Formation Doctorale
Systèmes Informatiques Répartis

Université Pierre et Marie Curie

DEA Systèmes Informatiques Répartis

Mémoire du Stage

Transmission Multimédia en direct dans le réseau pair-à-pair

LIU Yong

Sous la direction de
Olivier Fourmaux
Chercheur du laboratoire LIP6
Réseau et Performance

Année 2003 – 2004

Abstract

With the growth of the computing power of personal computer and the proliferation of broadband access to the Internet, media streaming has widely diffused. The high bandwidth required by live streaming multimedia greatly limits the number of clients that can be served by a source using unicast. IP Multicast should be a good solution to this problem, but the fact that the current Internet does not widely support IP Multicast force us to find other ways. Another existing solution is the proxy caching technique that can accomplish effective media streaming, but it cannot adapt to the variations of user locations and diverse user demands. Under this situation, the applying of P2P systems on live media streaming become a attracting direction. By using the P2P communication architecture, media streaming can be expected to smoothly react to network conditions and changes in user demands for media-streams.

In recent years, there are many end host multicast solutions have been proposed, for example, PeerCast, ZigZag, Promise,Narada, Delaunay, CoopNet, and SplitStream etc. The main problem they emphasized are slight different, but all of them would have addressed the common basic streaming problem, for instance the problem of transience, the problem of control of end-to-end delay, etc.

In this paper, we propose a PeerCast like P2P media streaming solution. We focused on the maintaining algorithm of a single source, receiver driven media streaming delivery tree with a desirable end host performance metrics. Our proposed solution is featured by its tree maintaining algorithm that organize the multicast tree according to the geographical characteristic of nodes. The main idea in our algorithm is the concept "nodelist" and the process of "sorting" and "adjusting". Both the theoretical analyses and the simulation studies proved that under a client-base size of hundreds, our solution can maintain a 'balanced" streaming tree and gain a relatively good end user performance.

Table of Contents

I. Introduction

It is generally believed that the streaming media will constitute a significant fraction of the Internet traffic in the near future. Almost all of the existing matured model for multimedia streaming is based on client/server models. Since multimedia streaming requires high bandwidth, the source server network bandwidth runs out rapidly if unicast client/server model is used. The multicast should be a good solution to the bandwidth bottleneck problem, but the lack of widespread support for IP multicast in today's Internet (especially at the interdomain level) forces us to find other ways. Recently, the concept of P2P has recently gained popularity thanks to the wide deployment of P2P file sharing applications over the Internet. Therefore, the combination of these two problems becomes an interesting idea. We dare say that P2P model is ideal one to solve the server link bottleneck problem and what we have interest is the application of P2P to real-time media streaming, which is a more challenging problem than ordinary file-sharing. In file-sharing systems, a client first downloads the entire file and then uses it, termed as the "open-after-downloading" model; while in media streaming systems, a client consumes the content of a media file while the file is being downloaded, termed as the "play-while-downloading" model.

The P2P media streaming network is dynamically constructed by instances of joining and leaving the network by peers. A consumer peer searches a desired media stream by itself and retrieves it from an appropriate provider peer. At this time, the consumer peer can become a provider peer of the media stream for other peers if the media stream is cached there. It is easy to understand but it is not easy to realize it in practice. The challenge is to design good peer join and leave strategies to realize high quality streaming sessions. The quality of a streaming session depends on a combination of factors, ranging from the characteristics of the streaming sources (e.g., link capacity, availability, and offered rate) to the characteristics of the network paths (e.g., available bandwidth, packet loss rate, and overlap of paths from sources to receiver). In this paper, we will only concentrate on the affection of the algorithm that maintain the streaming tree and ignored the other factors. When we consider the algorithm, there are several issues that are important in designing an efficient P2P media streaming solution, for example: the transience of nodes, the end-to-end delay, the control of overhead, the control of the height of tree, etc. The detail can be found in the following text.

In this paper, we present and simulate the design of a single-source, receiver-driven P2P live media streaming architecture that can scale to hundreds of autonomous, short-lived nodes under the scope of Internet. Our solution is featured by the algorithms that organize the multicast tree according to the geographical characteristic of nodes. Through the join/leave/adjusting algorithm, our solution will maintain a 'balanced" streaming tree and gain a good end user performance. In section V and VII, we have the detail about the description of the algorithm and the result of simulation. The rest of the paper is organized as follows. In Section II, we give an overview of content Distribution and P2P Model of Computing. Next, in Section III, we have an overview about the basic concept and problem about media streaming over P2P model. In section IV, we discussed several existing media streaming solution over P2P network. In section VI, we give a description about the environment of simulation.

Note: in this paper, the word “source”, “server” and “root” have the same meaning. It means the machine or group of machines that contain the entire multimedia object; this machine or group of machines have a public address that can be used by the coming client to find the server; it is the first member of the multicast tree. The word “client”, “node” and “peer” have the same meaning. It means the machine in the network (LAN or WAN, Intranet or Internet, depending on the environment to which the software/application is applied) who want to get the multimedia from the source. When it joins into the tree, it becomes a “node” or a “peer” of the tree.

II. Overview of Content Distribution and P2P Model of Computing

The problem of media streaming is a part of content distribution problem in Internet. For better understanding, firstly, we will have an overview to content distribution and P2P model.

1. Content Distribution

Content distribution on the Internet uses many different service architectures, ranging from centralized client/server to fully distribute.

1.1. World Wide Web

At the very beginning of Internet, the World Wide Web has become one of the main channels for distributing information in the world today [1]. The Web is client/server architecture. Content on the Web is hosted on web servers and is served using the Hypertext Transfer Protocol (HTTP). Users use browser software to access the content stored on the servers. But the problem of this architecture is obvious: it is inefficient to satisfy the quickly increasing demand of Internet users, especially the multimedia content such as audio and video.

1.2. Proxy Caching

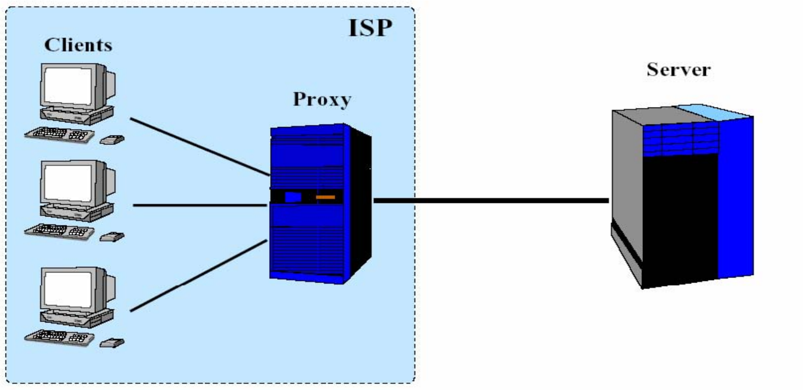


Figure1. Proxy caching

The first solution was to setup a caching proxy at the client site. This is show in Figure 1. In this approach the client sends its request to the caching proxy which checks if it has the requested object in its cache. If it does, it can return the object directly to the client and avoid contacting the server (also called origin server) which can be far away in the network [1]. If the caching proxy does not have the requested object, it will download it from the origin server, deliver it to the client, and cache a copy of the object.

The main problem with this approach is that the content provider has no control over the content once it has been retrieved from the origin server and placed into the caches. The content provider has no good way of ensuring that if the content changes, then all users would always get the new content. In order to avoid the problem of caches delivering stale, some content providers have resorted to marking all of their content as non-cacheable so that the caching proxies would not store it. This solution, while effective at solving the problem of stale content, defeats the purpose of installing caches because they cannot store content anymore. This problem was the primary motivation behind the development and success of content distribution networks (CDN).

1.3. Content Distribution Networks

Content Distribution Networks is a popular model of content distribution and have often been used among the large content providers. Figure 2 shows the architecture of a CDN.

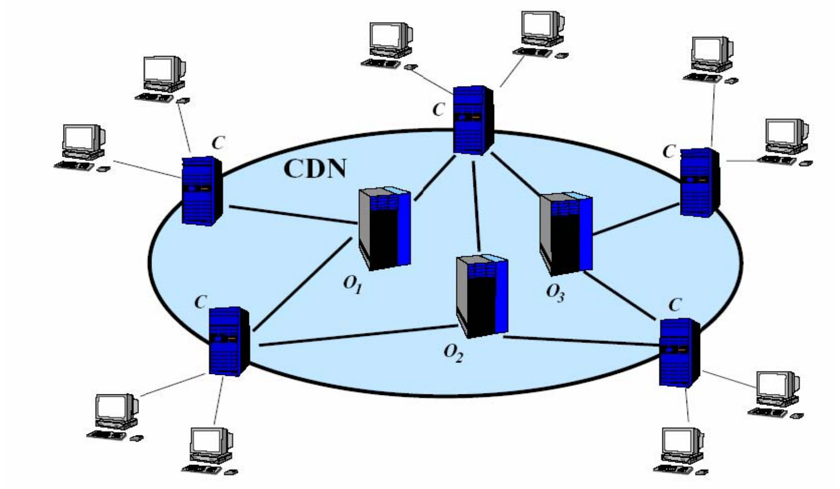


Figure2. CDN architecture

A CDN makes agreements with the content providers (O) to distribute their content to the users. The CDN operates content servers (C) that are typically placed near the users, for example at the dial-up ISPs. The user requests are redirected to these content servers which are able to serve the content fast. The internal network of the CDN connects the origin servers and content servers and is used to transfer content from the origin servers to content servers (or moving content from one content server to another). The main difficulty in creating a CDN is redirecting the clients to the content servers. Ideally, one would want this to be completely transparent to the clients so that no modifications to client software are needed nevertheless modern CDNs achieve this by manipulating information in the

Domain Name System (DNS) [1]. Because CDNs charge a high price for their services, they are mostly applicable to larger companies wanting to distribute their content. Individuals or small organizations would typically not be able to accord the services of a CDN; hence they would have to rely on client-side caching to help distribute their content.

1.4. Peer-to-Peer Networks

The latest development in content distribution is peer-to-peer networks. The first P2P was Napster [2] which allowed users to share MP3-files with each other. The main application for peer-to-peer networks has been file sharing, in which users make some files available on their computers and others can download these files. But now, more and more attention has been paid to the domain of live media streaming over P2P networks.

Generally, a P2P network can have three possible architectures:

Centralized Architecture

Figure 3 shows an example of a centralized and distributed P2P architecture. The centralized architecture requires that some central authority operates a single central server. This central server is responsible for answering the queries, hence all the query traffic is directed to it.

Distributed Architecture

In a distributed architecture all of the peers are equal each other and no peers keep any permanent information about which objects are stored where; also there is no directory of the peers which are a part of the network. An example of this architecture is Gnutella [3].

Hybrid architectures

This architecture want to keep a balance between the accuracy of the centralized architecture and the lower load of the distributed architecture.

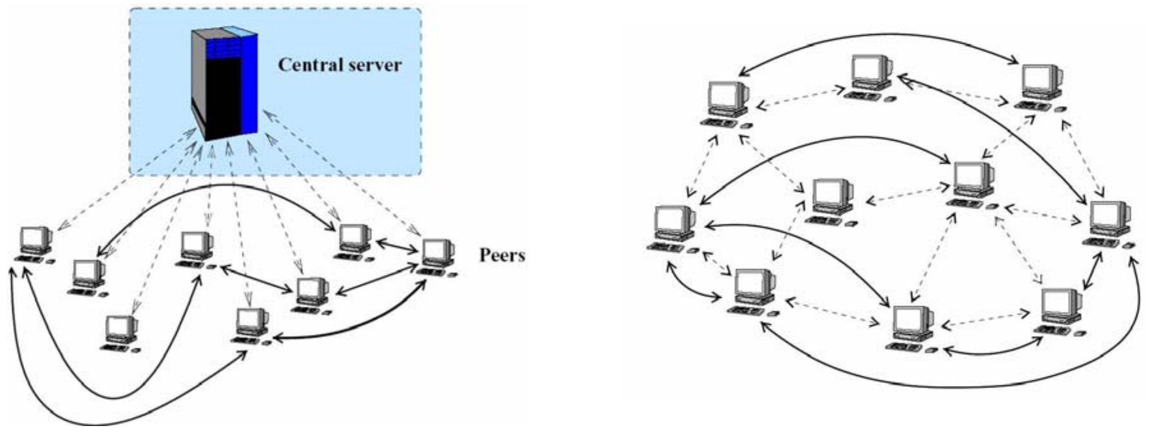


Figure3. Centralized versus distributed P2P architecture

2. The P2P Model of Computing

Below, we will have a further discussion on P2P architecture.

2.1. The concept of P2P

The concept of P2P is not new. Many earlier distributed systems have adopted a similar model, such as switched networks. The term P2P is also not new. In one of its simplest forms, it refers to the communication among the peers. Generally speaking, the term “peer-to-peer” (P2P) refers to the systems and applications that take use of the distributed resources to perform a function in a decentralized manner. With the rapid popularization of computers, P2P is increasingly receiving attention in research, product development, and investment circles. It is just because that P2P network have shown many advantages:

It enabled the resource aggregation [4];

It enabled direct communication among clients and therefore reduced the need for costly infrastructure;

It improved the extensibility of Internet application;

It made us avoid dependency on centralized points and therefore made the availability of information greatly increased.

We can predict that P2P computing is an alternative to the centralized and client/server models of computing, where there is typically a single or small cluster of servers and many clients. In its purest form, the P2P model has no concept of server; rather all participants are peers. Every single peer has the possibility to play the traditional role of server.

2.2. Goals of P2P

Like any other computing system, the goal of a P2P system is to support applications that satisfy the requirements of users. In a general way, a P2P approach has the following goals [4]:

(1). Resource aggregation and interoperability: Each node in the P2P system brings with it certain resources such as compute power or storage space. Applications that benefit from huge amounts of these resources, such as compute-intensive simulations or distributed file systems, naturally lean toward a P2P structure to aggregate these resources to solve the larger problem.

(2). Cost sharing and cost reduction: When that main cost becomes too large, a P2P architecture can help spread the cost over all the peers.

(3). Improved scalability/reliability: With the lack of strong central authority for autonomous peers, improving system scalability and reliability is an important goal. As a result, algorithmic innovation in the area of resource discovery and search has been a clear area of research, resulting in new algorithms for existing systems, and the development of new P2P platforms.

(4). Increased autonomy. In many cases, users of a distributed system are unwilling to rely on any centralized service provider. Instead, they prefer that all data and work on their behalf be performed locally. P2P systems support this level of autonomy simply because they require that the local node do work on behalf of its user.

(5). Anonymity/privacy. Anonymity and privacy are related to autonomy. A user may not want anyone or any service provider to know about his or her involvement in the system. With a central server, it is difficult to ensure anonymity because the server will typically be able to identify the client, at least by Internet address. By employing a P2P structure in

which activities are performed locally, users can avoid having to provide any information about themselves to anyone else.
(7). Dynamism. P2P systems assume that the computing environment is highly dynamic. That is, resources, such as compute nodes, will be entering and leaving the system continuously. When an application is intended to support a highly dynamic environment, the P2P approach is a natural fit.
(8). Enabling ad-hoc communication and collaboration. Related to dynamism is the notion of supporting ad-hoc environments. By ad hoc, we mean environments where members come and go based perhaps on their current physical location or their current interests. Again, P2P fits these applications because it naturally takes into account changes in the group of participants.

2.3. The location and classification of P2P

The classification of computer systems from the P2P perspective is showed in figure 4. All computer systems can be classified into centralized and distributed. Distributed systems can be further classified into the client/server model and the P2P model. The C/S model can be flat where all clients only communicate with a single server (possibly replicated for improved reliability), or it can be hierarchical for improved scalability. In a hierarchal model, the servers of one level are acting as clients to higher level servers. The P2P model can either be pure or it can be hybrid. In a pure model, a centralized server does not exist. An example of a pure P2P model is Gnutella. In a hybrid model, a server is approached first to obtain meta-information, such as the identity of the peer on which some information is stored, or to verify security credentials, then, the P2P communication is performed, examples of hybrid models are Napster or Groove.

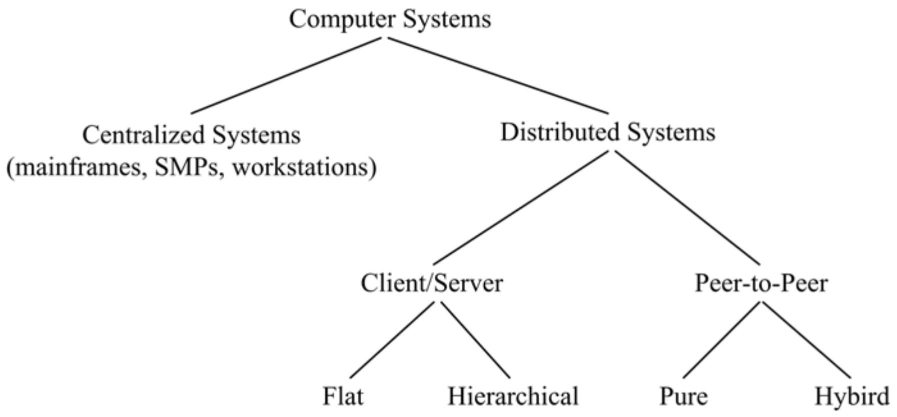


Figure4. Taxonomy of Computer Systems Architectures

2.4. Infrastructure Components

The components for an informal P2P systems architecture [4] include communication, group management, and robustness, class-specific and application specific.

(1). Communication

The P2P model covers a wide spectrum of communication paradigms. At one end of the spectrum are desktop machines mostly connected via stable, high-speed links over the Internet [4]. At the other end of the spectrum, are small wireless devices such as PDAs or even sensor-based devices that are connected in an ad-hoc manner via a wireless medium. The fundamental challenge of communication in a P2P community is overcoming the problems associated with the dynamic nature of peers.

(2). Group management

Peer group management includes discovery of other peers in the community and location and routing between those peers. Discovery of peers can be highly centralized such as in Napster, highly distributed such as in Gnutella, or somewhere in between.

(3). Robustness

There are three main components that are essential to maintaining robust P2P systems:
a. Security is one of the biggest challenges for P2P infrastructures, because, transforming a standard client device into a server poses risks to the system.
b. Resource aggregation in P2P systems is difficult because there are a wide variety of resources that may be aggregated across peers.
c. Reliability. The inherently distributed nature of peer networks makes it difficult to guarantee reliable behavior.

(4). Class-specific

Compute-intensive tasks are broken into pieces that must be scheduled across the peer community. Metadata describes the content stored across nodes of the peer community and may be consulted to determine the location of desired information.

(5). Application specific

Tools, applications, and services implement application-specific functionality, which correspond to certain P2P applications running on top of the underlying P2P infrastructure.

2.5. Algorithms

There are three most common algorithms implemented in P2P systems: Centralized directory model, flooded requests model, Document routing model.

Centralized directory model

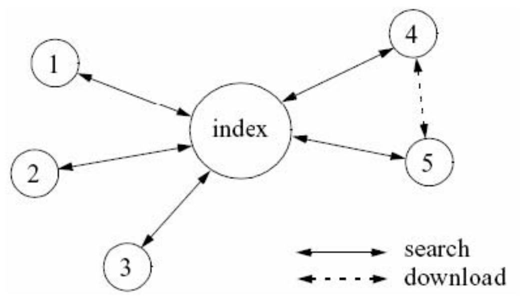


Figure5. Central index algorithm

In this model the peers connect to a central directory (Figure 5) where they publish information about the content they offer for sharing. Upon request from a peer, the central index will match the request with the best peer in its directory that matches the request. The best peer could be the one that is cheap cheapest, fastest, or the most available, depending on the user needs. Then a file exchange will occur directly between the two peers. This model requires some managed infrastructure (the directory server), which hosts information about all participants in the community. This can cause the model to show some scalability limits, because it requires bigger servers when the number of requests increase, and larger storage when the number of users increase. Napster [2] is based in this model.

Decentralized structured model (Document routing model)

In the document routing model, each peer from the network is assigned a random ID and each peer also knows a given number of peers (see Figure 6). When a document is published (shared) on such a system, an ID is assigned to the document based on a hash of the document's contents and its name. Each peer will then route the document towards the peer with the ID that is most similar to the document ID. This process is repeated until the nearest peer ID is the current peer's ID. Each routing operation also ensures that a local copy of the document is kept. When a peer requests the document from the P2P system, the request will go to the peer with the ID most similar to the document ID. This process is repeated until a copy of the document is found. Then the document is transferred back to the request originator, while each peer participating the routing will keep a local copy. FreeNet[5], Chord[6], CAN[7], Tapestry[8] and Pastry[9] are implemented in the model.

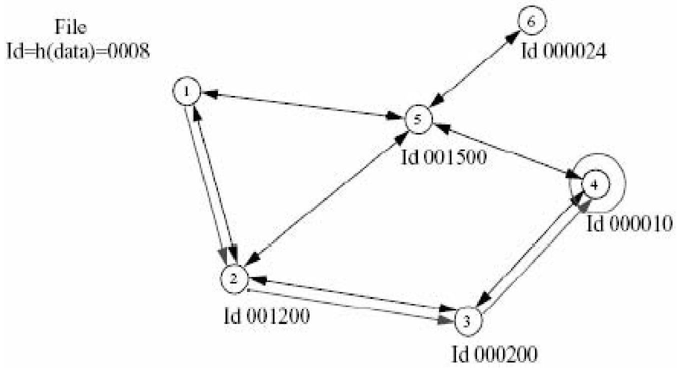


Figure6. Document Routing Algorithm

Decentralized unstructured model (flooded requests algorithm)

This is a pure P2P model in which no advertisement of shared resources occurs (see Figure 7). Instead, each request from a peer is flooded (broadcast) to directly connected peers, which themselves flood their peers etc., until the request is answered or a maximum number of flooding steps (typically 5 to 9) occur.

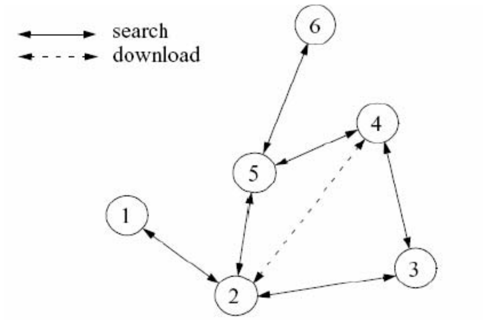


Figure 7 Flooded Requests Algorithm

This model requires a lot of network bandwidth and it does not prove to be very scalable, but it is efficient in limited communities such as a company network. By manipulating the number of connections of each node and appropriately setting the TTL parameter of the request messages, the flooded requests model can scale to a reachable horizon of hundreds of thousands of nodes. Gnutella[3], FastTrack [10], and Kaza[11] are based in this model.

Conclusion

Each of these models has their advantages and disadvantages, however, the decentralized unstructured systems are the most commonly used in today's Internet. The fundamental point is that in these unstructured systems, because the P2P network topology is unrelated to the location of data, the set of nodes receiving a particular query is unrelated to the content of the query. A host doesn't have any information about which other hosts may best be able to resolve the query. Thus, these "blind" search probes can not be on average more effective than probing random nodes.

2.6. The features of P2P technology

The features, which have a major impact on the effectiveness and deployment of P2P systems and applications, are:

Decentralization

While a centralized system is ideal to a more easily managed of the security and the access rights, its topology may yield inefficiencies, bottlenecks and wasted resources [4]. Therefore the decentralization put emphasis on the users' ownership and control of data and resources. In a fully decentralized system, every peer is an equal participant. This makes the implementation of the P2P models difficult in practice because there is no centralized server with a global view of all the peers in the network or the files they provide.

Scalability

An immediate benefit of decentralization is improved scalability. Scalability is limited by factors such as the amount of centralized operations (e.g, synchronization and coordination) that needs to be performed, the amount of state that needs to be maintained, the inherent parallelism an application exhibits, and the programming model that is used to represent the computation. Scalability also depends on the ratio of communication to computation between the nodes in a P2P system.

Anonymity

One goal of P2P is to allow people to use systems without concern for legal or other ramifications. There are three different kinds of anonymity between each communicating pair [4]: sender anonymity; receiver anonymity and mutual anonymity. Besides the kinds of anonymity, it is also very important to understand the degree of anonymity a certain technique can achieve. There are six popular techniques for enforcing different kinds of anonymity with different kinds of constraints: multicasting, spoofing the sender's address, identity spoofing, covert paths, intractable aliases, non-voluntary placement.

Self-Organization

In P2P systems, self-organization is needed because of scalability, fault resilience, intermittent connection of resources, and the cost of ownership. P2P systems can scale unpredictably in terms of the number of systems, number of users, and the load. It is very hard to predict any one of them, requiring frequent reconfiguration of centralized system. The significant level of scale results in an increased probability of failures, which requires self-maintenance and self-repair of the systems [4]. Similar reasoning applies to intermittent disconnection; it is hard for any predefined configuration to remain intact over a long period of time. Adaptation is required to handle the changes caused by peers connecting and disconnecting from the P2P systems.

Cost of ownership

Shared ownership reduces the cost of owning the systems and the content, and the cost of maintaining them. This is applicable to all classes of P2P systems, but it is probably most obvious in distributed computing. In P2P collaboration and communication systems, and in platforms, elimination of centralized computers for storing information also provides reduced ownership and maintenance costs.

Ad-Hoc Connectivity

The ad-hoc nature of connectivity has a strong effect on all classes of P2P systems. P2P systems and applications in distributed computing need to be aware of this ad-hoc nature and be able to handle systems joining and withdrawing from the pool of available P2P systems. While in traditional distributed systems this was an exceptional event, in P2P systems it is considered usual. In content sharing P2P systems and applications, users expect to be able to access content intermittently, subject to the connectivity of the content providers. In systems with higher guarantees, such as service-level agreements, the ad-hoc nature is reduced by redundant service providers, but the parts of the providers may still be unavailable. In collaborative P2P systems and applications, the ad hoc nature of connectivity is even more evident. Collaborative users are increasingly expected to use mobile devices, making them more connected to Internet and available for collaboration. To handle this situation, collaborative systems support transparent delay of communication to disconnected systems.

Performance

Performance is a significant concern in P2P systems. P2P systems aim to improve performance by aggregating distributed storage capacity and computing cycles of devices spread across a network. Because of the decentralized nature of these models, performance is influenced by three types of resources: processing, storage, and networking. Networking

delays can be significant in wide area networks. Bandwidth is a major factor when a large number of messages are propagated in the network and large amounts of files are being transferred among many peers. This limits the scalability of the system. Performance in this context tries to answer questions of how long it takes to retrieve a file or how much bandwidth a query will consume. There are three key approaches to optimize performance: replication, caching, and intelligent routing [4].
(1).Replication puts copies of objects/files closer to the requesting peers, thus minimizing the connection distance between the peers requesting and providing the objects.
(2).Caching reduces the path length required to fetch a file/object and therefore the number of messages exchanged between the peers.
(3).Intelligent routing and network organization. To fully realize the potential of P2P networks, it is important to understand and explore the social interactions among peers. For example, we can determine "good" peers based on interest, and dynamically manipulate the connections between peers to guarantee that peers with a high degree of similar interests are connected closely [14]. Establishing a good set of peers reduces the number of messages broadcast in the network.

Security

P2P systems share most of their security needs with common distributed systems: trust chains between peers and shared objects, session key exchange schemes, encryption, digital digests, and signatures [4]. Extensive research has been done in these areas, and new security requirements appeared with P2P systems, some are: Multi-key encryption, Sandboxing, Digital Rights Management, Reputation and Accountability, Firewalls.

Transparency and Usability

In distributed systems, transparency was traditionally associated with the ability to transparently connect distributed systems into a seamlessly local system. The P2P software should not require any significant set up or configuration of either networks or devices in order to be able to run. Also, P2P systems should be network and device transparent (independent). They should work on the Internet, intranets, and private networks, using high-speed or dial-up links. They should also be device transparent, which means they should work on a variety of devices, such as handheld personal digital assistants (PDAs), desktops, cell phones, and tablets. Another form of transparency is related to security and mobility [4].

Fault Resilience

One of the primary design goals of a P2P system is to avoid a central point of failure. Although most P2P systems (pure P2P) already do this, they nevertheless are faced with failures commonly associated with systems spanning multiple hosts and networks: disconnections/unreachability partitions, and node failures. These failures may be more pronounced in some networks (e.g., wireless) than others (e.g., wired enterprise networks). It would be desirable to continue active collaboration among the still connected peers in the presence of such failures. Replication of crucial resources helps alleviate the problem [4].

Interoperability

Although many P2P systems already exist there is still no support to enable these P2P systems to interoperate. Some of the requirements for interoperability include [4]:

self-determination for interoperation, communication among systems, exchange of request and data, and execution task in higher protocol levels, advertise and maintain about same level of security, QoS and reliability. In the past, there were different ways to approach interoperability, such as standards, common specifications, and common source code, open-source and de facto standards [4].

III. Streaming media over P2P model

Now, we have already the basic concept about P2P network and content distribution in Internet, we can find that although there have been significant research efforts in peer-to-peer systems during the last years, one category of P2P systems has so far received less attention: the P2P media streaming system [12].

1. The reason to stream media over P2P model

At the present time, mainly used media streaming applications over Internet are still based on the traditional client/server model of Content Delivery Networks (CDN). The performance we can get through this model is very limited. Even the large streaming servers are not able to feed more than a few hundred streaming sessions simultaneously. The streaming server is the single point of failure and high rate network access is costly. In addition, the selection of the best streaming server in a CDN during a session is difficult. Peer-to-Peer (P2P) networks provide characteristics and possibilities that can deal with these limitations.

A P2P communication infrastructure is built upon an overlay network whose topology is independent of the underlying physical network. Currently, many researchers are investigating how to implement media streaming in P2P networks in order to provide an efficient and scalable multimedia distribution service. Stringent Quality of Service (QoS) requirements for media distribution as well as dynamically changing transmission capacity and availability in P2P-networks pose many challenges.

2. What are a stream and a live Streaming?

A stream is a time-ordered sequence of packets belonging to a specific media file. This sequence of packets is not necessarily downloaded from the same serving node. Neither is it required to be downloaded in order. It must be displayed by the client in a specific order. It is the responsibility of the scheduler of final application to download the packets from a set of possible nodes before their scheduled display time to guarantee non disruptive playing of the media.

Live streaming refers to the synchronized distribution of streaming media content to one or more clients. (The content itself may either be truly live or pre-recorded.). A live stream has the important property of being history-agnostic: the group-member is only interested in the stream from the instant of its subscription onwards [13]. Live streaming media will form a significant fraction of the internet traffic in the near future. Recent trade reports indicate that if the current acceptance rate among end-users persists, streaming media could overtake television with respect to the size of the client base. The adoption trends for streaming media are expected to be particularly remarkable in the context of peer-to-peer (P2P) systems. Since video streams are high bandwidth applications, even a small number of clients receiving the stream by unicast are often sufficient to saturate bandwidth at the source. Thus, IP Multicast is a natural paradigm for distributing such content, but the deployment of IP Multicast has been slowed by difficult issues related to scalability, and support for higher layer functionality like congestion control and reliability, the service is to be simulated over unicast links between hosts, forming an overlay network over end-hosts. The application-layer assumes responsibility for providing multicast features like group management and packet replication. An end-host multicast solution then seems to be an ideal fit. However, a P2P system presents a challenging domain. A P2P system can have hundreds of nodes at any given instant. More significantly, nodes are autonomous, unpredictable, and may have short (of the order of minutes) lifetimes. Thus, a solution must be capable of good performance over a large and dynamic system. In [14] is proposed a tree-based end-host multicast architecture for streaming live media that can scale to hundreds of nodes in a P2P system.

3. Multiple Description Coding (MDC)

Multiple description coding is a method of encoding the audio and/or video signal into $M>1$ separate streams, or descriptions, such that any subset of these descriptions can be received and decoded into a signal with distortion (with respect to the original signal) commensurate with the number of descriptions received; that is, the more descriptions received, the lower the distortion (i.e., the higher the quality) of the reconstructed signal. This differs from layered coding, in MDC every subset of descriptions must be decodable, whereas in layered coding only a nested sequence of subsets must be decodable. For this extra flexibility, MDC incurs a modest performance penalty relative to layered coding, which in turn incurs a slight performance penalty relative to single description coding. Several multiple description coding schemes have been investigated over the years [15]. A particularly efficient and practical system is based on layered audio or video coding, Reed-Solomon coding, priority encoded transmission, and optimized bit allocation [16]. In such a system the audio and/or video signal is partitioned into groups of frames (GOFs), each group having duration $T = 1$ second or so. Each GOF is then independently encoded, error protected, and packetized into M packets, as shown in Figure 8.

If any $m \leq M$ packets are received, then the initial R_m bits of the bit stream for the GOF can be recovered, resulting in distortion $D(R_m)$, where $0 = R_0 \leq R_1 \leq \ldots \leq R_M$ and consequently $D(R_0) \geq D(R_1) \geq \ldots \geq D(R_M)$. Thus all M packets are equally important; only the number of received packets determines the reconstruction quality of the GOF. Further, the expected distortion is $M_{m=0}\ p(m)D(R_m)$, where $p(m)$ is the probability that m out of M packets are received. Given $p(m)$ and the operational distortion-rate function $D(R)$, this expected distortion can be minimized using a simple procedure that adjusts the rate points $R_1, .., R_M$ subject to a constraint on the packet length [16]. By sending the m_{th} packet in each GOF to the m_{th} description, the entire audio and/or video signal is represented by M descriptions, where each description is a sequence of packets transmitted at rate 1 packet per GOF. It is a very simple matter to generate these optimized M descriptions on the fly, assuming that the signal is already coded with a layered codec.

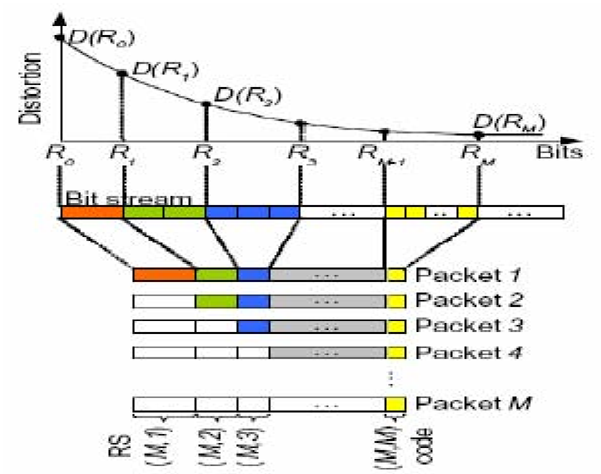


Figure8. Priority encoded packetization of a group of frames

4. Receiver driven (on-demand) streaming

On-demand streaming refers to the distribution of pre-recorded streaming media content on demand (e.g., when a user clicks on the corresponding link). As such, the streams corresponding to different users are not synchronized. When the server receives such a request, it starts streaming data in response if its current load condition permits. However, if the server is overloaded, say because of a flash crowd, it instead sends back a response including a short list of IP addresses of clients (peers) who have downloaded (part or all of) the requested stream and have expressed a willingness to participate. The requesting client then turns to one or more of these peers to download the desired content. Given the large volume of streaming media content, the burden on the server (in terms of CPU, disk, and network bandwidth) of doing this redirection is quite minimal compared to that of actually serving the content, this redirection procedure will help reduce server load by several orders of magnitude [16]. Many solution that are proposed are base on the model on-demand streaming, like PeerCast [14], ZigZag[17], etc. Our solution in this paper is also a kind of receiver driven, on demand streaming solution.

5. Strategies for the media distribution

For getting a relatively satisfied QoS (Quality of Service) requirement for media distribution over P2P networks. People have developed many techniques and strategies. Here, we will have an overview on replication, caching, pre-fetching, reputation, trust and semantic. And here, what we have interested in is the multimedia streaming, but the strategy of replication is often used in the algorithm of content/file sharing, so we will only give a more detailed discuss on the strategy caching and prefetching that are related with the media streaming.

5.1. Replication

In P2P systems, for providing a high availability, we need many copies of the same content to be replicated across peers in the systems. At the same time, content should not be excessively replicated, because it will waste bandwidth and storage resources.

Article [18] proposed replication schemes that solve the following difficult problems: availability, adaptive and distributed. Article [19] proposed replication strategies in the decentralized unstructured P2P systems and this kind of system is the most commonly used in today's Internet. In these decentralized unstructured P2P systems, the hosts form a P2P overlay network; each host has a set of "neighbors" that are chosen when it joins the network. A host sends its query to other hosts in the network. The fundamental point is that in these unstructured systems, because the P2P network topology is unrelated to the location of data, the set of nodes receiving a particular query is unrelated to the content of the query. A host doesn't have any information about which other hosts may best be able to resolve the query. Thus, these "blind" search probes can not be on average more effective than probing random nodes. To improve system performance, one wants to minimize the number of hosts that have to be probed before the query is resolved. One way to do this is to replicate the data on several hosts. That is, either when the data is originally stored or when it is the subject of a later search, the data can be proactively replicated on other hosts.

5.2. Caching

In the domain of streaming media delivery, Caching is a well known and proven concept [20]. The client who is playing the content can retrieve the cached media content from many different sources. A kind of Metadata has been attached to the content. It can help clients determine if they should retrieve the content from the cached copy or from the original source, refreshing the cache.

At early stage, the P2P caching systems needed to keep redundant copies of data because there was very little metadata about the files. Besides, the cached copy could only be retrieved from a single source. Quickly, people developed the systems that could retrieve data from different cache sources. And these sources could be selected based on hop count, latency, and some other factors. With the maturing of the solutions, more and more caching techniques could be combined with the new features of peer-to-peer such as dynamic peer detection in order to provide a very fault-tolerant system.

Nowadays, the sizes of multimedia have becoming more and more lager, for example the music, the films and cartoons. The large sizes of multimedia objects make the full-object caching strategies impossible. A film can easily consume the whole cache space. For solving this problem, a number of segment-based proxy caching strategies have been proposed [21]. These strategies cache segments of a media object instead of the entire object to reduce network traffic as well as the disk bandwidth requirements of media server. Besides, the client startup delay is reduced if beginning portions of media object are cached.

These caching methods can be classified in two types. The first type focuses on reducing client startup delay by always giving a high priority to cache the beginning segment of media object. Among this type of caching methods prefix caching, has been proposed to segment the media object as a prefix segment and a suffix segment. The proxy caches the prefix segments only and tries to cache as many prefix segments as possible. Prefix caching is effective when clients mostly access initial portion of media objects [21]. More recently, the uniform and exponential segmentation strategies have been developed. The uniform segmentation strategy segments all media objects into fixed-length segments while the exponential segmentation strategies segments media objects with an exponentially increased segment length. This strategy is based on the assumption that later segments of media objects are less likely to be accessed. Thus, the beginning segments are given a high priority for being cached.

The second type of method tries to improve the server traffic reduction. An example of this type of caching methods is found in the adaptive-lazy segmentation [22], in this strategy, each object is fully cached by the aggressive admission policy when it is accessed for the first time. The fully cached object is kept in the cache until it is chosen as an eviction victim by the replacement policy. At which time, the object is segmented using the lazy segmentation method and some segment are evicted by the replacement policy. From then on, the segments of the object are adaptively admitted or adaptively replaced segment by segment.

5.3. Prefetching

Segment-based proxy caching schemes have been effectively used to deliver streaming media objects. However, this approach does not always guarantee the continuous delivery because the to-be-viewed segments may not be cached in the proxy in time. The potential consequence is the playback jitter at the client side due to the proxy delay in fetching these uncached segments, thus this problem is called proxy jitter [23].

The key to solve the proxy jitter problem is to develop prefetching schemes to preload the uncached segments in time. Prefetching refers to media that the user might use in the near future. Thus, future parts of media can be stored at intermediate node in the networks to enable play back. In order to improve the media delivery quality for segment-based caching schemes, [23] proposed two simple and effective prefetching methods, namely, look-ahead window based prefetching and active prefetching to address the problem of proxy jitter. The major action of the look-ahead window based prefetching is to prefetch the succeeding segment if it is not cached when the client starts to access the current one. The window size is thus fixed for the uniform segmentation strategy and is exponentially increasing for the exponential segmentation strategy. The basic idea of the second method, active prefetching, is to preload uncached segments as early as the time when the client starts to access a media object.

Prefetching can either be speculative, where the knowledge about future accesses is not perfect, or informed, where the look-ahead to future accesses is certain. In [24] is investigated speculative prefetching using an analytical model which incorporates retrieval delay. To the best of our knowledge, even though retrieval delay is the main justification for prefetching, it has never been incorporated into previous models for prefetching. Previous studies focus on building access models for the purpose of access prediction and investigate these models empirically. In [24] is assumed the existence of one, such model to provide some knowledge about future access and investigate analytically the performance of a prefetcher that utilises this knowledge. Currently, prefetching is more often used in mobile computing, because resources are severely constrained, performance prediction is as important as access prediction. The growing trend of using multimedia aggravates the resource problem. Due that peers can be either mobile or stationary a prefetching scheme that anticipates the information needs of users could be useful.

IV. Existing media streaming solution over P2P networks

Now, we need to have an overview about several existing solution of live media streaming on P2P networks.

1. PeerCast

PeerCast [14] is a tree-based end-hosts multicast architecture for streaming live media to multiple peers, it focuses on dynamic construction of a multicast tree which connects peers requesting the live media and it is designed to scale to hundreds of short lifetime nodes that participate in long-durational multicast session transmitted over unreliable unicast UDP/RTP.

1.1. Peering layer

PeerCast proposed a peering layer design philosophy. The peering layer runs between the application and transport layers on each client peer (reference figure9.) and hide topology changes from final application. It controls maintenance of streaming tree: joining and leaving of peers. PeerCast use peering layer to separate the Data transfer sessions from Application sessions. Data transfer sessions are composed of the data channel and control channel of a media stream, they are established between the peering layers at two nodes. Application sessions are established between the peering layer and the end application. All communication between application and transport layers passes through the peering layer. The final application (application layer) can specify which media stream to obtain, but it is the peering layer to locate the server which can provide the stream, and establishes a data-transfer session with the server. When a termination of data transfer happened, it is the peering layer to locate a new server and restart the flow of data. Like that, the changes of data transfer session are hided from the application session. The final application does not need to care about the changing of multicast topology, it is to say that, using peering layer, we can mask peer transience form end application.

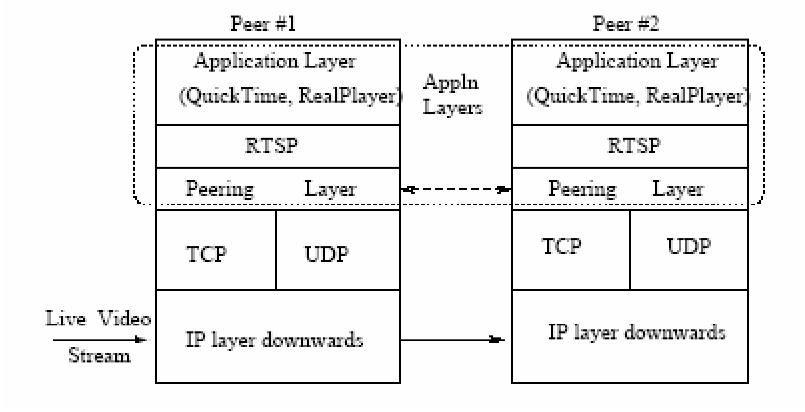


Figure9. Peering Layer

1.2. Redirect primitive

The main primitive supported by peering layer is Redirect. PeerCast use it as a basic means to build topology-maintaining algorithms. Most of the basic operations are finished by the help of redirect primitive, including discovering an unsaturated node to connect when a new node incoming, enabling the recovery process of the connecting of descendant nodes when a father node unsubscribed and handling the failures of nodes. We should know that the redirect messages are produced and used just at the peering layer and the final application know nothing about such message. The process of Redirect is very simple:

Node p sends a redirect message to node c;

Node c closes its data transfer session with p and tries to connect to node t.

Note that the information of target node t is included in the redirect message sent from p to c; when c received the redirect massage, c close the session with p immediately regardless of the situation of data session between node p and node c.

1.3. Basic policies for maintaining topology

Having discussed the mechanism of peering layer and redirect primitive, article [14] give us some basic policies for maintaining topology. We have known that PeerCast uses an overlay spanning tree, and the shape of the tree has great affection on the end-system metrics. We can prove that an almost-complete spanning tree is the optimal overlay tree for packet loss, packet delay, and time to first packet metrics. So, our goal now is to find a policy that can maintain an almost-complete spanning tree. Article [14] gives us several options based on a relatively simple cost function. For Join Policies, we suppose that each node to know only its local topology apart from the source, a saturated node will redirect the requesting node only to its immediate children. Therefore on choosing the redirect target, we will have several options:

1. Random: chooses one of its children at random as the target t.
2. Round-Robin (RR): choose the target according to a list of its children, beginning with the head of the list and then move the used one to the end of the list.

3. Smart-Placement (SP): Each node need maintain the network locations of its children. The client sends traceroute information along-with its request to node n and the node redirects to a child that has least access latency.

For Leave/Recover Policies, we have known that each node can get stream from at least two target nodes: 1. its parent; 2.the source. When a node unsubscribed/failed, all of its descendants need to find a new target node to recover the stream. We have two choices of operation: 1. Each transient descendant itself try to contact a target node; 2. Only some children C(one or several) of unsubscribed node try to contact a target node, the rest of its descendants will become the descendants of C.
Combining these two groups of choices, we have following four possibilities.
1. Root-All (RTA): redirect all descendants of specifying to source.
2. Grandfather-All (GFA): redirect all descendants of specifying to its parent.
3. Root (RT): only redirect C to source.
4. Grandfather (GF): only redirect C to the node's parent.

One point that is worth to be mentioned here is that PeerCast use heart beat message to detect the status of the peers, when a node failed, its parent and descendants will detect this failure after a predefined period, so the policy to recover from failure is similar to the leave policy.

The PeerCast solution can work with short-lived nodes, and allows relatively cheap deployment. Importantly, existing applications at a peer do not need to be changed to use the peering layer, and are thus conveniently enabled for such a stream distribution. In the context of P2P systems, the contributions of Peercast is an architecture for streaming media over a dynamic P2P network that uses a basic peering layer to efficiently enable a connected topology. PeerCast insulates end-applications from router transience, thus enabling good end-system performance that degrades gracefully with increasing numbers and decreasing lifetimes of clients.

2. Zigzag

Like PeerCast, ZigZag [17] is a P2P streaming technique proposed for the problem of streaming live media from one single source to a large numbers of clients on Internet.

The transience of peers is naturally one of the main designing objectives of ZigZag. This solution also takes the control of the tree height and the overhead of controlling information into account for the reason of the scalability of the system with a large number of clients. The main difference between PeerCast and ZigZag is the later one organizes the nodes into a hierarchical cluster structure and introduces a set of rule to confine the multicast tree. The number of hierarchical level and the size of cluster can be controlled and the multicast tree is built based on these constraints of predefined structure and rules. And the key characteristic of ZigZag's design is the use of a foreign head other than the head of a cluster to forward the content to the other members of that cluster.

ZigZag solution includes the description of:
1. The administrative organization that represents the logical relationships among the

nodes;
2. The multicast tree that represents the physical relationships among the nodes;
3. The rules applied to construct administrative organization and multicast tree;
4. The control protocol with which nodes exchange state information;
5. The Join/Leave/Optimization Policies.

2.1. Administrative organization

An administrative organization is a multi-layer hierarchy structure organized into clusters.

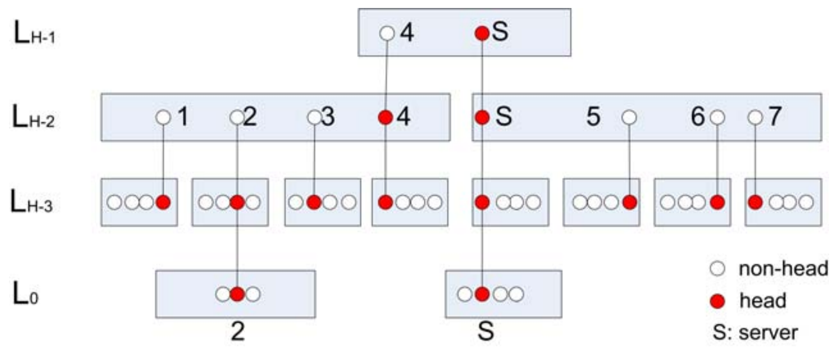


Figure10. Administrative organization of peers

All of the peers are the members of the lowest layer (layer 0) and are divided into clusters of predefined size; Layer H is composed by the "head" peers of layer H-1, a head is a peer selected form one cluster of lower layer (layer H-1). H is the number of layers; The size of cluster is predefined, for example [k, 3k] (k >3 is a constant); And according the proposed structure, we have H = Θ ($\log_k N$). Please reference figure10.

2.2. The predefined rules

In ZigZag, the building of multicast tree and the applying of Join/Leave/Optimization Policies must obey a set of predefined rules, the main contents are:
1. A peer not at its highest layer cannot have any link to or from any other peer.
2. A peer at its highest layer can only link to its foreign subordinates.
3. Non-head members of a cluster must get the content from a foreign head.
(In fact, rule 2 and 3 is the two sides of one rule.)

The terms used in rules:
Subordinate: Non-head peers of a cluster headed by a peer X are called "subordinate" of X.
Foreign head: A non-head (or server) clustermate of a peer X at layer j > 0 is called a "foreign head" of layer (j-1) subordinates of X.
Foreign subordinate: Layer-(j-1) subordinates of X are called "foreign subordinates" of any layer-j clustermate of X.

Foreign cluster: The layer-(j-1) cluster of X is called a "foreign cluster" any layer j clustermate of X.

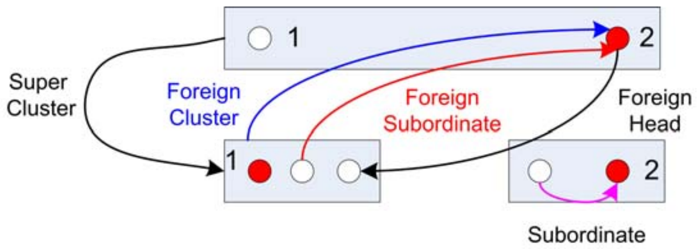


Figure11. Relationships among clusters and peers

With the description of administrative organization and constraints rules, we can already try to present a ZigZag multicast tree, reference the following figure12 and figure13 in next page.

2.3. Control protocol and Join/Leave Policies

For maintaining the shape of multicast tree and the structure of administrative organization, ZigZag use a control protocol to exchange controlling information among nodes. Each node in a cluster must periodically communicate with its clustermates, its children and parent in the multicast tree. The exchanged information include the peer degree, the relation between foreign head and its descendants, the flag Reachable(x) (representing the reachability from x to a layer 0 peer) and the flag Addable(x) (representing the possibility of joint of new client under x), depending the role played by the peers.

Making use of the flags Reachable(x), Addable(x) and cost functions D(Y) and d(Y , P) (D(Y)=end-to-end delay from the server observed by a peer Y; d(Y , P)=the delay from Y to P measured during the contact between Y and P), ZigZag proposed a join algorithm. In fact, the main idea of this algorithm has no essential difference with that proposed by PeerCast: using flag to judge the possibility of reaching and joining and than using a cost function to get a better result against the end system performance.

The differences rest with that:

1. ZigZag's join algorithm can not violate the rules described above, it is to say, the forming of multicast tree is constrained by the administrative organization. In contrast, PeerCast have no this kind of limitation.
2. For the reason of existing of limited size cluster, ZigZag need to apply periodically a split algorithm. It will split evenly an oversize cluster into two clusters. In contrast, PeerCast does not need this kind of operation.

With regard to the Leave/Failure policies, the one proposed by ZigZag is a different one with PeerCast. Here, no matter purposed or accidental departure of a node will be known by its parent peer, all subordinate peers and all children peers thanks to the control protocol. If the leaving node x is a non head peer, its cluster head is responsible to select an x's clustermate to replace x; If x is a head peer, a subordinate of x in layer 0 will be chose to replace x. So, we can find that this leave policy based totally on the structure of administrative organization and is totally different with PeerCast. Besides, ZigZag need a periodical merge procedure of the undersized cluster caused by the departure of nodes to reduce overhead and maintain the balance of tree structure.

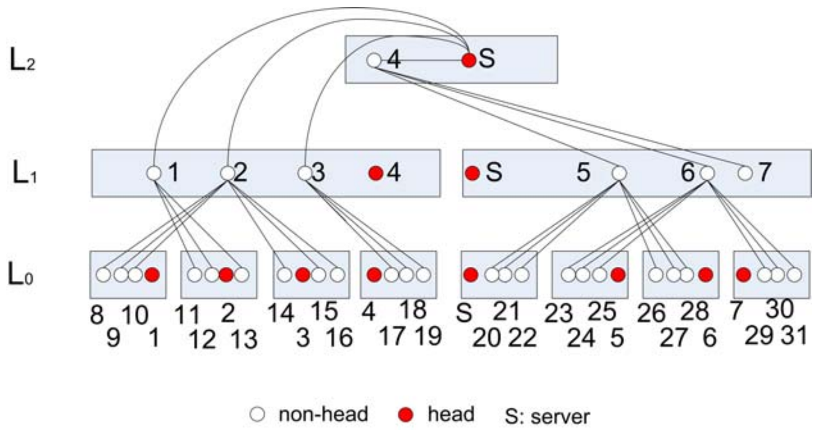


Figure12. The multicast tree atop the administrative organization

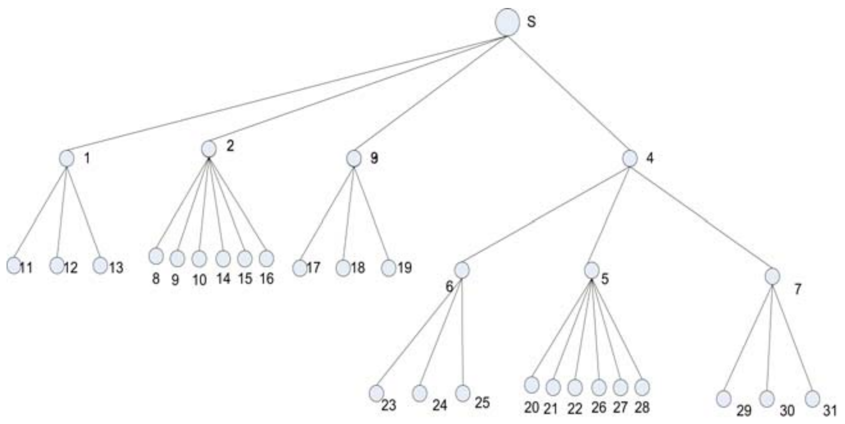


Figure13. The multicast tree in another perspective

2.4. A conclusion for PeerCast and ZigZag

Just like anyone solution, PeerCast and ZigZag have their limitation and large room to improve on. For example:
(1). One big and common limitation of PeerCast and ZigZag is that the peers in the tree can only have one parent, this presupposition heavily limits the utilization of bandwidth of peers and the efficiency of streaming even it simplified the problem. So the multi source problem is an interesting direction to work on.
(2). Both of PeerCast and ZigZag does not give enough attention to the problem of heterogeneity, the algorithm is still not smart enough and affect the efficiency.
(3). To PeerCast, the effective control of height of tree is still a problem.
(4). To PeerCast, the performance when extending to thousands or larger number of clients in Internet is still very limited.
(5). To ZigZag, the structure of administrative organization brought some unbalanced

characteristic, such as some peer can only have one child and some peer supports many children just like in a Star topology.
(6). Both PeerCast and ZigZag are application-level multicast technique and by far, in general practice, the multicast infrastructure in the network is still not widely adopted, so these end system level multicast solution that use end host to route and distribute multicast message have to use only unicast network service.

3. PROMISE

This system is presented in [25] as a P2P media streaming system with key functions of peer lookup, peer-based aggregated streaming, dynamic adaptations to network and peer conditions. PROMISE is based on an application level P2P service called CollectCast [26].

3.1 The major functions of PROMISE

(1). inferring and leveraging the underlying network topology and performance information for the selection of senders;
(2). monitoring the status of peers and connections and reacting to peer/connection failure or degradation with low overhead;
(3). dynamically switching active senders and standby senders. In Promise the dynamic and diversity are reflected in both peers and network connections among peers. A sender may stop contributing to a P2P streaming session at any time, the outbound bandwidth contributed by a sender may change, the connection between a sender and the receiver may exhibit different end-to-end bandwidth, loss, and failure rate, the underlying network topology determines that the connections between the senders and the receiver are not independent of each other, with respect to their loss and failure rate.

3.2. PROMISE architecture

The architecture (Figure14) consists of peers interconnected through a P2P substrate. The P2P substrate maintains connectivity among peers, manages peer membership, and performs object lookup. Operations are independent of the underlying P2P substrate. Therefore, PROMISE can be deployed on top of P2P substrates such as Pastry, Chord and CAN. In PROMISE prototype Pastry is used and modified. PROMISE assumes that peers exhibit heterogeneous characteristics and they do not have server-like capability: They contributed limited capacity, and may fail or reduce their sending rates unexpected. Each peer p is associated with two parameters: offered rat (R_p) and available (A_p). The offered rate is the maximum sending rate that a peer can contribute to the system. A lower bound on the offered rate (R_{pmin}) is imposed by the system to limit the maximum number of peers required to serve a request. This limited the number of concurrent connections that the requesting peer needs to maintain. The availability is the fraction of time a peer is available for serving. The available of peer p is represented as a binary random variable Ap, 1.0 indicate “available for streaming” and 0.0 otherwise. The rate and availability information is collected by a daemon running in each participating peer. The rate could be a parameter set by the user during initial set up. The availability can either be set by the user or measured by collecting statistics during the regular operation of the peer. Due PROMISE is based on CollectCast and it chooses the sending peers and orchestrates them in order to yield the best quality for the receiver. CollectCast encompasses a number of techniques for selecting the best sending peers, inferring and monitoring the characteristics of the underlying network, assigning streaming rates and data segments to the sending peers, and deciding when a change of the sending is need.

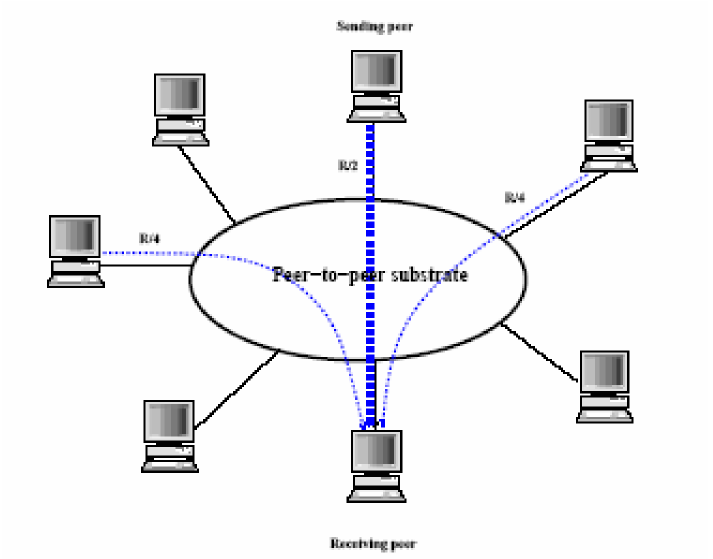


Figure14. PROMISE architecture

3.3. PROMISE operation

In PROMISE a peer requesting a movie runs the receiver procedure and the procedure first issues a lookup request to the underlying P2P substrate, which will return a set of candidate peers (10 to 20 peers) who have the movie. The protocol then constructs and annotates the topology connecting the candidate peers with the receiver. The selection algorithm determines the active sender set using the annotated topology; the active set is the best subset of peers that is likely to yield the best quality for this streaming session. (The rest of candidate peers are kept in a standby sender set). Once the active set is determined, the receiver establishes parallel connections with all peers in the active set. The receiver assigns a sending rate to each of the active senders based on the sender's offered rate and the goodness of the path from that sender to the receiver. The streaming session continues as far as there is not need to switch to a different active sending set. A switch is needed if a peer fails or the network path becomes congested. At which time, the topology is update with new values measured passively during streaming and a new active set is selected. Upon receiving a control packet, the sender determines the rate and the subset of data it is supposed to send. The sender keeps sending till a new control packet arrives with new assignment.

3.4. Selection of the best peers

Search return a set of candidate peers from which the receiving peer chooses the active set to start streaming the movie. In PROMISE tree selecting techniques are possible: random, end-to-end and topology-aware. The random technique randomly chooses a number of peers that can fulfill the aggregate rate requirement, even though these peers may have low availability and share a congested path [19]. The end-to-end technique is based on the quality of the individual paths and on the availability of each peer, the technique chooses the active set and estimates the "goodness" of the path from each candidate peer to the receiver.

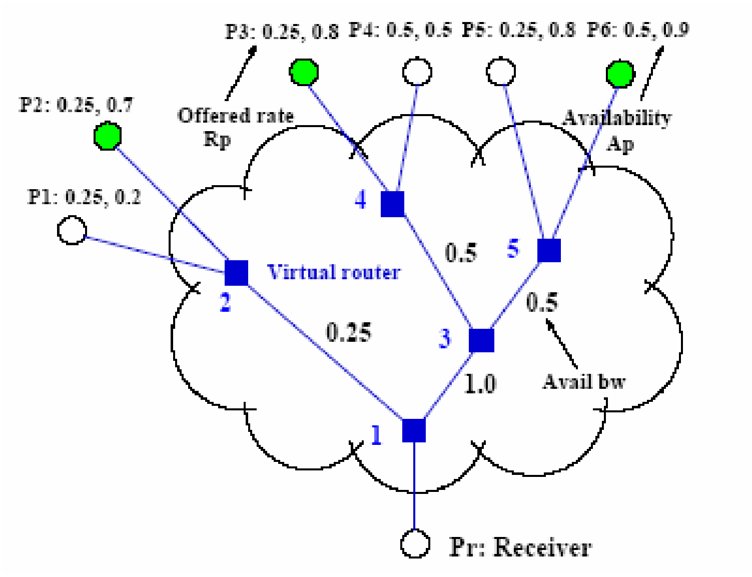


Figure15. Topology-aware selection

The end-to-end technique does not consider shared segments among paths. The topology-aware technique infers the underlying topology and its characteristics and considers the goodness of each segment of the path. Thus, it can make a judicious selection by avoiding peers whose paths are sharing a tight segment (Figure15). First is defined the goodness topology and how it is annotated by network performance metrics and peers characteristics. Then the goodness topology is used to estimate the peer goodness for the session being established. Goodness topology T is a directed graph that interconnects the candidate peers and the receiving peer. Each segment $i \rightarrow j \in T$ is annotated with a goodness random variable $g_{i \rightarrow j}$. Each leaf node represents a peer p from the set of candidate peers P and has two attributes: a fixed offered rate R_p and a random variable A_p that describes the availability of p for streaming.

4. CoopNet

CoopNet, described in [16] is an approach to content distribution that combines aspects of infrastructure-based and peer-to-peer content distribution; it is focused on distributing streaming media content, both live and on-demand, in order to replace the traditional client-server framework. Specifically, Coop-Net considers the problem that arises when the server is overwhelmed by the volume of requests from its clients. For instance, a news site may be overwhelmed because of a large "flash crowd" caused by an event of widespread interest, such as a sports event or an earthquake. A home computer might be overwhelmed even by a small number of clients because of its limited network bandwidth.

In fact, the large volume of data and the relatively high bandwidth requirement associated with streaming media content increases the likelihood of the server being overwhelmed in general. Server overload can cause significant degradation in the quality of the streaming media content received by clients.

CoopNet addresses this problem by having clients cooperate with each other to distribute content, thereby alleviating the load on the server. In the case of on-demand content, clients cache audio/video clips that they viewed in the recent past. During a period of overload, the server redirects new clients to other clients that had downloaded the content previously. In the case of live streaming, the clients form a distribution tree rooted at the server. Clients that receive streaming content from the server in turn stream it out to one or more of their peers. The key distinction between CoopNet and pure P2P systems like Gnutella is that CoopNet complements rather than replaces the client-server framework of the Web. There is still a server that hosts content and (directly) serves it to clients. CoopNet is only invoked when the server is unable to handle the load imposed by clients. The presence of a central server simplifies the task of locating content. In contrast, search for content in a pure P2P system entails an often more expensive distributed search. Individual clients may only participate in CoopNet for a short period of time, say just a few minutes, which is in contrast to the much longer participation times reported for systems such as Napster and Gnutella [19]. For instance, in the case of live streaming, a client may tune in for a few minutes during which time it may be willing to help distribute the content. Once the client tunes out, it may no longer be willing to participate in CoopNet. This calls for a content distribution mechanism that is robust against interruptions caused by the frequent joining and leaving of individual peers. To address this problem, CoopNet employs multiple description coding (MDC). The streaming media content, whether live or on-demand, is divided into multiple sub-streams using MDC and each sub-stream is delivered to the requesting client via a different peer. This improves robustness and also helps balance load amongst peers.

A distribution tree rooted at the server is formed, with clients as its members. Each node in the tree transmits the received stream to each of its children using unicast. The out degree of each node is constrained by the available outgoing bandwidth at the node. In general, the degree of the root node (i.e., the server) is likely to be much larger than that of the other nodes because the server is likely to have a much higher bandwidth than the individual client nodes. One issue is that the peers in CoopNet are far from being dedicated servers. Their ability and willingness to participate in CoopNet may fluctuate with time. For instance, a client's participation may terminate when the user tunes out of the live stream. In fact, even while the user is tuned in to the live stream, CoopNet-related activity on his/her machine may be scaled down or stopped immediately when the user initiates other, unrelated network communication. Machines can also crash or become disconnected from the network. With a single distribution tree, the departure or reduced availability of a node has a severe impact on its descendants. The descendants may receive no stream at all until the tree has been repaired. This is especially problematic because node arrivals and departures may be quite frequent in flash crowd situations. To reduce the disruption caused by node departures, we advocate having multiple distribution trees spanning a given set of nodes and transmitting a different MDC description down each tree. This would diminish the chances of a node losing the entire stream (even temporarily) because of the departure of another node. The distribution trees need to be constantly maintained as new clients join and existing ones leave. A centralized approach is advocated to tree management, which exploits the availability of a resourceful server node, coupled with client cooperation, to greatly simplify the problem.

For the evaluation of potential in CoopNet in the case of on-demand streaming the goals are study the effects of client cooperation on load reduction at the server and additional load incurred by cooperating peers [16]. The cooperation protocol used in simulations is based on server redirection. The server maintains a fixed size list of IP addresses of CoopNet clients that have recently contacted it. To get content, a client initially sends a request to the server. If the client is willing to cooperate, the server redirects the request by returning a short list of IP addresses of other CoopNet clients who have recently requested the same file. In turn, the client contacts these other CoopNet peers and arranges to retrieve the content directly from them. Each peer may have a different portion of the file, so it may be necessary to contact multiple peers for content. In order to select a peer (or a set of peers when using distributed streaming) to download content from, peers run a greedy algorithm that picks out the peer(s) with the longest portion of the file from the list returned by the server. If a client cannot retrieve content through any peer, it retrieves the entire content from the server. Note that the server only provides the means for discovering other CoopNet peers. Peers independently decide who they cooperate with.

In CoopNet a receiver can receive all the descriptions in the best case. A peer failure only causes its descendant peers to lose a few descriptions. The orphaned are still able to continue their service without burdening the source. However, this is done with a quality reduction. Furthermore, CoopNet puts a heavy control overhead on the source since the source must maintain full knowledge of all distribution trees.

5. Other architectures

There are others architectures proposed such as MAPS [27] and SplitStream [28]. In MAPS is proposed systems architecture extends existing P2P infrastructures with several modules in order to achieve improved media service over heterogeneous peer networks. Modules are collated into the Media Accelerating Peer Service (MAPS), thus the system’s software layers are, from bottom to top: the traditional underlying operating system and related services, the P2P service layer, the MAPS media support modules; and the application layer. The P2P service layer provides basic functionality such as peer discovery, user and group management, name resolution, and delivery primitives. MAPS extends this layer with a plug-in architecture for customizing basic search and delivery operations. MAPS provides two extensions: a module enabling real-time media streaming (via RTP over UDP) and a module for enhanced search capabilities (via distributed SQL). The MAPS media support modules use the P2P service layer for network and resource access and provide transformation and computation services on data obtained from or sent to the P2P service layer.

Other architecture called SplitStream concerned with application-level multicast in cooperative environments. In such environments the participants contribute resources in exchange for using the service and they expect that the forwarding load be shared among all participants. SplitStream enables efficient cooperative distribution of high-bandwidth content, whilst distributing the forwarding load among the participating nodes and it can also accommodate nodes with different network capacities and asymmetric bandwidth on the inbound and outbound network paths. Subject to these constraints, it balances the forwarding load across all the nodes. In SplitStream the key idea is to split the multicast content into kstripes, and multicast each stripe in a separate multicast tree. Participants join as many trees as there are stripes they wish to receive. The aim is to construct this forest of multicast trees such that an interior node in one tree is a leaf node in all the remaining trees. In this way, the forwarding load can be spread across all participating nodes. It is possible, for instance, to efficiently construct a forest in which the inbound and outbound bandwidth requirements of each node are the same, while maintaining low delay and link stress across the system. The approach also offers improved robustness to node failure and sudden node departures. Since ideally, any given node is an interior node in only one tree, its failure can cause the temporary loss of at most one of the stripes. With appropriate data encodings such as erasure coding of bulk data or multiple description coding (MDC) of streaming media, applications can thus mask or mitigate the effects of node failures even while the affected tree is being repaired.

In [29, 30, 31, 32 and 33] are presented others more recent advances about media distribution on peer-to-peer, such as PALS, where the main contribution of it is a receiver-driven coordination and adaptation framework for streaming from multiple, congestion-controlled, sender peers that is able to cope with unpredictable variations in throughput from each sender peer, as well as with dynamics of peer participation. PALS allows a receiver to orchestrate the adaptive delivery of stored layer encoded streams from multiple sender peers to a single receiver. Given a set of sender peers, PALS progressively evaluates various combinations of senders to determine a subset of the senders that can collectively provide maximum throughput. Once such a subset of senders is selected, the receiver monitors the overall throughput and periodically determines what is the target overall quality (i.e., the total number of coding layers) that can be delivered from all senders. Then, the receiver determines a proper distribution of the overall throughput among active layers (i.e., what portion of the overall throughput should be allocated for delivery of each layer), and finally divides allocated bandwidth to each layer among active senders (i.e., which segments of each layer should be delivered by each sender) in order to effectively cope with any sudden change in throughput of individual senders.

V. Our solution

Now, we will propose a P2P live media streaming solution.

1. Background and Related Research

Streaming media will constitute a significant fraction of the Internet traffic in the near future. Since multimedia streaming requires high bandwidth, server bandwidth will run out rapidly if client/server model is used. A peer-to-peer model is ideal to serve the server link bottleneck problem. Recent years, many researches have been done on investigating the applicability of P2P to the problem of streaming live media. And as a result, there are many solutions have been proposed, such as PeerCast, ZigZag (one to many), PROMISE (many to one), CoopNet etc. We have a simple description of these solutions in section IV.

All of these solutions have common ground:
(1). The purpose of these solutions is to construct a tree-based application-level multicast architecture that can return a good end user performance.
(2). The whole delivery tree is built rooted at one source;
(3). Only a subset of receivers can get the content directly from the source, the most part of the others would get the multimedia content from the receivers in its upstream;
(4). The core of these solutions is the algorithm that can be used to construct and maintain the multicast tree and manage the behavior of the nodes in the trees;

Some of these architectures have different assumption or precondition; these differences come from the different complexities and the scale of problem that the solution tries to focus on. For example, Zigzag is a "one to many" solution that means each node in Zigzag's multicast tree would have only one parent (the coming source of the multimedia contents) and PROMISE is a "many to one" solution that permit the nodes in the tree can have more than one parents; PeerCast contributes itself to solve the problem of transience of nodes but SplitSteam pay more attention to the heterogeneity of nodes and the problem of multisource; PeerCast can only scale to hundreds of nodes but Zigzag can easily extend to thousands of nodes.

So, if we try to propose a new P2P live streaming architecture, firstly we must take the common problems into account, the second step is to find a new algorithm to maintain the delivery tree and finally we must prove that the new schema is worth to be developed through the results of experiments or simulations.

In this paper, we want to propose a new join/leave algorithm that can manage the joining or leaving behavior of nodes according to its geographical characteristic, and though this algorithm, to maintain a relatively balanced multicast tree that can give us a satisfied end user performance. In fact, our architecture is a kind of PeerCast like solution, some basic come from the PeerCast solution. Because from article [14], we can find that the PeerCast solution is just a frame of structure instead of a detailed design like ZigZag or Splitstream. Its shortcoming is obvious. For example, in the join policies, it is very hard to build a enough balanced and efficient multicast tree using Random and RR because the choosing of redirect target is not based on any functional factor but a list order; in the leave policies, RTA and GFA redirect all of the nodes affected by the leaving of certain node, make the overhead of recovery is hard to control. Like the result of simulation, only SP and GF/RT is recommended. Other disadvantages include the insufficient of control of height of tree, adaptation to the heterogeneity of peers, insufficient control to overhead, insufficient of optimization algorithm, etc. At the same time, this basic frame have its advantages, based on the proposed basic concepts (the idea of peering layer; basic primitive redirect; basic Join/Leave policies) we can have enough space to improve on by developing new algorithm or adding new characteristics.

Therefore, based on the basic concepts of PeerCast, we will try to propose a new algorithm to maintain the delivery tree. We will have the details in the following chapter.

2. Basic assumptions and preconditions

(1). The delivery tree is a single-source media streaming tree**.** The source would contain the entire contents of the multimedia object. One source does not mean one machine; maybe there is more than one machine simultaneously supplying the multimedia content, but they will have the same address that is used by the clients to find the server. From here, we can find that, in fact, our live streaming media P2P network is hybrid P2P architecture. This architecture can keep a balance between the accuracy of the centralized architecture and the lower load of the distributed architecture and therefore it is ideal to deal with the live streaming media problem.

(2). Except the source (we suppose that the source is stable and has a relative high out bandwidth), the behavior of all of the nodes in the delivery tree is unpredictable; they are dynamic, autonomous, short-lived. Every one of them is free to join and leave the service at any time. When a node leaves it will abandon all of its descendant peers, this will make the descendants leave the streaming tree too. Therefore the whole tree is totally dynamic, with the joining and leaving of the peers, it changes continuously.

(3). Here we would only deal with the “one to many” situation that means each node in the delivery tree would have only one parent and several children. In practice, especially to the normal public internet user, the incoming bandwidth is often bigger than out coming bandwidth, the well known example is ISDN or ADSL. It means that the “one to many” model is not the optimal solution when considering the end user’s bandwidth. It will be more ideal and efficient if a node can download media content from multiple sources and at the same time take into account the heterogeneity of each node and not just regulate a uniform maximum number of children for each node but adjust this number dynamically. But apparently, this will make the problem more complicated, like SplitStream. For simplicity and easy to realize, we suppose that each node has only one parent and Cmax children. (Cmax is a parameter that we predefined.)

Note, in this paper, the word “father”, “parent” and “ancestor” have the same meaning. It represents the nodes that provide media streaming to their downriver nodes; the word “children”, “offspring” and “descendant” has the same meaning. It represents the nodes that receive media streaming form their upriver nodes.\

(4). Under the given network environment (LAN or WAN, here we suppose our solution will be applied to Internet), we suppose that all of the potential clients (who possibly have interest in the streaming media) are:

Very well geographically decentralized (distributed);
Every potential client has the same probability to join in the multicast tree;
Every client that has already joined into the tree has the same probability to leave the multicast tree, including the normal leaving (before it's departure, the node would inform its parent and children) and abnormal leaving, namely failure of node(it's departure is suddenly happened and neither it's parent nor it's children would be informed it's departure).

3. Our purpose

We want to propose a live media streaming solution over P2P systems that can scale to hundreds or thousands of autonomous, short-lived nodes under the circumstance of Internet. This architecture will organize the multicast tree according to the geographical characteristic of nodes. Through the join/leave/adjusting algorithm, our solution will maintain a balanced streaming tree and gain a good end user performance.

4. Our solution for basic problems

Before giving the detail of tree-maintaining algorithm, we must have a discussion on some of the basic relative problems**.**

4.1. The problem of Transience.

The primary challenge in a P2P end-host multicast system is the transience of peers (routing-elements). In practice, the behavior of peers is unpredictable: they are free to join and leave the tree at any time. And what make the situation worse is that the role of router in IP-Multicast is replaced by end-host (now, the routing-elements is end host) with the migration of multicast functionality from network-layer to the application-layer. Compared with IP-Multicast under which the Router failures are not so frequent, the unpredictable leaving of a peer in a multicast tree of an end host system will affect its entire descendant peers. Therefore, we need a mechanism to insulate end-applications from peer (router) transience in order to return a good performance.

Here, we have to measure to respond this problem:
a. Hide the changes of the topology of streaming tree from the final application at each client. When we want to carry out a design schema into practice, for example, developing software to realize the function on the client machine, it is not very difficult to separate the Data transfer sessions from Application sessions. A Data transfer sessions are composed of the data channel (media content) and control channel (control information) of a media stream, they are established between two nodes. Application sessions run at each node itself. The final application (application layer) can specify which media stream to obtain, but it does not need to know how to locate the server which can provide the stream, and how to establishes a data-transfer session with the server. When a termination of data transfer happened, it is not the final application but the other functional part of the software running on transport layer to locate a new server and restart the flow of data. Like that, the changes of data transfer session are hided from the application session. The final

application does not need to care about the changing of multicast topology, it is to say that, using this can of designing philosophy, we can mask peer transience form end application.

b. A recovery mechanism. When a node leaves, consequently, all its descendants are left out of the tree. And at this moment, the most important thing is to make the descendants continue receiving the feed as quick as possible, and guarantee that there is a minimal effect on the QoS perceived by its descendants. For achieving this goal, we must have an effective rejoin algorithm; we discuss recovery mechanisms in the following chapter.

4.2. The end-to-end delay

Compared with the underlying network traffic, the end-to-end delay is mostly due to local delays at intermediate clients due to queuing and processing. The local delay at such an intermediate client is mostly affected by its bandwidth contention. But we can not control the bandwidth and the only thing we can do is to adapt it using the proposed algorithm.

Our solution can keep small end-to-end delay small thanks to the geographical feature of the join algorithm. Besides, the number of hops, namely, the height of the tree will be the main factor that affects the end to end delay. We will have a discussion in the section of optimizing algorithm.

4.3. The clients how to locate the media source server?

All of the client who wants to get the streaming media should know where to find the media content server. It is easy to realize: the sources sever only needs to have an address---- IP address or URL. Here we have a actual instance, on the yellow page of PeerCast organization sit [40], we can find about one hundred channel of radio station, in this page each channel very provide a link address with that we can be directly connected to the radio station server. For example, if you want to hear the Japan-A-Radio pop music station, the link address will be:
peercast://pls/E8E52DF85616E9F6337EEB6E7191CAE1?tip=68.5.0.195:7144. The format of this address is determined by the software that you want to use. We can find that in this address there is the IP address. Using this IP address, the clients would be able to send the request of joining to the server. Of course, before you want to enjoy the music you must install the software of PeerCast.

4.4. Control of overhead

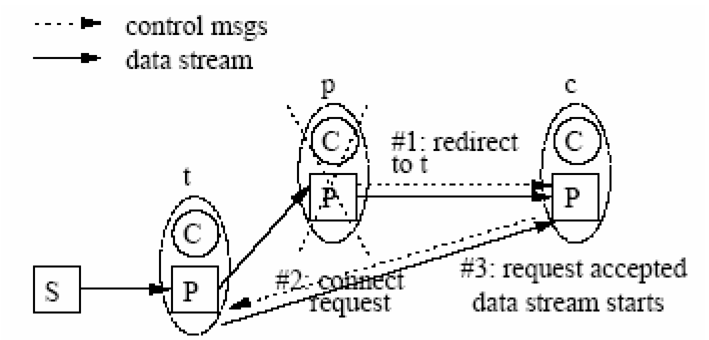
There are two kind of overhead for each node:
(1). The load of local resources in every node. It includes the processing power, memory, or bandwidth capacity. But in the P2P network, each peer is independent unit and it knows its own capacity, besides, the load of running a P2P live media streaming software like PeerCast is very low to current PC computer, it is so low as to be ignored. As for the bandwidth capacity, each node can adjust the maximum limit on the number of children according to it.

So, there is no the problem of overload.

(2). The control overhead. In the streaming session, each node needs to communicate and change information with the other nodes. Every operation during the streaming session will contribute to the control overhead, for instance, the join operation needs to find a proper node to connect and that will include tens of even hundreds time of sending request and receiving information; the leave operation will include a recovery action of its descendants and it means a rejoin procedure that contribute overhead, besides that the leave node itself will send message to inform its parent and children that it would leave, this also increases the overhead. It is easy to understand that the larger the number of nodes to witch an operation needs to consult is the higher the control over head is. But for efficient use of network resources and due to the resource limitation at each node, the control overhead at each node should be kept low. This is important to the scalability of a system with a large number of receivers. In next section, the part of mechanism of administration, we will discuss the overhead problem in our solution.

4.5. The Redirect Primitive

From above paragraph we have known how to locate a media server. At the begging of the construction of the streaming tree if the serve is unsaturated it is easy to solve, we only need to connect the client directly to the server. But our tree is not a star topology; the maximum number of children of server is very limited, at most hundred nods. Therefore in the situation server is already full the new comer must know how to find the next destination to request the streaming media. In our solution, we will use "redirect" mechanism, the same thing just like in PeerCast. As shown in Figure 16, a redirect message is sent by a peer p to another peer c which is either opening a data-transfer session with p, or has a session already open. The message specifies a target peer t. On receipt of the redirect message, c closes its data-transfer session with p, and tries to establish a data-transfer session with t, for the same stream URL. Note that such redirect messages are produced and used at the peering layer, and the application above is unaware of such messages. Thus, the application session persists despite changes in the data-transfer sessions. The redirect primitive proves to be an effective, efficient and economic (Its needs very low communication overhead) mechanism used in algorithm of tree maintaining.



Figur16. Redirect primitive

5. Algorithm of maintaining the media streaming tree

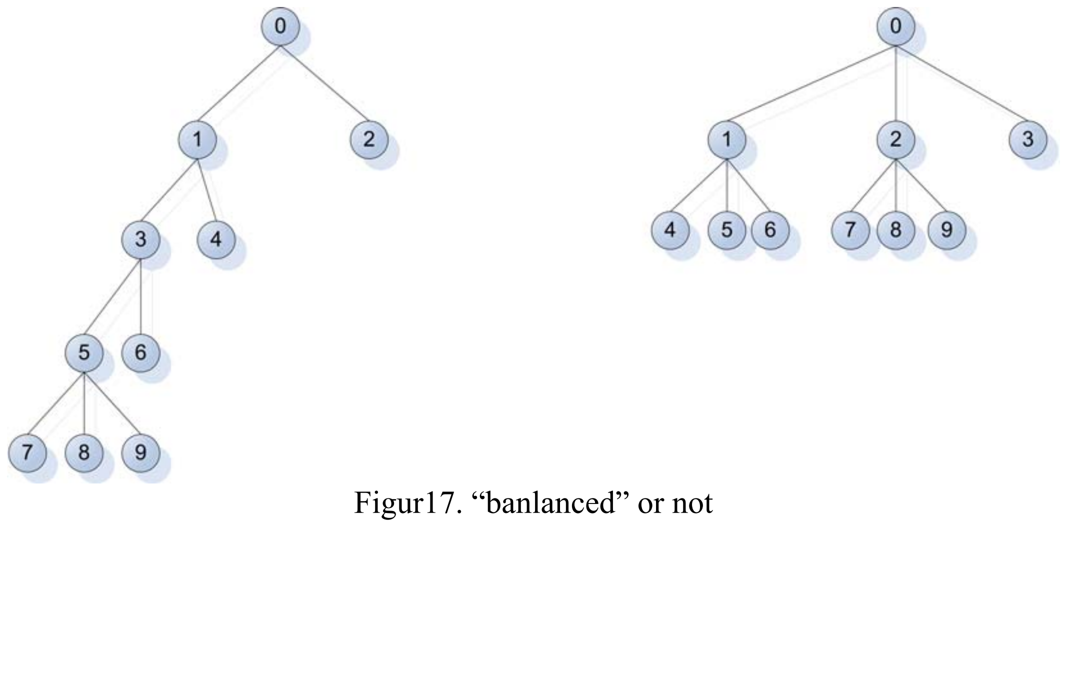
Like we have mentioned above, our purpose is to propose a media streaming tree that can provide us an acceptable end user performance, this performance is related to parameters like packet loss, end to end delay, control of overhead, etc. And these parameters will be affected by the shape of the delivery tree that is the direct result of the topology-maintaining algorithm. Without doubt, the join/leave/optimizing algorithms that control the client joining, normal leaving, failure of the nodes is the core of our solution of P2P live media streaming network.

Among these algorithms, the most important one is the join algorithm, because the leave algorithm is based on the join algorithm. After a node's leaving, we need a quick rejoining of its descendants, the rejoining action apply the join algorithm. And if the optimizing algorithm wants to adjust the position of node, it also needs the join/leave algorithm. In our solution, we will propose a geographic characteristic join algorithm that is efficient and effective.

5.1 Join algorithm

What we finally concerned is end user performance. Normally, the end-system performance is characterized by packet loss, packet delay, and time to first packet. In article [14], it has been proved that under homogeneous unicast edge and node characteristics, an almost-complete spanning tree is the optimal overlay tree for packet loss, packet delay, and time to first packet metrics. But we have said that we want to apply our solution on Internet. Under the circumstance of Internet, the nodes that will have interest in the media streaming would be so distributed and so different in local conditions and network condition, it is to say, there is no a homogeneous environment. An almost-complete spanning tree would not be enough to provide us satisfied end user performance. So, here, when applying to Internet, the tree that we want to construct should be a "balanced" one.

5.1.1. So called "balanced" tree



Figur17. "banlanced" or not

Reference figure17, can we say the tree in the right is a more “balanced” one compared with the left tree? Absolutely not, the so called “balanced” is not pointed to the shape of the spanning tree, the shape of the tree is only a logical topology that represent the relation between the nodes, the real physical location of the nodes have been concealed. The criterion to weight a tree is balanced or not, is not the shape of the tree, but the final performance at end host. The criterion that we would use to construct the tree is the “network condition”. Just like in above figure, if there is a 100Mbps connection among nodes 1, 3, 5,7,8,9 in left figure and it has only a 56kbps connection among nodes 2,7,8,9 in right figure; there is a bigger possibility that node 9 would get a better end user performance. And of course, the depth of the tree will heavily affect the performance, it will result in more packet loss, packet delay, here we just simply suppose that the depth of tree in this discussion will be kept in a acceptable level.

But the question still continues, under the circumstance of Internet, how can we know the “network condition” between nodes and then decide where to connect them? These peers that we will treat with are so distributed and autonomous, can we simply say the network condition between two nodes is direct ratio with the geographical distance? In Internet, perhaps we can, the larger is the physical distance between two nodes, the worse the network condition between them are, although that is not always exact.

Our way to solve this problem is to use time parameter to represent the distance parameter, and then indirectly representing the network condition parameter. The simplest one is the command “ping”. The “ping” will send a ICMP ECHO_REQUEST packets to network hosts and it can calculate round-trip times and packet loss statistics. From these returned value result we can get a simple conclusion about the network condition. We can choose other more complicated method to judge the network condition more exactly, but here, for simplicity, we will only choose the command “ping”.

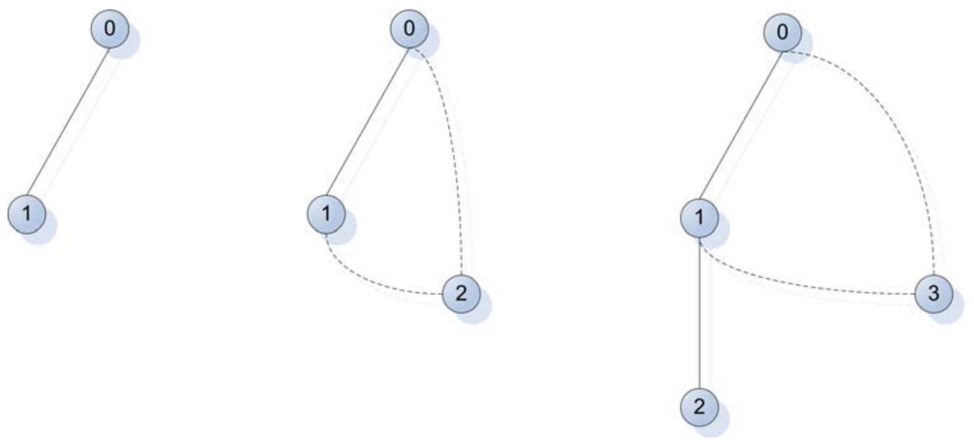
After knowing using what kind of parameter to judge the network condition, we would decide how to construct the tree using the parameter.

The ideal situation is:
when a new client comes, it will “ping” all of the existing nodes in the spanning tree, return the “round-trip times” for each node; and then we choose the node that have the least round-trip times to connect; if this node is unsaturated, it is ok, just connect to it; if this node has already the maximum number of children, we choose the node that take the second place, it is to say, we will sorting all of the existing nodes according the value of “round-trip times” from the incoming node. Obviously, by that, we can construct a almost balanced tree. But, it is impossible to adopt this kind of method. When the spanning tree is relatively large, for example, hundreds of nodes or even thousands of nodes, the overhead of communication will be enormous; the value of time to first packet will be unacceptable. So, this method can be descripted but it is impossible to realize it in practice.

But we can improve it. The method “sorting” can still be used. And this time, we introduce a concept of “nodelist” that used to choose the proper node to connect.

5.1.2. Presentation of our algorithm

Next, for better understanding the processes of join and the nodes how to maintain the "nodelist" for themselves, we will use figure and text together to explain our join algorithm. For simplicity, we make the maximum number of children of nods equal 3, including the source server. Besides, suppose we have two node x and y, we use distance[x][y] represent the "round-trip times" from x to y(x "ping" y). We can understand it just like physical distance, but in fact, it represents the network condition.



Figure[1] Figure[2] Figure[3]

Figur18. Join algorithm

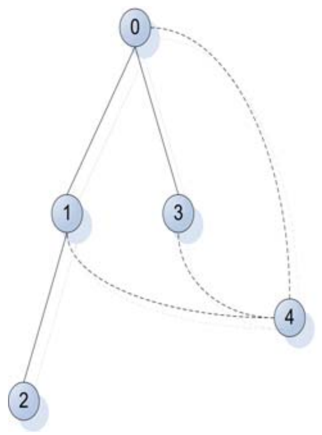
The node joining process begin, new nodes come and would be added to source server node 0. In figure [1], Node 1 would directly connect to server node 0, because there is available place, the server is not saturated, the nodelist maintained by node 1 would be: {0}.

In figure [2], Node 2 comes, sends a joining request to server node 0, it would be able to get the returned information that server node 0 have child node 1; so 2 would get a nodelist: {0,1}; compare distance[2][0] and distance[2][1]; if node 2 is closer to node 0,nodelist would be {0,1}; if node 2 is closer to node 1,the nodelist would become {1,0}; So there is a process of sorting distance order.

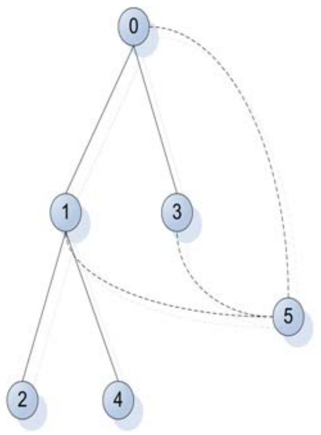
Now, pay attention to the first node number in the nodelist and judge that if there is available place to connect the new comer node2; Here, either node 0 or node 1 can accept node2, we suppose distance[2][1] is smaller, the final nodelist for node 2 would be {1,0}; node 2 would connect to node 1.

In figure [3], node 3 comes and it would send request to server node 0, then get it childe nodes list, one child node 1; so node 3 get a nodelist: {0,1}; if node 3 is closer to node 0,nodelist would become {0,1}; if node 3 is closer to node 1,nodelist would become {1,0}. Here, we suppose distance [3][0] is smaller, the final nodelist for node 3 would be {0,1}; node 3 would connect to node 0.

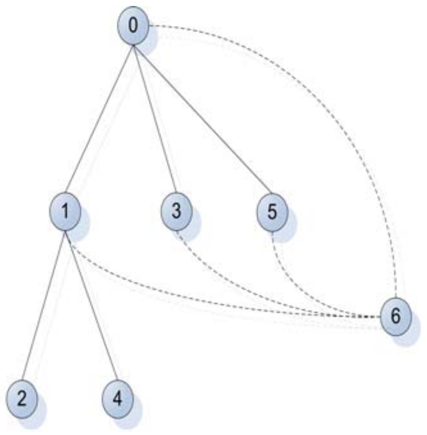
(Note: node 3 does not need to ask node 2 because now the node1 and node0 have enough places available for node 3 ;)



Figure[4]



Figure[5]



Figure[6]

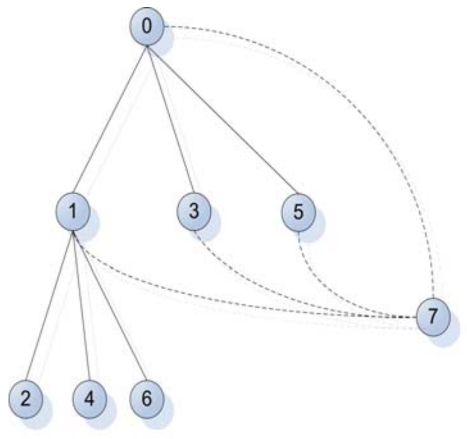
Figur18. Join algorithm

In figure [4], node 4 comes and it would send request to server node 0, then get it childe nodes list, node 1 and node 3; so node 4 would get a nodelist: {0,1,3}; the next step is sorting the nodelist according to comparing result among distance[4][0], distance[4][1], and distance[4][3]. Suppose finally we have the nodelist for node 4 would be {1, 3, 0}, it is to say node1 is closest; Now, judge that if node 1 have available place, here it has, add node 4 into node1;

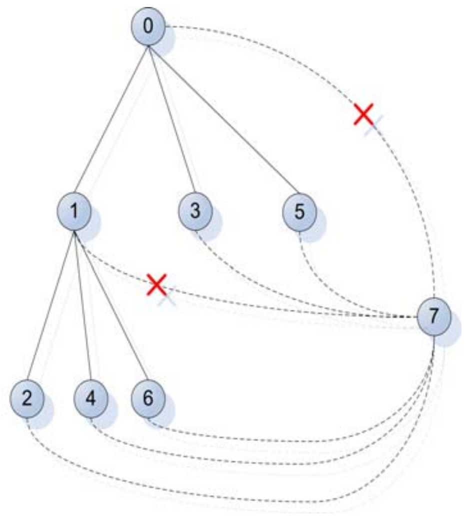
In figure [5], node 5 comes, repeat the requesting and sorting process, Suppose finally we have the nodelist for node 5 would be {0, 1, 3}, add node 5 into node 0;

In figure [6], repeat the same process and we suppose after sorting we get the nodelist for node 6 is: {0, 1, 3, 5}; Now, try to connect node 6 to node 0, but node have reached its maximum number or children, under this kind of situation, we must have adjusting process of nodelist: if the first node in the nodelist is saturated, we would reference the rest of node number, if all of children nodes of the first node, here is node 0, are already listed in the nodelist, we must remove the first saturated node number from the nodelist. Here, node 0 has children node 1, 3, 5 and all of them is in the nodelist, so we must remove node 0 from the list. Like that, the **final nodelist for node 6** would be **{1, 5, 3}**; Node 6 would be connected to node 1.

(Note: the step of adjusting the nodelist and removing the saturated node number is just for the purpose to get a reasonable nodelist, and in next part we will have the detail; and in this part, at the begging of joining process, the server node 0 is a little special, compared with the normal peer in the tee.)



Figure[7]



Figure[8]

Figur18. Join algorithm

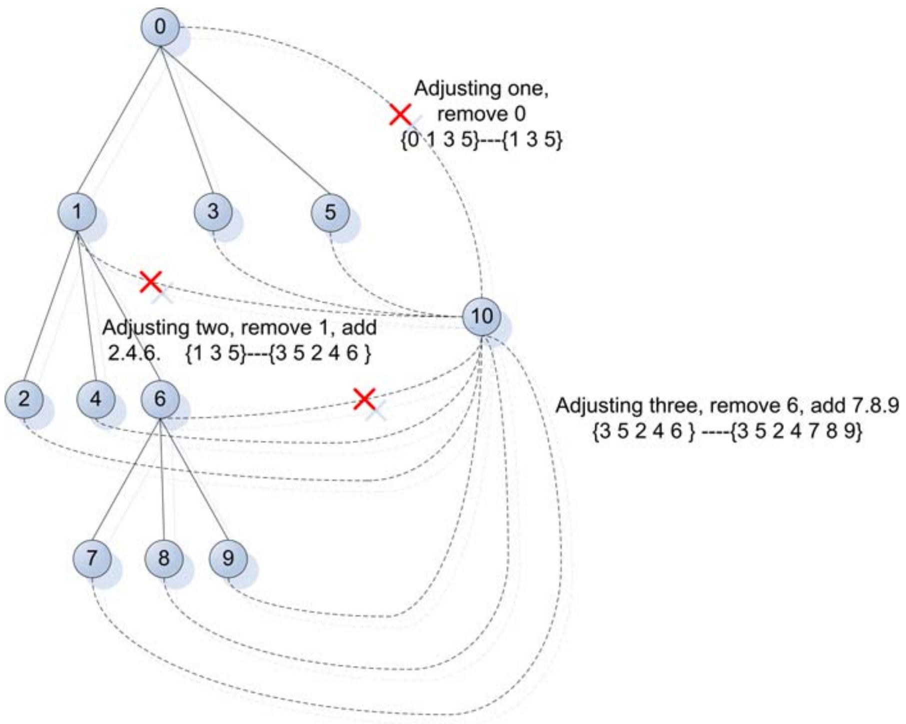
In figure [7], new client 7 comes; it would get the original nodelist: {0,1,3,5}; if sorting result is {3,x,x,x},added node7 to node 3; if sorting result is {5,x,x,x}, added node 7 to node 5;

If sorting result is {0, x, x, x}; like in figure [6], we have a process of adjusting of nodelist, node 0 is removed from the nodelist. The rest nodes in list include 1, 5 and 3. we suppose that we have **the result as {1,x,x}**({1,5,3} or {1,3,5}, the other node number except the first one is not important).

Now, judge that if node 1have available places, here no. so **adjusting** the nodelist, put all of children of node 1 (node 2, 4, 6) into the nodelist, at the same time remove node 1 from nodelist; the new nodelist would become {3, 5, 2, 4, 6} (node 0 and node 1 have been removed). **Resort it** according the distance between {3, 5, 2, 4, 6} and node7; if the sorting result is:
{2,x,x,x,x} or {4,x,x,x,x}or {5,x,x,x,x}or{6,x,x,x,x}, ,we can directly connect node7 to node 2 or node 4 or node 5 or node 6.

In this example, we can find that the adjusting process of nodelist including not only the removing of saturated node but also the processes of add children nodes.

In fact, no matter how to change the nodelist, we focus on the first unsaturated node that is the closest to the now coming node. From next example, it will be clearer.


Adjusting one, remove 0
{0 1 3 5}---{1 3 5}
Adjusting two, remove 1, add 2.4.6. {1 3 5}---{3 5 2 4 6 }
Adjusting three, remove 6, add 7.8.9
{3 5 2 4 6 } ----{3 5 2 4 7 8 9}
0
1
3
5
10
2
4
6
7
8
9

Figure[9] of Figur18. Join algorithm

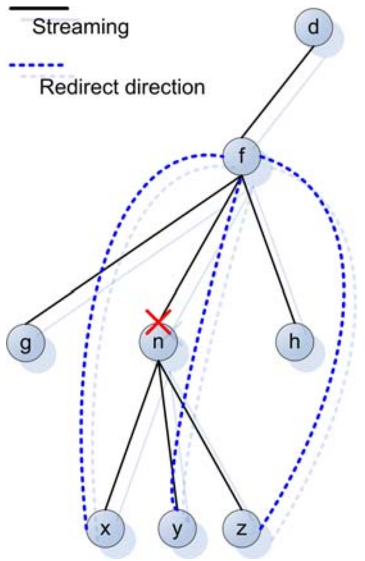
Following figure [8], we suppose add node 7 into node 6; We suppose that node 8 and node 9 have the same situation with node 7, we discuss node 10: now, node 10 comes; after joining request, it would get the original nodelist {0, 1, 3, 5}; After sorting, suppose we get {0, x, x, x}, node 0 is saturated, we need adjusting, all of node 0's children are in the nodelist, remove node 0 from the nodelist; sorting among {1, 3, 5};If result is {1,x,x}; the first node 1 is saturated, adjusting, add all of its children into the nodelist and remove node 1 from the nodelist. Sorting among {3,5,2,4,6} ; if the sorting result is :{6,x,x,x,x},the first node 6 is full, adds all of its children into the nodelist and remove itself form the list; sorting among {3,5,2,4,7,8,9};

In one word, our join algorithm is a process like that:
Send joining request---get original nodelist---sorting it---adjusting it---sorting it ---adjusting it---sorting it ………………….. repeat this process until the new comer find a node to join.

5.2. Leave algorithm

No matter a normal leave or an abnormal failure would cause a transient problem of nodes. Only when the leaving node is in the lowest layer, namely, a leaf, it would not affect others. In reality, a leaf node and a non leaf node have the same possibility to fail. As we have mentioned above, one of our method to deal with the transient problem is an effective and efficient relatively recovery mechanism. This recovery procedure must be included in the leave algorithm. In the following paragraph, we provide two kinds of algorithm for our solution.

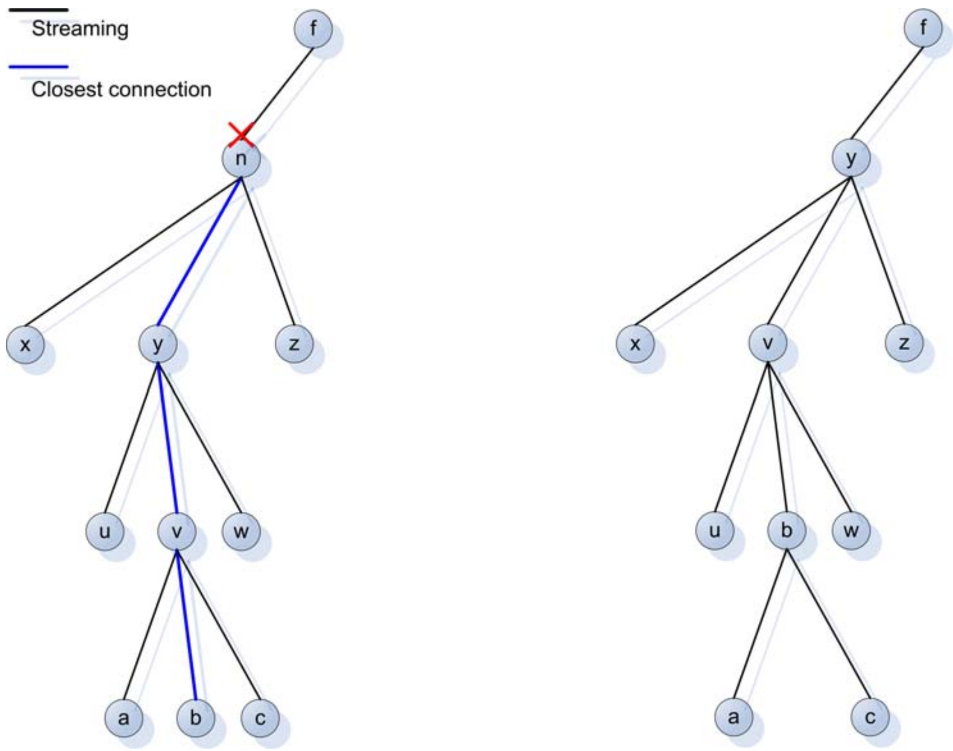
The first one is to redirect all of the descendants of the leaving node to a proper target node. Each node is definitely aware of two nodes in the network at least: its parent, and the source. Thus, there are two least candidate values for the target to redirect. Here, we choose the parent of the leave node as our redirect target instead of choosing the source server. The reason to do so is easy to understand. The recovery process includes a rejoin process that would use the join algorithm, and recall our join policy, the main ideal is to construct the tree according its geographical features, it means that a node is closer to the nodes in the same branch of tree than the nodes in the other branches. If we redirect all the descendants to the source server respectively, according to the join algorithm, they would be reconnected to the same branch just like the situation before the node leaving. So, here, we just redirect all the descendants to the parent of the leaving node. Like in the figure19, node n has failed; all of the descendants would be redirected to the parent of n, namely, node f, and use the join policy to reconnect them. The advantage of such a policy is that the effect of the fail of node is limited to the sub-tree rooted at node f. Moreover, the source s is protected from such requests in the event of multiple simultaneous failures. Yet the tree is expected to remain balanced as the sub-tree is reconstructed from the same nodes as before.



Figur19. Leave algorithm, method 1

For a normal leave, there is no problem to apply this policy, before leave, node n would inform all its descendent its leave and the address of the node f. But for abnormal leave, node n would have no time to do so. Reference the section mechanism of administration, we use the heartbeat message to detect the failure of a node and then we let the node x, y, z keep the information about node f. so, when the abnormal leave of node n has been detected, node x, y, z would know where to go and inform its descendants the information of node f.

The second one is to choose the closest child to replace the place of leaving node. The detail is below.



Figur20. Leave algorithm, method 2

In figure20, node n fail, no matter normal or abnormal fail, its children would know that. Now, compare the distance[f][x], distance[f][y] and distance[f][z], notice that we do not compare distance[n][x],[n][y],[n][z]. not only for the reason that node n has failed we can not get the distance value, but also for the reason that the future streaming source of this branch would be node f, so the distance from node f is more important. We suppose here node y is the closest, and then node y replace the place of node n; after that we can only compare the distance between node y and its direct child, suppose node v is the closer one, then v replace the place of y, reconnect the other ancient child of node y, namely, node u and node w, to node v; and just repeat the same procedure until reach the leaf layer.

The advantage of this method is that the number of affected nodes is very limited; the change of the tree is very small. And this process does not affect the source server at all, decrease the load of source. The disadvantages the time of recovery, each step need a comparing calculation and it would waste time.

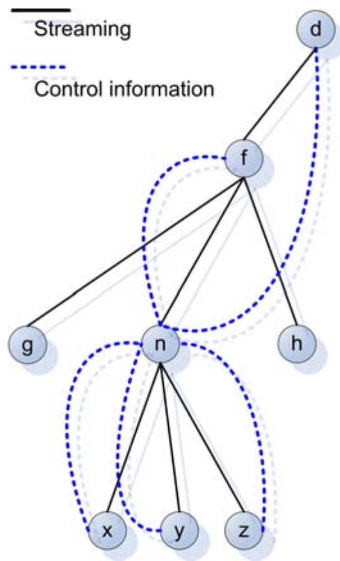
5.3. Mechanism of administration

Our solution is a P2P network solution, therefore there is no a central administration mechanism. But we still need to describe the procedures of changing information among nodes during the streaming session, because it is related to the problem of control of overhead.

In our join algorithm, we proposed a nodelist and we only need to compare the distance value between the new comer and the nodes in the list, by that we find the node that can be connected. Why we do not put the entire nodes in the tree into the nodelist and compare of all of them? That will be more accurate for choosing the closest nodes. It is easy to understand that in reality this kind of policy is inapplicable because when the tree is big enough the communication overhead will be too large to tolerate. In our leave algorithm, the children of leaving node will execute the rejoin procedure from the parent of leaving node instead of the source server; it is also measure that limits the control overhead.

Besides during these operation, even after the node has been added to the tree it also need to keep some information about other nodes for steaming purpose and for dealing with the unpredictable failure.

In our solution, we say, each node in the tree need to keep the information about all of its children, parent and it grandparent. This information mainly includes the address of the node.



Figur21. Control information

Reference figure 21, in the schema we have proposed, one node in the tree has only one parent, and then the grand parent must be one; if we suppose the maximum number of children is 3, then in the tree, the number of node with which one node need to communicate is only 3+1+1=5. Every given interval, 30 seconds for example, the node n send a heartbeat message to node d, f, x, y, z to verify the status of these nodes. For every node in the tree has the same situation, we can find that the communication overhead like that is relatively low, because it does not need to communicate with its neighbor in the same layer like in ZigZag [17].

5.4. Optimizing

Our solution have many shortcomings, here we try to propose some method to repair them and increase the adaptability and performance our streaming schema. One of the main disadvantages of our join algorithm is scarcity of a mechanism to control the height of the tree.

5.4.1. Control of height of the tree

Our geographical featured join algorithm has no a mechanism to control the height of the tree. But the height of tree would affect the end to end delay and then affected the end host performance. We can make the nodes keep the information about its layer, it knows in which layer it rest. We give out a parameter, for example, 20 layers. If the number of layer of a node is smaller than 20, we do nothing; if this value is greater than 20, this node would execute a rejoin procedure, this time it would use still the join algorithm we propose, it has a big possibility for the node to join into a layer smaller then 20, because the tree is changing at every moment with the node leaving. If this time this node is still connected to a layer that is deeper the 20, it means that geographically there are too many nodes that is in the same area with this new comer have been connected to the tree and have occupied the possible place. So when we redirect it to the source the second time, we would use a almost complete tree constructing algorithm instead our algorithm, in section of simulation, we have the detail of the algorithm. In this way, we can limit the height of the tree.

5.4.2. Heterogeneity of nodes

The bandwidth and network situation of each peer is differed from each other. If we choose one peer as an interior node to burden with forwarding stream only base on its geographical location, we can say that it is reasonable, but it is not enough efficient and powerful. In this paper, we suppose each node in the tree would have only one parent, it means a single source of media stream, it is simple but not reasonable, there are already many solution to address problem like Splitsteam. If we do not change this assumption, we have other space to improve: the number of children of the node. We can make each node have the ability to choose a parameter value about its number of children instead of only giving an invariable (like we have chosen 3 in simulation program). In reality, it would be the final application running on the node control the number of children according to the type of steaming media and the situation of inbound/outbound bandwidth.

These two points are just conceived methods with which we do not consider the detail and the feasibility in reality. To any solution, there will be a lot of room to ameliorate.

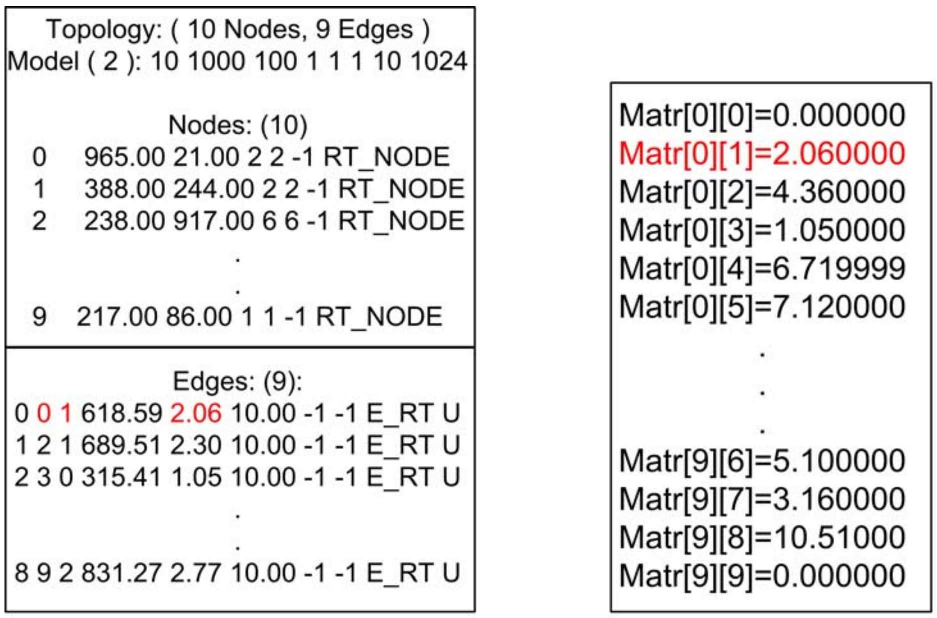
VI. Simulation

In order to verify that whether our P2P live media streaming solution works or not and to investigate the performance of join/leave algorithm under various scenarios, we would carry out a simulation based study. We would consider performance metrics like depth of tree, nodes degrees, the factors that can affect the shape of tree like the division of nodes and the other parameters like peer stretch.

1. The environment of simulation

1.1 The Internet topology generator and the distance matrix

We have mentioned that we want to apply our solution in the environment of Internet, therefore for the purpose of simulation; we must model an Internet topology. We chose to use BRITE [34], a universal topology generation tool to generate several graphs for the simulation purpose. We want to know whether our solution can work well with thousands of nodes, so we have generated 10 nodes, 100 nodes, 1000 nodes, 2000 nodes, 3000 nodes, 4000 nodes and 5000 nodes, 8 graph of topology. The 10 nodes and 100 nodes network topology are used for testing program and representing purpose. The larger networks form 1000 nodes to 5000 nodes are used for the purpose of statistic.



```
Topology: ( 10 Nodes, 9 Edges )
Model ( 2 ): 10 1000 100 1 1 1 10 1024

Nodes: (10)
0    965.00 21.00 2 2 -1 RT_NODE
1    388.00 244.00 2 2 -1 RT_NODE
2    238.00 917.00 6 6 -1 RT_NODE
.
.
9    217.00 86.00 1 1 -1 RT_NODE

Edges: (9):
0 0 1 618.59 2.06 10.00 -1 -1 E_RT U
1 2 1 689.51 2.30 10.00 -1 -1 E_RT U
2 3 0 315.41 1.05 10.00 -1 -1 E_RT U
.
.
8 9 2 831.27 2.77 10.00 -1 -1 E_RT U
```

```
Matr[0][0]=0.000000
Matr[0][1]=2.060000
Matr[0][2]=4.360000
Matr[0][3]=1.050000
Matr[0][4]=6.719999
Matr[0][5]=7.120000
.
.
.
Matr[9][6]=5.100000
Matr[9][7]=3.160000
Matr[9][8]=10.51000
Matr[9][9]=0.000000
```

The BRITE file of 10 nodes   The distance Matrix file of 10 nodes

Figure22. The format of BRITE file and the distance Matrix

Each BRITE Output file includes Nodes section and Edges section. In nodes section, we can get the unique id for each node, x-axis coordinate in the plane, y-axis coordinate in the plane, using this information, helped by proper tools like xfig, we can draw out the physical topology of the generated network. In edges section, we can find node id of source, node id of destination, Euclidean length between them and the propagation delay between them. Using the information of propagation delay between nodes, we can easily build up a distance matrix for the network. In such a distance matrix (a binary file), we can get the distance between any two node. We use distance[x][y] to represent the distance between x and y, and in fact the value distance [x][y] represent the propagation delay between them, We use this value as the standard of calculating when we sort the order of nodelist.

1.2.The event list

 We have already the topology and distance matrix for simulation, but for studying the performance of our algorithm, that is not enough. We still need an event list that describes the behavior of nodes. Here, the event list means the order of nodes that would join, leave the tree with the passing of time.

For some statistical function, the result has no relation with the time or the order of nodes. We only need to select the joining nodes randomly to construct a spanning under a given topology. For instance, if we need to study the division of nodes in each level of tree under different scale of interesting clients, we do not need to propose a given event list, because what we want is an average value, we can get it by calculating. In order to make the result more reasonable and believable, we must construct the tree using the nodes that is randomly chosen. Like what we have done in the section VII.

But for some statistical purpose that has relation with the time, we need a given event list. For example, if we want to study with the time passes how the shape of the streaming tree change and what is the effect when a node leaves the tree, we need a proposed event list. In figure 12, the left side is our proposed event list file, in this file we can find the time of joining the tree for each node. To make the simulation closer to the reality, we arrange these joining nodes according to a Gaussian distribution function, like the right side of figure 12. The 3600 is a central point of time we chosen, it means in this point of time there are most number of nodes come to join into the streaming tree.

Id of Node	The time to join
0	3640.176
1	3573.570
2	4102.323
3	3820.092
4	3899.257
5	3216.749
6	2880.985
7	3396.216
8	3588.273
9	3868.067

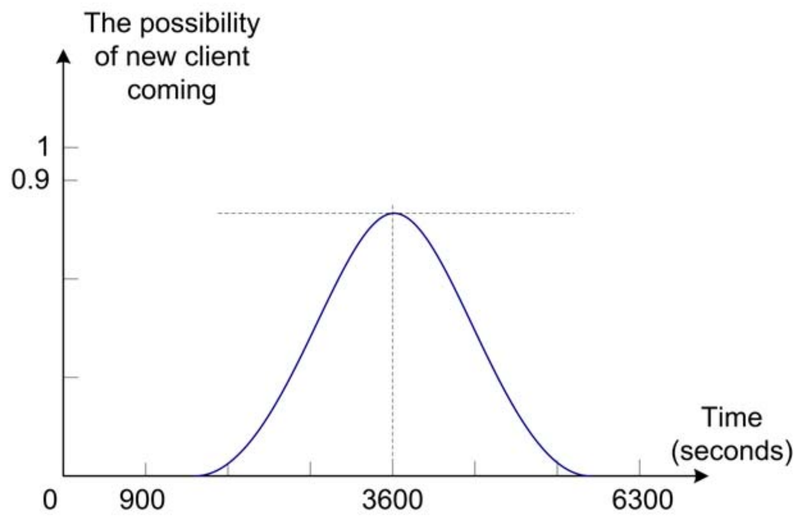


Figure23. Event list file and the distribution of joining node with time

1.3. Leaving possibility of nodes

The leave or failure of a node is totally unpredictable, with the beginning of the tree constructing, there would be node leaving. For the purpose of simulation, we must define clear leaving behaviors. We have several methods to deal with it. First one: we suppose a percentage of failure, for instance, 10%, than in an eventlist that including 1000 different joins we would randomly choose 100 different nodes ID from the existing tree as the failed one. This method should be used to the statistical function that has no relation with time. Second one: We let a number (for example 100) of peers fail sequentially with time and evaluated the affection of leaving of node on the shape of tree. Third one: see the discussion of eventlist in part 1.2, for simulating a reality situation, we would propose a failure model that would also accord a Gaussian function with the pass of time. It is reasonable, because the more the nodes add the higher the possibility of leave is. When the client-base is 1000, we suppose 10% leave possibility and that we can propose an eventlist that include both join and leave just like in reality.

2. The algorithm that used for comparing purpose

For the purpose of simulation, we have programmed the algorithm that we proposed. But we need a comparing object to make the statistical result clearer. So we propose a algorithm of constructing the streaming tree that is used for comparing.

We use this algorithm to construct an almost complete tree. We have two reasons to do this. The first one is this kind of algorithm is relatively simple to realize. The second one is this kind of algorithm is a good object for comparing. Form article [1], we know that under homogeneous unicast edge and node characteristics, an almost-complete spanning tree is the optimal overlay tree for packet loss, packet delay, and time to first packet metrics, an optimal policy is one that maintains an almost complete spanning tree. Although here we have no homogeneous unicast edge and node characteristics, but for simulation purpose, this algorithm that accord with the PeerCast design schema would be a good object.
We call this algorithm as algo1 and call our own algorithm as algo2.

2.1. Join policy of algo1

Algo1 is propose for construct a almost complete tree, so the biggest characteristic would be that the streaming tree is construct layer by layer. Only when the upper layer is full, the node will create a lower layer.It would be better to use the figure to describe the process of joining. Reference figure24: Figure [1][2][3] mean that the first 3 (the maximum number of children of source server) coming node would be added to node 0 (source server) directly. Figure [4] means if the node 0 has no more places, the new comer node 4 would be redirected to node 0’s children. Now, comparing the distance [4][1], distance[4][2] and distance[4][3], there is **a process of sorting** like algo2. Suppose distance [4][2] is the smallest one, node 4 is added to node 2. We suppose that node 5 and 6 have the same situation with node 4. Reference figure [5][6][7].Figure [8] means when node 7 come, it would be redirect to node 1, 2, 3. If the sorting result is {2, 1, 3}, but now node 2 is saturated, and the purpose of algo1 is to construct an almost complete tree that have the minimum number of layers, so node 7 would be connected to node 1 if node 1 have available place. If node 1 is saturated, node 7 would be added to node 3, it means the coming node would be added to according to sorting result and only be operated in the same layer. In figure [9][10], node 8 and node 9 would be added to the same layer with node 7 because there is still place in this layer.

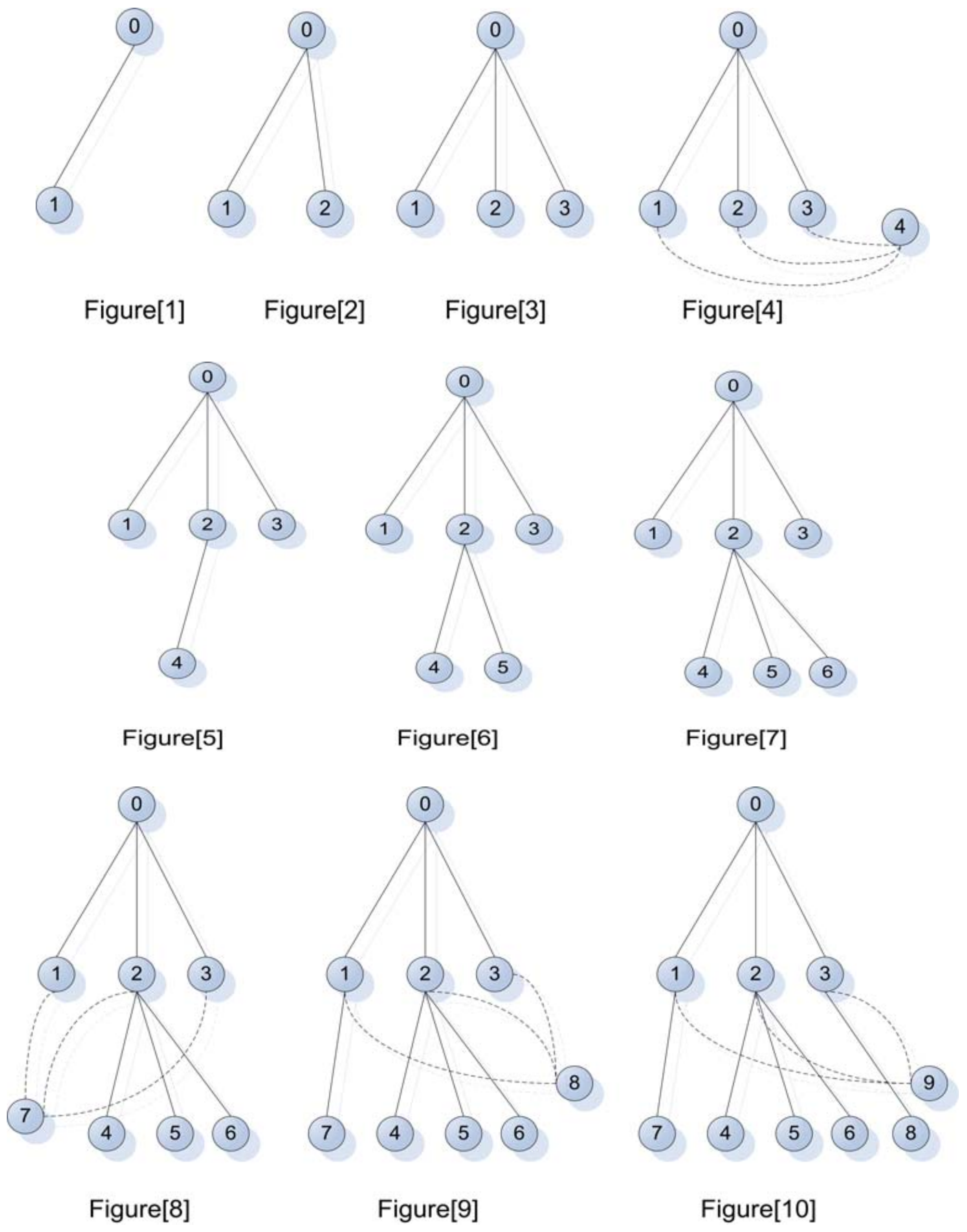


Figure24. Join algorithm of algo1

2.2. Leave policy of algo1

Node N leaves, all of descendants lost connection with streaming. Here, we will use a recovery mechanism just the same as our solution algo2; the detail is in section V, part 5.2, leave policy method 1.

VII. The result of simulation

In this section, we study the performance of our P2P media streaming algorithm by a simulating method. We choose the C as our programming language and the Linux as our testing platform.

Before studying, we should know to choose which network topology for simulation, 100 nodes or 5000 nodes? It means that we must know what the size of client-base is, namely, our algorithm can be applied to how many clients, hundreds, thousands, or ten thousands? The most important factor that limits the size of clients is end to end delay that would effect the end host performance heavily. The end-to-end delay from the source to a node may be excessive because the content may have to go through a number of intermediate receivers, namely, hops or layers. In order to shorten this delay, the tree height (the depth of the tree) must be kept small. Therefore, no matter there is the mechanism to control the depth of tree or not, there must be a bounded degree of node. To us, we would use the simulation result to determine the size of client-base.

1. The division of nodes

By studying the division of node, we would know the percentage of nodes in each layer; in fact, it is also a kind of the information about the shape of tree and shows us how the join policies (here, the result is given in the absence of unsubscribe or failures of clients) affect the shape of tree and the number of layer of the tree.

From article [14], we know that the buffering mechanism at the application level ensures that cumulative delay is acceptable until approximately 15 hops. Thus, if the percentages of nodes that fall at the depths greater than 15 are too high, for example, higher than 5%, it means that there is a relative high ratio of nodes that can not ensure a reasonable end-system performance.

As in figure25 and figure26, the X-axis plots the level number in the tree. The Y-axis plots the average percentage of nodes subscribed to the tree at a certain level.

Figure 25 is the result of a client-base that includes no more than one thousands nodes. From this figure, we find that the deepest layer of the tree is smaller than 20, the layer in that the most percentage of nodes stays is between layer 5 and layer 6, The greatest percentage of nodes in certain layer is between 17% and 14%, only a little of fraction of nodes fall at the layer deeper than 15. From these result we can say the distribution of nodes is relatively equilibrated.

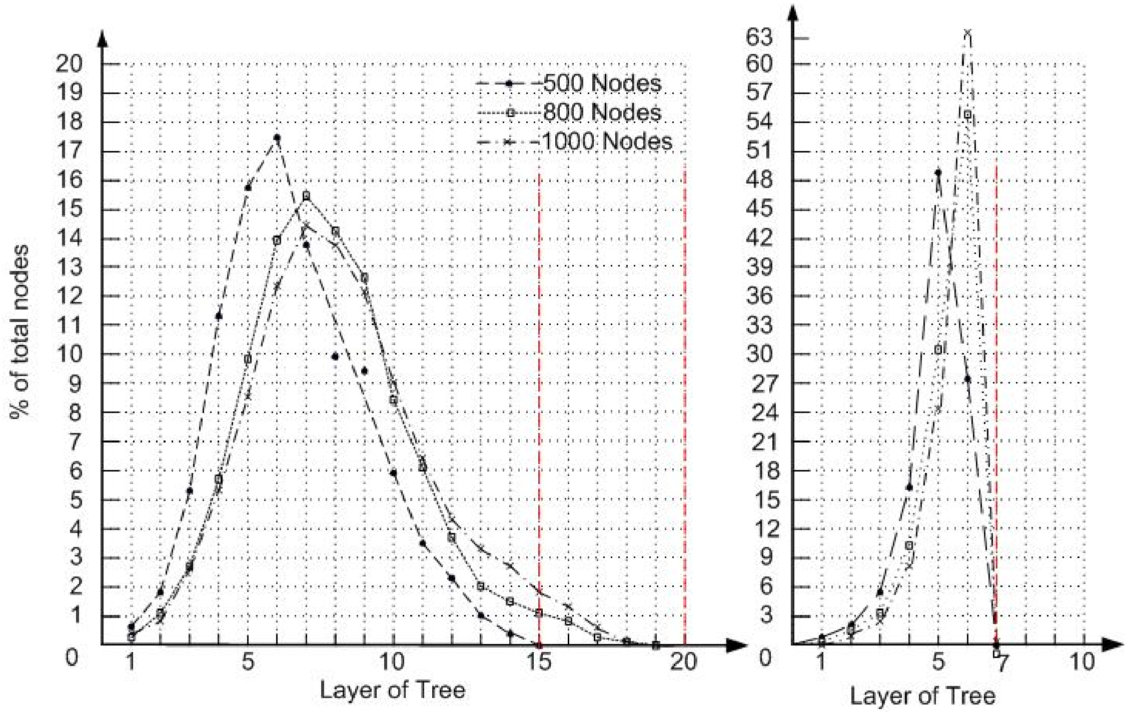


Figure25. The division of nodes based on the client-base size of 1000 nodes

In fact, this figure gives us the information about the shape of the tree, with the increasing of the client-base, the shape of the tree is kept stable, and only the position moved a little, it means that regarding the shape of tree the adaptability of the join algorithm under different size of clients group is good. The right side of the figure is the result of algo1, it curves peak to a very high value (between 63% and 48%) and fall to 0 very sharply, it is the result of the almost complete tree constructing join algorithm.

Instead, the percentage of nodes of algo2 fall off slowly along higher levels, there is till several nodes at depths greater than 15. Its show that one of the disadvantages of our solution: the absence of an effective level control mechanism.

Figure26 is the result of a client-base that includes thousands nodes. From this figure, we find that the layer in that the most percentage of nodes stays is about 9, the greatest percentage of nodes in certain layer is between 12% and 10%, comparing with the result of figure 18; the nodes are much more distributed talking of the layer of the tree. There is a meaningful number of nodes fall at the layer deeper than 15. Even though the shape of the tree is kept, we can not say that this result would return us a good end user performance, after all the depth of the tree is one of the most important factor that affect the end to end delay between the source of and the nodes.

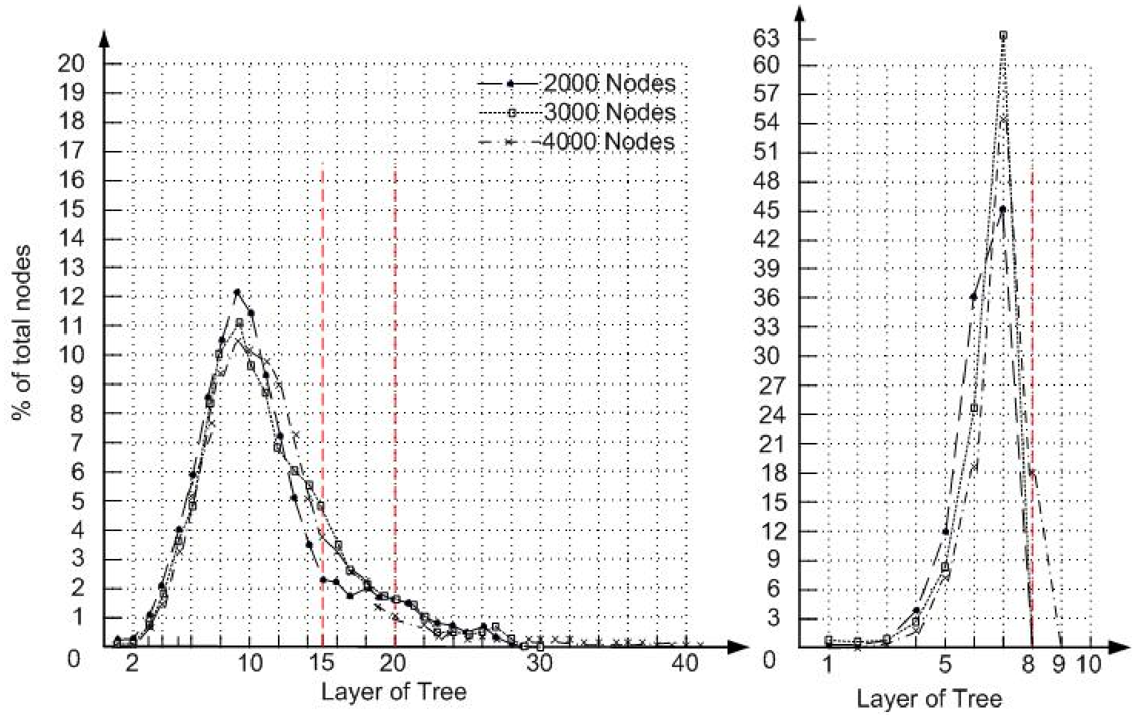


Figure26. The division of nodes based on the client-base size of 5000 nodes

client-base Size	500	800	1000	1500	2000	3000	4000
A*	14	18	18	23	30	33	37
B*	0	2.61%	3.39%	7.72%	12.17%	18.28%	21.87%
C*	0	0	0	3.27%	4.67%	6.93%	8.50%
D*	6	7	7	7	9	9	9
E*	17.53%	15.42%	14.4%	13.21%	12.18%	11.15%	10.47%

Table 9 the statistical result of division of nodes based on different size of client-base

A*: The deepest layer;
B*: The percentage of nodes that fall at the depths greater than 15;
C*: The percentage of nodes that fall at the depths greater than 20;
D*: The layer in that the most percentage of nodes stays;
E*: The greatest percentage of nodes in certain layer;
Note: this table just includes the statistical result for algo2; for algo1, it is not necessary to do so

From table 9, the data obviously show a trend that with the increasing of the size of client-base, the degree of distribution of nodes referring to the level of tree increase in a direct ratio. With the help of these data, we can conclude that to our join algorithm, the suitable size of client-base is between 1000 and 1500; this algorithm can ensure a good end user performance when there are hundreds of clients. Therefore, in the following statistic, we will base on the size of 1000 nodes client-base.

Note: there is one point need our attention. The maximum number of children in simulation is set to 3, if this value is changed, for instance to 10, the result will be dramatically different. But for us, like mentioned above, we want to apply our solution to Internet, for the general application of audio and video, the average bandwidth of the media stream are from tens Kbps to hundreds kbps, for instance, the radio station West Michigan Scanner http://wmsr.tripod.com/ has a bit rate 35kbps and the TV station Star Ray TV http://tobroadcast.com/ has a bit rate 208kbps. And the prevail accessing technique and speed for ordinary Internet user now is 512k DSL or cable, but normally the upstream bandwidth is limited to only 214kbps or lower, so if we set the average maximum number of children to 3, we can get a result that is closer to reality.

2. The depth of the tree

The concept of depth of the tree means the number of the lowest layer connecting the deepest nodes, namely, leaf of the tree. For instance, the deepest level of a tree is 15, and then we say the depth of this tree is 15. From the above simulation result, we know that our algorithm would return a good end user performance on a client-base of hundreds of clients. Figure 20 present the depth of the tree changes with the increasing of number of clients. The X-axis plots the number of nodes in the tree. The Y-axis plots the depth of the tree.

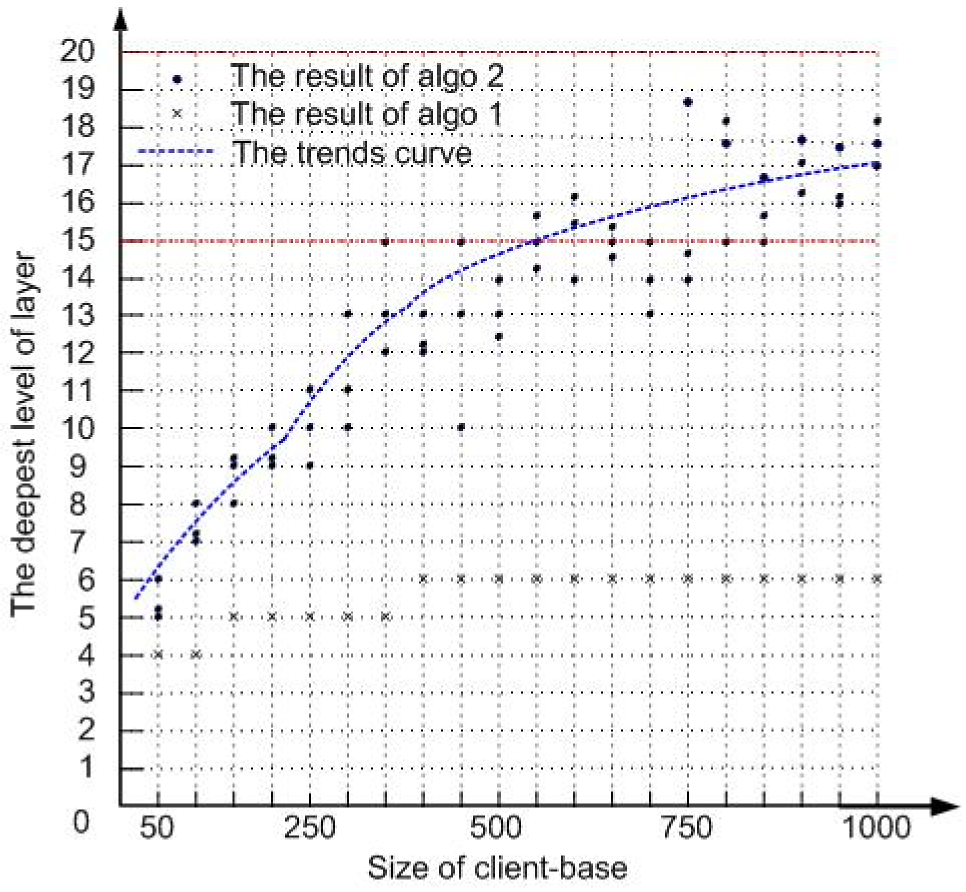


Figure27. The deepest layer with the different size of client-base

Form figure 20, we observe that to algo 1, the almost complete tree, the depth of tree is very limited and low thanks to it algorithm that would fill in all of the layer by nodes except the lowest leaf layer. To algo 2, we sampling 3 times of each given size of client-base, here we use the point instead of one average value, because this kind of result is a range of data, one value is meaningless and from the result we can find that there is a trends of exponential increasing with the augment of clients number. In fact, this statistical result is just the complementarities for the above statistical result no 1.

3. The longest path delay

The path delay of node x means the end to end delay from the source server to node x. Suppose from source server to node x, there are H hops, and this steaming path passes node N1, N2……Nh, and the path delay of node x
= distance [0][N1]+distance[N1][N2]+distance[N2][N2]+…+distance[Nh-1][Nh]. The longest delay path means the longest path delay. Note that the node who has the longest delay is not always the node in the lowest layer.

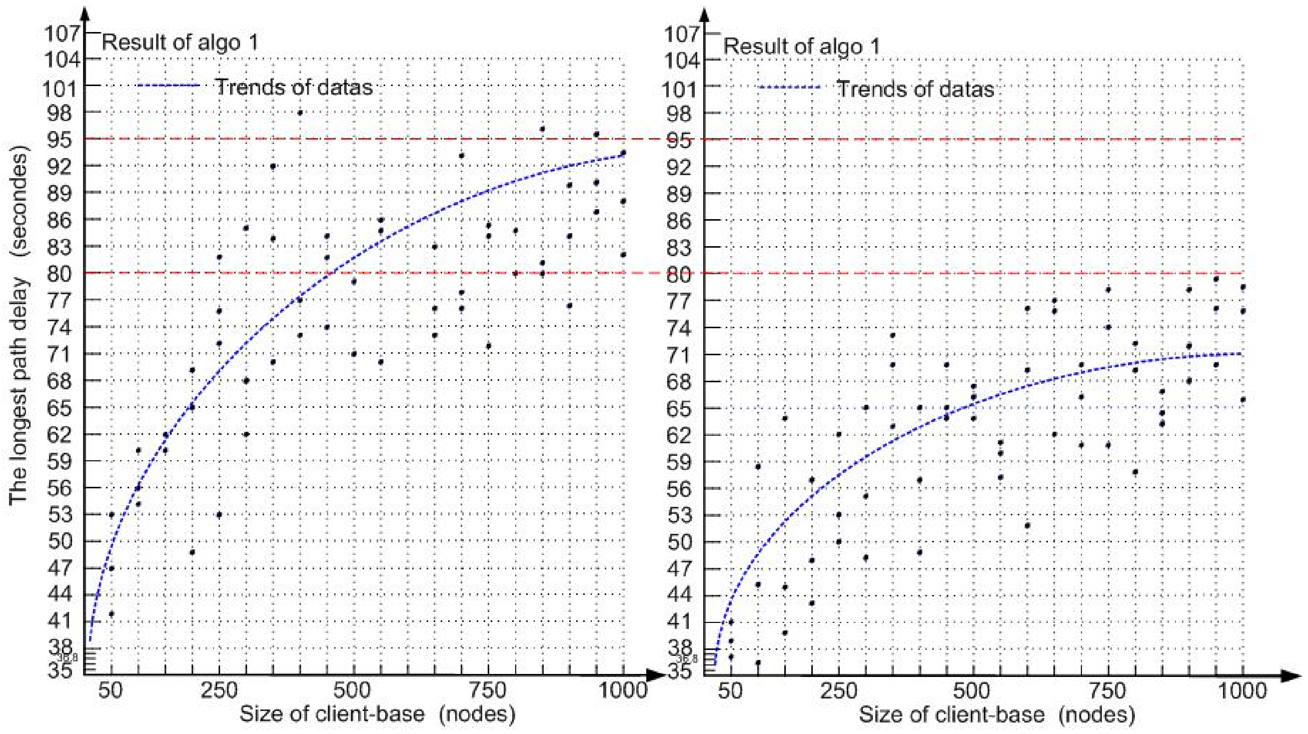


Figure28. The longest path delay

In figure 28, The X-axis plots the size of the client-base. The Y-axis plots the longest path delay.

We sampling 3 times of each given size of client-base, here we use the point instead of one average value, because this kind of result is a range of data. We observe that for both two algo there is a trends of exponential increasing with the augment of clients number. But obviously, the algo 1, the almost complete tree, has a relatively higher longest path delay than the algo 2. When N=500, algo1:algo2=80:65≈1.23; when N=800, algo1:algo2=89:70≈1.27;when N=1000,algo1:algo2=95:75≈1.26. Therefore, we can say that here there is a relation of direct ratio between these 2 algo regarding the longest path delay.

But the problem is that algo2 have much deeper of the depth of tree. Combining with the simulation result 1 and 2, we find that even the algo 1 have a very limited and controlled depth of the tree, it can not give us a controlled path delay. We know that normally the lower the path delay is the better the end host performance is. So we can conclude that our algorithm can effectively limit the longest path delay and help to provide a good end user performance thanks to the geographical close connecting mechanism in our join algorithm.

4. The cost of tree and the sum of distance

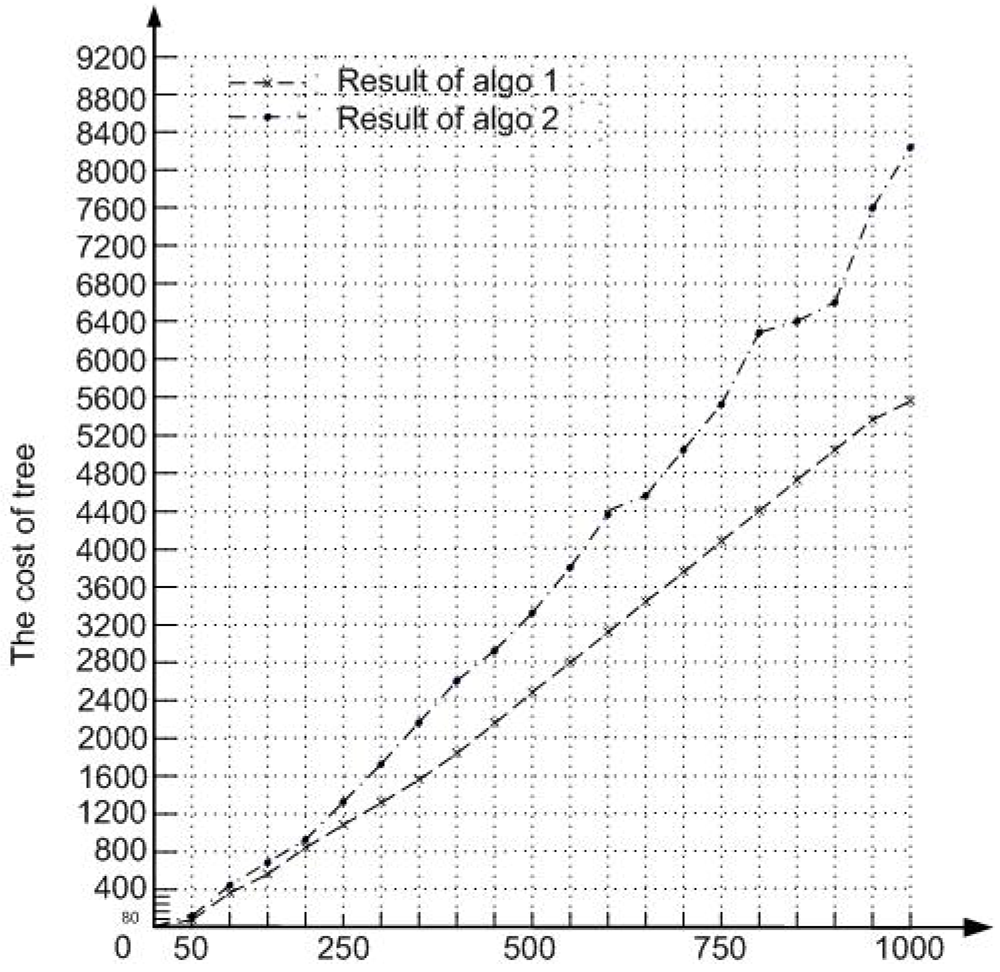


Figure29. Cost of tree for various sizes of client-base

We define [14] the cost C of a tree T as CT $\propto \sum_{1}^{n} h_n$. Here, let n be a node at depth h_n in a source-rooted overlay tree. Since packet delays are additive across hops, the delays observed at n are proportional to h_n. If p_{loss} is the probability of a packet loss at a hop, then the probability of not receiving a certain packet at n is given by $1-(1-p_{loss})^{h_n} \approx p_{loss}h_n$, for small p_{loss}. Thus, packet losses observed at n are proportional to h_n. Therefore, we can define the cost of a tree as C_T. According these theory analyses, it is easy to find that an almost-complete tree will have the minimal cost. Thus, an optimal algorithm is one that maintains an almost complete spanning tree.

Figure 29 shows the cost of the tree in algo 1 that construct an almost complete tree and algo 2 that we proposed for different sizes of client-base. The X-axis plots the size of the client-base. The Y-axis plots the cost of the tree. The cost of the tree was taken to be the average of the result by sampling in 1000 nodes base with the maximum number of children set to 3. We observe that the algo 2 curves diverge from algo 1 as the size of the client-base increases. At N=500, the cost of algo 2 is 1.33 X the cost of algo 1, and at N=1000, the cost of algo 2 is 1.46 X the cost of algo 1. The algo 1 construct an almost-complete source-rooted tree that minimizes C, therefore we conclude that our algo 1 result in a near-optimal trees for small client-base sizes (hundreds of clients), but degrade with increasing size of client-base.

Note: The result would change dramatically when the maximum number of children in simulation changes, here it is 3.

Next, see the figure30,

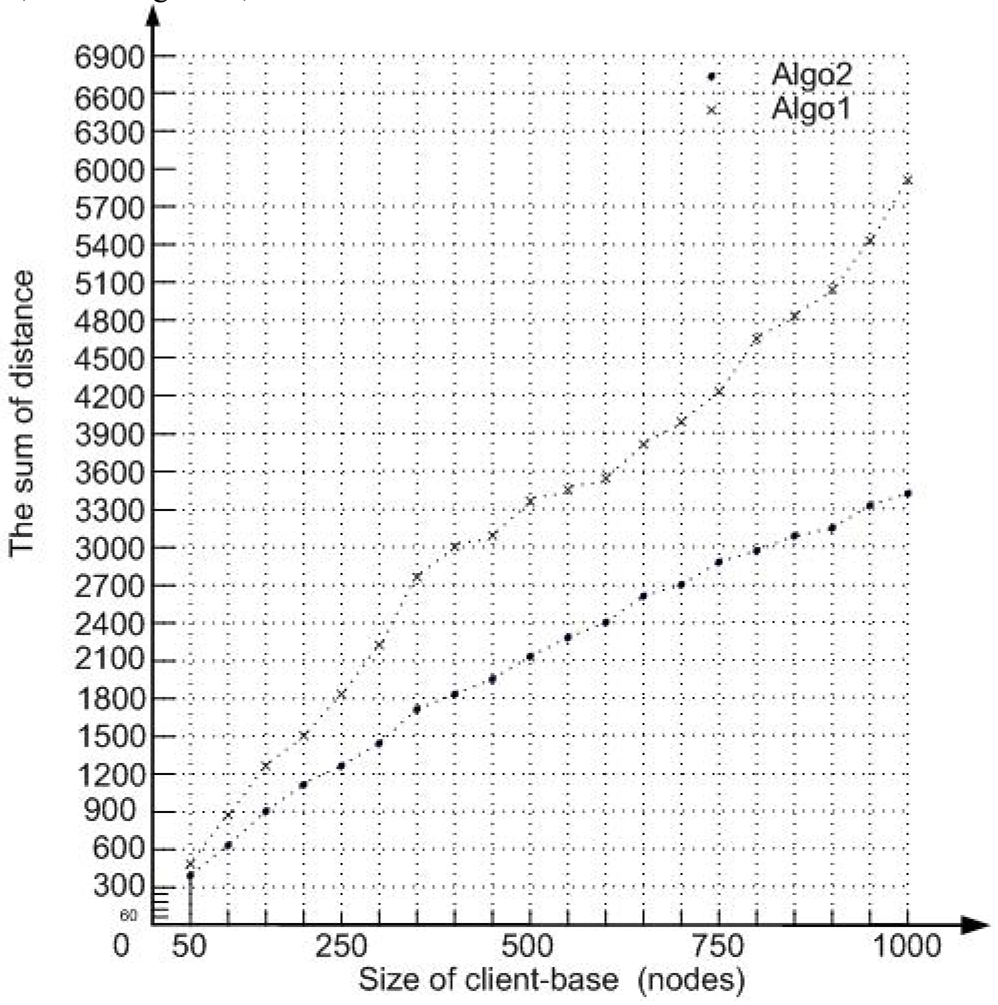


Figure30. The sum of distance

We define the sum of distance of a tree as the sum of path delay of all of the nodes in the tree. Suppose there are N nodes in the tree, the sum of distance of tree= $\sum_{x=1}^{N}$(path delay of node x) . The sum distance has a sense like the cost of the tree; the lower this value is the better the final node performance is. And it is parameter that describing the characteristic of the whole tree. We observe that algo 2 has a smaller value of the sum of distance compared with algo 1.

Comparing figure29 and figure 30, we can conclude that even algo 2 have more layer in the tree and a higher cost of tree, but mentioned the sum of distance, it has a better result. Therefore in geographic, the tree constructed by algo2 is more compact than algo1.

5. The effect of leave algorithm

Now, we would study the effect of the leave algorithm on the shape of the tree under the help by an eventlist.

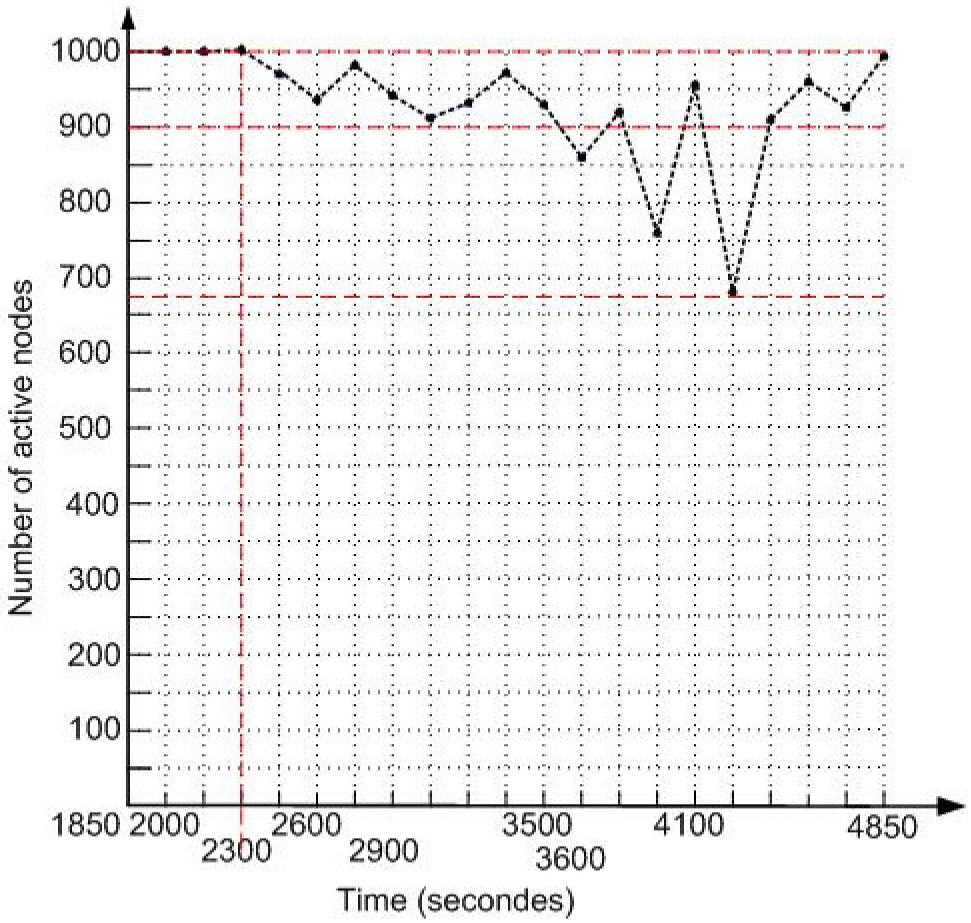


Figure31. The effect of leave algorithm

In figure 31, the X-axis plots simulation time, while the Y-axis plots the number of active nodes in the overlay tree. For the purpose of study, we started with the system consisting of 1000 clients and we suppose that at the very beginning, all of these 1000 client have been added to the tree and there is no failure. We let a number of peers fail sequentially and each failure happened at a given time. Here, we have a sequential failure of 16 nodes, account for 16% of total nodes. And we suppose from the time point 2300, the fail begin to happen. What we want to study is how many nodes would be affected when a node failure occur. When the node leave happen to the leaf layer, the failure would do not affect the system because the node has no dependences. For those failures happening to higher layer nodes, the number of being affected nodes are mostly less than 100 (10% of client population). In the worst situation, in figure 23, time 4250, there are about 300 nodes being affected. We have discussed the leave algorithm in section V, from the theory side, it can ensure a rapid recovery, combing the simulation result here, the nodes that are affected by the action of leave would be very limited, therefore we can conclude that our leave algorithm are desirable,

6. A session that simulate reality

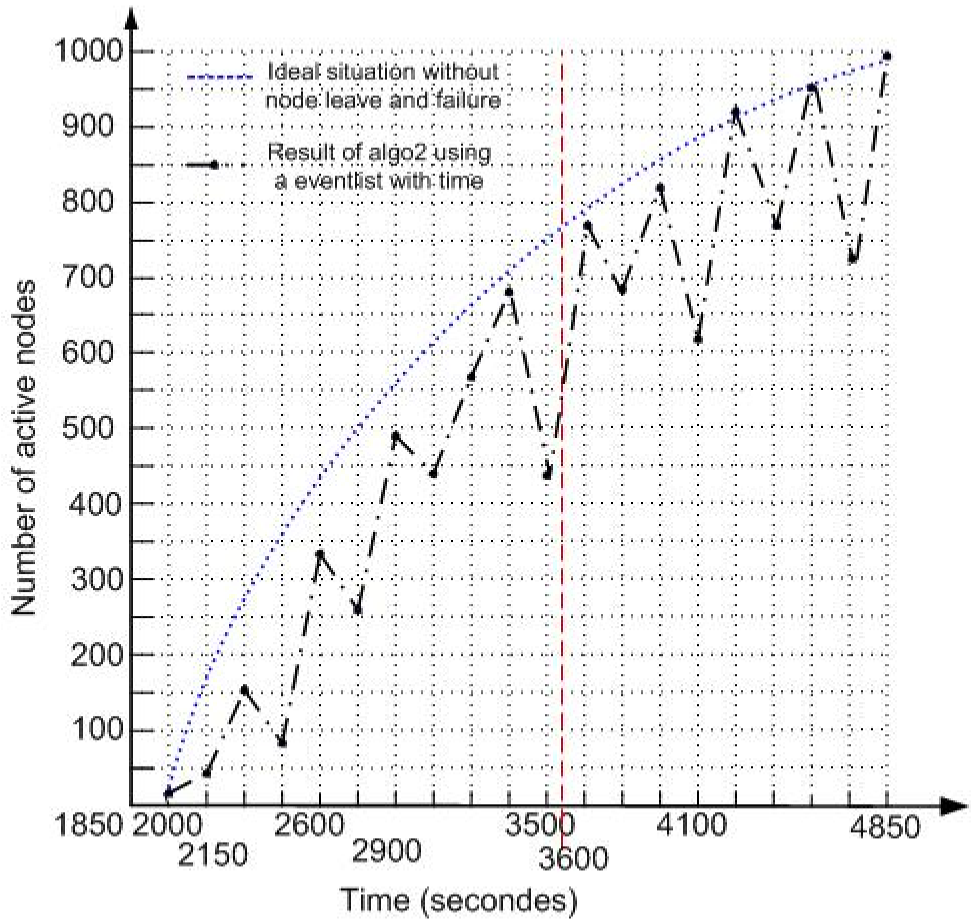


Figure32. Simulating procedure of a streaming session

Finally, we use a eventlist that include continues join and random leave in the simulation program and we represent the result in figure 24, the X-axis plots simulation time, while the Y-axis plots the number of active nodes in the overlay tree. These join node in the eventlist would be in a Gaussian function model like we have discussed in section VI.

At the very beginning, there would be already the node leave; the action of leave including the possibility of leave and the leaving model with the time would accord with the model of node failure that we discussed in section VI. We observe that under the ideal situation that has no failure or leave of node, the curve would be an exponential function, which is the reasonable result because the node joining is in the mode of a Gaussian function. As for the result including the node failure, we find that the trend of the curve accord with an exponential model and the effect of the leave action are limited. Therefore we can conclude that the result accord with the theory analysis and it show that our algorithm can return a desirable result.

VIII. Conclusion and future works

Peer-to-peer media streaming systems are expected to become as popular as the peer-to-peer file sharing systems. That is what we want to discuss in this paper. We focused on the maintaining algorithm of a single source, receiver driven media streaming delivery tree and though those tree constructing policies we try to get a reasonable, and acceptable, or even a desirable result in term of the end host performance metrics. Our proposed solution is featured by its tree maintaining algorithm that organize the multicast tree according to the geographical characteristic of nodes. The main idea in our algorithm is the concept “nodelist” and the process of “sorting” and “adjusting”. Both the theoretical analyses and the simulation studies proved that under a client-base size of hundreds (in fact the adaptable size is between 1000 nodes and 2000 nodes); our solution can maintain a ‘balanced” streaming tree and gain a relatively good end user performance.

1. The ability to deal with transience problem

We use the mechanism of hiding the changes of the topology of streaming tree from the final application at each client and a fast recovery mechanism to address this problem. The simulation result of show that the effecting of nodes failure can be control under a reasonable range.

2. End-to-end delay

In theory, our geographical sorting join policy and fast recovery mechanism would ensure a small end-to-end delay at the time of the constructing of the streaming delivery tree. And the mechanism of control of the height of the tree would limit the worst end-to-end delay. The simulation result of the longest path delay show us that the worst delay in our tree is very limited compared with the almost complete tree under the same network topology environment and the same enventlist.

3. Control overhead

In section V, part 2.4, we have discussed our control overhead. Under our schema, a node in the spanning tree only needs to periodically communicate with its parent, grandparent and all of its children. It does not need to change information with its neighbors in the same layer. It means the number of nodes that need to change information is very small. This kind of measure can greatly limit the control overhead. And the overhead is a constant (if the maximum number of children is a constant) regardless of the client population, which is a big advantage.

4. Efficient join and failure recovery

Our method of applying a “nodelist” reduce dramatically the number of nodes that a new client need to request. And our recovery policy changes the shape of the existing tree as little as possible. Therefore, we can say that our join and failure recovery mechanism are efficient.

5. The stability of the streaming delivery tree

Both from the algorithm analysis and the result of simulation we can find that every operation pursues to affect the existing tree as little as possible. That means the most part of nodes in the tree would enjoy a relatively stable service.

6. Future works

Of course, just like we have mention is the discussion of optimize policy, our tree constructing and maintaining schema have still many shortcomings, for example, a effective mechanism to control the height of the tree. These shortcomings mean a big room for improvement and ameliorate the end host performance. Our solution can be extended in many directions.

First, the dealing with of multi-source problem, the assumption of single-source is just for the simplifying the problem, and obviously, a multi-source tree will be a good direction to develop our schema.

Second, the extensibility to a larger size of client-base, from the discussion of simulation result we can find that the main factor that limits the client-base is the height of the tree and the maximum number of children. The height of tree can be control by an effective mechanism.

Third, the problem of heterogeneity of peers, the bandwidth and network situation of each peer is differed from each other. If we choose one peer as an interior node to be burden with forwarding stream only base on its geographical location, it is reasonable, but not enough efficient and powerful. The simplest improving method is dynamically changing the number of maximum children.

In one word, the live media streaming over P2P network has a promising bright future and at the same time still has a long way to go.

References

[1] Kangasharju, Jussi Antti Tapio: Internet Content Distribution, PhD thesis, April 2002.
[2] Napster Inc. The napster homepage. In http://www.napster.com/, 2004.
[3] Open Source Community. Gnutella. In http://gnutella.wego.com/, 2004.
[4] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-Peer Computing. Hewlett-Packard Technical Report. 2002.
[5] Open Source Community. The free network project rewiring the internet. In http://freenet.sourceforge.net/, 2001.
[6] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnana. Chord: A scalable peer-to-peer lookup service for internet applications. In Proceedings of the ACM SIGCOMM’01 Conference, 2001.
[7] S. Ratnassamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In Proceedings of the ACM SIGCOMM’01 Conference, 2001.
[8] B. Y. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141,University of California at Berkeley, Computer Science Department, 2001.
[9] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems". IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany, pages 329-350, November, 2001.
[10] FastTrack Peer-to-Peer technology company. FastTrack. In http://www.fasttrack.nu/, 2001.
[11] KaZaA file sharing network. KaZaA. In http://www.kazaa.com/, 2002.
[12] Dongyan Xu_, Mohamed Hefeeda, Susanne Hambrusch, Bharat Bhargava; On Peer-to-Peer Media Streaming, In Proceeding of IEEE-ICDCS’02, Vienna, Austria, July 2002.
[13] M. Bawa, H. Deshpande, and H. Garcia-Molina. Transience of peers and streaming media, ACM SIGCOMM Computer Communication Review, Volume 33, Issue 1, January 2003.
[14] H. Deshpande, M. Bawa, and H. Garcia-Molina. “Streaming Live Media over a Peer-to-Peer Network”, Technical Report, Stanford University, August 2001.
[15] V. K. Goyal, Multiple Description Coding: Compression Meets the Network, IEEE Signal Processing Magazine, vol. 18, no. 5, pp. 74 – 94, Sept. 2001.
[16] V. Padmanabhan, H. Wang, P. Chou, K. Sripanidkulchai; Distributing Streaming Media Content using Cooperative Networking, Procc. of the 12th international workshop on network and operating system support for digital audio and video, Miami, Florida 2002.
[17] Duc A. Tran, Kien A. Hua, Tai Do. ZIGZAG: An Efficient Peer-to-Peer Scheme for Media Streaming. IEEE INFOCOM 2003
[18] Jussi Kangasharju, Keith W. Ross, David A. Turner. Optimal Content Replication in Peer-to-peer communities, Submitted, 2003.
[19] E. Cohen, S. Shenker; Replication Strategies in Unstructured Peer-to-Peer Networks, Proceeding of the 2002 Conference on applications, technologies, architecture and protocols for computer communications SIGCOMM 2002.

[20] D. Stolarz; Peer-to-Peer Streaming Media Delivery, IEEE-International Conference on Peer-to-Peer (P2P'01) Lingköping, Sweden, August, 2001.
[21] S.Chen, B. Shen, S. Wee and X. Zhang; Investigating performance insights of segments-based proxy caching of streaming media strategies, Proceedings of ACM International Conference on Multimedia Computing and Networking, (MMCN' 04), Santa Clara, California, January 21-22, 2004.
[22] S.Chen, B. Shen, S. Wee, and X. Zhang; Adaptive and lazy segmentation based proxy caching for streaming media delivery, Proceedings of 13th ACM International Workshop on Network and Operating Systems Support for Design Audio and Video (NOSSDAV'03), Monterey, California, USA, June 2003.
[23] S.Chen, B. Shen, S. Wee, and X. Zhang; Streaming flow analyses for prefetching in segment-based proxy caching to improve media delivery quality, Proceedings of the 8th International Workshop on Web Content Caching and Distribution (WCW' 03), Sep.2003.
[24] N. J. Tuah, M. Kumar, S. Venkatesh; Investigation of a Prefetch Model for Low Bandwidth Networks, Proceedings of the 1st ACM international workshop on Wireless mobile multimedia, October 1998.
[25] M. Hefeeda, A. Habib, B. Botev, D. Xu, and B. Bhargava. PROMISE: peer-to-peer media streaming using Collect-Cast. Technical report, CS-TR 03-016, Purdue University, August 2003. Extended version.
[26] M. Hefeeda, A. Habib, B. Botev, D. Xu, B.Bhargava, "CollectCast: A Peer-to-Peer Service for Media Streaming," submitted to ACM Multimedia Systems Journal, Oct 2003.
[27] R. Lienhart, M.Holliman, Y.Chen, I. Kozintsev and M. Yeung; Improving Media Service on P2P networks, IEEE Internet Computing January-February 2002.
[28] M. Castro, P. Druschel, A.M. Kermarrec, A. Nandi; SplitStream: High-bandwidth content distribution in cooperative environments, IPTPS'03, Berkeley, CA, Feb.2003.
[29] M. Yang, Z. Fei, A Fine-Grained Peer Sharing Technique for Delivering Large Media Files Over the Interne", Proceedings of the Eighth International Workshop on Web Content Caching and Distribution (WCW 2003), Hawthorne, NY, September 2003
[30] Shoaib Khan, Rüdiger Schollmeier, and Eckehard Steinbach, A Performance comparison of Multiple Description Video Streaming in Peer-to-Peer and Content Delivery Network", IEEE International Conference on Multimedia & Expo, ICME' 04, Taipei, Taiwan, June 2004.
[31] M. Hefeeda, B. Bhargava, D. Yau, A Hybrid Architecture for Cost -Effective On-Demand Media Streaming, Journal of Computer Networks, 44(3), 2004.
[32] M. Hefeeda, A. Habib, B. Bhargava, Cost-Profit Analysis of a Peer-to-Peer Media Streaming Architecture, CERIAS TR 200237, Purdue University, October 2002.
[33] Z.Xiang, Q. Zhang, W.Zhu, Z.Zhang and Y.Zhang; Peer to-Peer Based Multimedia Distribution Service, IEEE transactions on Multimedia , Vol. 6. No.2, April 2004.
[34] http://www.cs.bu.edu/BRITE/
[35] A. Medina, A. Lakhina, I. Matta, and J. Byers. BRITE: An approach to universal topology generation. In Proceedings of MASCOTS'01, August 2001.
[36] Suman Banerjee, Bobby Bhattacharjee, Christopher Kommareddy: Scalable Application Layer Multicast.

[37] A.R owstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. IFIP/ACM Middleware 2001.
[38] M.C astro, P.D ruschel, A.-M.Kermarrec, and A.Ro wstron. SC RIBE: A large-scale and decentralized application-level multicast infrastructure. IEEE JSAC, 20(8), Oct.2002.
[39] M.Cast ro, P.D ruschel, Y.C.H u, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. Technical Report MSR-TR-2002-82, Microsoft Research, 2002.
[40] Robbert van Renesse, Ken Birman, Adrian Bozdog, Dan Dumitriu, Manpreet Singh, Werner Vogels. Heterogeneity-Aware Peer-to-Peer Multicast. Dept. of Computer Science, Cornell University, Sony Corporation, Oct.2003
[41] http://www.peercast.org
[42] http://www-db.Stanford.edu/peers/

Appendix1.

The codes for simulation our algorithm: algo2

```
/*****************************************************************
** The name of file: simu_algo2.c
** The date: 2004.08.07
** Description: The join/leave algorithm;
                              The statistic function;
** Version: 2004_07_28_DEBUG
*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* Nodes */
#define NODE_SOURCE        0
#define MAX_PARENTS 1      /*    the number of the parent of a node    */
#define MAX_CHILD 3      /*    the maximum number of children of a node    */
#define M_pV(pHead, c, x, y) (pHead+c*y+x)
  /*    finish the function of dynamic two dimension array    */

/*    define the structure of Node    */
typedef struct _Node
{
    int Id;                  /*    Node Id    */
    float Path_Delay;      /*    Path Delay    */
    struct Node* Parent[MAX_PARENTS];    /* sender = Parents[0] */
    struct Node* Child[MAX_CHILD];
}Node;

/*    define the structure representing layer information    */
typedef struct _Layer_Info
{
    int iLayer;              /*    current layer    */
    int iFull;               /*    the pointed node is full or not，0～MAX_CHILD represent
the iFull th node is empty，MAX_CHILD means full    */
    Node* pNode;            /*    the pointer that point to the Node structure    */
}Layer_Info;

/*    define double link table structure    */
typedef struct _DLinkList
{
    Layer_Info* pLayer_Info;
    struct _DLinkList* prior, * next;
}DLinkList;

/*    define global variable    */
float* g_Matr;
Node* g_NodeinTree[1];
int g_numberofnodeinTree;              /*    the total number of nodes in the tree    */
int g_MatrixSize;                      /*    the size of the distance matrix    */
DLinkList* g_pHeader = NULL;           /*    conserve the pointer of link table */
DLinkList* g_pSeekHeader = NULL;       /*    conserve the pointer of link table */
FILE* g_result;
float g_Sum;
int g_ichoice, g_iseeds, g_iremoved;
int* g_pSelectArray;
int* g_pRemoveArray;
int Longest_Layer = 0;
void ShowOnScreen();

/*****************************************************************
** The name of function:       RandomFun
** Input:      int iTotalNum, int iSelectNum, int iRemoveNum
** Titanium:      the total size
** iSelectNum:      the number of random chosen nodes, keep it in the array
```

```
                    "g_pSelectArray"
** iRemoveNum:   choose random number of node from iSelectNum, keep it in the
array "g_pRemoveArray"
** Output:     0——normal;   1——fail to allocate memery
** Description:      random function

****************************************************************/
int
RandomFun(int iTotalNum, int iSelectNum, int iRemoveNum)
{
   int i, j, tmp, ibool;
   g_pSelectArray = (int *) malloc(sizeof(int) * iSelectNum);
   g_pRemoveArray = (int *) malloc(sizeof(int) * iRemoveNum);
   if (g_pSelectArray == NULL || g_pRemoveArray == NULL)
         return 1;
   srand((unsigned) time(NULL));

   for (i = 0; i < iSelectNum; i++)
      {
         ibool = 0;
         while (ibool != 1)
            {
               tmp = (int) (((rand() / 10000) * iTotalNum) / (RAND_MAX / 10000));
               ibool = 1;
               for (j = 0; j < i; j++)
                  {
                     if (tmp == g_pSelectArray[j] || tmp == 0)
                        {
                           ibool = 0;
                           j = i + 1;
                        }
                  }
            }
         g_pSelectArray[i] = tmp;
      }

   for (i = 0; i < iRemoveNum; i++)
      {
         ibool = 0;
         while (ibool != 1)
            {
               tmp = (int) (((rand() / 10000) * iTotalNum) / (RAND_MAX / 10000));
               ibool = 1;
               for (j = 0; j < i; j++)
                  {
                     if (g_pSelectArray[tmp] == g_pRemoveArray[j])
                        {
                           ibool = 0;
                           j = i + 1;
                        }
                  }
            }
         g_pRemoveArray[i] = g_pSelectArray[tmp];
      }
   return 0;
}   /*   End of RandomFun   */

/****************************************************************
** The name of function:      ClearRandomArray
** Input:   No
** Output:   No
** Description:      release the memory used by random function
****************************************************************/
void
ClearRandomArray()
{
    if (g_pSelectArray != NULL)
            free(g_pSelectArray);
    if (g_pRemoveArray != NULL)
            free(g_pRemoveArray);
}    /*    End of ClearRandomArray    */

/*****************************************************************
** The name of function:       InitMatrix
** Input:      No
** Output:      No
** Description:    Initialize the distance matrix, keep it in “g_matr”, use
“*M_pV(g_Matr, the column of matrix, visited x, visited y)” to find.
*****************************************************************/
void
InitMatrix()
{
    int i, j, imaxchild;
    FILE* fp;

    g_ichoice = 0;
    while (g_ichoice != 1 && g_ichoice != 2 && g_ichoice != 3)
        {
            printf("\nPlease Choose 1.'1000.adj'    2.'2000.adj'    3.'3000.adj'    \n");
            scanf("%d", &g_ichoice);
        }

    printf("Please Input the Number of Nodes Which should be Add to the Tree\n");
    scanf("%d", &g_iseeds);

    if (g_ichoice == 1)
            fp = fopen("r1000.adj", "rb");
    else if (g_ichoice == 2)
            fp = fopen("r2000.adj", "rb");
    else
            fp = fopen("r3000.adj", "rb");

    g_result = fopen("result.txt", "wt+");

    if (fread(&g_MatrixSize, sizeof(int), 1, fp) != 1)
        {
            fprintf(stderr, "Error: reading matrix size in");
            exit(21);
        }

    printf(" ----      Result    ----\n");
    fprintf(g_result, " ----      Result    ----\n");

    imaxchild = MAX_CHILD;
    printf("\nMAX_CHILD = %d\n", imaxchild);
    fprintf(g_result, "\nMAX_CHILD = %d\n", imaxchild);

    printf("Loading Matrix Size: %d x %d\n", g_MatrixSize, g_MatrixSize);
    fprintf(g_result, "Loading Matrix Size: %d x %d\n", g_MatrixSize,
                g_MatrixSize);

    g_Matr = (float *) malloc(sizeof(float) * g_MatrixSize * g_MatrixSize);
    if (g_Matr == NULL)
        {
            fprintf(stderr, "Error: allocate memory for the    distance matrix");
            exit(22);
        }
    i = g_MatrixSize * g_MatrixSize;
    j = fread(g_Matr, sizeof(float), i, fp);
    if (j != i)
        {
            fprintf(stderr, "Error: reading matrix size in");
            exit(23);
        }
    printf("Loading matrix: File Read Succeed!\n");
    fprintf(g_result, "Loading matrix: File Read Succeed!\n");

    for (i = 0; i < g_MatrixSize; i++)
        {
            for (j = 0; j < g_MatrixSize; j++)
                {
                    /*fprintf(g_result,     "Matr[%d][%d]=%f\n",     i,     j,     *M_pV(g_Matr,
g_MatrixSize, i, j));*/
                }
        }
    fclose(fp);
}
/*****************************************************************
** The name of function:        InitNode
** Input:      No
** Output:      0---normal，  1---fail
** Description:    initialize the node
*****************************************************************/
int
InitNode()
{
    int j;

    /*    initialize node 0    */
    g_NodeinTree[0] = (Node *) (malloc(sizeof(Node)));
    if (g_NodeinTree[0] == NULL)
            return 1;

    g_NodeinTree[0]->Id = 0;
    g_NodeinTree[0]->Type = 2;
    g_NodeinTree[0]->Path_Delay = 0;
    g_NodeinTree[0]->Parent[0] = NULL;

    for (j = 0; j < MAX_CHILD; j++)
        {
            g_NodeinTree[0]->Child[j] = NULL;
        }
    InitMatrix();

    return 0;
} /*    End of InitNode    */

/*****************************************************************
** The name of function:        FindMin
** Input:      int iNodeID, DLinkList ** pHeader
** iNodeID:    the node ID of targeted node
** pHeader:      the pointer of link table for the serching object
** Output:      the pointer of the closest node
** Description:    find the closest node
*****************************************************************/

Node*
FindMin(int iNodeID, DLinkList** pHeader)
{
    int k, x;
    float distance;
    Node* pNode;
    DLinkList* p_tmpDLL;

    distance = -1;
    p_tmpDLL = *pHeader;

    while (p_tmpDLL != NULL)
```

```
        {
          if (distance == -1)
            {
              k = p_tmpDLL->pLayer_Info->pNode->Id;
              pNode = (Node *) p_tmpDLL->pLayer_Info->pNode;
              distance = *M_pV(g_Matr, g_MatrixSize, iNodeID, k);
            }
          x = p_tmpDLL->pLayer_Info->pNode->Id;
          if (distance > *M_pV(g_Matr, g_MatrixSize, x, iNodeID))
            {
              distance = *M_pV(g_Matr, g_MatrixSize, x, iNodeID);
              k = x;
              pNode = (Node *) p_tmpDLL->pLayer_Info->pNode;
            }
          p_tmpDLL = p_tmpDLL->next;
        }

    return pNode;
} /*    End of FindMin    */


/*****************************************************************
** The name of the function:        Traversal_Tree
** Input:      Node* pNode, int iLayer, DLinkList ** pHeader
** pNode:      the pointer of root node
** iLayer:        the layer of pNode，the layer number of source is 0
** pHeader:      to preserve the pointer of link table
** Output:      0——normal
** Description:        traverse the tree
*****************************************************************/

int
Traversal_Tree(Node* pNode, int iLayer, DLinkList** pHeader)
{
  int i;
  DLinkList* p_tmpDLL;

  if (pNode != NULL)
    {
      /*    scan the whole tree    */
      p_tmpDLL = (DLinkList *) (malloc(sizeof(DLinkList)));
      (DLinkList *) (p_tmpDLL->pLayer_Info) = (DLinkList *)

(malloc(sizeof(DLinkList)));

      p_tmpDLL->pLayer_Info->pNode = pNode;
      p_tmpDLL->pLayer_Info->iLayer = iLayer;
      p_tmpDLL->next = *pHeader;
      p_tmpDLL->prior = NULL;
      if (*pHeader != NULL)
          (*pHeader)->prior = p_tmpDLL;
      *pHeader = p_tmpDLL;

      /* store the whole tree in the double linked list, not forget to free it */
      iLayer++;
      /*    continue to traverse    */
      for (i = 0; i < MAX_CHILD; i++)
        {
          Traversal_Tree((Node *) pNode->Child[i], iLayer, pHeader);
        }
    }

  return 0;
} /* End of Traversal_Tree */

/*****************************************************************
```

```
** The name of function:      FreeDLinkList
** Input:     DLinkList ** pHeader
** pHeader:     preserve the pointer of link table
** Output:     no
** Description:      release the memory
*****************************************************************/

void
FreeDLinkList(DLinkList** pHeader)
{
    DLinkList* p_tmpDLL;
    while (*pHeader != NULL)
        {
            p_tmpDLL = *pHeader;
            *pHeader = (*pHeader)->next;
            if (p_tmpDLL->pLayer_Info != NULL)
                    free(p_tmpDLL->pLayer_Info);
            p_tmpDLL->pLayer_Info = NULL;
            free(p_tmpDLL);
            p_tmpDLL = NULL;
        }
} /* End of FreeDLinkList */

/******************************************************************
** The name of function:      CountDelay
** Input:     Node * pNode
** pNode:     the node that we want to calculate distance
** output:     the path delay from 0 to node pNode
** Description:      the path delay from 0 to node pNode
*****************************************************************/

float
CountDelay(Node* pNode)
{
    float distance;
    Node* p_tmpNode;

    if (pNode->Id == 0)
            return 0;

    p_tmpNode = (Node *) (pNode->Parent[0]);
    distance = *M_pV(g_Matr, g_MatrixSize, p_tmpNode->Id, pNode->Id);

    while (p_tmpNode->Parent[0] != NULL)
        {
            pNode = p_tmpNode;
            p_tmpNode = (Node *) pNode->Parent[0];

            distance += *M_pV(g_Matr, g_MatrixSize, p_tmpNode->Id, pNode->Id);
        }

    return distance;
} /* End of CountDelay */

/******************************************************************
** The name of function:      _AddNode
** Input:     int iNodeID , Node* pNode
** iNodeID:   the ID of new comer
** pNode:     the requested node
** Output:    0——normal， 1——the inputting pointer is null， 2——fail to
allocate memory to new pointer
** Description:      join a new client
*****************************************************************/

int
_AddNode(int iNodeID, Node* pNode)
```

```
{
   int i, j;

   if (pNode == NULL)
         return 1;

   for (i = 0; i < MAX_CHILD; i++)
      {
         if (pNode->Child[i] == NULL)
            {
               j = i;
               break;
            }
      }

   (Node *) pNode->Child[j] = (Node *) (malloc(sizeof(Node)));
   ((Node *) (pNode->Child[j]))->Id = iNodeID;
   ((Node *) (pNode->Child[j]))->Type = 0;
   (Node *) ((Node *) (pNode->Child[j]))->Parent[0] = pNode;
   ((Node *) (pNode->Child[j]))->Path_Delay = CountDelay((Node *)

(pNode->Child[j]));

   for (i = 0; i < MAX_CHILD; i++)
      {
         ((Node *) (pNode->Child[j]))->Child[i] = NULL;
      }

   if (pNode->Child[j] == NULL)
         return 2;

   return 0;
} /* End of _AddNode */

/*****************************************************************
** The name of function:      AddToLinkList
** Input:     DLinkList** pHeader, Node* pNode
** pHeader:     the pointer of link table
** pNode:     the pointer of node that want to join the link table
** Output:     no
** Description:      add certain node into the link table
*****************************************************************/

void
AddToLinkList(DLinkList** pHeader, Node* pNode)
{
   DLinkList* p_tmpDLL;
   if (pNode != NULL)
      {
         p_tmpDLL = (DLinkList *) (malloc(sizeof(DLinkList)));
         (DLinkList *) (p_tmpDLL->pLayer_Info) = (DLinkList *)

(malloc(sizeof(DLinkList)));

         p_tmpDLL->pLayer_Info->pNode = pNode;
         p_tmpDLL->pLayer_Info->iLayer = -1;
         p_tmpDLL->next = *pHeader;
         p_tmpDLL->prior = NULL;
         if (*pHeader != NULL)
               (*pHeader)->prior = p_tmpDLL;
         *pHeader = p_tmpDLL;
      }
} /* End of AddToLinkList */

/*****************************************************************
** The name of function:      RemoveFromLL
** Input:      DLinkList** pHeader, Node* pNode
** pHeader:      the pointer of link table
** pNode:      the pointer of the node that we want to delete from the link table
** Output:      0——normal；   1——can not find the node
** Description:       delete a certain node from a given link table
****************************************************************/

int
RemoveFromLL(DLinkList** pHeader, int iNodeID)
{
    DLinkList* p_tmpDLL;

    p_tmpDLL = *pHeader;
    while (p_tmpDLL != NULL)
        {
            if (p_tmpDLL->pLayer_Info->pNode->Id == iNodeID)
                {
                    if (p_tmpDLL->next != NULL)
                        {
                            (DLinkList *) (p_tmpDLL->next)->prior = (DLinkList *)

(p_tmpDLL->prior);
                        }
                    if (p_tmpDLL->prior != NULL)
                        {
                            (DLinkList *) (p_tmpDLL->prior)->next = (DLinkList *)

(p_tmpDLL->next);
                        }
                    else
                        {
                            *pHeader = (DLinkList *) p_tmpDLL->next;
                        }
                    if (p_tmpDLL->pLayer_Info != NULL)
                        free(p_tmpDLL->pLayer_Info);
                    p_tmpDLL->pLayer_Info = NULL;
                    free(p_tmpDLL);
                    p_tmpDLL = NULL;
                    return 0;
                }
            p_tmpDLL = p_tmpDLL->next;
        }
    return 1;
} /* End of RemoveFromLL */

/*****************************************************************
** The name of function:       IsNodeFull
** Input:      Node* pNode

** pNode:      the pointer of the node that we want to detect
** Ourput:      0——this node is saturated；   1——unsaturated
** Description:       detect whether a node is saturated
****************************************************************/

int
IsNodeFull(Node* pNode)
{
    int i;
    for (i = 0; i < MAX_CHILD; i++)
        {
            if (pNode->Child[i] == NULL)
                    return 1;
        }
    return 0;
}   /* End of IsNodeFull */
```

```
/*****************************************************************
** The name of the function:        Add_Node
** Input:      int iNodeID,    Node* pNode, int istartID
** iNodeID:      the node ID of new comer
** pNode：          starting node
** istartID：        the ID of connected node
** Output:      0——normal，  1——the inputting pointer is null，  2——fail to
allocate the memory for new pointer
** Description:        add a new node
****************************************************************/


int
Add_Node(int iNodeID, Node* pNode, int istartID)
{
    int i, ret;
    DLinkList* p_tmpDLL;
    Node* tmp_Node;

    p_tmpDLL = NULL;

  /*    add the starting node and its children    */
    if (pNode->Id == istartID && IsNodeFull(pNode) == 1)
/*    the starting node is source and it is unsaturated    */
        {
            AddToLinkList(&g_pSeekHeader, pNode);
        }
    else
        {
            RemoveFromLL(&g_pSeekHeader, pNode->Id);
        }

    for (i = 0; i < MAX_CHILD; i++)    /*    add its children    */
        {
            AddToLinkList(&g_pSeekHeader, (Node *) pNode->Child[i]);
        }

    /*    find the closest node    */
    tmp_Node = FindMin(iNodeID, &g_pSeekHeader);
    if (IsNodeFull(tmp_Node) == 0) /*    saturated,  recursion    */
        {
            Add_Node(iNodeID, tmp_Node, istartID);
        }
    else/*    unsaturated, directly connect    */
        {
            ret = _AddNode(iNodeID, tmp_Node);
            if (ret != 0)
                    return ret;
        }
    FreeDLinkList(&g_pSeekHeader);

    return 0;
}    /* End of Add_Node */

/*****************************************************************
** The name of function:        Remove_Node
** Input:      int iNodeID,    Node* pNode
** iNodeID:    the ID of the node that we want to delete
** pNode：          starting node
** Output:    0——normal，  1——the inputting node pointer is null，  2——fail to
allocate memory for new pointer
** Description:        remove node
****************************************************************/
int
Remove_Node(int iNodeID)
{
    int i, j, ilayer, ih, il;
```

```
Node* pNode;
Node* pFatherNode;
DLinkList* p_tmpDLL, * pDlinklist;

pNode = NULL;
pDlinklist = NULL;

printf("\n\nNode ID = %d have left, all its descendant are disconnected;\n",
       iNodeID);
fprintf(g_result,
        "\n\nNode ID = %d have left, all its descendant are disconnected;\n",
        iNodeID);
/* search for the target node */
Traversal_Tree(g_NodeinTree[0], 0, &g_pHeader);

p_tmpDLL = g_pHeader;

while (p_tmpDLL != NULL)
  {
    if (p_tmpDLL->pLayer_Info->pNode->Id == iNodeID)
      {
        pNode = (Node *) p_tmpDLL->pLayer_Info->pNode;
        ilayer = p_tmpDLL->pLayer_Info->iLayer;
      }
    p_tmpDLL = p_tmpDLL->next;
  }
FreeDLinkList(&g_pHeader);
if (pNode == NULL)
    return 3;

if (pNode->Id != 0)
    pFatherNode = (Node *) pNode->Parent[0];
else
    pFatherNode = pNode;

/* find the offspring nodes of the target node */
Traversal_Tree(pNode, ilayer, &pDlinklist);
p_tmpDLL = pDlinklist;
j = 0;

while (p_tmpDLL != NULL)
  {
    j++;
    if (p_tmpDLL->pLayer_Info->pNode->Id == iNodeID)
      {
        if (p_tmpDLL->prior != NULL)
            p_tmpDLL->prior->next = NULL;

        if (p_tmpDLL->pLayer_Info != NULL)
            free(p_tmpDLL->pLayer_Info);
        p_tmpDLL->pLayer_Info = NULL;

        free(p_tmpDLL);
        p_tmpDLL = NULL;

        break;
      }
    p_tmpDLL = p_tmpDLL->next;
  }
for (i = 0; i < MAX_CHILD; i++)
  {
    if (((Node *) pNode->Parent[0])->Child[i] != NULL)
        if (((Node *) ((Node *) pNode->Parent[0])->Child[i])->Id == iNodeID)
            ((Node *) pNode->Parent[0])->Child[i] = NULL;
  }
free(pNode);/* delete the node */
```

```
pNode = NULL;
if (j == 1)
        pDlinklist = NULL;

ShowOnScreen();/*    show on screen the tree after node leaving    */
printf("\nAdd it's offspring the tree! \n");
fprintf(g_result, "\nAdd it's offspring the tree! \n");

/* add the offspring nodes */
p_tmpDLL = pDlinklist;
if (p_tmpDLL != NULL)
    {
        il = p_tmpDLL->pLayer_Info->iLayer;
        ih = p_tmpDLL->pLayer_Info->iLayer;
    }
while (p_tmpDLL != NULL)
    {
        if (p_tmpDLL->pLayer_Info->iLayer > ih)
                ih = p_tmpDLL->pLayer_Info->iLayer;
        if (p_tmpDLL->pLayer_Info->iLayer < il)
                il = p_tmpDLL->pLayer_Info->iLayer;

        p_tmpDLL = p_tmpDLL->next;
    }

for (i = il; i <= ih; i++)
    {
        p_tmpDLL = pDlinklist;
        while (p_tmpDLL != NULL)
            {
                if (p_tmpDLL->pLayer_Info->iLayer == i)
                    {
                        Add_Node(p_tmpDLL->pLayer_Info->pNode->Id, pFatherNode,
                                                pFatherNode->Id);
                    }
                p_tmpDLL = p_tmpDLL->next;
            }
    }

    FreeDLinkList(&pDlinklist);

    return 0;
} /* End of Remove_Node */

/*******************************************************************
** The name of function:        ReclaimMem
** Input:      no
** Output:      no
** Description:        release the allocated memory
*******************************************************************/

void
ReclaimMem()
{
    DLinkList* p_tmpDLL;

    Traversal_Tree(g_NodeinTree[0], 0, &g_pHeader);

    p_tmpDLL = g_pHeader;

    while (p_tmpDLL != NULL)
        {
            if (p_tmpDLL->pLayer_Info->pNode != NULL)
                {
                    free(p_tmpDLL->pLayer_Info->pNode);
                    p_tmpDLL->pLayer_Info->pNode = NULL;
            }
        p_tmpDLL = p_tmpDLL->next;
        }
    FreeDLinkList(&g_pHeader);
    /*   release the Matrix memory，close the output file   */
    fclose(g_result);
    if (g_Matr != NULL)
        free(g_Matr);
} /* End of ReclaimMem */

/*****************************************************************
** The name of function:      ShowOnScreen
** Input:    no
** Output:   no
** Description:      show all of the node in the tree
*****************************************************************/
void
ShowOnScreen()
{
    int i, j;
    int k;
    int Number_Layer[25];

    int Sum_Layer = 0;
    float Longest_Delay = 0;

    DLinkList* p_tmpDLL;

    for (k = 0; k < 26; k++)
        {
        Number_Layer[k] = 0;
        }
    Traversal_Tree(g_NodeinTree[0], 0, &g_pHeader);
    p_tmpDLL = g_pHeader;
    i = 0;

    printf("\n\nList all the nodes in the current tree! \n");
    fprintf(g_result, "\n\nList all the nodes in the current tree! \n");
    while (p_tmpDLL != NULL)
        {
        if (p_tmpDLL->pLayer_Info->pNode->Id == 0)
            j = 0;
        else
            j = (int)
                ((Node *)
                 ((Node *) (p_tmpDLL->pLayer_Info->pNode)->Parent[0])->Id);
        printf("Node ID = %d, Layer = %d Parent = %d Path_Delay = %f \n",
               p_tmpDLL->pLayer_Info->pNode->Id,
p_tmpDLL->pLayer_Info->iLayer,
               j, p_tmpDLL->pLayer_Info->pNode->Path_Delay);

        if (Longest_Layer < p_tmpDLL->pLayer_Info->iLayer)
            Longest_Layer = p_tmpDLL->pLayer_Info->iLayer;

        if (Longest_Delay < p_tmpDLL->pLayer_Info->pNode->Path_Delay)
            Longest_Delay = p_tmpDLL->pLayer_Info->pNode->Path_Delay;

        Sum_Layer = Sum_Layer + p_tmpDLL->pLayer_Info->iLayer;

        for (k = 0; k < 26; k++)
            {
            if (p_tmpDLL->pLayer_Info->iLayer == k)
                Number_Layer[k] = Number_Layer[k] + 1;
            }

        fprintf(g_result,
                  "Node ID = %d, Layer = %d Parent = %d Path_Delay = %f \n",
                  p_tmpDLL->pLayer_Info->pNode->Id,
p_tmpDLL->pLayer_Info->iLayer,
                  j, p_tmpDLL->pLayer_Info->pNode->Path_Delay);
        p_tmpDLL = p_tmpDLL->next;
        i++;
      }

  printf("\nThere are %d nodes in the tree! \n", i);
  fprintf(g_result, "\nThere are %d nodes in the tree! \n", i);
  fprintf(g_result, "\nThe deepest layer is %d ! \n", Longest_Layer);
  fprintf(g_result, "\nThe sum of layer is %d ! \n", Sum_Layer);
  fprintf(g_result, "\nThe longest delay    is %f ! \n", Longest_Delay);

  for (k = 0; k < 26; k++)
      {
        fprintf(g_result, "\nThe layer %d includes %d nodes ! \n", k,
                  Number_Layer[k]);
      }

  FreeDLinkList(&g_pHeader);
} /* End of ShowOnScreen */


/****************************************************************
** The name of function:      CountSum
** Input:      Node * pNode, int sum
** pNode:      calculation starting point
** Output :      the sum of distance
** Description:      calculate the sum of distance
****************************************************************/

float
CountSum(Node* pNode)
{
  int i;
  Node* ptmpNode;
  ptmpNode = NULL;
  if (pNode != NULL)
    {
      /*    traverse the tree    */
      for (i = 0; i < MAX_CHILD; i++)
        {
          ptmpNode = (Node *) pNode->Child[i];
          if (ptmpNode != NULL)
            {
              g_Sum = g_Sum + *M_pV(g_Matr, g_MatrixSize, pNode->Id,
                                    ptmpNode->Id);
            }
          CountSum(ptmpNode);
        }
    }
  return g_Sum;
} /* End of CountSum */

int
main(int argc, char** argv)
{
  int i, it;
  float sum;
  InitNode();
  it = 1;
  for (; g_ichoice > 0; g_ichoice--)
      it = it * 10;
  RandomFun(it, g_iseeds, 1);
```

```
    printf("\n Add Node ID: ");
    fprintf(g_result, "\n Add Node ID: ");
    for (i = 0; i < g_iseeds ; i++)
        {
            printf(" %d, ", g_pSelectArray[i]);
            fprintf(g_result, " %d, ", g_pSelectArray[i]);
            Add_Node(g_pSelectArray[i], g_NodeinTree[0], 0);
        }
    printf("\nTatal %d Nodes.\n", g_iseeds);
    fprintf(g_result, "\nTatal %d Nodes.\n", g_iseeds);
    ShowOnScreen();
    g_Sum = 0;
    sum = CountSum(g_NodeinTree[0]);
    printf("Sum = %f", sum);
    fprintf(g_result, "\nThe Sum = %f.\n", sum);
    ReclaimMem();
    ClearRandomArray();
    return 0;
}
```

Appendix2.

The codes for simulation our algorithm: algo1

```
/*****************************************************************
** The name of file: simu_algo1.c
** The date: 2004.07.07
** Description: The join/leave algorithm;
                          The statistic function;
** Version: 2004_06_28_DEBUG
*****************************************************************/
#include <stdio.h>
#include <stdlib.h>

/*    define invariabel    */
#define MATRIXSIZE 1000            /*     The size of distance Matrix    */

/* Nodes */
#define NODE_SOURCE      0
#define NODE_RECEIVER 10
#define MAX_PARENTS 1    /*    the number of the parent of a node    */
#define MAX_CHILD     3      /*    the maximum number of children of a node    */

/*    define the structure of Node    */
typedef struct _Node{
        int Id;                                          /*    Node Id    */
        float Path_Delay;              /*    Path Delay    */
        struct Node *Parent[MAX_PARENTS];   /* sender = Parents[0] */
        struct Node *Child[MAX_CHILD];

}Node;

/*    define the structure representing layer information    */
typedef struct _Layer_Info{
        int iLayer;                        /*    current layer    */
        int iFull;                         /*    the pointed node is full or not，0～MAX_CHILD
represent the iFull th node is empty，MAX_CHILD means full    */
        Node * pNode;         /*    the pointer that point to the Node structure    */
}Layer_Info;

/*    define double link table structure    */
typedef struct _DLinkList{
        Layer_Info * pLayer_Info;
        struct _DLinkList * prior, * next;
}DLinkList;


/*    define global variable    */
float g_Matr[MATRIXSIZE][MATRIXSIZE];
Node * g_NodeinTree[MATRIXSIZE];
int g_numberofnodeinTree;      /*    the total number of nodes in the tree    */
int g_MatrixSize;                     /*    the size of the distance matrix    */
float g_Sum;
int Longest_Layer=0;

int g_CurrentLayer = -1;           /*    recent filling layer number    */
DLinkList * g_pHeader = NULL;
FILE * g_result;

void initmatrix()
          {
                 int i,j,imaxchild;
                 FILE *fp;
                 fp=fopen("r1000.adj", "rb"); /* open a read only binary file */
                 g_result=fopen("result.txt","wt+");

if ( fread( &g_MatrixSize, sizeof(int), 1, fp) != 1 )
```

```
        {
          fprintf ( stderr, "Error: reading matrix size in");
          exit(21);
        }

             printf("       ----      Result    ----\n");
        fprintf(g_result, "      ----      Result    ----\n");

        imaxchild = MAX_CHILD;
        printf("\nMAX_CHILD = %d\n", imaxchild);
        fprintf(g_result, "\nMAX_CHILD = %d\n", imaxchild);

        printf("Loading Matrix Size: %d x %d\n", g_MatrixSize, g_MatrixSize);
        fprintf(g_result,  "Loading  Matrix  Size:  %d  x  %d\n",  g_MatrixSize,
g_MatrixSize);

for ( j = 0; j < MATRIXSIZE; j++ ) /* running on columns */
        {
          for ( i = 0; i < MATRIXSIZE; i++ ) /* running on lines */
        {
          if ( fread( &g_Matr[i][j], sizeof(float), 1, fp) != 1 )
              {
                fprintf ( stderr,"Error: reading at column=%d, line=%d\n", j, i );
                exit(24);
              }
        }
          printf(".");
        }

          for(i=0;i<MATRIXSIZE;i++)
           { for(j=0;j<MATRIXSIZE;j++)
               {
                 /* printf("Matr[%d][%d]=%f\n",i,j,g_Matr[i][j]);*/
                       /*    fprintf(g_result,"Matr[%d][%d]=%f\n",i,j,g_Matr[i][j]);*/
               }
           }
               fclose(fp);       /*    close the file */
        }

/******************************************************************
** The name of function:       InitNode
** Input:      No
** Output:     0---normal，  1---fail
** Description:    initialize the node
******************************************************************/

int InitNode()
{
        int i, j;

        for(i=0; i < MATRIXSIZE; i++)
        {
                g_NodeinTree[i]=NULL;
        }

        g_NodeinTree[0] = (Node *)(malloc(sizeof(Node)));
        if(g_NodeinTree[0] == NULL)return 1;

        g_NodeinTree[0]->Id = 0;
        g_NodeinTree[0]->Type = 2;
               g_NodeinTree[0]->Path_Delay = 0;
        g_NodeinTree[0]->Parent[0] = NULL;

        for(j=0; j < MAX_CHILD; j++)
        {
                g_NodeinTree[0]->Child[j] = NULL;
        }


        return 0;
}       /*      End of InitNode     */

/*****************************************************************
** The name of function:        FindMin
** Input:      float a[], int n
** a[ ]::    the array that need sorting
**n:      the size of array
** Output:      the order number of the smallest value
** Description:    find the smallest value
*****************************************************************/

int FindMin(float a[], int n)
{
        int i,k;
        k = 0;
        for(i = 1; i < n; i++)
        {
                if(a[i] < a[i-1])
                {
                        k = i;
                }
                else
                {
                        a[i] = a[i-1];
                }
        }
        return k;
}       /*      End of FindMin     */
```

```
/*****************************************************************
** The name of the function:        Traversal_Tree
** Input:      Node* pNode, int iLayer, DLinkList ** pHeader
** pNode:      the pointer of root node
** iLayer:        the layer of pNode，the layer number of source is 0
** pHeader:      to preserve the pointer of link table
** Output:      0——normal
** Description:        traverse the tree
*****************************************************************/

int Traversal_Tree(Node* pNode, int iLayer, DLinkList ** pHeader)
{
        int i;
        DLinkList * p_tmpDLL;

        if(pNode != NULL)
        {
                /*    scan the whole tree    */
                p_tmpDLL = (DLinkList *)(malloc(sizeof(DLinkList)));
                (DLinkList            *)(p_tmpDLL->pLayer_Info)            =            (DLinkList
*)(malloc(sizeof(DLinkList)));

                p_tmpDLL->pLayer_Info->pNode = pNode;
                p_tmpDLL->pLayer_Info->iLayer = iLayer;
                p_tmpDLL->next = *pHeader;
                p_tmpDLL->prior = NULL;
                if(*pHeader != NULL)(*pHeader)->prior = p_tmpDLL;
                *pHeader = p_tmpDLL;

                /* store the whole tree in the double linked list, not forget to free it */
                iLayer++;
                /*    continue to traverse    */
```

```
            for(i=0; i < MAX_CHILD; i++)
            {
                Traversal_Tree((Node *)pNode->Child[i], iLayer, pHeader);
            }
        }

        return 0;
}       /* End of Traversal_Tree */

/****************************************************************
** The name of function:      FreeDLinkList
** Input:      DLinkList ** pHeader
** pHeader:      preserve the pointer of link table
** Output:      no
** Description:      release the memory
****************************************************************/

void FreeDLinkList(DLinkList ** pHeader)
{
        DLinkList * p_tmpDLL;
        while(*pHeader != NULL)
        {
            p_tmpDLL = *pHeader;
            *pHeader = (*pHeader)->next;
            if(p_tmpDLL->pLayer_Info != NULL)free(p_tmpDLL->pLayer_Info);
            p_tmpDLL->pLayer_Info = NULL;
            free(p_tmpDLL);
            p_tmpDLL = NULL;
        }
}       /* End of FreeDLinkList */

/****************************************************************
** The name of function:      CountDelay
** Input:      Node * pNode
** pNode:      the node that we want to calculate distance
** output:      the path delay from 0 to node pNode
** Description:      the path delay from 0 to node pNode **  函数名:
        CountDelay
****************************************************************/

float CountDelay(Node * pNode)
{
        float distance;
        Node* p_tmpNode;

        if(pNode->Id == 0)return 0;

        p_tmpNode = (Node*)(pNode->Parent[0]);
        distance = g_Matr[p_tmpNode->Id][pNode->Id];

        while(p_tmpNode->Parent[0] != NULL)
        {
            pNode = p_tmpNode;
            p_tmpNode = (Node*)pNode->Parent[0];

            distance += g_Matr[p_tmpNode->Id][pNode->Id];
        }

        return distance;
}       /* End of CountDelay */

/****************************************************************
** The name of function:      _AddNode
** Input:      int iNodeID , Node* pNode
** iNodeID:    the ID of new comer
** pNode:      the requested node
** Output:    0——normal， 1——the inputting pointer is null， 2——fail to
allocate memory to new pointer
** Description:      join a new client
*************************************************************/

int _AddNode(int iNodeID, Node* pNode)
{
        int i, j;

        if(pNode == NULL)return 1;

        for(i=0; i < MAX_CHILD; i++)
        {
                if(pNode->Child[i] == NULL)
                {
                        j = i;
                        break;
                }
        }

        (Node *)pNode->Child[j] = (Node *)(malloc(sizeof(Node)));
        ((Node *)(pNode->Child[j]))->Id = iNodeID;
        ((Node *)(pNode->Child[j]))->Type = 0;
        (Node *)((Node *)(pNode->Child[j]))->Parent[0] = pNode;
        ((Node        *)(pNode->Child[j]))->Path_Delay        =        CountDelay((Node
*)(pNode->Child[j]));

        for(i=0; i < MAX_CHILD; i++)
        {
                ((Node *)(pNode->Child[j]))->Child[i] = NULL;
        }
        if(pNode->Child[j] == NULL)return 2;

        return 0;
}
/*************************************************************
** The name of the function:      Add_Node
** Input:     int iNodeID
** iNodeID:     the node ID of new comer
** Output:    0——normal， 1——the inputting pointer is null， 2——fail to
allocate the memory for new pointer
** Description:      add a new node
*************************************************************/

int Add_Node(int iNodeID)
{
        int i,j,k,l,t,x,ret,ideep;
        float * distance;
        DLinkList * p_tmpDLL;
        Node ** pNodetmp; /* the last two layers of the tree */

        distance = NULL;
        pNodetmp = NULL;



        Traversal_Tree(g_NodeinTree[0], 0, &g_pHeader);

        p_tmpDLL = g_pHeader;
        ideep = 0;
        while(p_tmpDLL != NULL)
        {
                if(ideep        <        p_tmpDLL->pLayer_Info->iLayer)ideep        =
p_tmpDLL->pLayer_Info->iLayer;
                p_tmpDLL->pLayer_Info->iFull = MAX_CHILD;
                for(j=0;j < MAX_CHILD;j++)
```

```
				{
					if(p_tmpDLL->pLayer_Info->pNode->Child[j]                    ==
NULL)p_tmpDLL->pLayer_Info->iFull = j;
				}
				p_tmpDLL = p_tmpDLL->next;
		}
		for(i = 0;i <= ideep;i++)
		{
			p_tmpDLL = g_pHeader;
			k = 0;
			while(p_tmpDLL != NULL)
			{
				if(p_tmpDLL->pLayer_Info->iFull      ==      MAX_CHILD      &&
p_tmpDLL->pLayer_Info->iLayer == i)k++;
				p_tmpDLL = p_tmpDLL->next;
			}
			t = 1;
			for(l = 0; l < i; l++)
			{
				t = t * MAX_CHILD;
			}
			if(k != t)
			{
				ret = i;
				break;
			}
		}
		if(ideep == 0)
		{
			ret = _AddNode(iNodeID, g_NodeinTree[0]);
			if(ret != 0)return ret;
		}
		else
		{
			j = t - k;
			pNodetmp   = (Node **)(malloc(j*sizeof(Node *)));
			distance = (float *)(malloc(j*sizeof(float)));

			p_tmpDLL = g_pHeader;
			k = 0;
			while(p_tmpDLL != NULL)
			{
				if(p_tmpDLL->pLayer_Info->iFull      !=      MAX_CHILD      &&
p_tmpDLL->pLayer_Info->iLayer == i)
				{
					pNodetmp[k] = p_tmpDLL->pLayer_Info->pNode;
					k++;
				}
				p_tmpDLL = p_tmpDLL->next;
			}
			for(t=0; t < k; t++)
			{
				x = ((Node *)(pNodetmp[t]))->Id;
				distance[t] = g_Matr[x][iNodeID];
			}
			x = FindMin(distance, k);
			ret = _AddNode(iNodeID, (Node *)pNodetmp[x]);
			if(ret != 0)return ret;


			if(distance != NULL)free(distance);
			distance = NULL;
			if(pNodetmp != NULL)free(pNodetmp);
			pNodetmp = NULL;
		}
		FreeDLinkList(&g_pHeader);
```

```
        return 0;
}

/*****************************************************************
** The name of function:       Delete_Node
** Input:     int iNodeID
** iNodeID:   the ID of the node that we want to delete
** Output:     0——normal，  1——the inputting node pointer is null，  2——fail to
allocate memory for new pointer, 3—— this node does not exist
** Description:      remove node
****************************************************************/

int Delete_Node(int iNodeID)
{
        int i,j,ilayer,ih,il;
        Node * pNode;
        DLinkList * p_tmpDLL, * pDlinklist;

        pNode = NULL;
        pDlinklist = NULL;

        printf("Node ID = %d have left, all its descendant are
                        disconnected;\n",iNodeID);

         /* search for the target node */
        Traversal_Tree(g_NodeinTree[0], 0, &g_pHeader);

        p_tmpDLL = g_pHeader;

        while(p_tmpDLL != NULL)
        {
                if(p_tmpDLL->pLayer_Info->pNode->Id == iNodeID)
                {
                        pNode = (Node*)p_tmpDLL->pLayer_Info->pNode;
                        ilayer = p_tmpDLL->pLayer_Info->iLayer;
                }
                p_tmpDLL = p_tmpDLL->next;
        }
        FreeDLinkList(&g_pHeader);
        if(pNode == NULL)return 3;

        /* find the offspring nodes of the target node */
        Traversal_Tree(pNode, ilayer, &pDlinklist);
        p_tmpDLL = pDlinklist;
        j = 0;

        while(p_tmpDLL != NULL)
        {
                j++;
                if(p_tmpDLL->pLayer_Info->pNode->Id == iNodeID)
                {
                        if(p_tmpDLL->prior != NULL)p_tmpDLL->prior->next = NULL;

                        if(p_tmpDLL->pLayer_Info                                        !=
NULL)free(p_tmpDLL->pLayer_Info);
                        p_tmpDLL->pLayer_Info = NULL;

                        free(p_tmpDLL);
                        p_tmpDLL = NULL;

                        break;
                }
                p_tmpDLL = p_tmpDLL->next;
        }
        for(i =0; i < MAX_CHILD; i++)
        {
            if(((Node *)pNode->Parent[0])->Child[i] != NULL)
                if(((Node *)((Node *)pNode->Parent[0])->Child[i])->Id == iNodeID)
                ((Node *)pNode->Parent[0])->Child[i] = NULL;
        }
        free(pNode);/* delete the node */
        pNode = NULL;
        if(j == 1)pDlinklist = NULL;

        /* add the offspring nodes */
        p_tmpDLL = pDlinklist;
        if(p_tmpDLL != NULL)
        {
            il=p_tmpDLL->pLayer_Info->iLayer;
            ih=p_tmpDLL->pLayer_Info->iLayer;
        }
        while(p_tmpDLL != NULL)
        {
    if(p_tmpDLL->pLayer_Info->iLayer > ih)ih = p_tmpDLL->pLayer_Info->iLayer;
    if(p_tmpDLL->pLayer_Info->iLayer < il)il = p_tmpDLL->pLayer_Info->iLayer;

            p_tmpDLL = p_tmpDLL->next;
        }

        for(i = il;i <= ih;i++)
        {
            p_tmpDLL = pDlinklist;
            while(p_tmpDLL != NULL)
            {
                if(p_tmpDLL->pLayer_Info->iLayer == i)
                {
                    Add_Node(p_tmpDLL->pLayer_Info->pNode->Id);
                }
                p_tmpDLL = p_tmpDLL->next;
            }
        }

        FreeDLinkList(&pDlinklist);

        return 0;
}       /* End of Delete_Node */

/*******************************************************************
** The name of function:      ReclaimMem
** Input:     no
** Output:     no
** Description:      release the allocated memory
*******************************************************************/

void ReclaimMem()
{
        DLinkList * p_tmpDLL;

        Traversal_Tree(g_NodeinTree[0], 0, &g_pHeader);

        p_tmpDLL = g_pHeader;

        while(p_tmpDLL != NULL)
        {
            if(p_tmpDLL->pLayer_Info->pNode != NULL)
            {
                free(p_tmpDLL->pLayer_Info->pNode);
                p_tmpDLL->pLayer_Info->pNode = NULL;
            }
            p_tmpDLL = p_tmpDLL->next;
        }
        FreeDLinkList(&g_pHeader);
}       /* End of ReclaimMem */

/*****************************************************************
** The name of function:        CountSum
** Input:      Node * pNode, int sum
** pNode:      calculation starting point
** Output :      the sum of distance
** Description:        calculate the sum of distance
*****************************************************************/

float CountSum(Node * pNode)
{
        int i;
        Node * ptmpNode;

        ptmpNode = NULL;

        if(pNode != NULL)
        {
                /* traverse the tree    */
                for(i=0; i < MAX_CHILD; i++)
                {
                        ptmpNode = (Node *)pNode->Child[i];
                        if(ptmpNode != NULL)
                        {

                        g_Sum = g_Sum + g_Matr[pNode->Id][ptmpNode->Id];
                        }

                        CountSum(ptmpNode);
                }
        }
        return g_Sum;
}       /* End of CountSum */

/*****************************************************************
** The name of function:        ShowOnScreen
** Input:      no
** Output:    no
** Description:        show all of the node in the tree
*****************************************************************/
void ShowOnScreen()
{
        int i,j;
               int k;
        DLinkList * p_tmpDLL;
                    int Number_Layer[25];

                      int Sum_Layer=0;
                        float Longest_Delay=0;

        for (k=0;k<26;k++)
                    {
                                    Number_Layer[k]=0;
                                }

        Traversal_Tree(g_NodeinTree[0], 0, &g_pHeader);
        p_tmpDLL = g_pHeader;
        i = 0;

        printf("\nList all the nodes in the current tree! \n");
        while(p_tmpDLL != NULL)
        {
                if(p_tmpDLL->pLayer_Info->pNode->Id == 0)j=0;
```

```
else                j                 =                 (int)((Node                *)((Node
*)(p_tmpDLL->pLayer_Info->pNode)->Parent[0])->Id);

printf("Node ID = %d, Layer = %d Parent = %d Path_Delay = %f \n",
                p_tmpDLL->pLayer_Info->pNode->Id,
                p_tmpDLL->pLayer_Info->iLayer,
                j,
                p_tmpDLL->pLayer_Info->pNode->Path_Delay
                );

if (Longest_Layer<p_tmpDLL->pLayer_Info->iLayer)
             Longest_Layer=p_tmpDLL->pLayer_Info->iLayer;


if(Longest_Delay<p_tmpDLL->pLayer_Info->pNode->Path_Delay)

Longest_Delay=p_tmpDLL->pLayer_Info->pNode->Path_Delay;
Sum_Layer=Sum_Layer+p_tmpDLL->pLayer_Info->iLayer;

          for (k=0;k<8;k++)
             { if(p_tmpDLL->pLayer_Info->iLayer==k)
                         Number_Layer[k]=Number_Layer[k]+1;
                       }

fprintf(g_result, "Node ID = %d, Layer = %d Parent = %d Path_Delay = %f \n",
                p_tmpDLL->pLayer_Info->pNode->Id,
                p_tmpDLL->pLayer_Info->iLayer,
                j,
                p_tmpDLL->pLayer_Info->pNode->Path_Delay
                );
```

```
          p_tmpDLL = p_tmpDLL->next;
          i++;
     }
     printf("\nThere are %d nodes in the tree! \n",i);
          fprintf(g_result, "\nThere are %d nodes in the tree! \n",i);

fprintf(g_result, "\nThe deepest layer is %d ! \n",Longest_Layer);
          fprintf(g_result, "\nThe sum of layer is %d ! \n",Sum_Layer);
          fprintf(g_result, "\nThe longest delay    is %f ! \n",Longest_Delay);

     for (k=0;k<10;k++)
             {
   fprintf(g_result, "\nThe layer %d includes %d nodes ! \n",k,Number_Layer[k]);
                        }

     FreeDLinkList(&g_pHeader);
}    /* End of ShowOnScreen */

int main (int argc, char **argv)
{
     int i;
     int Eventlist[] = {a eventlist we proposed};

     float sum;

          initmatrix();
     InitNode();

            for(i=0; i < 850; i++)
          {
          Add_Node(Eventlist[i]);
          }
```

```
        ShowOnScreen();

g_Sum = 0;
        sum = CountSum(g_NodeinTree[0]);
        printf("Sum = %f", sum);
        fprintf(g_result, "\nThe Sum = %f.\n", sum);
        fprintf(g_result, "\nAdd Node ID:");
        for(i=0; i < 50; i++)
        {
            /*fprintf(g_result, "%d,",Eventlist[i]);*/
        }
        ReclaimMem();
        return 0;
}
```