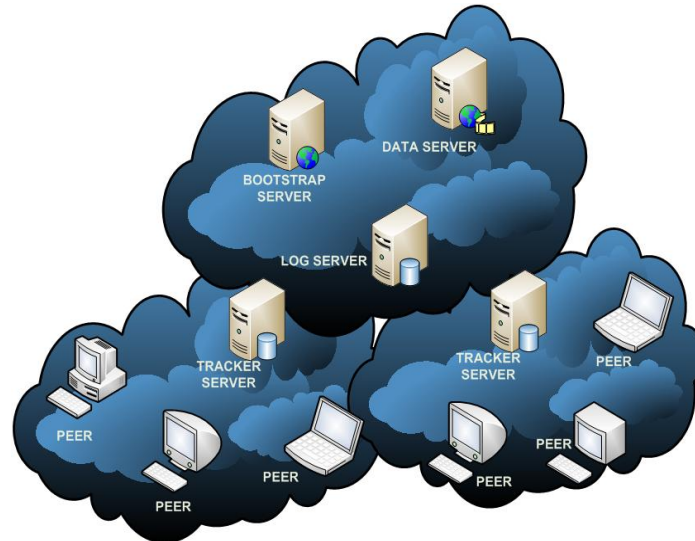
*Partnership Manager*-It is based on the main idea of Buffer Map (BM). A video is divided into segments and each partner consists of some these segments. The information of BM is continuously exchanged in certain time. The availability of data and partner determines the direction. The path for the flow of data and the number of partner for a node never fixed. It varies with the availability of data.

*Scheduler*-Scheduling is to know which segment of data is to be fetched from the available partners. Once we have the Buffer Map of the node and the partner than it can be determined from which partner the data is to be fetched. According to the authors for stable and static environment simple round – robin algorithm would be enough to get the information but for the dynamic and heterogeneous environment this algorithm would not work because dynamic environment consists of those nodes which joins and leaves the Cool-Streaming simultaneously.

**PPLive:**

Another P2P application which proved to deliver better content and was widely used during the Beijing Olympics was PPLive. The PPLive system works as other P2P applications but consists of some additional components such as bootstrap server (it helps peers to find the suitable tracking server), content server (which is considered as a data center and holds most of the data), tracking server (which contains the list of available peers and find the neighboring peers) and the peers (which request data). The individual

peers in PPLive network installs PPLive software which implements a protocol through which the peers can communicate with the server as well as with the other peers.



**Figure 2.2 The system architecture of PPLive P2P-VoD**

A peer initially request for a particular chunks of data with the bootstrap server and the server complete all the bootstrapping functionalities and forward its request to an appropriate tracking server. Chunks are the parts of movie which different peer holds. A tracker server than establishes a neighborhood of peers who have similar request. The peers than share and download data with other peers within their neighborhood. If in case there is no available data within the neighborhood than the peers download the data directly from the server. Since the PPLive architecture is based on P2P notion each individual peer contributes a small amount of their hard disk storage. The chunks of the requested movie are stored in this space. When there are enough available chunks in the disk than the peer advertises to other peer about their data information.

The PPLive architecture mainly shows that it diminishes the server workload and gives good quality of service for popular movies or videos. Also PPLive offers good playback continuity. The performance of PPLive degrades for unpopular movies. The performance may deteriorate because of one of the following reasons [20] i) There are not enough replicas of the desired movie existing in the system. ii) Peers that hold the desired movie are with limited upload bandwidth iii) Peers may not want to fully exploit the available bandwidth iv) The available upload bandwidth of peer holding the desired movie has been fully occupied.

## **2.2 Multicast Tree and Reliability**

Reliability of P2P streaming has lately become a topic of discussion among the researcher. Lots of ideas have been proposed to enhance the reliability of the overlay topology. Especially in Live Streaming reliability of nodes and topology is of prime concern. Lots of researchers has discussed about the enhancement of topology, construction of tree, proposed new architecture, optimized the tree algorithm to increase the reliability of the topology and node itself.

Among these discussions the author in [22] describes an approach to stream videos over the internet with resilience known as CoopNet. This paper focuses on two key issues, firstly – Most of the ISPs charges for upstream bandwidth so it's not a wise decision to exploit a single node. In order to resolve this issue the peer in CoopNet will only contribute in uploading as long as the node is interested in the content. Secondly – Nodes in P2P streaming are mostly dynamic in nature, they may leave the network at their own

will or may be disconnected due to unwanted disruption. This makes the network topology vulnerable. To address this problem author suggests an approach for live streaming with large amount of peers which is done by replicating data for peers' availability and creating multiple paths for peers to get the data. The multiple paths for the peers can be achieved by using efficient tree management algorithm so, that in case of failure of one link the peers can choose the alternate path. To increase the quality of stream, the content is encoded using multiple descriptions coding (MDC) [23]. The MDC content is replicated so that peers can access data from multiple sources.

The paper suggests of constructing multi tree to overcome the vulnerability of dynamic nodes. CoopNet tends to build short trees so that it would minimize the probability of disruption due to failure, congestions at parent nodes. It also suggests making the tree as diverse as possible which means interior nodes in one tree must likely be the leaf nodes in other tree. The tree construction algorithm preferred by CoopNet is deterministic rather than randomized tree construction. In case of deterministic algorithm it is first determined whether a node in a tree will be a fertile node (node which has children) or a sterile node (leaf node). If a node is fertile node in one tree than the node in other tree is mostly likely to be a sterile node. This approach does increase the diversity of the node as well as make the tree topology resilient to dynamic nodes but still it suffers from single point of failure as pointed out by the author. If the root node of the tree fails than the streaming content will not be available to any of the other nodes in the tree.



Dynamic Users have huge influence in the stability of the P2P network and delay induced churn. Kunwoo Park in his paper [12] describes the different dynamic users in the network which is a considerable factor for building an overlay tree topology. One of the constraints that affect the performance of p2p application is the dynamic nature peers or nodes. Based on this paper users can be classified into two types: short-term users and long term users. Long term users are those who stay in the system for 50% of the session and short-term are those who stay for 1% of the session. The paper discuss about determining these types of users. Long term users help to improve the system performance. So, after identifying the stable users they can be placed on the core of the distribution structure while short term users can be placed on the leaf nodes. Placing the long term nodes at the top or core structure of the architecture provides resilience of the topology. Also the author describes about the concept of mutual exchange of message to determine the existence of an individual node known as heartbeat message. This message helps to keep alive records of the neighboring peer.

Dynamic users are one of the areas of research to enhance the efficiency of P2P Streaming. Unlike dynamic user the construction of tree topology itself is an integral and important aspect of enhancing P2P Streaming. The topology optimization formulated by the author in paper [24] is focused on minimizing the average height of the sub-stream tree and average propagation latency in each tree. The basic theme of the paper is focused on the construction and maintenance of the multi-tree structure by introducing interior tree connection. This approach is obtained by optimal node placement and interior tree

connection using heuristic algorithm. Numerical analysis shows that this approach can efficiently build streaming trees with low delay. The multi tree construction approach in this paper depends upon the number of streams for the given video content. Assume there are  $M$  streams for a video content with a streaming rate of  $r$ . The overall  $M$  sub tree will be constructed for each available stream. The basic idea for the node placement in the tree is if a node is an interior in any one of the  $M$  tree than in the rest  $M-1$  tree that particular node will have a position of leaf node. This approach certainly does reduces the churn occurrence in the system but due to the each node placement at a higher level, the sub tree which has a node with less bandwidth will be a victim of low video quality or video lag. The approach suggested by the author is good for building trees with low delay performance.

Our method of increasing reliability of the tree is in closely related to the approach suggested by the author in [24]. But instead of constructing the multiple trees for each stream we will be constructing a single backup tree for the existing base tree. Also the backup tree that we will construct will come into action in case of node failure so that there is no interruption of flow of data. Also we will be using the process of normalization for picking of optimal nodes in the trees for the construction of base tree as well as backup tree. This will help to determine an optimal node based on two parameters bandwidth and delay. The concept of backup tree is to provide an alternate parent to a node in case of failure. Also data only flows in the base tree backup tree is a support tree which only maintains connection to other nodes.

## Chapter 3

### Reliable Multicast

We consider the following problem. We have a mesh wireless network, with a set of  $n$  nodes  $v = \{v_1, v_2, \dots, v_n\}$ . Each node  $v_i$  also has a probability of failure,  $p_i$ . We assume that download bandwidth are much more than upload bandwidths, and therefore do not need to be taken into consideration. The objective is to construct a multicast tree for  $v$  so that the entire node in  $v$  can receive streamed data from an origin source  $s$ , with the quality-of-service (QoS) requirement of a minimum streaming rate  $r$ , and moreover, this multicast tree is maximally reliable. We observe that a consequence of the QoS requirement of minimum streaming rate is that a given node  $v_i$  cannot have more than  $c_i$  children in the multicast tree where  $\frac{u_i}{c_i} \geq r$  and  $\frac{u_i}{c_{i+1}} < r$ .

Where,  $u_i$  is the uploading bandwidth of the node  $v_i$ .

$c_i$  is the children of node  $v_i$ .

$r$  is the rate of uploading bandwidth.

We need a precise definition of the reliability of a multicast tree, the property that we wish to maximize in our proposed problem.

#### **3.1 Definition (Reliability):**

Given a multicast tree  $T$  for a set of nodes  $V$ , suppose each node  $v_i$  is either alive or has failed (failure includes departure of a node). Recall that each node  $v_i$  has an associated probability of failure  $p_i$ . We begin by defining the indicator variable  $x_i$  as

$$X_i = \begin{cases} 1 & \text{if the indicator variable } v_i \text{ is alive.} \\ 0 & \text{if node } v_i \text{ has failed} \end{cases}$$

We call the vector  $\vec{x} = (x_1, x_2, \dots, x_n)$  the state vector; it indicates the state of the multicast tree. The set of nodes that are still able to receive the data stream is completely determined by this state vector. We observe that if node  $v_i$  has failed, then all descendants of  $v_i$  (or all nodes in the sub tree with  $v_i$  as root) are disconnected from the multicast tree and cannot receive the data stream.

Given a state vector  $\vec{x}$ , we can determine the set of nodes that are disconnected. We define a function  $\phi(\vec{x})$  such that

$$\phi(\vec{x}) = \text{the number of nodes that are disconnected in } T \text{ with the state vector } \vec{x}.$$

We call  $\phi(\vec{x})$  the structure function of  $T$ .

We suppose that the state of node  $v_i$ , or its indicator variable, is a random variable such that  $P\{x_i = 1\} = 1 - P_i = P\{x_i = 0\}$ , where  $P_i$  is the probability of failure of node  $v_i$ . Thus, with the probabilities of failures  $\vec{p} = \{p_1, \dots, p_n\}$  and a multicast tree  $T$  with its structure function  $\phi$ , it is possible to compute  $E[\phi(\vec{x})]$ , the expected number of nodes that will be disconnected. We define the reliability of  $T$  to be  $R_{(T)} = 1 - \frac{E[\phi(\vec{x})]}{n}$ .

The rationale is that  $n - E[\phi(\vec{x})]$  is the number of nodes still connected and receiving data in the tree and the higher this number is, the more reliable the tree is. We simply normalize this number to the total number nodes and define that to be  $R_{(T)} \in [0, 1]$ , a value of 1 means  $T$  is perfectly reliable it follows then that in order to maximize the reliability, we must construct a multicast tree that minimizes  $E[\phi(\vec{x})]$ .

For convenience of discussion and without loss of generality, we assume that the nodes are sorted in order of increasing probabilities of failure, i.e.  $p_1 \leq p_2 \leq \dots \leq p_n$ . Recall that the constructed multicast tree must have a streaming rate of  $e$  and therefore each node  $v_i$  can have at most  $c_i$  children where  $c_i$  is the largest integer such that  $\frac{u_i}{r} \geq r$ . Even though our current objective is not to minimize end to end delay, we still wish to keep end to end delay as low as possible while maximizing the reliability of the tree. We note that a deep and narrow tree results in higher delays, whereas a wider shorter tree yields lower delays. Thus while we construct the tree to be reliable, we require every node  $v_i$  to have exactly  $c_i$  children. This will ensure that the resulting tree is as wide and short as possible, while still providing the required rate of  $r$ .

To minimize  $E[\phi(\vec{x})]$  requires us to calculate it: From probability theory. Given the random variable  $x$ , and a function  $\phi(\vec{x})$ .

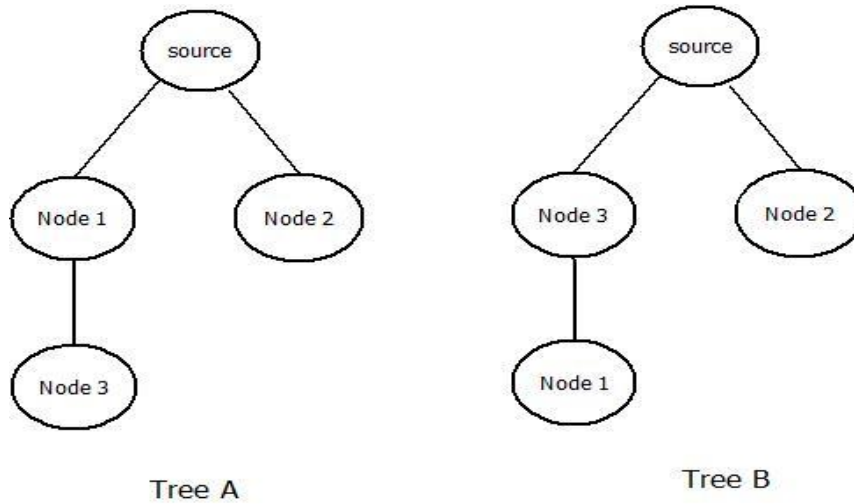
$$E[\phi(\vec{x})] = \sum_{x: p(x) > 0} \phi(x) p(x)$$

We use a simple example to develop some intuition over this problem (probability of failure = 0).

### 3.2 Example:

We have a source  $s$  that is perfectly reliable, and 3 receiver nodes  $v_1, v_2, v_3$ . Assume  $u_i = 2, i = 1, 2, 3$  and the required minimum streaming rate is  $r = 1$ . This means that each node including  $s$  has at most 2 children in the multicast tree, also  $p_1 < p_2 < p_3$ .

Here are two possible multicast trees constructed.



**Figure 3.1 Two Alternate Multicast Trees**

Let us consider the reliability of tree A and tree B. We simply compute  $E[\phi(\vec{x})]$  for A and B, and whichever tree has the smaller  $E[\phi(\vec{x})]$  is the more reliable of the two.

For tree A and B:  $E[\phi(\vec{x})] = \sum_{x: p(x) > 0} \phi(x) p(x)$  with  $\phi(\cdot)$  being a different function for A and for B.

We enumerate all possible values of the variable  $x = (x_1, x_2, x_3)$ ; they are all binary strings of length 3. E.g. (0, 0, 0) means  $v_1, v_2, v_3$  are all failed, (1, 0, 1) means only  $v_2$  is failed.

X: state variable (x)	P(x)	$\phi(\vec{x})$ for A	$\phi(\vec{x})$ for B
000	$p_1 p_2 p_3$	3	3
001	$p_1 p_2 (1-p_3)$	3	2
010	$p_1 (1-p_2) p_3$	2	2
011	$p_1 (1-p_2) (1-p_3)$	2	1
100	$(1-p_1) p_2 p_3$	2	3

101	$(1-P_1)p_2(1-p_3)$	1	1
110	$(1-P_1)(1-p_2)(1-p_3)$	1	2

**Table 3.1 All binary states for 3 nodes**

The structure functions for A and B are different precisely because the two trees have different structures or topologies. For instance, when node 1 is the only failed node, in the tree A it means 2 nodes (1 and 3) are disconnected whereas in tree B it means only 1 node disconnected. From the above tabulated values, it is simple to calculate  $E[\phi(\vec{x})]$  for A and B, given the probabilities of failure  $\vec{p} = (p_1, p_2, p_3)$ . We note that  $E[\phi(\vec{x})]$  decreases from A to B by  $p_1p_2(1-p_3) + p_1(1-p_2)(1-p_3)$  while increases from A to B by  $(1-p_1)p_2p_3 + (1-p_1)(1-p_2)p_3$ .

Taken together, the net difference is:

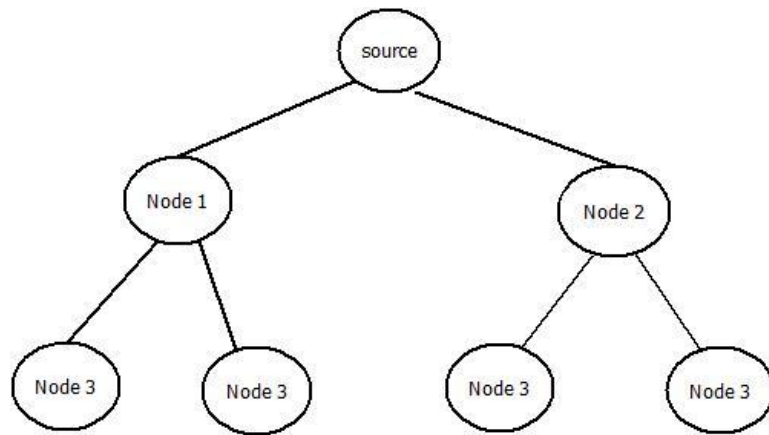
$E[\phi_B(x)] - E[\phi_A(x)] = \Delta p_2 + \Delta(1-p_2)$ , where  $\Delta = p_3 - p_1$ . i.e.,  $E[\phi_B(x)] - E[\phi_A(x)] = \Delta = p_3 - p_1 > 0$ . This means that tree B is less reliable than tree A, according to our definition of reliability.

The above simple example confirms our intuition that more reliable nodes (those with lower probability of failure) should be positioned higher in the multicast tree. As seen in our toy example, placing the more reliable node 1 higher in the tree results in a more reliable tree. The reason is very intuitive: In tree B, node 3 is a less reliable node but is allowed to affect more of the other nodes by its higher node is, the fewer nodes should it affect in the event of its failure and therefore it should be placed in the constructed tree.

It can be thought of this way: If there are two nodes  $v$  and  $w$  with probability of failure for  $v$  being smaller than for  $w$ , i.e.,  $v$  is more reliable than  $w$ . intuitively, it would be more reliable to have  $w$  have fewer descendants than  $v$ . Node  $w$  is more likely to fail, therefore  $w$  should have fewer descendants than  $v$ , which means  $w$  should be placed lower in the tree than  $v$ . We thus arrive at the following proposition which consider a restricted case of our problem. In this restricted version, we make the additional assumption that all nodes have the same upload rate ( $u_1 = u_2 = \dots = u_n$ ), hence all nodes must have the same number of children in the resulting multicast tree (except for when it runs out of nodes),

$$c_i = \left\lfloor \frac{u}{r} \right\rfloor, \forall i.$$

The proposition basically states that the optimally reliable tree is simply the one that positions all the nodes in increasing order of their probabilities of failure. For example, if there are 6 nodes: 1, 2, 3... 6 with  $p_1 \leq p_2 \leq \dots \leq p_6$ . Also each node has the upload rate of  $u = 2r$  (This means each node has 2 child nodes).The optimal tree in the following:



**Figure 3.3. An optimal Tree**



### 3.3 Proposition:

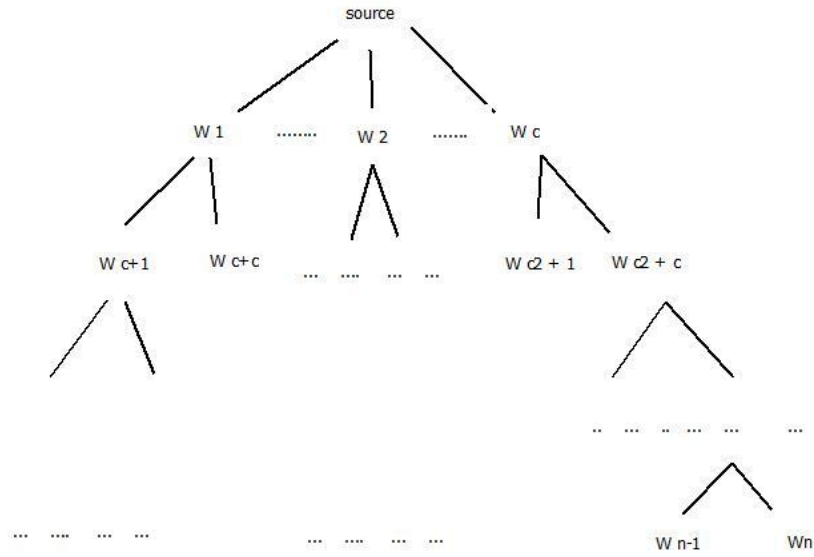
Given a set of nodes  $v = \{v_1, v_2, \dots, v_n\}$  ordered by their probabilities of failure.  $p_1 \leq p_2 \leq \dots \leq p_n$ , and each node have the upload rate  $u$ . The optimally reliable complete multicast tree  $T$  with streaming rate  $r$  is the following:  $T$  has the root node  $s$  which is the source. Node  $s$  has  $C = \left\lfloor \frac{u}{r} \right\rfloor$  children  $v_1, v_2, \dots, v_c$ , respectively. Node  $v_1$  has  $c$  children  $v_{c+1}, v_{c+2}, \dots, v_{c+c}$ . And so on and so forth. The breadth first traversal of  $T$  yields the nodes  $v_1, v_2, \dots, v_n$  in this order; and each node has  $c$  children until there are no more nodes left.

### 3.4 Proof:

The proof that the tree  $T$  is the optimal is straight forward. We start with any complete multicast tree  $T'$  of the  $n$  nodes and show that by replacing each node in turn with the one from  $T$ , the reliability of the tree is constantly increased until  $T'$  is transformed into  $T$ .

Without loss of generality, we assume that  $n = c^h - 2$  for some integer  $h$ . The proof works for any  $n$ . This just simplifies the explanation, by having every interior node in any complete multicast tree possess exactly  $c$  children.

Suppose we have any complete multicast tree  $T'$ , with the breadth first traversal of  $T'$  yielding  $\{s, w_1, w_2, \dots, w_n\}$ . Note that node  $w_i = v_k$  for some  $k$ ,  $1 \leq k \leq n$ .

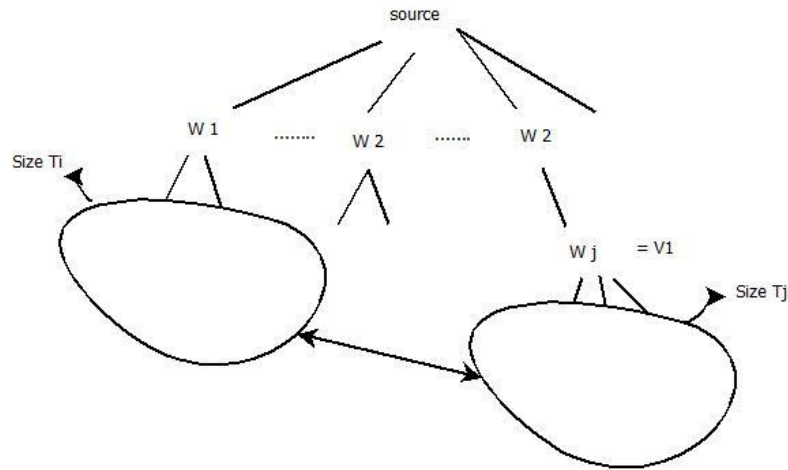


**Figure 3.4 A complete Multicast Tree T'**

Let  $i = 1$ , we first look at  $w_i = w_1$ . If  $w_1 = v_1$ , then do nothing. Otherwise we see what happens to  $E[\Phi(\vec{x})]$  the reliability of the tree if we replace  $w_1$  with  $v_1$  which is  $w_j$  for some  $j > 1$ . Also  $w_1 = v_k$  for some  $k > 1$ . In calculating  $E[\Phi(\vec{x})]$  there are only four possible combinations of  $p_k$  and  $p_i$ .  $p_k$  is the probability of failure for node  $w_1 = v_k$ ,  $K > 1$ , and  $p_i$  is the probability of failure for node  $v_i = w_j$ ,  $j > 1$ . The four possibilities are:

$$P(x) = \begin{cases} p_k \dots p_i \dots \\ p_k \dots (1 - p_i) \dots \\ (1 - p_i) \dots p_i \dots \\ (1 - p_k) \dots (1 - p_i) \dots \end{cases}$$

Note that with  $w_1 = v_k$ ,  $k > 1$ , the  $\Phi(\vec{x})$  multiplied to  $[p_k \dots (1 - p_i) \dots]$  includes the number of nodes in the sub tree rooted at  $w_1$ . However, if  $w_1$  is swapped with  $v_1$ , then  $[p_k \dots (1 - p_i) \dots]$  changes to  $[p_i \dots (1 - p_k) \dots]$ , that is the number of descendants of the former  $w_1$ ,  $T_1$  is now multiplied to  $[p_i \dots (1 - p_k) \dots]$ . Let  $E[\Phi(\vec{x})]'$  be the tree reliability prior to the swap of  $W_i$  with  $W_j = v_i$ , and  $E[\Phi(\vec{x})]$  be the after swap reliability.



**Figure 3.5** swapping the sub tree  $W_1$  and  $W_j$ .

Let  $T_i$  be the number of nodes in the sub tree rooted at  $W_i$ , and  $T_j$  be the number of nodes in the sub tree rooted at  $W_j = V_i$ .

$T_i$  comes from  $\phi(\vec{x})$ ; all the other node in  $\phi(\vec{x})$  do not contribute to any change because only  $W_i$  and  $W_j$  are affected, all the other nodes do not change.

Swapping  $W_i$  and  $W_j$  results in the change form  $[p_k \dots (1 - p_i) \dots]$  to  $[p_i \dots (1 - p_k) \dots]$  which contributes the following difference to  $E[\phi(\vec{x})] - E[\phi(\vec{x})]'$ :

$$= T_i P_{1k} * p_i (1-p_k) - T_i p_{1k} * p_k (1-p_i), \quad \text{where } p_{1k} \text{ is the product of all } p_i$$

except  $j = 1$  or  $k$

$$= T_i [p_i (1-p_k) - p_k (1-p_i)] P_{1k}$$

$$= T_i (p_i - p_k) p_{1k}$$

$$= - T_i |p_k - p_i| p_{1k}; \text{ since } p_k \geq p_i$$

Similarly, the swap also causes the changes from  $[(1 - p_k) \dots p_i \dots]$  to  $[(1 - p_i) \dots p_k]$  which contributes the following difference to  $E[\phi(\vec{x})] - E[\phi(\vec{x})]'$ :

$$= T_j P_{1k} (1 - p_i)p_k - T_j p_{1k} (1 - p_k)p_i$$

$$= T_j |p_k - p_i| p_{1k}$$

The other two possible combinations of  $[p_k \dots p_i \dots]$  and  $[(1 - p_k) \dots (1 - p_i)]$  do not contribute any change to  $E[\phi(\vec{x})]$  after swapping  $p_i$  and  $p_k$ . Therefore the net difference from  $E[\phi(\vec{x})] - E[\phi(\vec{x})]'$  is

$$= -T_i |p_k - p_i| p_{1k} + T_j |p_k - p_i| p_{1k} < 0 \quad \text{since } T_i > T_j$$

$T_i > T_j$  because the tree is complete and  $W_i$  is in front of  $W_j$  in the breadth first traversal of the trees.

This proves that there is a decrease in  $E[\phi(\vec{x})]$  from swapping  $W_i$  and  $W_j = V_i$ , which means an increase in reliability of the tree.

We simply repeat this procedure for  $W_2, W_3$  etc. At each iteration, wither  $W_i$  is already  $V_i$  or we swap  $W_i$  with  $W_j = V_i, j > i$ . After each swap, the same reasoning as above shows that the  $E[\phi(\vec{x})]$  decreases and thus reliability increases. This process ends until all the nodes have been considered in turn. Since reliability always increases or doesn't change at each iteration, the resulting tree  $T$  with the breadth-first traversal yielding  $\{v_1, v_2 \dots v_n\}$  is the optimally reliable tree.

## Chapter 4

### Implementation

#### 4.1 P2P Implementation in Twisted (Server and Node)

The twisted components used for client server architecture is similar to that of components used for P2P architecture. The major difference that set them apart is that every node in P2P structure behaves as a server (forwards or generates data) as well as client (receives the data). Basically a P2P network comprises of two major components server and the node.

A basic classes and functions outline used for P2P server is explained below [10].

##### 4.1.1 Server:

This server produces the data to the nodes in continuous fashion.

##### Producer:

Since it is a streaming server, *IPushProducer* is an ideal interface written for streaming purpose. The server and client maintain the relationship of producer and consumer. *resumeProducing()* informs the producer to produce the data. *pauseProducing()* tells the producer to pause as enough data has been produced for the time being to be processed. It calls *resumeProducing()* once the data has been processed.

```

class ContentProducer: implements(interfaces.IPushProducer)
    def __init__(self, protocol):
        # initialize variable for this class
        self.protocol = protocol
    def resumeProducing(self):
        self.protocol.transport.write('send some text here')
    def pauseProducing(self):
        # perform some actions.
    def stopProducing(self):
        # perform some actions.

```

**Figure 4.1 Content Producer Class**

### Protocol:

Twisted being an event-driven programming its protocol waits for an event from the network, upon receiving such events it makes a call to methods on protocol. All the protocol in twisted are either inherited or subclasses from *twisted.internet.protocol.Protocol*. For every new connection an instance of protocol is made which handles that particular connection and also manages the flow of data through that connection.

The protocol used here is *LineReceiver*. This protocol has two different event handlers: *lineReceived()* and *rawDataReceived()*. Since we are receiving data as lines so it plausible to use *lineReceived()*. For every line received this method is called once. The protocol class also handles the connection. Whenever a connection is made, the producer is registered first. As long as connection is on, the registered node continuously receives the data. In order to stop receiving the data, the connection has to be unregistered first and then the *loseConnection()* method has to be called. If *loseConnection()* is called without unregistering the producer, the data in the buffer is sent first and then only the connection is disconnected.

```

class ServerProtocol(basic.LineReceiver):
    def connectionMade(self):
        self.producer = ContentProducer(self)
        self.transport.registerProducer(self.producer, True)
        self.producer.resumeProducing()
    def lineReceived(self, line):
        # perform some actions like manipulate the line here
    def connectionLost(self, connector):
        # perform some actions.

```

**Figure 4.2 Server Protocol Class**

Factory:

The Factory class inherits from *twisted.internet.protocol.Factory*.

The factory class here simply instantiates instances of a protocol class. “`__name__ == '__main__'`” is the main function in python. Whenever a file is executed it first looks for this function to load the program. *reactor* is the event loop of twisted. For a server to establish TCP connection the reactor listens at a specific port. It connects the factory to the network which here is known as the instance of *ServerFactory()*.

```

class ServerFactory(protocol.ServerFactory):
    protocol = ServerProtocol

if __name__ == '__main__':
    reactor.listenTCP(port, ServerFactory())
    reactor.run()

```

**Figure 4.3 Server Factory Class**

So, putting all the classes together, here is a complete outline of a P2P Server.

```
from twisted.internet import reactor, protocol, interfaces
from twisted.protocols import basic
from zope.interface import implements

class ContentProducer: implements(interfaces.IPushProducer)
    def __init__(self, protocol):
        # initialize variable for this class
        self.protocol = protocol
    def resumeProducing(self):
        self.protocol.transport.write('send some text here')
    def pauseProducing(self):
        # perform some actions.
    def stopProducing(self):
        # perform some actions.

class ServerProtocol(basic.LineReceiver):
    def connectionMade(self):
        self.producer = ContentProducer(self)
        self.transport.registerProducer(self.producer, True)
        self.producer.resumeProducing()
    def lineReceived(self, line):
        # perform some actions like manipulate the line here
    def connectionLost(self, connector):
        # perform some actions.

class ServerFactory(protocol.ServerFactory):
    protocol = ServerProtocol

if __name__ == '__main__':
    reactor.listenTCP(port, ServerFactory())
    reactor.run()
```

**Figure 4.4 A P2P Server Class**

#### **4.1.2 Node:**

A basic classes and functions outline used for P2P Node is explained below.

##### Protocol:

The protocol class for individual node is the same as the server itself. As the node in P2P behaves both as a server and a client there is an 'if else' condition to distinguish the



factory for the initial connection between any two nodes. The name of the factory is defined in the factory class at factory before the connection is made. Other functions in protocol behave in the way as the server does.

```
class NodeProtocol(LineReceiver):
    def __init__(self):
        # initilaize NodeProtocol variable
    def connectionMade(self):
        if self.factory.name == 'server':
            # perform action as a server
        else:
            # perform action as a client
    def lineReceived(self, line):
        # manipulate line either forward it to child or perform actions on it
    def connetionLost(self, connector):
        # either try reconnecting or lose connection permanently
```

**Figure 4.5 Node Protocol Class**

Factory:

An individual node consists of both the server factory and the client factory. These both factories instantiates an instance of a protocol class. Each factory is given their name based on theirs factory type. The *NodeServerFactory* class represents the node as a server while the *NodeClientFactory* class represents the node as a client. For the node server, reactor listens for connection at a specific port number, while for the node client reactor connects to a specific port number and IP address. The instance of the server and client class is instantiated at the main class of the node so that when the node is started initially, an individual node will behave as both as a server and client depending upon the action that it have to perform.

```

class NodeServerFactory(Factory):
    protocol = NodeProtocol
    def __init__(self, name):
        # initializing the name as a 'server'
        self.name = name

class NodeClientFactory(ClientFactory):
    protocol = NodeProtocol
    def __init__(self, name):
        # initializing the name as a 'client'
        self.name = name

if __name__ == '__main__':
    sFactory = NodeServerFactory('server')
    cFactory = NodeClientFactory('client')
    reactor.connectTCP('ip address', 'port number', cFactory)
    reactor.listenTCP('port number', sFactory)
    reactor.run()

```

**Figure 4.6 Node Factory Class**

So, putting all the classes together, here is a complete outline of a P2P Node.

```

from twisted.internet import reactor, protocol, interfaces
from twisted.protocols import basic
from twisted.internet.protocol import ServerFactory, Factory, ClientFactory
class NodeProtocol(LineReceiver):
    def __init__(self):
        # initialize NodeProtocol variable
    def connectionMade(self):
        if self.factory.name == 'server':
            # perform action as a server
        else:
            # perform action as a client
    def lineReceived(self, line):
        # manipulate line either forward it to child or perform actions on it
    def connectionLost(self, connector):
        # either try reconnecting or lose connection permanently

class NodeServerFactory(Factory):
    protocol = NodeProtocol
    def __init__(self, name):
        # initializing the name as a 'server'
        self.name = name

class NodeClientFactory(ClientFactory):
    protocol = NodeProtocol
    def __init__(self, name):
        # initializing the name as a 'client'
        self.name = name

if __name__ == '__main__':
    sFactory = NodeServerFactory('server')
    cFactory = NodeClientFactory('client')
    reactor.connectTCP('ip address', 'port number', cFactory)
    reactor.listenTCP('port number', sFactory)
    reactor.run()

```

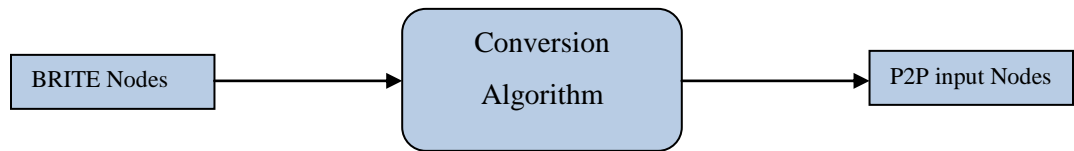
**Figure 4.7 A Complete Node Class**

## 4.2 Input Conversion:

The objective of this experiment is to determine a resilient solution for P2P Live Streaming using tree topology. To achieve the goal, the parameters that are considered are bandwidth and delay. Since, in a single tree topology a node can have only one parent but can have multiple children. The input we have got from the BRITE topology consists of nodes having multiple parent and multiple children. To attain a node with single parent but multiple children we use an input conversion algorithm. The algorithm receives the BRITE output format and generates a two dimensional matrix containing single parent

and multiple children if there are any. This 2 dimensional array is then mapped into a .txt file which is the required initial node input for the experiment.

The nodes in the P2P experiments are automatically generated from the BRITE topology generator. As explained in the section 1.5.4 BRITE is a universal topology generator not restricted to any particular way of generating topologies. The objective behind using a topology generator is to obtain a huge set of nodes which then would be fed into the designed P2P experiment. The BRITE output conversion to desired P2P nodes consists of three components.



**Figure 4.8 Input Conversion Process**

#### 4.2.1 BRITE Nodes:

The image below shows the initial state of the Input as generated from BRITE.

---

```

Topology: ( 5 Nodes, 7 Edges )
Model (3 - ASWaxman): 5 1000 100 1 2 0.15000000596046448 0.20000000298023224 1 1 10.0 1024.0

Nodes: ( 5 )
0      266    936    3      3      0      AS_NODE
1      844    279    3      3      1      AS_NODE
2      473    943    3      3      2      AS_NODE
3      70     603    3      3      3      AS_NODE
4      958    276    2      2      4      AS_NODE

Edges: ( 7 )
0      2      0      207.1183236703117    0.6908723623404551    10.0  2      0      E_AS  U
1      2      1      760.6161975661576    2.537142537342142    10.0  2      1      E_AS  U
2      3      2      527.2655877259581    1.7587686869893109    10.0  3      2      E_AS  U
3      3      0      386.4000517598309    1.2888918364978712    10.0  3      0      E_AS  U
4      4      1      114.03946685248927    0.3803947157753023    10.0  4      1      E_AS  U
5      4      3      946.2943516686549    1.1564981920547677    10.0  4      3      E_AS  U
6      0      1      875.0617121094946    2.9188916824234936    10.0  0      1      E_AS  U
  
```

**Figure 4.9 Nodes generated from BRITE**

This figure contains information about the number of nodes, edges and the model used to generate the topology. BRITE topology generator contains two different models: *Waxman* and *Barabasi*. Using any of these models is not of a great concern since we are interested in the nodes and edges generated by the model rather than the models itself. So, using any of these models won't affect our results. We need the BRITE topology generator for generating a topology. Feeding the basic parameter in BRITE topology generator we obtain a mesh topology. Later using an algorithm the mesh topology is transverse into tree topology. Basically, the entire output file from BRITE contains the same information except the number of nodes and edges, which depend upon the parameter (number of nodes), passed while generation of BRITE output file.

The table below describes what the fields are [28] in corresponding to the figure 3.2:

Field	Meaning	Field	Meaning
NodeId	Unique id for each node	EdgeId	Unique id for each edge
xpos	x-axis coordinate in the plane	from	node id of source
ypos	y-axis coordinate in the plane	to	node id of destination
indegree	Indegree of the node	length	Euclidean length
outdegree	Outdegree of the node	delay	propagation delay
Asid	id of the AS this node belongs to (if hierarchical)	bandwidth	bandwidth (assigned by <i>AssignBW</i> method)
type	Type assigned to the node (e.g. router, AS)	Asfrom	if hierarchical topology, AS id of source node
		Asto	if hierarchical topology, AS id of destination node
		type	Type assigned to the edge by classification routine
Explanation of Node fields		Explanation of Edge fields	

**Figure 4.10 Fields and its meaning of BRITE outputs fields**

The experiment designed for P2P Live Streaming requires only few of these fields so we ignore those fields that are not relevant to the experiment. From the node table we only consider the number of nodes and their ids, rest of the other fields are irrelevant to the experiment. From the edge table we only consider 'from', 'to', 'bandwidth' and 'delay' fields while the rest of the other fields are ignored.

#### 4.2.2 Input Conversion Algorithm:

The BRITE file received as an output is in .brite format containing lots of fields that are not required to the experiment. To remove these irrelevant fields we use input conversion algorithm. The algorithm basically reads through the given file and returns a two dimensional array which is later mapped to .txt file and fed to the system as nodes.

The pseudo-code below explains how the conversion from BRITE nodes to P2P nodes is attained.

```
def generateMatrixFormat(filename):
    # It reads the brite file and returns all the edges value in a 2D matrix format
    inputMatrixArray = readfileAsArray(filename)
    # defining the 2D Matrix array for the output which contains bandwidth and delay of an edge
    outputMatrixArray = [['' for x in range(noofnodes)] for y in range(noofnodes)]
    for col in range(0, noofnodes):
        i = 0
        for row in inputMatrixArray:
            parent = int(inputMatrixArray[i][1])
            child = int(inputMatrixArray[i][2])
            if ((col == parent) or (col == child)):
                band = str(inputMatrixArray[i][5])
                delay = str(abs(float(inputMatrixArray[i][4])))
                outputMatrixArray[parent][child] = band+', '+delay
            i = i + 1
    return outputMatrixArray
```

**Figure 4.11 Input Conversion Algorithm**

The function *generateMatrixFormat()* receives the topology generated from BRITE and converts into two dimensional matrix array which is represented by *outputMatrixArray* in the above code. The algorithm starts with passing filename as a parameter which also means the complete path to the filename.

*inputMatrixArray* is a two dimensional array which holds only the edge values of BRITE topology. The function *readfileAsArray(filename)* returns the values after the Edges (7) (refer to figure 3.2) as it is but in two dimensional matrix.

*outputMatrixArray* is the final matrix which array which holds only bandwidth and delay of each edge. Since not all node connects to each node, these types of edges which doesn't exists are represented by a dummy value as 'x' in the two dimensional array.

There are two loops in this algorithm; the first loop represents the node id, the second loop represents the rows in the *inputMatrixArray*. The second column of this array represents the parent node, the third column represents the child node, the fifth column represents the delay of the edge and the sixth column represents the bandwidth of the node. These are the fields that we are interested in fetching from the *inputMatrixArray*. So, the second and thirds column of the array is assigned as parent and child, if any of this node id matches the loop count or *col* (which represents the node id), the fifth and sixth column of that array is assigned as delay and bandwidth of that parent and child and is finally stored in the *outputMatrixArray*. This process continues as long as association of each node with another node is reached, in short the loop check for the connection of one node with every other node.

All those nodes which have association with another node are stored in the *outputMatrixArray* with their respective bandwidth and delay and for those edges without association the value is represented as 'x'.

#### 4.2.3 P2P Input Nodes:

After the conversion process of BRITE file to two dimensional arrays of P2P Nodes, these values in the arrays are mapped as a matrix in a text file separated with spaces.

The figure below shows the output of the matrix file referring figure 3.2 as the initial input.

```

x 100.0,1.0 x x x x 100.0,1.0 x x x
x x x 100.0,1.0 x x x 100.0,1.0 x x
100.0,1.0 100.0,1.0 x x x x x x x x
100.0,1.0 x 100.0,1.0 x x x x x x x
x x 100.0,1.0 100.0,1.0 x x x x x x
100.0,1.0 100.0,1.0 x x x x x x x x
x 100.0,1.0 100.0,1.0 x x x x x x x
100.0,1.0 x x x 100.0,1.0 x x x x x
100.0,1.0 x x x x 100.0,1.0 x x x x
x x x x x x 100.0,1.0 x 100.0,1.0 x

```

**Figure 4.12 P2P Input Nodes**

Each value in the array represents the edge value (relation of a node with other node). For example

- The first x value in the first row and first column means that there is no association (edge) for 1<sup>st</sup> – 1<sup>st</sup> node.
- The second value 100.0, 1.0 in the first row and second column means that there is a connection between 1<sup>st</sup> – 2<sup>nd</sup> nodes. 100.0 represent the bandwidth of that edge and 1.0 represents the delay of that edge.



- There exists no association (edge) for a node with itself. E.g. 1<sup>st</sup>- 1<sup>st</sup> node, 2<sup>nd</sup> – 2<sup>nd</sup> node etc.
- The association between  $i^{\text{th}} - j^{\text{th}}$  contains the same value as the association between  $j^{\text{th}} - i^{\text{th}}$  does.

### **4.3 Topology Formation:**

The P2P Node forms a tree topology for sharing the data and passing the information to the next level. The basic implementation of server and node id explained in section 3.1. These two components have their own attributes. Every node has same attribute in the topology. The attributes are as follow:-

*NodeId* : It represents the node uniquely in the topology. This is the id generated from the BRITE topology generator. For the streaming server, the node id is assigned '0' by default.

*Name*: The name of the node given by themselves. For this experiment it is automatically generated alphabetically starting from 'A'.

*Listen Port*: The port at which node waits for connection (any node connecting at this port is the child of particular node).

*Child*: It contains the ids of node to which it streams the data (a node can have multiple child which are separated by comma).

*Parent*: It contains the id of a node (a particular node has just one parent)

*Upload bandwidth*: It is the amount of the data the node streams to its children.

*Delay:* It is the amount of the time the data takes to travel from its parent to itself.

*Backup Parent:* It is the parent in the backup tree. Only connection is established between the nodes, there is no data flow in the backup tree. The data only flows in the base tree.

*Backup Child:* It is the children in the backup tree. A node can have multiple children which are separated by commas.

*Depart time:* The time which is generated randomly at the point when the node connects in the network. This time is allocated so that the nodes fall automatically after certain amount of time without doing it manually.

The main server holds a copy of all these information of each node in a dictionary. A server holds information in the following manner.

All the above attributes are represented by a different data structures present in the Python. There is a main data structure known as ‘dictionary’ which holds this information.

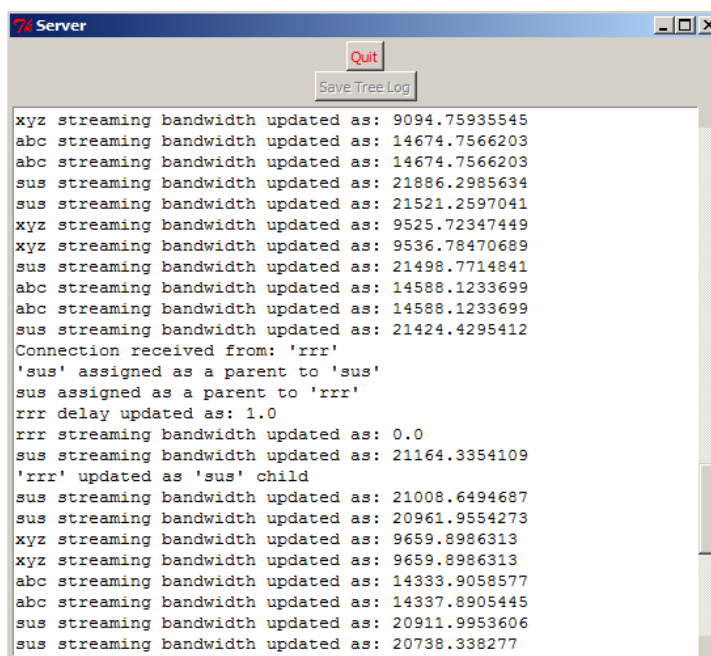
```
Main Dictionary = {Node 1 : nodeObj (This objects holds)
                  Node 2 : nodeObj all the attributes
                  Node 3 : nodeObj mentioned above)
                  .4 : .
                  . : .
                  . : .
                  . : .
                  Node ith : nodeObj
                  }
```

**Figure 4.13 Data structure for storing nodes**

### 4.3.1 Node Arrival (Joining of Peers):

The streaming server listens at a particular port waiting for the nodes to connect. The port of the server can be any integer value or any port which is not open already. This is the main server which is the only producer of the content. Other nodes just forward the content in the tree.

Below is the server UI.



```
Server
Quit
Save Tree Log
xyz streaming bandwidth updated as: 9094.75935545
abc streaming bandwidth updated as: 14674.7566203
abc streaming bandwidth updated as: 14674.7566203
sus streaming bandwidth updated as: 21886.2985634
sus streaming bandwidth updated as: 21521.2597041
xyz streaming bandwidth updated as: 9525.72347449
xyz streaming bandwidth updated as: 9536.78470689
sus streaming bandwidth updated as: 21498.7714841
abc streaming bandwidth updated as: 14588.1233699
abc streaming bandwidth updated as: 14588.1233699
sus streaming bandwidth updated as: 21424.4295412
Connection received from: 'rrr'
'sus' assigned as a parent to 'sus'
sus assigned as a parent to 'rrr'
rrr delay updated as: 1.0
rrr streaming bandwidth updated as: 0.0
sus streaming bandwidth updated as: 21164.3354109
'rrr' updated as 'sus' child
sus streaming bandwidth updated as: 21008.6494687
sus streaming bandwidth updated as: 20961.9554273
xyz streaming bandwidth updated as: 9659.8986313
xyz streaming bandwidth updated as: 9659.8986313
abc streaming bandwidth updated as: 14333.9058577
abc streaming bandwidth updated as: 14337.8905445
sus streaming bandwidth updated as: 20911.9953606
sus streaming bandwidth updated as: 20738.338277
```

Figure 4.14 Server Log

The server UI contains of a log report where every event occurring in the network is displayed. Also server holds all the information of the nodes and their events.

The nodes for the topology are automatically generated and convert to feed in the system.

A node initially connects to the server with a message which contains 3 parameters (information).

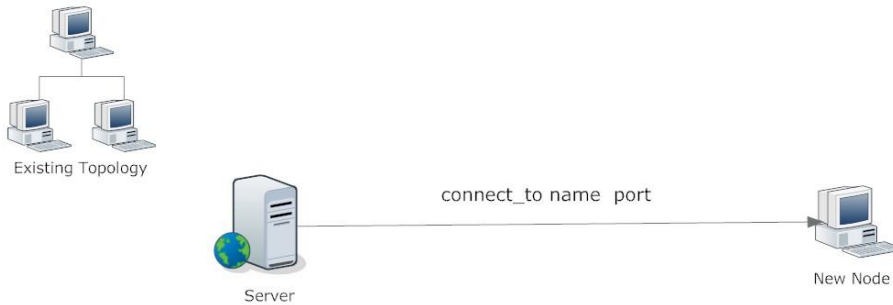
A step by step connection and formation of tree topology is described below.

Step 1: Establishing connection and sending initial message to server.



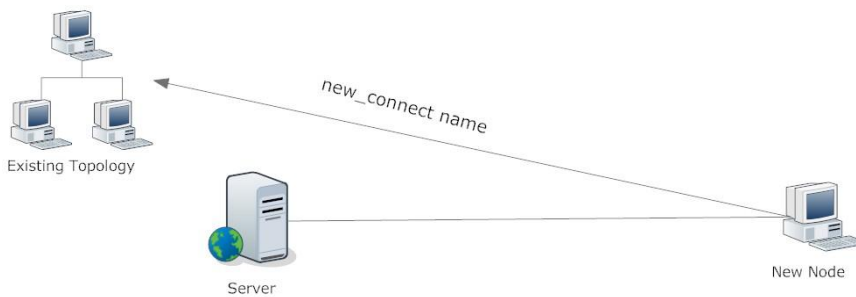
A new node establishes connection with server and sends message as 'initialInfo' along with its name. Since the experiment is carried in a single system IP cannot uniquely identify a node. But in real case IP is used to identify a node. For this experiment IP is replaced with name or Id to uniquely identify it. Port is the listening port for the particular new node.

Step 2: Parent connection Information.



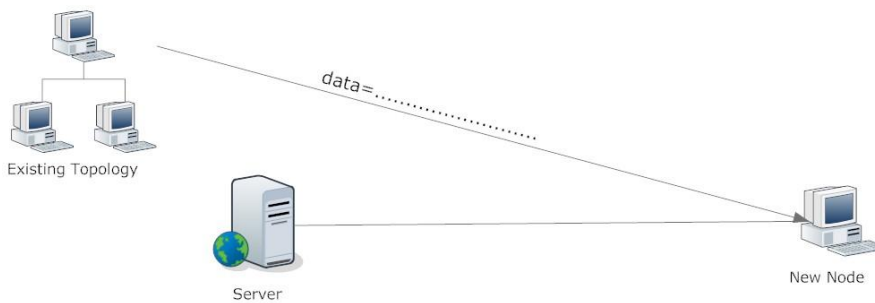
In this process the server looks in to its database and finds out a parent for the new node. To find a parent there is an algorithm known as 'Node Selection Algorithm' described in later section. This algorithm searches for the parent based on the two parameter bandwidth and delay. A node in database which has maximum streaming capacity and lowest possible delay is selected as a parent for the new node.

### Step 3: Establishing connection with the Parent Node.



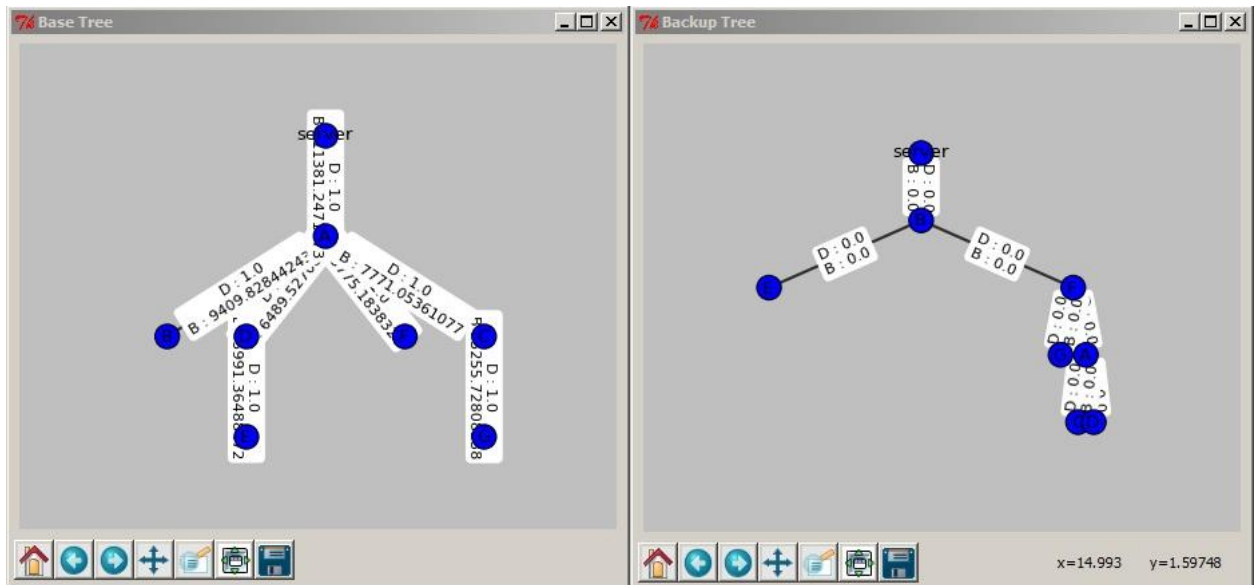
Once the new node receives the id and port number of its parent, it request for initial connection to the new parent which is already a part of the existing topology by sending message as 'new\_connect' along with its name (Id).

### Step 4: Receiving Data from the parent Node



Once the connection between the new node and its parent is established it starts to receive data as long as its parent gets the data. The parent node in this case just forward whatever data it has received from its parent. The data forwarded is a stream of image, but the image is forwarded in string format as a lot of chunks. The new node receives all these chunks and assembles it and displays the image and so on it forwards the received data to its children.

Here is a display of a tree topology with a server and seven nodes.

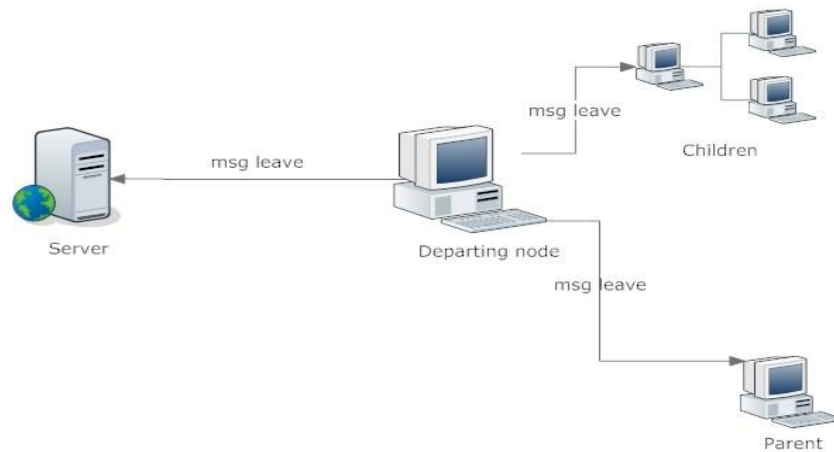


**Figure 4.15 Base tree and its backup tree**

### 4.3.2 Node Departure (Leaving of Peers)

In P2P environment dealing with node departure is one of the research topics. Users leave frequently in the P2P environment which causes tree instability and which in return causes playback problems. Users leave the network in two different ways, one is graceful leaving in which the server is pre informed before the departure and other case is sudden leave which may be caused due to hardware problem or network issues. To address this problem we use a backup tree where each node a backup parent to receive the data in case the current parent leaves.

In case of graceful departure a node sends a departure message to all its children, parent and server.



A departing node in graceful process sends a message as ‘msg’ leave to notify everyone that it is leaving. Once other node receives this message they inform their backup parent about it and receives data from them. Also everyone in connection with departing node removes the node from their list as well.

In case of sudden leave every node that is connected to departing node loses a connection which is notified by an API in twisted known as *connectionLost()*. As same in the case of graceful node, the node behaves exactly the same way in sudden leave. Once a node departs from the topology the server re construct the backup tree and sends the new backup parent information to all the nodes in the topology

#### **4.4 Base Tree Construction Algorithm:**

Base Tree is the major tree in the topology through which the data flows from the parent to the children. The top of the tree is always the streaming server. Base is formed due to the continuous arrival of new node and also due to the failure of a node. During a process of new node arrival the server has to select a parent for the new node from the existing topology. The algorithm used during this process is known as ‘Parent Selection

Algorithm'. The parent is selected based on two parameters – bandwidth and delay. Any node that has the best combination of high bandwidth with low delay will be considered as the parent for the new node. During this process first of the all the nodes in the existing topology are normalized on this factor and any node that has the highest value of normalization factor will be the parent node.

#### Normalization Factor:

The two parameter based on which the parent is selected are two different factors. So to combine these two factors is known as normalization. The factor that determines this process is known as the normalization factor.

Following program describes how a node in a network is normalized.

```
def getNormalizeFactor(hiDelay, loDelay, selfDelay, hiBandwidth, loBandwidth, selfBandwidth):
    bandFactor = 0.0
    delayFactor = 0.0
    if (selfBandwidth <= loBandwidth): bandFactor = 0.0
    elif (selfBandwidth == hiBandwidth): bandFactor = 100.0
    else: bandFactor = ((selfBandwidth - loBandwidth)/(hiBandwidth - loBandwidth)) * 100

    if (delay == loDelay): delayFactor = 100.0
    elif (delay == hiDelay): delayFactor = 0.0
    else: delayFactor = 100 - (((delay - loDelay)/(hiDelay - loDelay)) * 100)

    return ((bandFactor + delayFactor)/2)
```

**Figure 4.16 Normalization Algorithm**

The function *getNormalizeFactor()* takes six parameters:

'*hiDelay*' = highest delay value in the network.

'*loDelay*' = lowest delay value in the network.

'*selfDelay*' = delay of the node which is to be normalized.

'*loBandwidth*' = lowest uploading bandwidth of any node in the network.



'*hiBandwidth*' = highest uploading bandwidth of any node in the network.

'*selfBandwidth*' = uploading bandwidth of the node which is to be normalized

The delay of the above parameter is calculated from the server to a particular node. The uploading bandwidth of the node keeps decreasing as the number of the children of that particular node keeps on increasing. The delay of the node in the tree keeps on increasing as we move along downwards to the tree and the node that reside on the higher of the tree has less delay.

The function has two factors bandwidth factor and delay factor which are both initially set to 0. The above algorithm determines the both the factor for a node and returns the half of their sum. The both factor delay and bandwidth contains an equally important role in determining the factor.

#### Parent Node Selection:

When a new node connects to the content server it selects a parent for new node using 'parent selection algorithm'. The selected parent id is then forwarded to the new node and the new node connects to the id given by the server and gets data from that node.

Following pseudo code explains the parent selection algorithm.

```
def getParentNode(baseTreeDict, newNodeID):
    parentNode = ''
    if len(baseTreeDict) == 0:
        parentNode = 0
    if len(baseTreeDict) == 1:
        parentNode = baseTreeDict.get('1').getID()
    else:
        normDict = normalize(nodeDict, nodeID)
        parentNode = sort(normDict).getHighestNormID()
    return parentNode
```

**Figure 4.17 Node Selection Algorithm**

The function *getParentNode()* gives the id of the parent that the new node will stream data from. It contains parameter such as

*baseTreeDict*: This is the current tree topology of the base tree. This tree will be normalized and the new node will be added to this dictionary

*newNodeID*: The id of the new node.

It contains of three conditions: The first condition where there are no node at all in the tree. This is the very beginning phase of tree construction where the server initiates connection and waits for the peers to build the topology. The first node to join this tree will directly receive data from the server. The second condition where there is only one node in the tree. The server doesn't have to normalize the tree since there is only one node. The one node will be the parent for the new node. The third condition where there are two or more than two nodes in the tree. This is the most likely scenario for any new node joining the tree. In this case the server normalizes the nodes existing in the network and picks a node having the highest normalization factor as explained in the normalizing algorithm.

#### **4.5 Backup Tree Construction Algorithm:**

Backup Tree is the secondary tree in the network which contains an alternate parent to a particular node in case of node failure or departure. Backup Tree is constructed in two scenarios:

Node Arrival- Every time when a new node arrives in a network the backup tree is reconstructed even if a backup tree exists. It is necessary to reconstruct the backup tree one a new node has arrived because the new node becomes the leaf node in the base tree. To achieve a resilient tree all the interior node in the base tree are made the leaf node in the backup tree, and most of the leaf node in the base are made the interior node in the backup tree. Also, the arriving new node might have good bandwidth and less delay placing it at the top of the tree can make the topology stable.

Node Failure- When a node departs with pre-information or without any information, a small segment of existing backup tree becomes the path for the data flow and new backup tree is created again. If a node leaves the network with pre-information, the children node or the affecting node immediately receive the data from their backup parent and a new backup tree is created. The new information (parent name, port) is forwarded to all the nodes in the base tree by server.

In case of sudden departure where the children nodes do not receive any pre-information of their parent failure, the server provides the information about the failure and the children node switch to their backup parent for the data. The server initially maintains a connection with all the nodes, while in case of failure this initial connection of failed node is lost which helps the server to identify the failed node. Once a node has failed the server creates a backup tree of the recent base tree and forward the new information (parent name, port) to all the nodes in the base tree.

Following algorithm explains the backup tree construction.

```

def getBackupTree (baseTreeDict) :
    backupDict = {}
    innerNodeList = getInnerNodes (baseTreeDict)
    leafNodeList = getLeafNodes (baseTreeDict)
    while (len (leafNodeList)) :
        print 'The value of i is' + str(i)
        # nObj gives the node(obj) which highest bandwidth value
        nObj = getHighestBandWidthNode (leafNodeList)
        # it sets all the attribute to the new object nInfo
        nInfo = setChildNParentToBackupTree (backupDict, nObj, baseNodeDict)
        # adding the new node to the backup dictionary
        backupDict.setdefault(nInfo.getID(), nInfo)
        # It removes the desired object (nObj) from the leafNodeList
        leafNodeList = removeNodeObjectFromList (leafNodeList, nObj)
    while (len (innerNodeList)) :
        nObj = getHighestBandWidthNode (innerNodeList)
        nInfo = setChildNParentToBackupTree (backupDict, nObj, baseNodeDict)
        # adding the new node to the backup dictionary
        backupDict.setdefault(nInfo.getID(), nInfo)
        # It removes the desired object (nObj) from the leafNodeList
        innerNodeList = removeNodeObjectFromList (innerNodeList, nObj)
    return backupDict

```

**Figure 4.18 Backup Tree Algorithm**

The backup tree is made from the base tree. The algorithm starts with identifying the interior nodes from the leaf nodes of the base tree. The *innerNodeList* consists of interior nodes of the base tree while the *leafNodeList* consists of leaf nodes of the base tree. The objective of the constructing the backup tree is to make the interior node of the base tree as leaf node of the backup tree while the leaf node of the base is made the interior node of the backup tree. So, we start constructing the tree with leaf node of base tree. The first while loop starts constructing the backup tree, it chooses the node with the highest bandwidth in the base tree which is done by the function *getHighestBandWidth()*. The nodes with the highest bandwidth are sequentially chosen and placed at the top of the tree i.e. (*backupDict*). Once all the leaf node of the base tree is chosen the second while loop starts appending the interior node of the base tree to the backup tree. The

process for choosing the interior node of the base tree is same as that of choosing the leaf node from the base tree. Finally all the nodes are appended to the backup tree until each node in the base tree is finished. *backupDict* is a dictionary which consists the final backup tree of the given base tree.

## 4.6 Results

### 4.6.1 Comparison (Random vs. Algorithm)

Since the objective of this experiment is to achieve a resilient multicast using the algorithm developed. We have compared the result obtained of total node failure (number of nodes affected in the base tree + number of nodes affected in the backup tree) using the proposed algorithm and random selection algorithm. In case of random algorithm, the nodes in the backup tree are selected randomly instead of using the algorithm.

Below is the mathematical formulation to obtain the average node failure in the tree

Let  $n$  be the number of nodes in the tree

Let  $n_i$  be the node id

$$NF_{(base)} = \sum_i^n n_i \text{ where } i = 1, 2, \dots, n$$

$$NF_{(backup)} = \sum_i^n n_i \text{ where } i = 1, 2, \dots, n$$

$$NF_{(Total)} = NF_{(base)} + NF_{(backup)}$$

$$NF_{(Avg)} = NF_{(Total)} / n$$

Where,  $NF_{(base)}$  is the total number of node failed in base tree

$NF_{(backup)}$  is the total number of node failed in backup tree

$NF_{(Total)}$  is the sum of node failed in both the trees (base and backup tree)

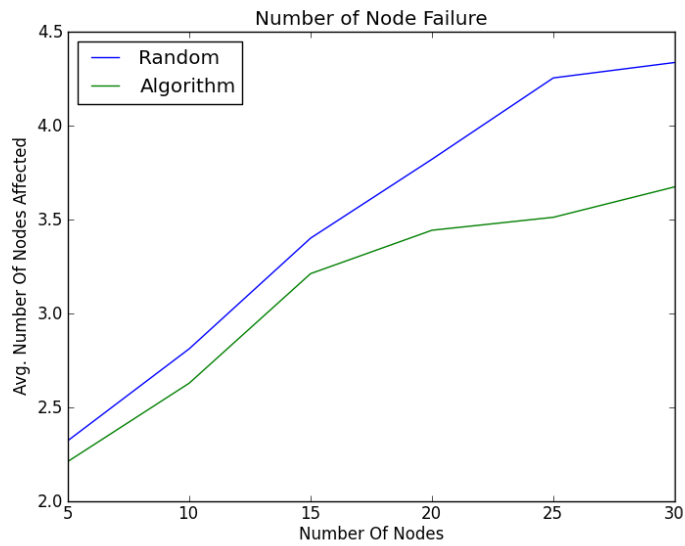
Based on the above mathematical formula we calculate the average number of node failed in the multicast for a particular number of node  $n$ .

The tabular data given below shows the average number of node failure in the both scenario for both the tree (base tree and backup tree).

Number of Nodes	Average Node failure (Using proposed algorithm)	Average Node Failure (Using random algorithm)
5	2.12	2.323
10	2.69	2.812
15	3.212	3.401
20	3.443	3.821
25	3.512	4.254
30	3.674	4.336

**Table 4.1 Average Node Failure (Algorithm vs. Random Selection)**

Below is the graphical representation of above obtained data.



**Figure 4.19 Average Node Failures**

The tabular and graphical data shows that the proposed algorithm provide better resilience to the multicast tree as compared to the random selection for construction of backup tree.

#### 4.6.2 Bandwidth Analysis

Since the data flows through the base tree we have analyzed how the bandwidth varies as the number of node increases in the base tree. The average flow of bandwidth is considered to be lowest bandwidth link in the multicast tree.

Let T be the multicast tree.

$$B_{(avg)} T = B_{(min)} T$$

Where,  $B_{(avg)}$  is the average bandwidth of the base tree T

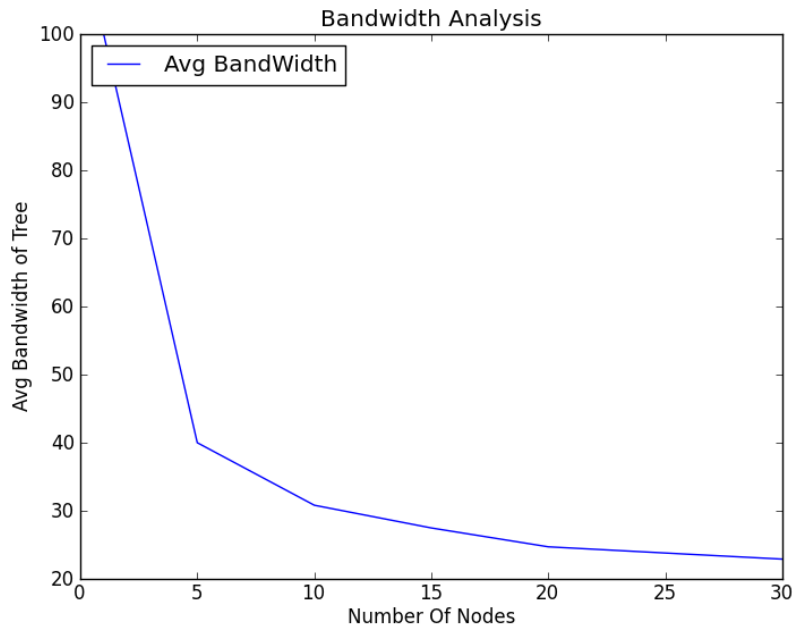
$B_{(min)}$  is the minimum bandwidth that any link has in the base tree T

The tabular data below shows the bandwidth variation as the number of node increases in the base tree. Assuming the server is streaming as the rate of 100 lines per second.

Number of Nodes	Bandwidth (lines per second)
1	100
5	40
10	30.819
15	27.481
20	24.710
25	23.802
30	22.905

**Table 4.2 Bandwidth Analysis**

Below is the graphical representation of above obtained data.



**Figure 4.20 Average Bandwidth**

The tabular and graphical data shows that as the number of node increases in the base tree the average bandwidth of the multicast tree decreases.

#### 4.6.3 Delay Analysis

Delay is the amount of the time it takes for a node to send a data and receive a reply for the sent message. We have used following mathematical formulation to calculate the average delay of the base tree.

$$T(\text{delay}) = \sum n_i \quad \text{where } i = 1, 2, \dots \text{ Number of nodes}$$

$$T(\text{avg}) = T(\text{delay}) / n$$

Where,  $T(\text{avg}) = \text{avg .delay of the base tree}$

$$T(\text{delay}) = \text{Total delay of the tree}$$

$n_i = \text{nodal delay of each node in the base tree}$



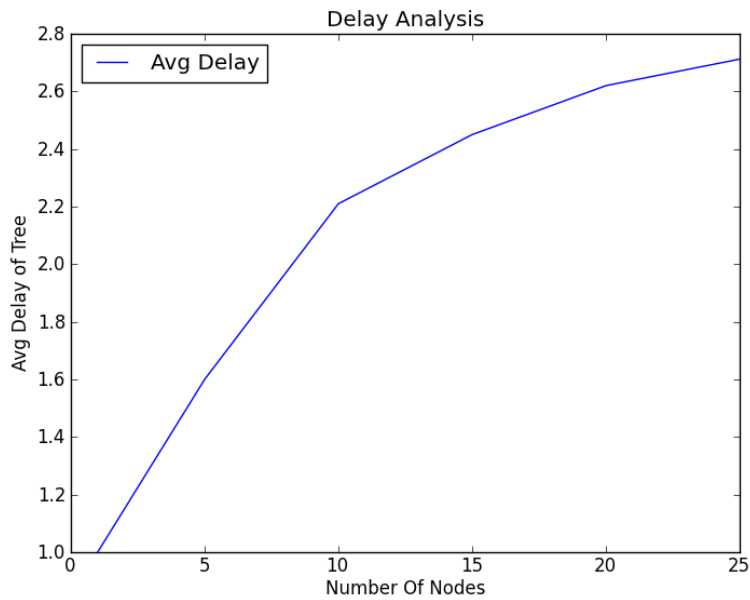
$n$  = Total number of nodes in the tree

Using the above formula we have obtained following results for the average delay of the multicast tree for different number of nodes.

Number of Nodes	Average Delay
1	1
5	1.6
10	2.21
15	2.45
20	2.62
25	2.712
30	2.78

**Table 4.3 Delay Analysis**

Below is the graphical representation of above obtained data.



### **Figure 4.21 Average Delay**

The tabular and graphical data shows that as the number of node increases in the base tree the average delay of the multicast increases

## **Chapter 5**

### **Conclusion**

This thesis studies about the most common problem faced in P2P Streaming that is churn. We present an approach to provide resilience to the P2P network caused by frequent joining and leaving of nodes. The proposed solution to this problem is to construct a backup tree for the existing base tree which provides continuous data or streaming to the failed nodes. Through mathematical interpretation and experimental results we show that introducing a backup tree can significantly increase the resilience of the tree which in return helps to provide effective tree topology structure.

For further step, we are interested in prototyping and evaluating the algorithms and proposed approach in real internet environment.

## Bibliography (or References)

- [1] Minor Gordon, “Twisted in knots”, IEEE Distributed Systems Online, vol. 8, no. 3, 2007, at. No. 0703-o3005
- [2] Lutz Mark, “Learning Python”, 4<sup>th</sup> Edition, 2007
- [3] Fettig Abe, “Twisted Network Programming Essentials” 2005
- [4] Networkx 1.7 Documentation, 30 March. 2012. Documentation of networkx.  
<http://networkx.lanl.gov/overview.html>
- [5] Networkx – Wikipedia, the free dictionary, 1<sup>st</sup> Nov. 2012, Wikipedia  
<http://en.wikipedia.org/wiki/NetworkX>
- [6] Hyunseok Chang, Sugih Jamin, Wenjie Wang, Live streaming performance of the Zattoo network”, In Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference, IMC '2009.
- [7] Kai Wang, Chuang Lin, "Insight into the P2P-VoD System: Performance Modeling and Analysis", 2009 Proceedings of 18th International Conference on Computer Communications and Networks, 2009 , Page1-6
- [8] Xinyan Zhang, Jiangchuan Liu, Bo Li, Yum, Y.-S.P. “CoolStreaming DONet: A Data-driven Overlay Network for Peer-to-Peer Live Media Streaming.” In Proceedings of IEEE INFOCOM 2005, 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Page 2102 - 2111 vol. 3.
- [9] Yaping Zhu, Rui Zhang-Shen, Sampath Rangarajan, Jennifer Rexford, “Cabernet: Connectivity Architecture for Better Network Services”, In Proceedings of

International Conference On Emerging Networking Experiments And Technologies, ACM CoNEXT 2008.

- [10] Documentation – Twisted, 30 March. 2012. Documentation of twisted  
<http://twistedmatrix.com/trac/wiki/Documentation>
- [11] Shicong Meng, Ling Liu, Jianwei Yin, “Scalable and Reliable IPTV Service through Collaborative Request Dispatching”, In proc. Of IEEE International Conference on Web Services (ICWS) 2010, Page 179-186.
- [12] Kunwoo Park, Dukhyun Chang, Junghoon Kim, Wonjun Yoon, and Ted Taekyoung Kwon, (2010), “An Analysis of User Dynamics in P2P Live Streaming” Services, Communications (ICC), 2010 IEEE International Conference on, Page 1-6
- [13] Mubashar Mushtaq, Toufik Ahmed, “P2P-based mobile IPTV: Challenges and opportunities”, Proceedings of the 2008 IEEE/ACS International Conference on Computer Systems and Applications, March 31-April 04, 2008, Page 975-980
- [14] Yu-Yi Chen, Jinn-Ke Jan, Yo-Yu Chi, Meng-Lin Tsai, (2009), A Feasible DRM Mechanism for BT-Like P2P System Information, Engineering and Electronic Commerce, 2009. IEEEC '09. International Symposium on , 323 -327
- [15] Prithula Dhungel, Xiaojun Hei, Keith, W. Ross & Nitesh Saxena. “The Pollution Attack in P2P Live Video Streaming Measurement Results and Defenses.” In proceedings of the 2007 workshop on Peer-to-peer streaming and IP-TV, August 2007, Page: 323-328.

- [16] Magharei N., and Rejaie, R. Prime: “Peer-to-peer receiver driven mesh-based streaming”, In Proceedings of IEEE INFOCOM, 2007.
- [17] Yong Liu, Yang Guo, Chao Liang, “A survey on peer-to-peer video streaming systems”. The Journal of P2P Networking and Applications, Springer, Volume 1, Number 1 / March, 2008, page 18-28
- [18] Castro, M., Druschel, P., Kermarrec, A.M., Nandia, Rowstron, A., and Singh, A. “SplitStream: High bandwidth multicast in cooperative environments”. In Proceedings of ACM SOSP (2003).
- [19] Cisco “Cisco visual networking index: Forecast and methodology, 2009-2014”, 2010
- [20] Susu Xie, Bo Li, Keung G.Y., Xinyan Zhang “CoolStreaming: Design Theory and Practice” In proceedings of IEEE transactions, December 2007, Page 1661 – 1671
- [21] Bo Li, Hao Yin “Peer-to peer Live Video Streaming on the internet: issues, existing approaches, and challenges” In proceedings of IEEE communication magazine, 2007, pages 94-99
- [22] Venkata N. Padmanabhan, Helen J.Wang, Philip A. Chou “Resilient peer-to-peer Streaming” In proceedings of IEEE international conference on Network Protocols, 2003, Page 16 - 27
- [23] V.K. Goyal. Multiple Description Coding: Compression Meets the Network. IEEE Signal Processing Magazine, September 2001, page 74-93

- [24] Chao Liang, Yong Liu & Keith W. Ross, (2009), "Topology Optimization in Multi-Tree Based P2P Streaming System." IEEE, Tools with Artificial Intelligence, 21<sup>st</sup> International Conference On, Page 806 -813
- [25] Y.-H. Chu, S. G.Rao, and H. Zhang, "A case for end system multicast," in Proceedings of ACM SIGMET- RICS, 2000.
- [26] V. Venkataraman, K. Yoshida, and P. Francis, "Chunkyspread: Heterogeneous unstructured end system multicast," in Proceedings of IEEE International Conference on Network Protocols (ICNP), 2006
- [27] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "SplitStream: High-bandwidth multicast in cooperative environments," in Proceedings of ACM SOSP, 2003.
- [28] Alberto Medina, Anukool Lakhina, Ibrahim Matta, John ByersThe, "BRITE Output Format", 30 March, 2012,