ROYAL INSTITUTE
OF TECHNOLOGY

TuxStream: A Locality-Aware Hybrid Tree/Mesh Overlay For Peer-to-Peer Live Media Streaming

MOHSEN HARIRI, MAHAK MEMAR

School of Information and Communication Technology (ICT)
The Royal Institute of Technology (KTH)
Supervisor: Amir H. Payberah
Examiner: Prof. Seif Haridi
December 2010
Stockholm - Sweden

TRITA-ICT-EX-2010:312

Abstract

Peer-to-peer (P2P) live media streaming has become widely popular in today's Internet. A lot of research has been done in the topic of streaming video to a large population of users in recent years, but it is still a challenging problem. Users must receive data simultaneously with minimal delay. Peer-to-peer systems introduce a new challenges: nodes can join and leave continuously and concurrently. Therefore, a solution is needed that is robust to node dynamics. Also, load of distributing data must be balanced among users so the bandwidth of all participating nodes in used. On the other hand, retrieving data chunks from the proximity neighborhood of the nodes leads to more efficient use of network resources.

In this thesis we present *tuxStream*, a hybrid mesh/tree solution addressing above problems. To achieve fast distribution of data, a tree of nodes that have stayed in the system for a sufficient period of time with high upload bandwidth is gradually formed. Further, we organize nodes in proximity-aware groups and from a mesh structure of nodes in each group. This way nodes are able to fetch data from neighbors in their locality. Each group has a tree node as its member that disseminates new data chunks in the group. To guarantee resiliency to node dynamics, an auxiliary mesh structure is constructed of all nodes in the system. If a fluctuation in data delivery happens or a tree node fails, nodes are able to get data from their neighbors in this auxiliary mesh structure. We evaluate the performance of our system to show the effect of locality-aware neighbor selection on network traffic. In addition we compare it with mTreebone, a hybrid tree/mesh overlay, and CoolStreaming, a pure mesh based solution, and show that tuxStream has better load distribution and lower network traffic while maintaining playback continuity and low transmission delays.

KEYWORDS: Live media streaming, Locality-awareness, Peer-to-peer systems, Push-pull solution.

Acknowledgments

First we want to thank our supervisor, Amir Payberah, who introduced the initial idea of tuxStream. He gave us consultation and helped us to summarize our work in a paper.

We are thankful to our examiner, Prof. Seif Haridi, for his advanced classes in peer-to-peer systems which provided us with the grounds for this thesis. We also gained an in-depth knowledge in distributed systems while participating in reading sessions in his research group.

We are also grateful to Dr. Jim Dowling for sharing his time and knowledge with us whenever we needed. We also wanted to thank Tallat Mahmood Shafaat for guiding us in our paper based on this thesis, and Cosmin Arad for answering all our questions regarding Kompics.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Streaming live media content to a group of clients in an scalable fashion over Internet has become popular in recent years. While a lot of research has been done in this topic, it is still a challenging problem. Applications with high quality of experience by users and minimal delay in receiving the stream are favored. Additionally, live media streaming requires a high bandwidth due to the large volume of data, so favored protocols are the ones with least overhead.

Traditionally, native IP multicast has been the preferred method for delivering data to a set of receivers in a scalable fashion. Although being efficient, it is not applicable over the Internet due to some political and technical issues [3].

Between application layer multicast topologies [4],[5],[6], peer-to-peer overlays are becoming a powerful paradigm for live media streaming considering their ease of large scale deployment. They also introduce new challenges [26]: as nodes can join and leave continuously and concurrently, called *churn*, a solution is needed that is robust to node dynamics.

Peer-to-peer solutions for live media streaming are classified into different categories. One solution is to construct a tree structure rooted at the source of the stream [7]. Although an efficient solution, it is highly vulnerable in presence of churn. In addition, a relatively small number of interior nodes carry the load of forwarding data, while the outgoing bandwidth of the leaf nodes is not utilized. Besides, placing a node with low bandwidth high in the tree results in waste of bandwidth for all of the downstream nodes. A multi-tree [8] structure is proposed as one of the solutions to cope with the shortcomings of a single tree structure. In this solution, the source splits the stream into sub-streams and multicasts each sub-stream using a separate tree. This increases the resiliency of the system to churn and improves potential bandwidth utilization of all nodes. Another category is mesh-based overlays [9],[10], where nodes dynamically establish peering relationships based on the

data-availability or bandwidth-availability of those nodes. Nodes periodically send data availability information to their neighbors and later fetch required data from them, considering the received information. As nodes may download/upload video from/to multiple nodes simultaneously, such systems are more robust to churn. The problem with these systems is that they suffer an efficiency-resiliency trade-off [11]. Another solution is a collaboration of tree and mesh forming an efficient and resilient hybrid overlay [12],[1]. The tree is used to achieve fast distribution of data, and the mesh is used to increase robustness to failure of nodes. To provide this robustness, nodes get data from a set of neighbors. There should be no partitioning in the overlay, so nodes have access to all data chunks in the system. To ensure this global connectivity, neighbors are selected from a random set of recommendations. Random neighbor selection makes it possible that two distant nodes (in terms of proximity) in the underlaying physical network become neighbors in the mesh. This relationship establishment has two disadvantages: (i) Data might travel two disjoint paths to reach two nodes that are in each other's locality. This results in a redundant and unnecessary traffic overhead in the physical network. (ii) Additionally, data chunks traveling unnecessary paths to reach destination results in an additional increase in the latency of content delivery.

Considering the problems addressed above, we need a system to achieve the following goals:

- Maximizes the quality of service experienced by user.
- Minimizes the end to end delay in the system.
- Increases robustness to node dynamics. Nodes can join and leave at any time. This should not result in any service interruption for other nodes in the system.
- Utilizes bandwidth of participating nodes by balancing the load of data dissemination in the system between all participating nodes.
- Reduces load on physical network by trying to get data from nodes that are closer regarding a proximity metric like network latency.

In this paper we present *tuxStream*, a peer-to-peer hybrid overlay for live media streaming that uses a tree to achieve fast distribution of data, and proximity-aware mesh structures, called *cluster*s, to increase resiliency to churn and reduce network latency. A tree structure rooted at the source of the stream is gradually constructed using stable nodes with higher upload bandwidth, called *super-node*s (Section 2.a). Clusters are gradually formed by other nodes around these super nodes (Section 2.b). Each node maintains

two partial views, one from the whole system, called a *global-view*, and the other from nodes in its cluster, called a *local-view*. Nodes in a cluster pull data from the super-node in that cluster and other nodes in their local-view. If a node observes any loss in data chunks that are close to their playback time, it initiates a gradual shift to pull data from nodes in its global-view. This data loss might happen in case of failure of the super-node in the cluster for non-super-nodes, and failure of parent for tree nodes. Details of how this data loss is detected and how gradual shift happens is explained in section 2.c.

We evaluate performance of tuxStream and compare it with mTreebone [1] and CoolStreaming [2], where the former is a collaboration of tree and mesh and the latter is a pure mesh-based solution, to show the effect of locality in our hybrid solution.

Chapter 2

Background and Related Work

IP multicast is an early solution to transmit packets to a subset of hosts. It uses routers as internal nodes of the multicast tree to forward data to end hosts in a scalable way. However it is not applicable due to some deployment issues [39].

Application layer multicast (ALM) approaches are proposed as another solution in delivering content to multiple users. Among these solutions, peer-to-peer (P2P) applications are proposed to multicast data to a group of users where each user acts as both receiver and sender of data in the system.

According to [26], peer-to-peer media streaming is challenging in two aspects: (i) finding supplying nodes and (ii) streaming topology.

In the following sections we describe existing approaches in solving these problems.

2.1 Finding supplying nodes

Locating a node in a peer-to-peer system with enough free bandwidth to get the stream with minimal delay is one of the main issues in these solutions. Some of the approaches addressing this issue are:

Centralized approach

In this method, information regarding all nodes' address and available bandwidth is stored in a centralized directory. Each node's join and leave is done through this directory server. Upon receiving a join request from a node, the global server returns addresses of supplying peer(s) to the node. Upon leave of a node or detecting its failure, it removes its membership information and introduces new supplier to other nodes connected to the failed node.

While it is simple and quick in replaying to join and failure of the nodes, this method has the draw backs of a single server system: it is not scalable; it can become overloaded with requests and is a single point of failure.

Hierarchical overlay approach

This approach is used in ZigZag 2.2.1 algorithm. In this method, nodes in the system are arranged into layers, where the lowest layer contains all the nodes in the system. In each layer, nodes are organized into clusters with a minimum and maximum size, according to a system property. In each cluster in layer L a node is selected as *cluster-leader* and becomes a member of a cluster in layer L+1. Cluster-leader is responsible for membership management of its cluster members. This layered clustering continues until reach a level that no more layers can be constructed on top of it due to minimum cluster size. The leader of the cluster in this topmost layer is called rendezvous point. Join of the nodes to the system are done through this rendezvous point which returns list of nodes in the level below it.

This approach overcomes the problems with centralized approach. As the load of the node membership management is distributed between participating nodes, one node is not overloaded. Also, there is no single point of failure in the system.

Distributed hash table based approach

In distributed hash tables (DHT), objects are stored as key-value pairs in one or multiple nodes in the system. The main operation in DHT systems is lookup, which takes a key and returns the corresponding value for that key. SplitStream 2.3.1 is a sample algorithm that uses DHTs to locate supplying peers. It builds multiple trees from nodes in the system. To locate parent of a node A in one of the multicast trees, it uses Pastry [34] to find a node B, which is the first node in the routing path of node A to multicast tree's ID.

Benefits of DHT approaches are that they are decentralized, scalable and fault tolerant.

Controlled flooding approach

Controlled flooding approach was first used by Gnutella [31]. In this approach, a node looking for a supplier sends a lookup message with a time to live (TTL) to its neighbors in the mesh overlay. Each neighbor upon receiving the message, checks if it satisfies the conditions of becoming the supplier. If so, it replies to the original sender of the message, otherwise after decreasing the

value of TTL by one, it forwards the request to its neighbors. This continues until the value of TTL becomes zero.

This method puts significant traffic on the network. In addition, based on the value of TTL nodes my not find the supplying node and query will fail and must be repeated again.

Gossip-based approach

In this approach, each node when joining the system gets a random partial list of all participating nodes in the system and becomes their neighbors in the mesh overlay. Node begins partnership relationships with a subset of its neighbors and periodically sends data availability information to its partners. Supplying nodes are found based on this data availability information exchange. CoolStreaming/DONET [5] uses this approach.

This method is scalable and performs well in presence of node dynamics.

In the following sections we describe existing topologies for live media streaming.

2.2 Push-Based solutions: Single Tree Structure

In single tree approach, nodes form a tree structure rooted at the source of the media stream. Each node gets data from its parent and forwards it to its children.

The main issues in this approach are the height (depth) of the tree and the number of children for internal nodes. As each node receives data from its parent, increase in depth of the tree results in higher delays in receiving data by bottom nodes. Therefore tree construction solutions try to decrease tree depth and consequently keep source to end host delay to a minimum. On the other hand, internal nodes' upload bandwidth is limited. Nodes can accept limited number of children considering their maximum upload bandwidth and streaming rate to be able to upload data to all their children without loss.

One of the major problems in single tree structure is their maintenance. Each node is connected to a single parent, therefore if a fluctuation happens in data delivery by the node's parent, either as a result of parent's leave or a problem in link between them, node and all its downstream nodes face data shortage. In peer-to-peer live streaming systems, users can join and leave arbitrarily at any time, so this problem can happen frequently. In this case, affected children and their downstream nodes are disconnected from the stream, while tree is recovering.

Another weakness of single tree approaches is their unfairness. Leaf nodes do not contribute any bandwidth to the system, while internal nodes carry all the bearing load of the system. As leaf nodes are a large fraction of system nodes, a great portion of potential upload bandwidth in the system is wasted.

In the following section we explain a single tree solution and how it addresses the weakness of this approach.

2.2.1 ZigZag

ZigZag[29] algorithm is an example of single tree solution which tries to address its shortcomings. ZigZag builds a tree structure from all of the nodes in the system with its height logarithmic to the total number of participating nodes. The node degree for internal nodes is a system parameter called k. Algorithm had two parts: administrative part and streaming part.

The administrative part of the algorithm organizes nodes in a hierarchy of clusters (shown in figure 2.1). The lowest layer contains all of the participating nodes, and arranges them in clusters with maximum size of $3k$. A node in each cluster in level L is selected as the head of that cluster (dotted nodes in figure 2.1). They become members of clusters in layer L+1. This administration organization does not infer the streaming topology. Meaning that nodes in each cluster do not get data from their cluster head.

To explain streaming part, first they define *foreign head* of a node as follows: if we consider that node N is a cluster mate of node A at layer L-1, they define *foreign head* of node N as a non-head cluster mate of node A at layer L>0. For example, node B is the foreign head of node N in figure 2.1.

The streaming is done based on the clustering in the administrative part. Nodes in each cluster deliver data from their foreign head rather than their cluster head. This means that node A at level L can only link to nodes that do not share the same cluster with node A at level *L-1*.

Altogether, each node has a cluster-head that is responsible for its placement in the administrative part and a parent that is responsible for pushing the stream to the node. This helps to increase resiliency of system to node dynamics: if a parent head of a cluster fails, its cluster head is responsible to find a new parent for members of that cluster.

The problem with ZigZag is that bandwidth of leaf nodes is not used, which is a weaknesses of single tree structures. On the other hand, it is still vulnerable to node dynamics as it takes time for a cluster-head to find a new parent for the affected children in case of failure of the parent.

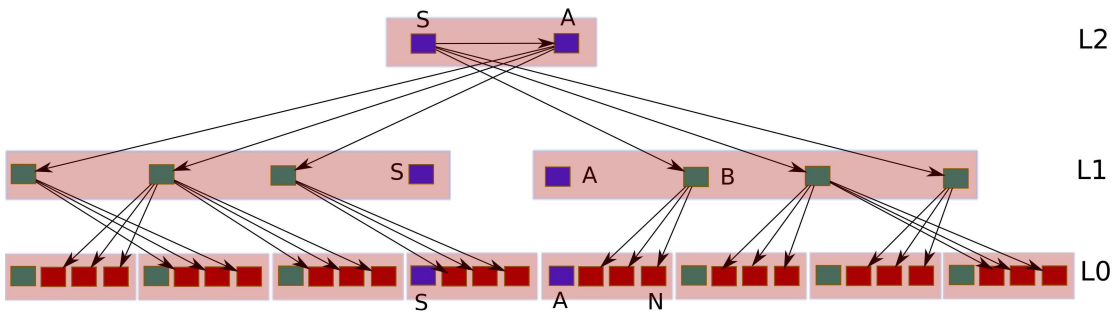


Figure 2.1: A model of ZIGZAG administrative organization and multicast tree with k=4

2.3 Push-Based Solutions: Multiple Tree Structure

Multiple tree structures overcome the weaknesses of single tree structures by building multiple trees(paths) between source of media stream and each receiver. The media stream in divided into sub-streams and pushed in each tree.

The main challenge in these kind of solutions is to build these trees in a way that an interior node in one tree is the leaf node in all other trees. This way, the burden of forwarding data is carried by a large portion of nodes in the system. As all of the nodes get to be an internal node in at least one of the trees and forward data chunks, the problem of bandwidth utilization in single trees is solved.

On the other hand, as each node is probably an internal node in one tree, failure of a node causes loss of one stripe of stream for a while. This should be handled by the selected encoding algorithm in a proper way. For example, by playing the stream in the receiver node in a lower quality.

Multiple tree structures seem to be resilient to node dynamics and utilize bandwidth of the nodes, while users experience a good play back continuity.

2.3.1 SplitStream

SplitStream [10] is an example of multi-tree streaming protocols. It divides stream into several stripes using a encoding application. For splitting the media stream Multiple Description Coding (MDC) [35] can be used that satisfies two requirements: (i) download of each stripe must require nearly equal

bandwidth, and (ii) nodes must be able to reconstruct content of the stream if they have received a sufficient subset of strips. It uses DHT for membership management of system nodes.

SplitStream uses Pastry [34] and Scribe [33] to build multicast trees. Pastry gives each node a random identifier, called *nodeId*. It routes a message toward nodes with nodeIds that share longer prefixes with the message key. Scribe gives each group of nodes a random key, called *groupId*. It builds a multicast tree by joining the Pastry routes from each member to the groupId's root.

SplitStream uses a separate multicast tree per stripe. It defines term *interior-node-disjoint* meaning that each node is an internal node in at most one of the trees and leaf node at other trees. To ensure this, groupIds of trees must be different in the most significant digit.A node must join all trees to receive the complete stream. In this case, when a node fails, only a portion of stripes are not delivered to its downstream nodes, and they still can play the stream but with lower quality.

A problem with SplitStream is that it does not guarantee that a node can not become an internal node in more than one of the trees. Therefore, a node with low bandwidth can become an internal-node in several trees and as a result become a bottle neck.

Another problem is that the number of children for each node in each multicast tree can be as high as its in-degree in Pastry, which can be higher than node's bandwidth. To overcome this, two *pushdown* and *anycast* processes are defined enabling nodes to reject/replace a child when number of children exceeds their bandwidth limit. The rejected/replaced child needs to find a new parent. This manipulated structure of DHT and constructs links that are not in the Pastry paths. Therefore these new links do not have the advantages of DHT-links: (i) route convergence in Pastry ensures that a message is routed to a member near its sender. While in these links this fact is not guaranteed. (ii) message paths in DHTs are loop-free, while these changes in tree structure might result in loops in the tree. The number of these links can increase in presence of node dynamics.

2.4 Pull-Based Solutions

Pull-based solutions are another category of solutions used in live media streaming. In these solutions, streaming content is divided into data chunks of equal size. Each node maintains a list of its neighbors and periodically exchanges data chunk availability information with them. Data chunks are pulled from neighbors by each node based on that information.

Taking advantage of this chunk availability gossiping, these solutions are

more resilient to peer dynamics. However, this chunk availability information exchange brings more delay in distributing data to all participating nodes. A node A must wait to receive a data availability information for a chunk from its neighbor node B, sent by node B at the end of its data availability exchange period. Upon receiving this information, node A will send a request for that chunk at the end of its scheduling period. If the values for data availability exchange period and scheduling period are too small algorithm puts an unnecessary overhead. On the other hand, bigger values for these parameters results in more delay in receiving data chunks. This term is usually referred to as efficiency-latency trade off.

PPLive [1] is an example of a successful real world system in this category [36].

Another successful deployment in this category is CoolStreaming/DONet [5], a data-driven and mesh-based solution that adopts a gossip-based protocol for membership management. Each node maintains and updates a partial list of system members. Data chunks are retrieved from system members by other nodes based on periodical data availability information exchanged between nodes. Details of how this system works is explained in section 2.6.2.

2.4.1 DagStream

As a pull-based solution, DagStream [30] is a multiparent, receiver-driven solution that addresses network connectivity and failure resiliency while considering locality-awareness in the system.

For membership management, a service called RandPeer [32] is used in making directed graphs of nodes, called *DAG*s. Nodes looking for a new parent send a look-up request to RandPeer and get address of a close by node in response. To join the system, node A gets address of a node B from RandPeer and checks if node B has a free place for a child. In that case node A adds node B as its parent and sets its level as $level_B + 1$. To maintain resiliency to failure, each node adds at least k parents which are located in lower levels of the tree than this node. It periodically looks for new parents and adds or replaces them, to keep at least k parents. When replacing a parent the ones with least delay and lowest level are preferred.

To stream the media each parent periodically sends data availability information to its children. Each child after applying a scheduling algorithm, sends requests for different data chunks to different parents. The scheduling algorithm takes into account two factors: data availability information sent from that parent and available bandwidth of that parent.

In DagStream, authors prove building locality-aware Dags by showing the reduced average parent-child delay using DagStream approach compared to

other parent discovery approaches. The evaluation results showing a good streaming quality for DagStream.

2.5 Push-Pull Solutions

Push-pull streaming solutions consider the fact that push-based and pull-based solutions are complementary in some aspects and by using a hybrid of these solutions, they manage to overcome their weaknesses. Push-based solutions have low overhead and low source to end delay, while pull-based solutions are robust to arbitrary join and leave of the nodes and use all the available bandwidth in the system.

mTreebone [4] is an example in this category. It introduces stable nodes to construct a tree-based backbone. Other non-stable nodes try to find a stable node and attach to it as its child. Moreover, to handle node dynamics and completely use available bandwidth of all overlay nodes, stable nodes together with non-stable-nodes form an auxiliary mesh overlay.The details of mTreebone approach are explained in section 2.6.1.

2.5.1 CliqueStream

CliqueStream [18] is a push-pull solution that addresses both failure resiliency and locality-awareness.

For membership management it uses a clustered hash table, eQuus [19], to organize nodes into proximity based clusters, called *cliques*. It assigns a unique id to each clique where cliques with nodes closer to each other in physical network have numerically adjacent ids. Cliques have a maximum size that is given as a system parameter. If join of a node to its closest clique causes its size to exceed this threshold, clique is split into two halves. The new clique occupies a new id that differs in one digit from the second half.

CliqueStream selects at least two stable nodes called *relay* and *backup-relay node* in each clique. These are the nodes with highest bandwidth, which have stayed more than a predefined threshold in the system. Data is pushed in a tree made of these stable nodes and nodes in cliques pull data from their partners in the same clique.

If a stable node fails, all the downstream nodes stop receiving the stream and initiate recovery of the link. The backup-relay node detects failed stable node and fulfills all its responsibilities. If both relay node and backup-relay node fail at the same time, then downstream nodes detect their failure after a certain amount of time. While facing an interruption in reception of the stream, each of these nodes re-join the stream independently. Therefore,

in scenarios with high churn, nodes still face interruptions in receiving data chunks.

2.5.2 Bullet

Bullet [16] is another tree/mesh solution in media streaming. It uses the tree structure to push disjoint segments of data in the system. A mesh structure is used, so nodes can retrieve missing segments from other members in the system. Therefore, it maximizes the amount of bandwidth delivered to each node in the system.

For streaming topology in Bullet, the source of media stream divides stream content into blocks and further to packets. Each parent in the overlay sends disjoint packets to each of its children considering their available bandwidth. Each node has to pull missing data from other nodes in the system using a mesh structure.

Bullet uses RanSub [37] to distribute data availability between nodes in the mesh. RunSub uses a tree structure and two *collect* and *distribute* messages to send a partial list of participating nodes to each node in the system. Bullet piggy backs data availability information to this message. Each node establishes peering relationship with limited number of nodes selected based on this periodical data availability information and keeps list of available blocks received from other nodes in a matrix. Nodes use the extra bandwidth to distribute data in the mesh that is formed as explained. Based on the information periodically updated in node's matrix, it send requests to other nodes to get the missing data.

2.6 Solutions in Details

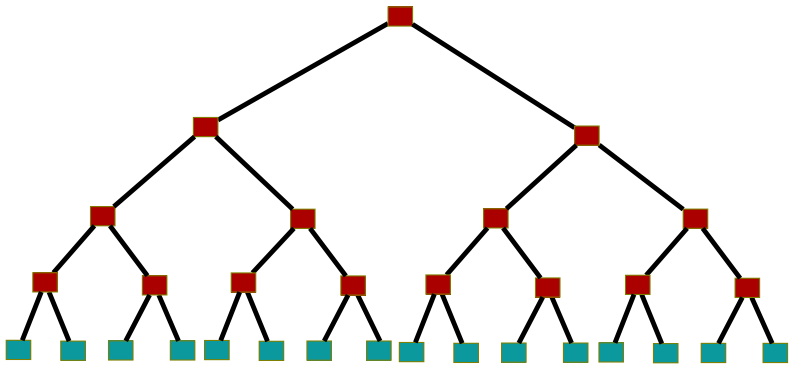
In the following sections we explain some solutions in details that we have used or referred to in our system. We explain mTreebone [4], a hybrid tree/mesh solution and CoolStreaming [5], a mesh-based protocol. We have implemented and compared both systems with tuxStream.

Later we refer to a survey and the results on join and membership durations of nodes in a peer-to-peer overlay.

2.6.1 mTreebone

Treebone: A Stable Tree-Based Backbone

The core of mTreebone [4] is a tree of stable nodes called *treebone*. Non-stable nodes in the system are attached to treebone nodes as their children.



–

Figure 2.2: Illustration of 2-array tree and number of internal nodes. mTreebone argues the existence of sufficient stable nodes,i.e treebone nodes, in each point in time in the system to support the whole overlay. Red nodes show internal nodes in a the tree.

Experiments done by authors show that there are enough stable nodes in the system at each point in time for building the treebone. For example if nodes in the system are organized in a K-ary tree, number of internal nodes in the tree is 1/K of all nodes in the system. This is illustrated in figure 2.2.

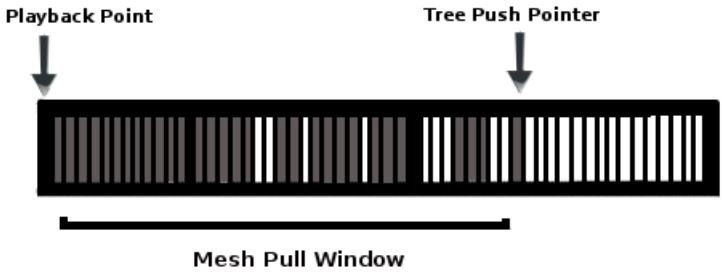
The studies done by authors suggest that this ratio of stable nodes are present in the overlay in each point in time.

Mesh: An Adoptive Auxiliary Overlay

In mTreebone in order to increase resiliency to node dynamics and improve bandwidth utilization of non-stable nodes, an auxiliary mesh overlay is used. Nodes keep a partial list of overlay nodes and update it using a light-wighted, random gossip algorithm. Neighbors in the mesh periodically send data availability information to their neighbors. If there is a data outage in the treebone, nodes fetch data chunks from their neighbors.

Joining of Nodes

When node A wants to join the system, it gets session-length, node's arrival time, a partial list of nodes in the system and address of a treebone node (e.g. node B) from the source. Node A tries to attach to the node B and initiates its mesh using the partial list received from the source.



–

Figure 2.3: mTreebone push-pull switch buffer

Treebone bootstrapping, Evolution and optimization

A function *T(t)* is defined that specifies the threshold of becoming a super-node according to t (node's arrival time). Each node periodically checks its own age in the overlay. Once it exceeds *T(t)*, node joins the treebone.

Considering the initiation period, the only treebone member is source. This is not efficient for fast data distribution. A randomized promotion algorithm increases number of treebone nodes in the initial period of the session. In this algorithm, each node can become a treebone node with the probability of *s/T(t)* where s is its current age.

When a node becomes a member of treebone it performs two periodical checks to optimize its place in the treebone:

- High-degree preemption: Each treebone node periodically checks number of its children to the number of other treebone nodes that are closer to source than itself. If node A finds a node B with less distance to the source and fewer children than itself, node A takes node B's place in the treebone and node B re-joins treebone.
- Low-delay jump: Each treebone node periodically checks for nodes that are closer to the source than its parent, and have a free place for new child. When a node finds such treebone node, it leaves its parent and adopts closer node as its own parent.

In case of failure of a treebone node, its children look for a new parent in the treebone. If affected child is a treebone node and it does not find another treebone node with free upload bandwidth for a child, it preempts the position of a non-stable node that is currently child of a treebone node. The non-stable node must find a new place in the tree.

Push-Pull Algorithm

In mTreebone data is pushed in the treebone. Nodes pull data from their neighbors in mesh overlay if they can not get required data chunks from their treebone parents. This can happen if a treebone node leaves the system. While its children are looking for another treebone node with free upload bandwidth for a child, they pull data chunks from their neighbors in the mesh.

On the other hand, when a node joins the system, it must look for a treebone node to accept it as its child. This might take time at the initiation period where number of treebone nodes are small. Therefore, newly joined node has to pull data from its mesh neighbors while it is looking for a treebone parent.

Another use of mesh neighbors is when a treebone node takes place of a non-treebone node while trying to join or re-join the treebone. Non-treebone node is detached from treebone for a while and uses its neighbors to get data chunks.

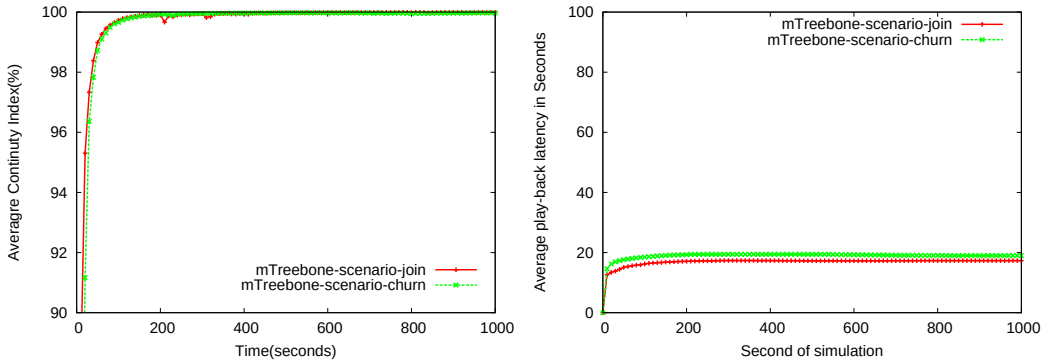
To manage this push-pull switching, an sliding mesh-pull window is defined. As shown in figure 2.3, mesh pull window contains data chunks between playback point and tree-push pointer. If a gap appears in these blocks, due to one of the above reasons, node pulls missing blocks through the mesh overlay. When a node is disconnected from treebone, its tree-push pointer is disabled. When it reconnects to the treebone, it resets its tree-push pointer to the first missing block and sends a request to its parent to push data chunks from that point.

Experiments

To check our implementation of mTreebone, we experimented our simulation with the same scenario defined in mTreebone. The results are provided in figure 2.4. The experiments are done considering a bandwidth uniformly distributed from 4 to 12 times of the bandwidth required for a full streaming. In this case, tree is formed very fast and non-stable nodes are attached to a treebone node to receive data.

In conclusion, the argument of mTreebone is:

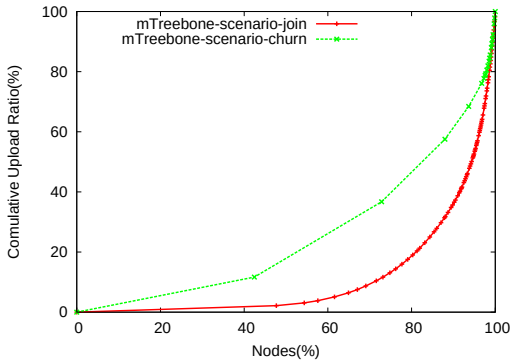
> There are generally enough stable nodes in each time instance for building a treebone. In fact, even a small set of stable nodes is sufficient to support the whole overlay. As an illustration, consider a simple K-ary tree of height H. The fraction of its internal nodes, i.e., those belong to the backbone, is no more than $1/K$ if the tree is complete and balanced.



(a) Continuity index (b) Playback Latency

Figure 2.4: Results of mTreebone experiments using the same scenario presented in the paper.

On the other hand, low values for bandwidth results in a non-balanced tree, which takes a lot of time to balance it using the *High-degree preemption* and *low-delay jump* algorithms. There fore, nodes mostly use mesh to pull data chunks from their neighbors. In our experiments using lower and more reliable values for bandwidth of the nodes show this weakness of mTreebone.



(a) Load distribution

Figure 2.5: Load distribution in mTreebone experiments using the same scenario presented in the paper.

As shown in figure 2.5, in join scenario nodes get most of the data from their parents in the tree, while in the churn scenario higher numbers of nodes are active in data dissemination.

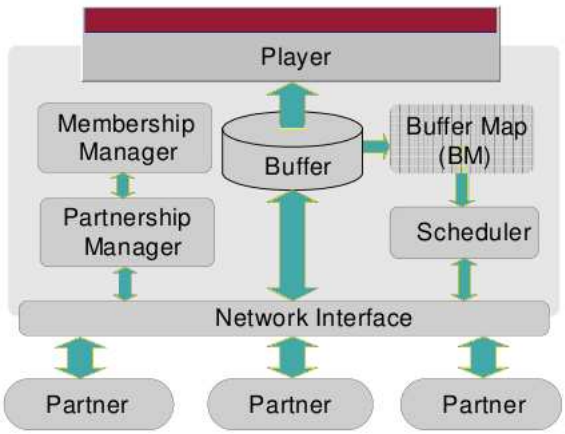


Figure 2.6: System Diagram for a CoolStreaming Node

2.6.2 CoolStreaming

In coolStreaming, media stream is divided into segments of equal length. Each node, except the source of media stream, can be either be provider or receiver of a segment. Availability of a segment in a node's buffer is represented by its buffer map. As shown in figure 2.6, a coolStreaming node has the following main modules:

Membership Management

Each node keeps a partial list of identifiers (e.g. IP addresses) of active nodes in the overlay, called *mCache*. Upon join of a node A to the system, it contacts the origin node (the source of the media stream) and gets the address of a node B, called *deputy node*. Node A then obtains a candid partner list from node B.

To keep this list updated a *membership message* is periodically sent by each node. This membership massage has four parts: <sequence-number, node-IP-address, number-of-partners, time-to-live>. mCache entry of a node is updated in three ways: (i) when a partners gets a membership message with a new sequence number, updates its mCache entry for the node with that IP address and saved the last update time of membership management message for that node. (ii) a deputy node adds the new node to its mCache entry in the join procedure. (iii) the intermediate nodes in the gossiping procedure, update their mCache entry for the node while, forwarding its membership

message. A gossip membership management can be used to distribute membership messages.

Partnership Management

Each node periodically exchanges its buffer map with its partners and schedules to pull unavailable segments from its partner. It is suggested in coolStreaming that if a segment represents 1-second of media stream, a buffer map containing 120 segments of video stream is enough. Each node periodically refreshes its partners with replacing a portion of them by new partners selected from its mCache. The nodes that have lower average of uploaded segments to this node are selected to be replaced by new ones.

Scheduler

In the scheduling algorithm in coolStreaming, when selecting a segment two constraints are considered: the playback time of that segment and the upload bandwidth of partners. In each scheduling period, number of suppliers for each segment is calculated. The rarest segments are selected first. If there are multiple suppliers for a segment, the one with highest bandwidth is selected.

When a node is replying to its partners' requests, segments are sent in order of their index. This increases the probability that these segments meet their playback deadline.

2.6.3 Join/Leave Behavior of Nodes in Live Media Streaming

The join and leave behavior of nodes in a streaming session are analyzed in [3][23][24][25]. Based on their statistics inter-arrival of nodes follows an exponential distribution and the membership durations for a non-stop live streaming session is based on a heavy-tailed Pareto distribution. These statistics give α values ranging from 0.7 to 2. Figure 2.7 shows the result of statistics from [23]. This distribution shows that most memberships are short, while a few nodes stay for a long period during streaming.

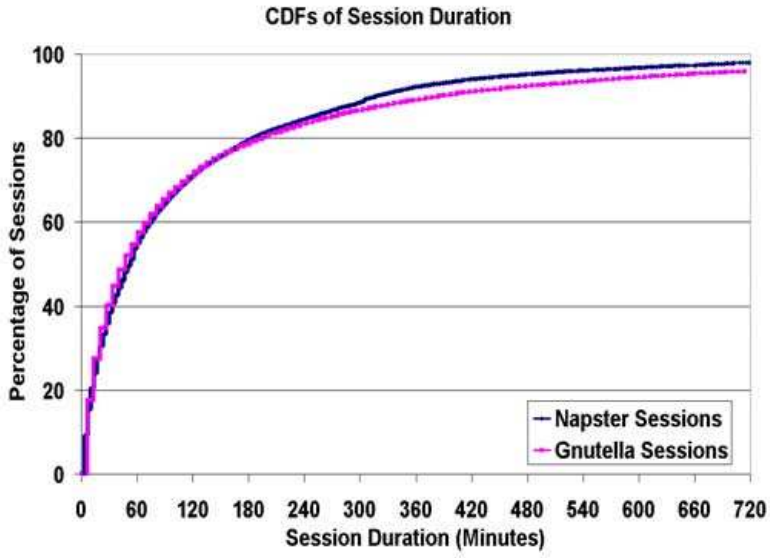


Figure 2.7: The distribution of Napster/Gnutella session durations

Figure 2.8 from [4] shows duration of nodes in the system monitored for CCTV3 and approximated with a Limited Pareto distribution with $\alpha = 1.03$ calculated as : $pdf(x) = \frac{\alpha L^{\alpha} x^{-\alpha-1}}{1-(\frac{L}{H})^{\alpha}}$.

The threshold from which we can guess a node is stable is very important. If the threshold is selected too small, nodes will be identified as stable that will not stay in the system for a long period. On the other hand, if the threshold is defined too high, the service time of stable nodes in the roles defined for them will be low. To calculate this optimal threshold of becoming a stable node we take the same approach as mTreebone [4] explained below.

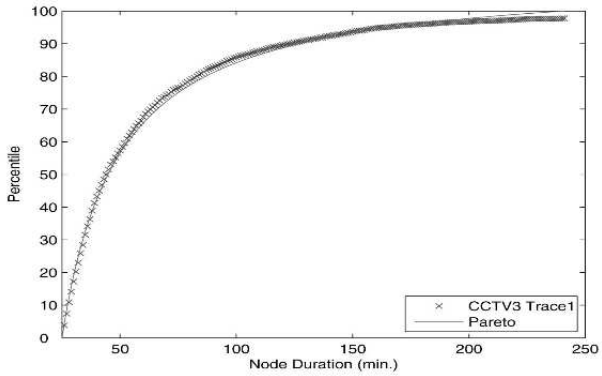
First we explain the following definitions:

- $EST(t)$: The estimated service time of a node that has joined the system at time t, after it is detected it as a stable node.
- $T(t)$: The age threshold of becoming a stable node.
- L : The session length.
- $f(x)$: The probability distribution function of node duration in the system. As explained, it is calculated using a *pdf* for bounded Pareto distribution.

The expected service time of node arrived at time t is the expected duration of the node minus the time it becomes a stable node:

$$EST(t) = \frac{\int_{T(t)}^{L-t} x f(x)\, dx}{\int_{T(t)}^{L-t} f(x)\, dx} - T(t)$$

Calculating the above function using the given *pdf* we can conclude that the optimal age threshold if the value of α is 1 is 0.3 of residual session length $(L - t)$.



–

Figure 2.8: The distribution of node duration approximated by a Pareto distribution.

2.6.4 Bandwidth Distribution

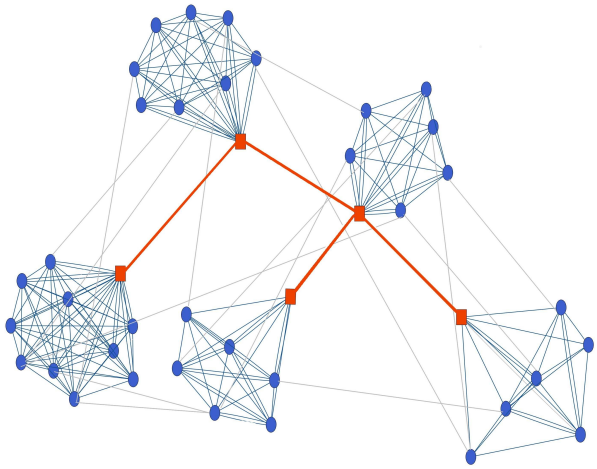
For bandwidth estimation in our scenarios we use statistics from dslreport.com, also used for statistics in [2]. They report results of speed tests voluntarily done in past two weeks. The summery of the results from approximately 9000 test is shown in table x. Based on this information 1% of the nodes in the system have upload bandwidth between 3500 Kbps to 6500 Kbps, which is the minimum and maximum bandwidth of becoming super-peer. Other nodes have the upload bandwidth of 200 Kbps to 2500 Kbps which is the minimum and maximum bandwidth of non-super-peers in our system. We consider download bandwidth of each node 4 times of its upload bandwidth.

Chapter 3

Architecture Of TuxStream

In this section we present the details of *tuxStream*. The contributions of the proposed solution are :

- Maximizing quality of service experienced by users. The important thing from the users point of experience no interruptions while playing the stream.
- Minimizing playback latency in the system by fast distribution of data in the system. In a live streaming system, it is important that the latency



–

Figure 3.1: A view of tuxStream with 5 super nodes in the tree with their clusters. Nodes in the clusters form local-mesh structures. Light lines between clusters show the global-view of the nodes in each cluster.

between the playback time of the source of media stream and users is kept minimum.

- Utilizing upload bandwidth of all participating nodes, such that a few nodes do not carry most of the load of data dissemination, while upload bandwidth of other nodes is not used.

- Increasing resiliency to churn. Arbitrary join and failure of nodes must not decrease quality of service experienced by other participating nodes.

- Considering properties of the underlying physical network while constructing overlay of nodes. This results in nodes fetching more data chunks from nodes in their locality than other nodes in the overlay. The locality-awareness results in traffic reduction on the physical network.

To achieve these goals, we propose a hybrid tree-mesh solution: (i) To facilitate fast dissemination of data in the system and consequently good playback continuity, a tree of participating nodes is constructed (shown in dark lines in figure 3.1). (ii) To utilizes upload bandwidth of participating nodes, all system nodes are members of mesh structures. To accomplish this clusters of nodes are gradually formed around each tree node, which we also refer to as *cluster-head*. Nodes in each cluster pull data from their cluster-mates using a mesh structure. (iii) To increase resiliency to node dynamics there are two modifications. First one is that only nodes which are expected to stay for a long time in the system are allowed to join the tree structure (square-shaped nodes in figure 3.1). This decreases the probability of changes in the tree structure due to node dynamics. On the other hand, all of the participating nodes in the system form a mesh structure. This enables them to pull data from other nodes in the system in case they can not fetch it from nodes in their locality. (iv) To obtain locality-awareness, clusters are formed based on a proximity metric, e.g. latency. This means that nodes in the system try to find cluster-heads that are closer to them and join their mesh structure to pull data from the nodes in their locality.

We consider a single dedicated *source node*, that stays connected during the streaming session. Source node generates the stream and divides it into pieces of equal size, called *chunk*s. Chunks of the media stream are generated by the source and pushed in the tree and disseminated in the clusters.

In the following section, we first explain the algorithm used in overlay construction. In this part we explain how the tree and mesh structures are gradually formed. We also explain algorithms to deal with join and failure of the nodes. In the end, the gradual optimization algorithm is described to show how nodes find the optimal cluster in the system.

In the next section, the push-pull algorithm is explained. We explain how pull algorithm works and how nodes handle node dynamics. In this section the algorithms for chunk selection and chunk distribution are explained.

3.1 Overlay Construction

In this section the details of the gradual construction of tree structure and the gradual formation of locality-aware mesh structures is explained.

3.1.1 Stable-nodes and Super-nodes

In this system, we identify potential *stable node*s. Stable nodes are nodes with long lifespan that tend to stay longer in the system. The idea of using stable nodes is supported by Wang et al. [20]. They conduct studies both on real traces and analytical models, and show the existence of one or multiple representative backbone trees in a mesh structure. The majority of data is delivered through this backbone tree. Nodes in this backbone tree are mainly stable nodes.

In our framework we identify stable nodes with upload bandwidth higher than a specific threshold, called *super-node*s. This bandwidth limit is given as a system parameter and is a product of streaming rate. The reason for using a bandwidth constraint is the relaying role we assign to these super-nodes: These super-nodes are organized into a tree structure rooted at source; so each super-node, as a member of the tree, must push data chunks it has received from its parent to its children. On the other hand, each node in this tree structure has a cluster around it that is gradually formed with nodes in the locality of that super-node. Super-nodes must be able to dedicate sufficient bandwidth to their neighbors. If a super-node dedicates all of its bandwidth to its children, and does not preserve free upload slots to reply to the requests of its neighbors in the local mesh structure, nodes in its cluster have problem maintaining their playback continuity. Consequently, each super-node has two roles: one as a tree node to push data to its children and the other as a cluster-head to disseminate data in its cluster. As a result, a super-node must be able to dedicate enough bandwidth to its children and still have enough free upload bandwidth to send chunks to its neighbors in the mesh structure. We have performed experiments on the value of this bandwidth limit threshold. We consider this threshold a product of streaming rate. This threshold has two parts: (i) one part is the fraction we expect it to be dedicated to super-node's cluster. (ii) The second part indicates the minimum number of children we

expect from a super-node to guarantee that the tree of super-nodes does not become a chain of nodes.

3.1.2 Global-Mesh Structure

Each node in the system manages to keep a partial view of all of the nodes in the system, called *global-view*. This view can be achieved and updated by running a light weighted peer sampling algorithm like Cyclon [21]. The global-view is a partial view of all nodes in the system, regardless of their distance to the node. Nodes use this view to construct a *global-mesh* structure.

3.1.3 Formation of Tree of Super-nodes

When the system bootstraps, the source is the only super-node. Later, a periodic check by each node determines whether a node is qualified to become a super-node considering its age and bandwidth. If a node is eligible, it joins the backbone tree formed with the source as its root. First, a node joins as a child of a super-node, which is selected from its view. Later, the node periodically checks the possibility of moving up in the tree. A child with higher bandwidth swaps its place with its parent. As we have not implemented any aggregation algorithms to check the size each super-node's cluster, we consider the fact that a super-node with higher bandwidth can disseminate data faster in the system. Therefore it is better to place it closer to the source of media stream.

In the process of promoting a non-super-node to super-node, to keep the tree structure consistent, a node informs its new parent of the status of its buffer. The new parent begins to push data to that child, in a rate higher than streaming rate, until they reach the same status of buffer.

3.1.4 Formation of Clusters and Local Mesh Structures

All nodes in the system keep monitoring their global-view to find new super-nodes in the system. After formation of the tree of super-nodes, with this monitoring, nodes find the address of new super-nodes in the system. When a node A finds a new super-node in its global-view, node A checks the distance of the newly found super-node to itself. If new super-node is closer to node A than its current cluster-head, which is the source of media stream, node A joins the cluster with new super-node as its head.

Each node keeps a partial view of all the nodes that share the same cluster with this node, called *local-view*. When there is no super-node other than the source in the system, local-view and global-view of all nodes consists of

nodes from same cluster. As new super-nodes and consequently new clusters are formed, each node keeps nodes with same cluster-head as its local-view members. Nodes in each cluster construct a mesh structure over each cluster called *local-mesh* structure.

The distance between nodes can be network latency (round trip time), hop counts or even certain economic cost of the path between two nodes. We choose latency as the distance metric in our simulation.

3.1.5 Joining of the Nodes

When a node A wants to join the system, it has to know the address of a node B that is currently a member of the system. Node A sends a request to node B asking for its local-view and cluster-head. Afterwards, node A joins the same cluster as node B. Although this cluster-head may not be the closest super-node, not looking for the optimal cluster to join in the beginning decreases start-up delay. As shown in algorithm 1, node A joins node B's cluster by adopting its local-view and cluster-head as its own (lines 4-5). Subsequently, node A begins membership management to update its local-view and global-view, updates its partners based on these two views and participates in data distribution. Nodes gradually try to find the optimal cluster.

Algorithm 1 Node A's Join Procedure Through Node B

1: **procedure** JOIN
2: **Send** ReqJoin **to** b
3: **Recv** $bLocalView, bClusterHeadAddress$ **from** b
4: $aLocalView \leftarrow bLocalView$
5: $aClusterHeadAddress \leftarrow bClusterHeadAddress$
6: **trigger** $< InitiateMembershipManagement >$ ▷ Initiates Cyclon to update node's local-view and global-view.
7: **trigger** $< InitiateRefreshPartnership >$ ▷ Periodically refresh node's partner-list based on its local-view and global-view (algorithm 4).
8: **trigger** $< InitiateGradualOptimization >$ ▷ Periodically check for a closer super-node to join its cluster (algorithm 2).
9: **trigger** $< InitiateDataDistribution >$ ▷ Begin the periodical Buffer-map exchange and chunk selection and chunk distribution algorithms
10: **end procedure**

3.1.6 Gradual Optimization of Clusters

As mentioned previously, nodes join the first cluster introduced to them by the node they have joined through, and begin to fetch data from their neighbors. Meanwhile, they improve their location in the overlay by trying to find the optimal cluster. To do so, nodes periodically look for new super-nodes in their global-view by checking the the cluster-head of their neighbors in their global-view. In the algorithm 2, GetNewSuperNodes() finds potentially close super-nodes (line 2). They check the distance of the newly found super-nodes with their distance to their own cluster-head. If the new super-node is closer to them than their own, node moves to the closer cluster. In algorithm 2, GetRTT() function returns the delay between the two nodes. Moving from one cluster to another cluster is done by setting the closer super-node as the node's cluster-head and choosing the new cluster-head's local-view as node's local-view (lines 9-12 in algorithm 2).

This periodical check also has another benefit. When a node is checking the latency of the link between itself and its cluster-head, in case of failure of the cluster-head, it detects the failure and prefers any super-node to its old cluster-head. As Node selects the closest super-node between all newly found super-nodes (lines 4-8 in algorithm 2), its will join the closest possible cluster.

Algorithm 2 Gradual Optimization Algorithm

```
1: procedure GradualOptimization
2:     superNodes ← GetNewSuperNodes(globalView)
3:     closestSuperNode ← currentClusterHead
4:     for all node_i ∈ superNodes do
5:         if GetRTT(node_i) < GetRTT(closestSuperNode) then
6:             closestSuperNode ← node_i
7:         end if
8:     end for
9:     if closestSuperNode ≠ currentClusterHead then
10:        currentClusterHead ← closestSuperNode
11:        localView ← closestSuperNode.GetLocalView()
12:    end if
13: end procedure
```

3.1.7 Failure of the Nodes

If a tree node fails, it can affect data delivery to both its cluster members and downstream nodes. Failed node's children detect the failure and re-join

the tree structure. The re-join algorithm is the same as join algorithm with the orphan node sending a request to be accepted as the child of a super-node that it has found in its local-view or global-view. To accelerate re-join process, orphan node tries to join super-nodes that are cluster-heads of other non-super-nodes in its global-view. Meanwhile, nodes with a failed cluster-head try to find a new cluster using the gradual optimization algorithm (algorithm 2).

On the other hand, if a non-super-node fails, it is deleted from its neighbors' views by the gossip algorithm used for membership management. We use Cyclon [21] as the membership manager in our implementation. In Cyclon, nodes periodically exchange their partial views of the system with another nodes, called *shuffling*. When a node A begins to shuffle with another node B in its view, if that node does not reply in a predefined period, node A will remove node B from its view. This way, failed nodes are gradually removed from local-view and global-view of other nodes.

3.2 Push-Pull Data Dissemination

For data dissemination in this system a push-pull algorithm is employed. The media source pushes data in the tree of super-nodes. Each super-node pushes delivered data chunks to its children and disseminates them in its cluster. Below we explain the pull algorithm in details that is similar to CoolStreaming [5].

3.2.1 Buffer-map

As explained, the media source divides stream into the pieces of equal size called data chunks. A node informs its partners of the availability of a chunk in its buffer using *buffer-map*. Buffer-maps are sent by each node to its partners periodically. A buffer-map can be a sliding window of 160 data chunks.

3.2.2 G-percent and High-priority Set

There is a specified number of chunks after the playback point whose availability is crucial to maintain playback continuity. This means, if node has not been able to get those chunks from its partners, there must be a problem and there is a risk that those chunks will not be delivered before their playback time. We define this specified number of chunks after the playback point as *high-priority* set.

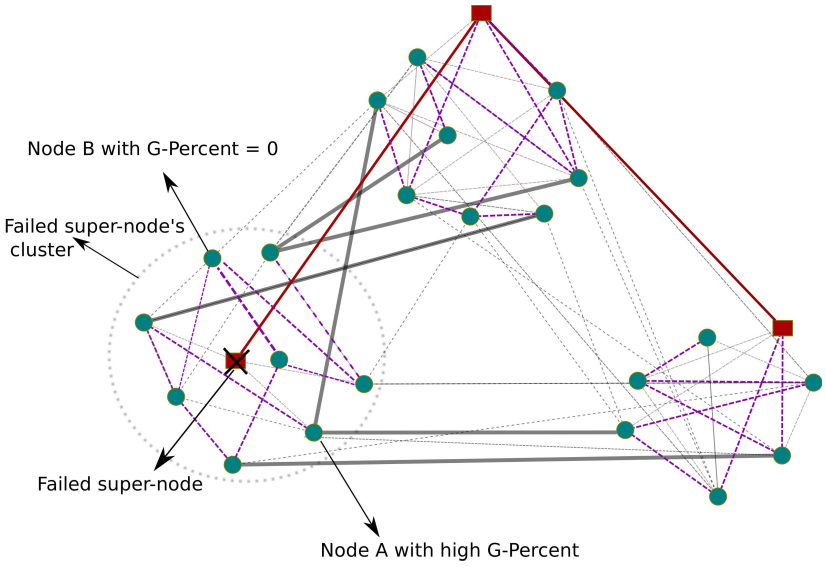


Figure 3.2: GPercent: nodes in the cluster with failed super-node try to fetch data chunks from other nodes in their global-view

Streaming of data to a node can be interrupted due to failure of its parent or the cluster-head. This results in missing data chunks in high-priority set. In this case the node must be able to get required chunks from other nodes in the system. To guarantee playback continuity, we define a parameter called *G-percent*. This parameter indicates what percentage of partners are selected from the neighbors in local-view and what percentage from global-view of the node.

For example, if the value of G-percent is 20 and size of partner-list is 10, it means that 2 out of 10 partners are selected from global-view and others from local-view. Periodical calculation of GPercent is shown in algorithm 3. The value of this parameter is initially set to *MinimumGPercent*. The reason for mentioning a MinimumGPercent value is to decrease the probability of removing a node with high GPercent from its global partners' partner-list when periodically refreshing partner's partner-list. If there exists a missing chunk in the high-priority set, the node increases the value of G-percent (line 4). On the other hand, if there is no missing chunk in this set, the node decreases the value of G-percent considering that its value must not be less than MinimumGPercent (lines 7-9). The amount of changes in G-percent, *GPercentIncrementalParameter*, indicates the eagerness of the system to use the global-view in case of node experiencing potential data loss.

In figure 3.2 the light dashed lines show both local-mesh and global-mesh overlays. The dark dashed lines show partnership with local-view members and dark solid gray lines show partnership between global neighbors. As we can see, nodes in the cluster with failed super-node have more global nodes as their partners than nodes in the two other clusters. It is useful to mention that some nodes in this cluster do not increase their GPercent value as other nodes fetch new data chunks from their neighbors in global-view and disseminate them in the cluster. For example, node B's partners are all from its local-view as other nodes (e.g node A) will pull data chunks from their partners from global-view and later send them to their partners in local-view.

Algorithm 3 G-Percent Calculation

```
1: procedure CALCULATE GPERCENT
2:     if buffer.highPrioritySet.HasMissedBlocks() then
3:         if gPercent < 100 then        ▷ Maximum value for G-Percent is 100
4:             gPercent ← gPercent + GPercentIncrementalParameter
5:         end if
6:     else
7:         if gPercent > MinimumGPercent then
8:             gPercent ← gPercent − GPercentIncrementalParameter
9:         end if
10:    end if
11: end procedure
```

3.2.3 Partner List and Resiliency to Free-riders

Nodes are members of two mesh structures in this system. To pull data from their neighbors in the mesh structures, a *partner-list* is formed from members of each node's local-view and global-view. To omit a partner from partners list, node refreshes its partner-list periodically by keeping most of the partners the same and replacing a portion of them with new ones selected from its local-view or global-view. In algorithm 4, new number of members of local-view and global-view in partner-list is calculated and extra members are removed. If gPercent is the same and no partner is removed based on its locality, node makes sure that at least "PartnerRefreshThreshold" number of partners are removed and replaced by new partners (lines 6-15).

To increase resiliency to free riders, a node gives a score to each partner based on the number of uploaded chunks from that partner in that period. When refreshing its partners, the node selects partners with lower score to be replaced by new nodes selected from one of its views. The

three functions RemoveWeakestLocalPartner(), RemoveWeakestGlobalPartner() and RemoveWeakestPartner() in algorithm 4 first sort the partners list based on each partner's score and then omit the ones with least score. By this algorithm, a free-rider node is gradually omitted from partner-list of other nodes in the system. It is possible that node does not strive, because it has the chance of being randomly selected to be added to partner-list of a node. But it is soon replaced, as it does not have a high score.

Algorithm 4 Refresh Partnership Algorithm

```
1: procedure RefreshPartnership
2:     gPercent ← CalculateGPercent()          ▷ Explained in algorithm 3
3:     pl ← partnerList
4:     newGlobalPartnersSize ← pl.size() * gPercent/100
5:     nGPS ← newGlobalPartnersSize
6:     newLocalPartnersSize ← pl.size() − nGPS
7:     nLPS ← newLocalPartnersSize
8:     while pl.GetLocalPartners().Size() > nLPS do
9:         RemoveWeakestLocalPartner()
10:    end while
11:    while partners.GetGlobalPartners().Size() > nGPS do
12:        RemoveWeakestGlobalPartner()
13:    end while
14:            ▷ at least 'PartnerRefreshThreshold' number of partners must be
   refreshed
15:    minRefreshValue ← MaxPartnerSize − PartnerRefreshThreshold
16:    while pl.Size() > minRefreshValue do
17:        RemoveWeakestPartner()
18:    end while
19:    while pl.GetLocalPartners().size() < nLPS do
20:        pl.Add(localView.GetRandomPartner())
21:    end while
22:    while pl.GetGlobalPartners().Size() < nGPS do
23:        pl.Add(globalView.GetRandomPartner())
24:    end while
25: end procedure
```

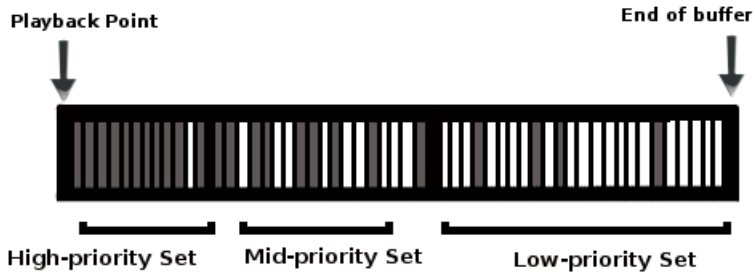


Figure 3.3: tuxStream Buffer-map: chunk selection strategy.

3.2.4 Pull Algorithm

A node periodically sends its buffer-map to its partners to inform them of its available data chunks. To pull data from its neighbors, a node sends a request for missing data chunks to its partners based on the received buffer-maps in that period.

3.2.5 Handling node dynamics

All of the nodes in the system, regardless of being a member of the tree or mesh, periodically monitor their high-priority set. In this case, they find some undelivered chunks in this set and increase the value of G-Percent, which results in an increase in the number of partners selected from the global-view. Consequently, the node begins to pull data from its global neighbors. In other words, nodes begin to gradually pull data chunks from their global-view members if they observe any gap in receiving the stream. We have to consider the fact that, if any of the nodes in a cluster fetches requested chunks, it disseminates them in the local-mesh structure; therefore, some nodes in the system may not get chunks from nodes in their global-view as other members in the cluster have already pulled them.

A non-super-node's failure does not have any effect on the continuity index of its neighbors as they continue to pull data chunks from other partners.

3.2.6 Chunk Selection Policy

Each node saves the buffer-maps sent to it from its partners in each period. Based on these buffer-maps, at the end of that period, node sends requests to fetch available data chunks from its neighbors. Chunk selection strategy indi-

cates the order of selection of chunks between these available chunks. When deciding this order two fast are important. It is important in live media streaming that user experience a continuous play back. To Achieve this, nodes must request data chunks in order. Therefore, there is a high probability that they are received before their play back time.

On the other hand, as described, nodes give a score to their partners based on the number of data chunks they have uploaded to them and will decide on keeping o removing a partner based on this score. Therefore, it is important for nodes to obtain data chunks needed by other nodes in the system.

To satisfy above goals for chunk selection policy, similar to give-to-get [22], we divide the buffer-map into three sets (figure 3.3):

1. *High-priority* set: Including data chunks i where *playback-point*$<i<$*playback-point* $+ h$. h is equal to the size of previously defined high-priority set (section 3.2.2).

2. *Mid-priority* set: Including data chunks i where *playback-point* $+ h \leq i <$ *playback-point* $+ \mu * h$.

3. *Low-priority* set: The remaining chunks until the end of buffer belong to this set.

In the high-priority set, chunks are requested in-order, while in the mid and low priority sets, downloading rarest chunks is favored. The probability of selecting a chunk from each set is a system parameter.

3.2.7 Chunk Distribution Policy

As mentioned in section 3.2.3 to guarantee free-riding resiliency a node keeps total number of uploaded chunks from each partner in each period, by defining a *score* for each of them. A higher score shows more uploaded chunks from that partner. When replying requests, the node prioritizes partners by their score. This means that node first replies to request of partners with higher score. Among those with same score, one is randomly selected.

On the other hand, as the upload bandwidth of a node is considered a limited and valuable resource, node should first reply to the requests that are most probably useful to the receiver. To maintain this, when a node receives a request from one of its partners, it attaches a time stamp to that request or updates time stamp of that request if it already has a copy of that request. While replying to requests, if a certain time T has passed from receiving a request and the node has not sent the requested chunk yet, there is no point in sending it after T has passed. Consequently, a node just deletes those

requests. T is fraction of scheduling period. Therefore, after T has passed from receiving a request there are two possibilities: requester has either sent a new request for that chunk again to another partner or that chunk has passed its play time.

In this system chunk distribution policy for source of the media stream and super-nodes is different from other nodes that are not member of the tree structure. There are two differences between normal and super nodes from chunk distribution policy perspective:

- Super nodes do not receive any blocks from the neighbor nodes under normal conditions, ie when they receive all blocks from their parents. So the give-to-get policy will not apply for super nodes.
- Super nodes are the only nodes distributing new blocks into the system, so they should prioritize distribution of new blocks over the old requested blocks. In general we can say they should prioritize rare blocks, but this is not possible because they cannot have a full view of the system to distinguish rare blocks from non-rare blocks.

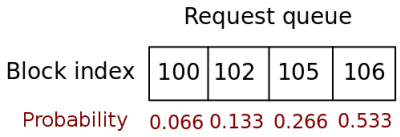


Figure 3.4: Chunk distribution strategy of super-nodes

Based on these two issues, we proposed a new distribution policy for super nodes including the source. This policy assigns probabilities to requests for each block based on the index of the block, in which requests for newly generated blocks have more probability to be answered than older blocks. The probability is based on a geometrical series with 'ratio' as the parameter. for example if the ratio of .5 is specified, and we have requests for 4 block indexes, the probabilities will be:

- 1st highest index probability: 0.533
- 2nd highest index probability: 0.266
- 3rd highest index probability: 0.133

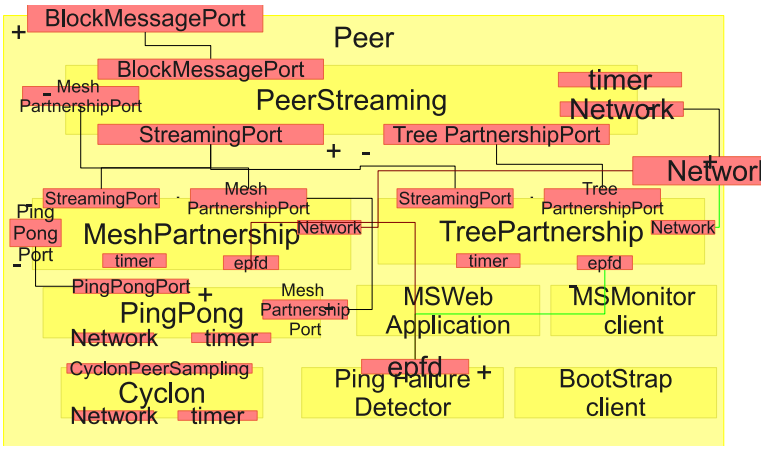


Figure 3.5: Component Diagram of a Node in tuxStream

- 4th highest index probability: 0.066

So with ratio specified as .5, each request for N+1'th index has half probability of being answered compared to requests for block N. This is illustrated in figure 3.4. This is called the *futuristic* chunk distribution strategy.

This chunk selection and distribution strategy helps system to be resilient to free-riders while providing chunks in-order to play the video without interruption.

3.3 Implementation

To implement tuxStream, we have used Kompics[27] framework. The component diagram of a node's components and events is shown in figure 3.5. There are five main components in a node in tuxStream. These are explained in details in the following sections.

3.3.1 Mesh Partnership

This component has the following three responsibilities :

- Partnership management:
  The list of partners is refreshed periodically in this component. The first step is to calculate G-Percent. Here we check the number of missed

blocks in the high-priority set, and based on that calculate G-Percent value. After that it refreshes partners in the partner list. We have defined a system parameter called "minimum-refresh-size", which indicates the minimum number of partners removed from partner-list and replaced by new partners selected from the nodes's neighbors. On the other hand, based on calculated G-Percent, number of partners from each local or global view is updated.

To guarantee resiliency to free riders, we keep a score for each partner. Partner's score is increased by each data chunk updated to this node by that partners in the last period. When removing partners, those with least score are removed first.

- Chunk selection and distribution:
  This component is in charge of sending buffer map periodically to partners and schedule to get data chunks based on received buffer-maps in the last period. In the chunk selection algorithm, based on the received buffer-maps in the last period, chunks are organized into three high-priority, mid-priority and low-priority sets.

  Considering kompics framework and its network model, the nodes begin to process next request after it has finished uploading the last request to output buffer of the node. As download bandwidth of a node is limited, it limits the number of requests a node can send in each period. This value is a product of total number of data chunks this node can download in one scheduling period. It is calculated considering time required to download one data chunk based on the node's download bandwidth.

  Requests in high-priority set are sent in order of data chunk index, and in low and mid priority set are sent randomly. Probability of selecting a chunk from each set is defined as a system parameter.

  A crucial decision each node must make is the first data chuck it will send the request for. We consider the average of the last chunk index for all received buffer maps minus high-priority set size when deciding the first data chunk to send a request for. This approach helps to reduce playback latency between the nodes in the system.

- Gradual optimization:
  In each period, each super-node looks into its global-view for new super-nodes. If the node finds any new super-nodes, it triggers a ping request on pingPongPort. *pingPong* component determines the nodes distance to the newly found super-node. In our implementation, the distance represents the round trip time (RTT) between these two corresponding

nodes. Later, node checks all the information it has gathered regarding the distance of all new super-nodes it has found in the last period to itself. The node decides if it wants to change its cluster based on this information and its distance to its current cluster-head.

For changing cluster, the node sends a request to the new super-node and asks for its local-view. Later, when updating its partners, the node considers its previous cluster members as global members and replaces them with new partners from its new local-view.

3.3.2 Tree Partnership

A periodical check in peer component checks if a node is eligible to become a super-node. When a node becomes eligible, tree partnership component is initiated. This component is in charge of keeping parent's address and children's list updated. When a node is assigned as this node's parent, tree-partnership begins to probe the node's parent. Upon detection of the failure of the parent, this component is responsible to look for a new parent. It is done by invoking the *gradual-optimization* event.

3.3.3 Peer Streaming

- Streaming data chunks:
  This component is in charge of streaming. It handles the *send-block-message* and *receive-block-message* events by updating the status of node's buffer. If the node is a member of tree of super-nodes, it has to push received data chunks to its children. Therefore, this component keeps an updated copy of the node's children list to be able to send the received block message to node's children.

- Playing the Stream:
  Another responsibility of this component is to decide when to begin playing the stream. Upon receiving a predefined number of data chunks, node begins to play the stream. We will later experiment on the optimal size for this set of delivered data chunks. Each node keeps a small number of data chunks after playing them, to ensure their availability in the system for the other nodes.

  To play the stream a periodical *playNextChunk* event is triggered with a timer equal to the stream rate. When playing the stream, if the data chunk that must be played is not received yet, the node waits until the next invoke of the *playNextChunk* event; if the data chunk is not received yet, the node considers it as missed chunk and jumps over it.

3.3.4 Ping Pong

This component is invoked by the mesh partnership when a node wants to calculate its round trip time to another node. This component calculates the time taken from sending a *ping* event to the other node and receiving the corresponding *pong* event.

3.3.5 Cyclon

Here we use Cylcon as membership management algorithm. Cyclon builds a large connected overlay by each node maintaining a small, partial view of the overlay nodes and keeps it updated by periodically exchanging it's partial view (called *shuffling*) with its neighbors in the overlay.

In our implementation of Cyclon, each node keeps two views of the system: local-view, a partial view of the nodes in the system that share the same cluster-head as the node and global-view, a partial view of all of the nodes in the system with different cluster-heads. To do so, each cache entry contains cluster-head address of the node besides its network-address and age. Therefore, in each period a node shuffles with two nodes: one selected from its local-view and the other one from its global-view.

Chapter 4

Evaluation

4.1 Experimental Scenarios

We have defined four different scenarios to evaluate tuxStream. In all scenarios stream rate is 500 Kbps.

Join Only

1200 nodes join the system following a Poisson distribution with an average inter-arrival time of 200 milliseconds. Among these nodes 15 of them have the bandwidth of 5 to 7 times of stream rate. Other 1185 nodes have the bandwidth uniformly distributed between half to 2.5 times of streaming rate.

Churn

First 500 nodes join the system following a Poisson distribution with an average inter-arrival time of 500 milliseconds, and then till the end of the simulations 1200 nodes join and fail continuously following the same distribution with an average inter arrival time of 1000 milliseconds. Failure of the nodes follows a Pareto distribution with skew of 1.03 based. 17 out of 1100 nodes joining the system have a bandwidth between 5 to 7 times of streaming rate. Other nodes have bandwidth of half to 2.5 times of streaming rate.

Flash Crowd

First, 100 nodes join the system following a Poisson distribution with an average inter-arrival time of 100 milliseconds. 3 out of these 100 nodes have the bandwidth of 5 to 7 times of stream rate and others half to 2.5 times of streaming rate. Then, 1000 nodes join following the same distribution with

an average inter-arrival time of 10 milliseconds. 12 nodes are joined with the bandwidth of 5 to 7 times of stream rate.

Catastrophic Failure

1000 nodes join the system following a Poisson distribution with an inter-arrival time of 100 milliseconds, where 15 of them have the bandwidth of 5 to 7 times of stream rate and others half to 2.5 times of streaming rate. Then, 500 existing nodes fail following a Poisson distribution with an average inter-arrival time 10 milliseconds.

Free Riders

1000 nodes join the system following a Poisson distribution with inter-arrival time of 100 milliseconds and where 200 of them are free-riders and 15 of them have the bandwidth of 5 to 7 times of stream rate and others half to 2.5 times of streaming rate.

4.2 Metrics

Continuity-index

It is a metric to show the quality of service in the client side. It is the percentage of successfully played data chunks to the total number of played chunks in each node. In the diagrams the average of this value for all of the nodes in the system is calculated.

Playback-latency

It shows the difference in seconds between the playback time in the source of media stream and a node. The lower values for this metric is more desirable as it shows more consistency between nodes in the system.

This value is highly dependent to two factors in the system. The first is the pre-buffer time, which is the number of data chunks we wait to receive before beginning to play the stream. The lower values for this parameter results in the node beginning the stream faster. But, if the selected value is too small, node does not have enough time to retrieve following data chunks. This results in node experiencing dis-continuity in playing the stream. The second factor is the index of first data chunk node will send a request for. This means that node must have a estimation on the average situation of buffer-map of other

nodes in the system. If it sends a request for a small chunk index, it has two disadvantages: (i) It increases the playback latency between node and the source of media stream. (ii) As nodes keep only a limited number of data chunks after they have played them, the newly joined node may not be able to retrieve all the data chunks it requires. This results in data loss in the beginning of playing the stream. In our implementation, node sends its first request for the average value of the last index in all received buffer maps in the first invocation of the scheduling algorithm.

Network-latency

In our implementation, latency is selected to show the distance between two nodes. We calculate the average of the total latencies proposed in the system by sending each data chunk. The higher value for latency in sending a chunk from one node to another shows more distance between the two nodes. Therefore higher values for this average shows data chunk exchange between nodes that are not in each others locality.

Startup-delay

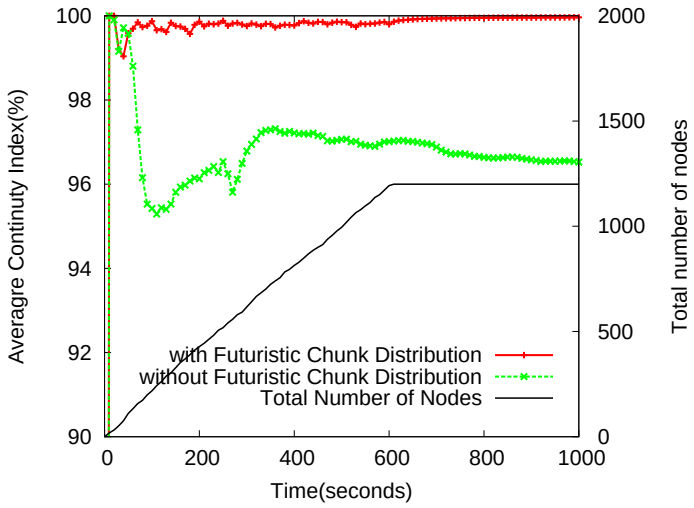
This metric shows the time taken between joining of the node to the system and its initiation of playing the stream. In the diagrams the average value of startup for all participating nodes is calculated. The lower values are more desirable.

Transmission-delay

This metric shows the end-to-end delay in the system, regardless of other factors that can affect it. We measure it by calculating the average of transmission-delay for all of the data chunks transmitted in the system. Transmission-delay for a data chunk is the time taken for it to be transfered from the source of media stream to its receiver.

4.3 Self Experiments

In the experiments we first evaluate performance of tuxStream, showing the effect of changes in the following parameters and deciding what is the best value for each of the parameters. For each parameter the join only and churn scenarios are experimented.



(a) Continuity-index

Figure 4.1: Effect of Different Chunk Distribution policy for Super-nodes in Join Scenario on Continuity-index

4.3.1 Chunk distribution strategy

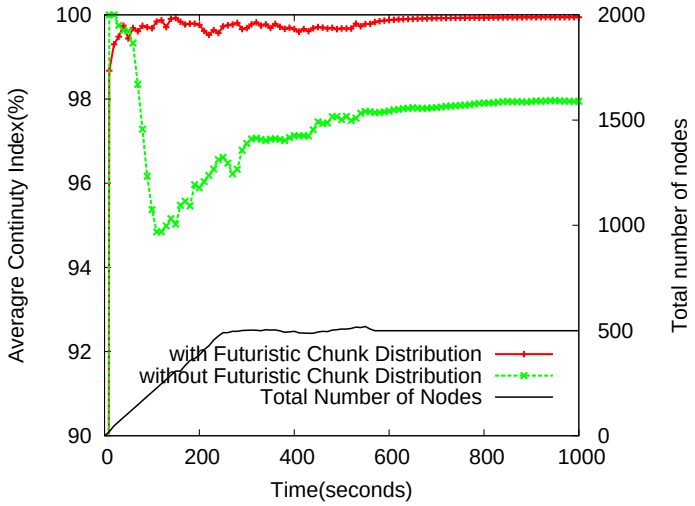
As explained in 3.4, chunk distribution of super-nodes in tuxStream is in favor of distributing new chunks between the nodes in their local-mesh structure. If these super-nodes, like other non-super-nodes in the system reply to the requests from their mesh neighbors based on chunk index, i.e. based on the order their neighbors have sent their requests, this results in strive of some nodes for new data chunks which are owned by these super-nodes, while super-nodes are replying to the requests for chunks which are not rare in the system.

In figures 4.1, 4.2 and 4.3 the effect of changes in chunk distribution on continuity-index and average playback-latency in shown.

4.3.2 pre-buffer time

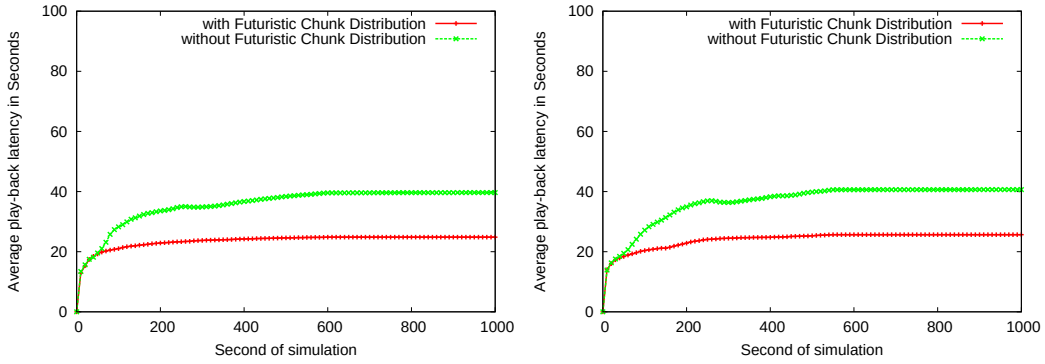
pre-buffer time is the number of data chunks a node must receive before playing the stream. We evaluate three values for this parameter: 10, 20, 30. As each second of streaming is divided into two data chunks, waiting for receiving e.g 20 data chunks before beginning to play the stream means waiting for receiving 10 seconds of the stream before beginning to play.

As shown in figure 4.4, in both join only and churn scenarios by selecting the 10 for pre-buffer time nodes experience low qualities in playing the stream. The reason is they begin too early to play and they must wait for the following



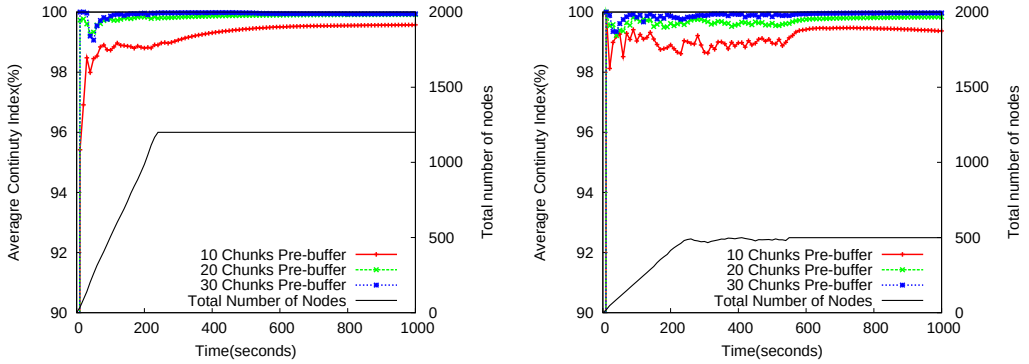
(a) Continuity-index

Figure 4.2: Effect of Different Chunk Distribution policy for Super-nodes in Churn Scenario on Continuity-index



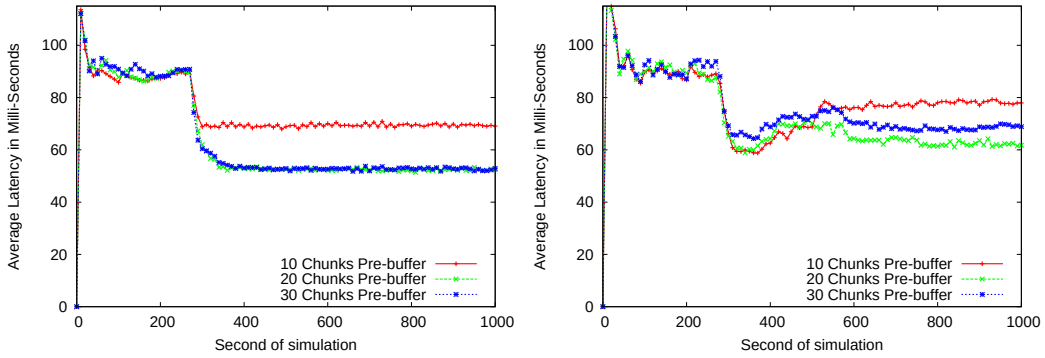
(a) Join Scenario (b) Churn Scenario

Figure 4.3: Effect of Different Chunk Distribution policy for Super-nodes On Playback-latency



(a) Join Scenario (b) Churn Scenario

Figure 4.4: Effect of Different Pre-buffer-times on Continuity-index



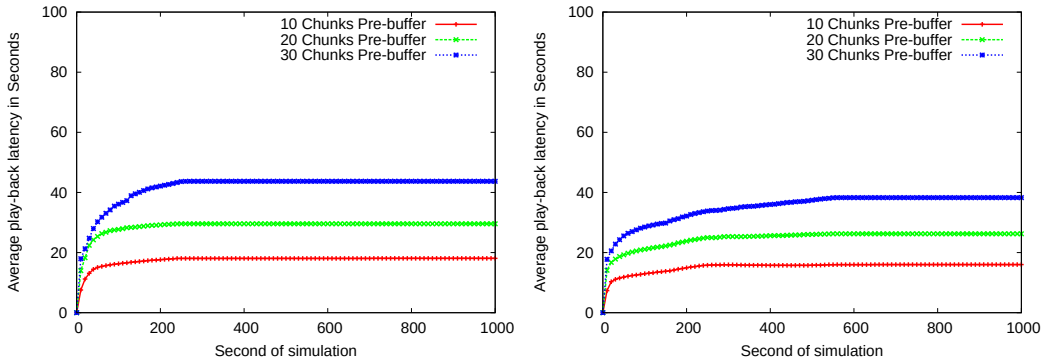
(a) Join Scenario (b) Churn Scenario

Figure 4.5: Effect of Different Pre-buffer-times on Network-latency

data chunks to be delivered. With values 20 and 30 we experience better continuity index.

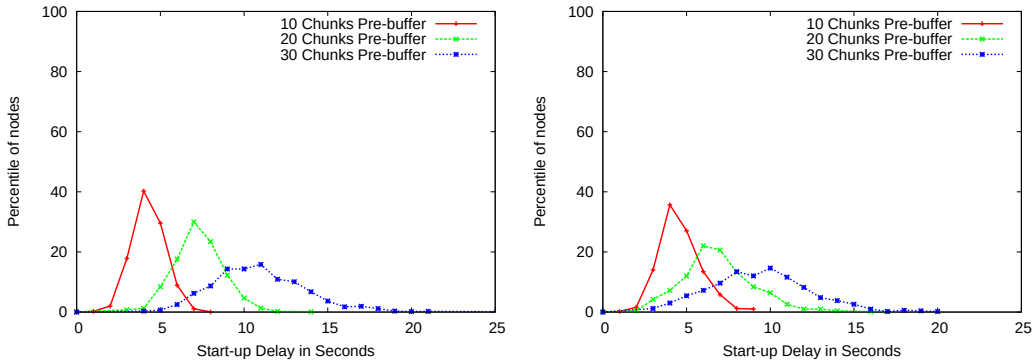
In figure 4.5, the effect of changes in pre-buffer time value on network-latency in the system is shown in both join only and churn scenarios. As we can see, selecting the value 10 for pre-buffer time results in higher amounts of network latency. The reason is the missing data chunks in the high-priority set and consequently higher values for g-percent. Nodes begin to fetch data chunks from their global neighbors rather than their local neighbors which is not desirable. Average network latency for both 20 and 30 pre-buffer time values are good.

In figure 4.6 the effect of changes in pre-buffer time of average playback-latency in the system is shown. As expected the higher the values for pre-



(a) Join Scenario (b) Churn Scenario

Figure 4.6: Effect of Different Pre-buffer-times on Playback-latency



(a) Join Scenario (b) Churn Scenario

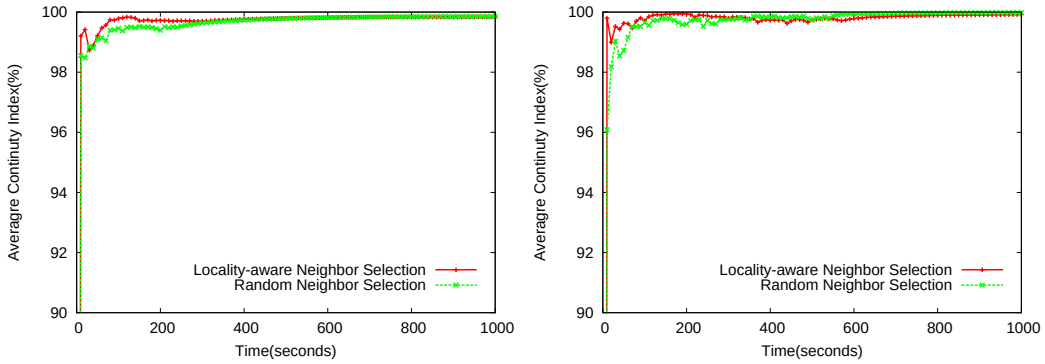
Figure 4.7: Effect of Different Pre-buffer-times on Startup-delay of the nodes

buffer time result in higher values for playback-latency. The reason is that nodes must wait for longer period of time to receive all the required chunks while the media source and other nodes are playing the stream.

As shown in figure 4.7, startup-delay in case we have 30 as the value of pre-buffer time is the most. The reason is that nodes must wait longer period of time to begin to play the stream.

4.3.3 Locality Awareness

To check the effect of organizing nodes in locality-aware clusters, in the following experiments we consider two algorithms for neighbors selection. In the first algorithm we use tuxStream neighbor selection which is locality-aware neigh-



(a) Join Scenario (b) Churn Scenario

Figure 4.8: Effect of Locality Awareness on Continuity-index

bor selection. In the second algorithm, we use random neighbor selection. This means that when node wants to update its partners both its local-view and global-view are the same and node selects random partners between these nodes, ignoring their distance.

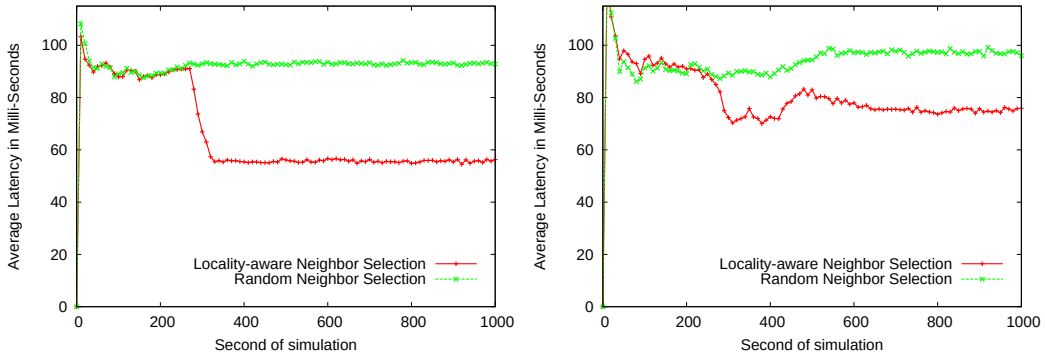
As shown in figure 4.8 in both implementations nodes have a nearly 99% continuity-index.

The effect of locality-awareness on network-latency is shown in figure 4.9. As expected network latency increases in random neighbor selection as nodes send requests for data chunks not considering their distance to them. As we can see after 300 seconds has passed from initiation of the simulation and the first super-nodes and consequently clusters begin to form, the value of latency in locality-aware neighbor selection decreases. This is the result of nodes fetching data chunks from their local neighbors. The fast decrease in the network latency shows the fast formation of clusters in the system using the gradual optimization algorithm.

As shown in the figure 4.10 the value for playback latency is different in a small value as a result of local neighbor selection. This is expected as the win-no lose situation is favorable in locality-aware systems [38].

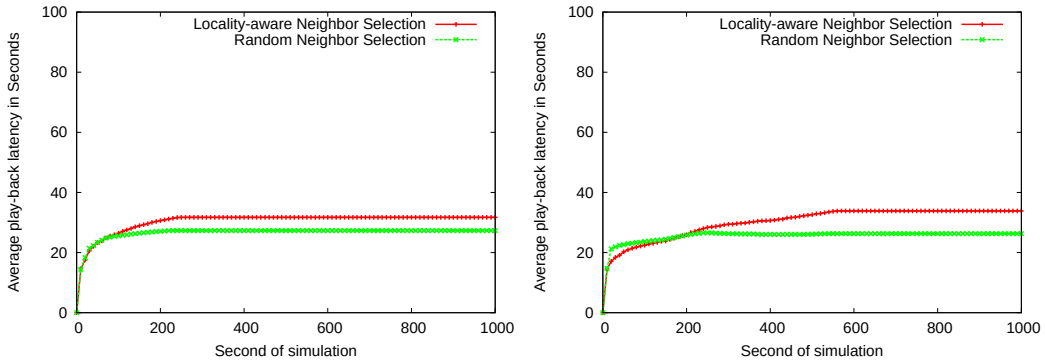
4.3.4 G-Percent

As explained in section 3.2.2, there are two parameters that have effect on G-percent calculation: *MinimumGPercent* and *GPercentIncrementalParameter*. In this section we explore the optimal value for each of these parameters. The higher values for *GPercentIncrementalParameter* shows node's eagerness to fetch data from global-view neighbors when a fluctuation happens in retrieving



(a) Join Scenario (b) Churn Scenario

Figure 4.9: Effect of Locality Awareness on Network-latency



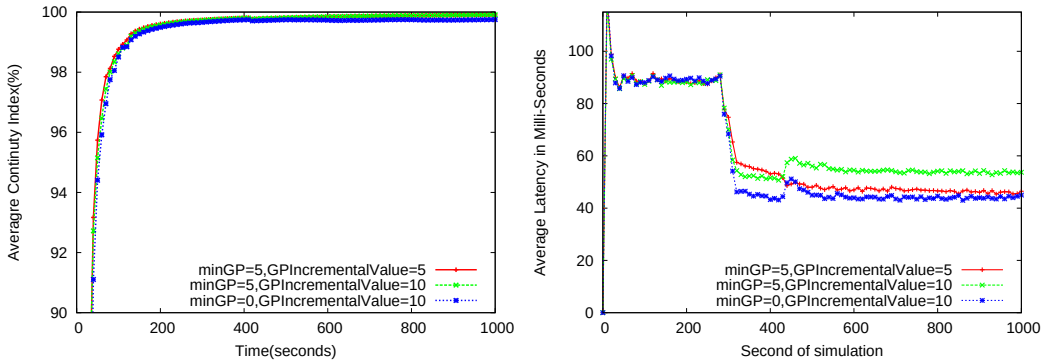
(a) Join Scenario (b) Churn Scenario

Figure 4.10: Effect of Locality Awareness on Playback-latency

data chunks from local-view members.

The result of experiment on these two parameters in churn scenario is displayed in figure 4.11. Setting *MinimumGPercent* equal to 0 lowers continuity-index. The reason is a node with missing chunks in its high priority set increases its G-percent value and adds global neighbor(s), e.g node B, as its partner(s). On the other hand, nodes in global neighborhood(e.g node B), with G-percent equal to 0 will not add any non-local members in its partner list. There fore, although *GPercentIncrementalParameter* is 10, node is not able to fetch data chunks from its global members. This results in lower continuity-index in these nodes (shown in figure 4.11a). Consequently, network-latency is lower in this case (figure 4.11b).

In the other two experiments, as the value of *MinimumGPercent* is 5 (this



(a) Continuity index (b) Network latency

Figure 4.11: Effect of GPercent on Network-latency and Continuity-index In Catastrophic Scenario

means that out of 20 partners in each node, at least one is always from global-view), nodes manage to keep a good continuity index and the network-latency is higher in these experiments for nodes with higher G-percent successfully pull data from their global-view partners.

4.4 Comparison With Other Solutions

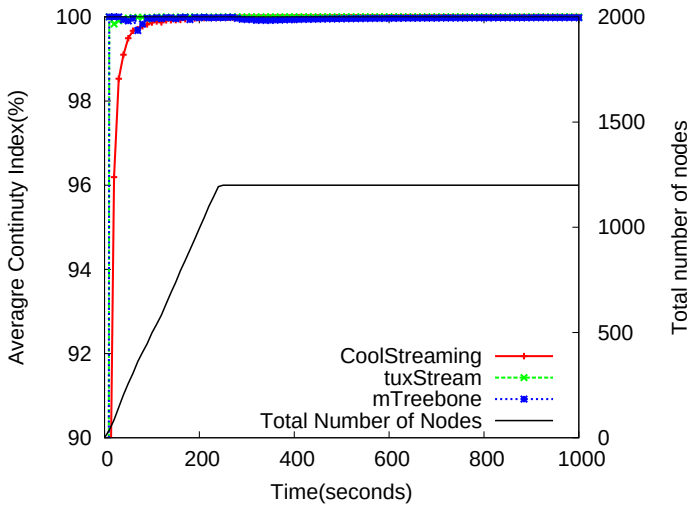
We compare tuxStream with two other solutions: CoolStreaming and mTreebone. The results of experiment for different scenarios are explained in the following sections.

4.4.1 Comparison with High-bandwidth in Join-only Scenario

First we show the result of experiments for all solutions while the all nodes in the system have a high bandwidth.

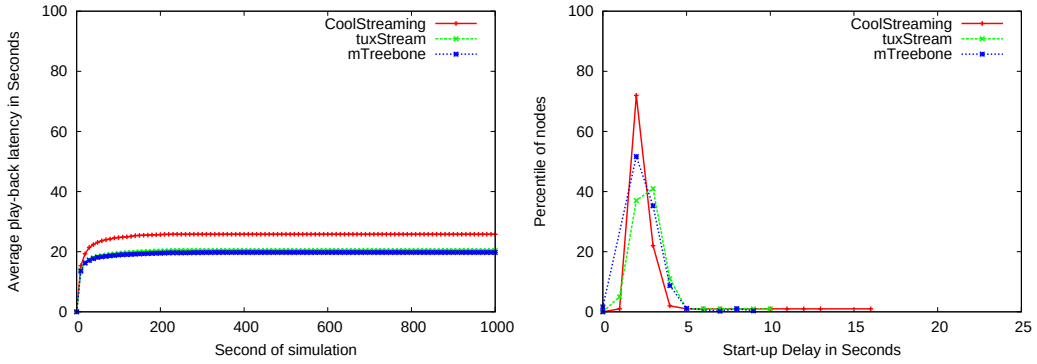
For high bandwidth scenario, we assign each nodes an upload bandwidth of randomly selected between 4 to 12 times of streaming rate. As shown in figure 4.12a all three solutions have a good continuity-index in presence of high bandwidth.

Also, with regard to the value of pre-buffer time (10 second of stream), we can see a good playback-latency experiences by all nodes (figure 4.13a). The results for startup-delay is shown in figure 4.13b.



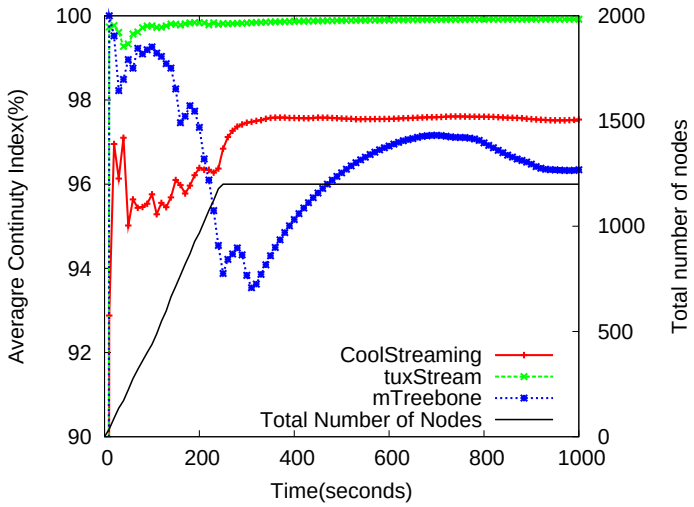
(a) Continuity index

Figure 4.12: Continuity-index in Join Scenario for all Solutions with High Bandwidth



(a) Playback Latency (b) Startup Delay

Figure 4.13: Playback-latency and Startup-delay in Join Scenario for all Solutions with High Bandwidth



(a) Continuity index

Figure 4.14: Continuity-index in Join Scenario for all Solutions

4.4.2 Join-only Scenario

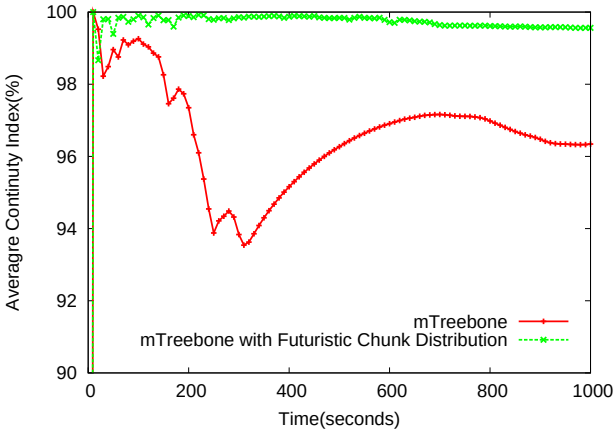
In this part we explain the result of experiments on the three solutions for join-only scenario.

Continuity index

In the join scenario (figure 4.14a) we can see that tuxStream shows a better quality of experience compared to other two solutions. We expected continuity-index in mTreebone to increase to 100%, but the gradual optimization mechanisms affects the continuity index by changing the structure of the tree. This effect of change in structure is more visible when system consists of nodes with low upload bandwidth.

In mTreebone, for the first drop in the continuity index in around 300th second of the simulation, the cause is the distribution policy of the nodes in the tree. The random selection of nodes to respond to, and the in order distribution of the chunks in the system or in other words rarity of recently generated chunks in the system, causes some nodes to face a lot of missing chunks. To justify this we have experimented mTreebone with chunk distribution algorithm proposed by tuxStream. The results for continuity index of nodes is shown in figure 4.15.

In coolStreaming, as the distribution policy differs with tuxStream, first join of the nodes can have a tremendous effect on the continuity index. After



(a) Continuity index

Figure 4.15: Continuity-index in Join Scenario for mTreebone with tuxStream Chunk Distribution Algorithm

join period has finished(600th second of simulation), it increases to an average of 97% in continuity index. This result is in accordance with coolStreaming paper [5].

Playback latency

As shown in figure 4.16b Playback-latency of tuxStream is better than the other two solutions. The reason for higher playback latency of mTreebone is the drop in the continuity-index.

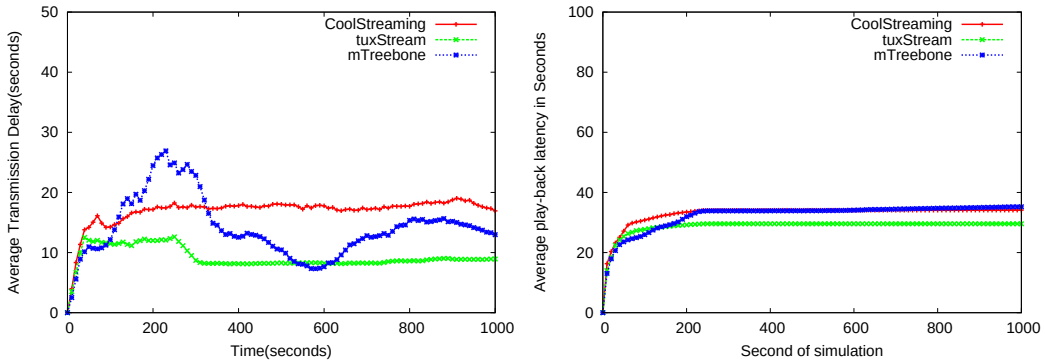
In our implementation of buffer status, when a node has no delivered chunks in its buffer, node pauses playback until it receives a chunk. As a result of this wait, the playback of those nodes increases the average playback-latency of the system.

For mTreebone, after formation of the treebone, transmission delay decreases(shown in figure 4.16a), but as nodes do not jump over chunks the increased playback remains the same.

In coolstreaming, the delay caused by pure mesh based solution behavior, results in more difference in playback of the source of media stream and the nodes.

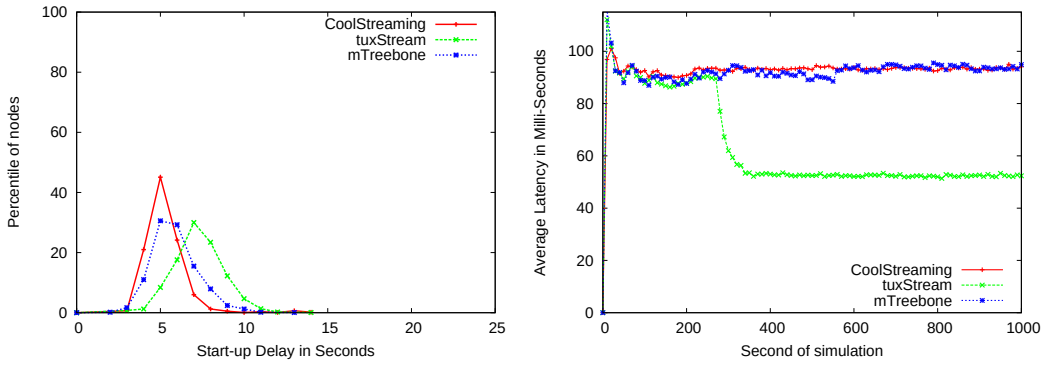
Network Latency

The network-latency for all three scenarios is shown in figure 4.17b. The network-latency for tuxStream will be eventually least of all, which is the



(a) Transmission delay (b) Playback latency

Figure 4.16: Transmission-delay and Playback-latency in Join-only Scenario



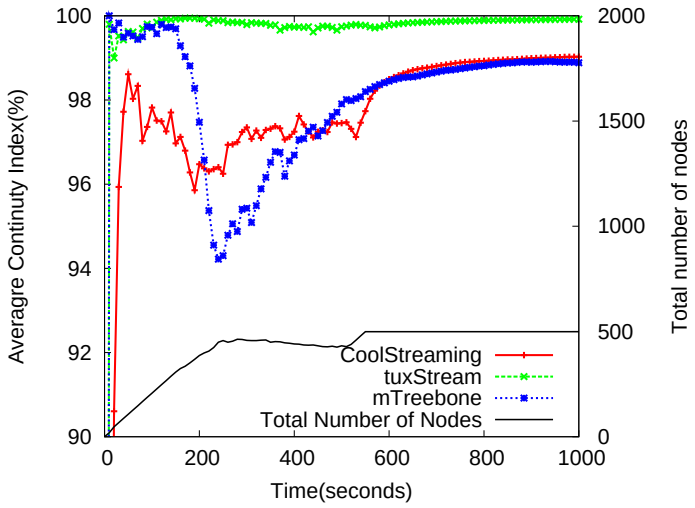
(a) Startup delay (b) Network latency

Figure 4.17: Startup-delay and Network-latency in Join-only Scenario

effect of formation of locality-aware clusters.

Startup delay

In figure 4.17a the startup-delay for all three scenarios are shown. It was expected that mTreebone, as it forms a treebone, show lower startup-delay than the other two solutions. This does not happen in practice. The reason is that it takes quite time for nodes to find a place in the treebone and practically, nodes begin to play the stream pulling data from their mesh neighbors.



(a) Continuity index

Figure 4.18: Continuity-index in Churn Scenario

4.4.3 Churn Scenario

In this section we explain the results of experiments on three solutions for the churn scenario.

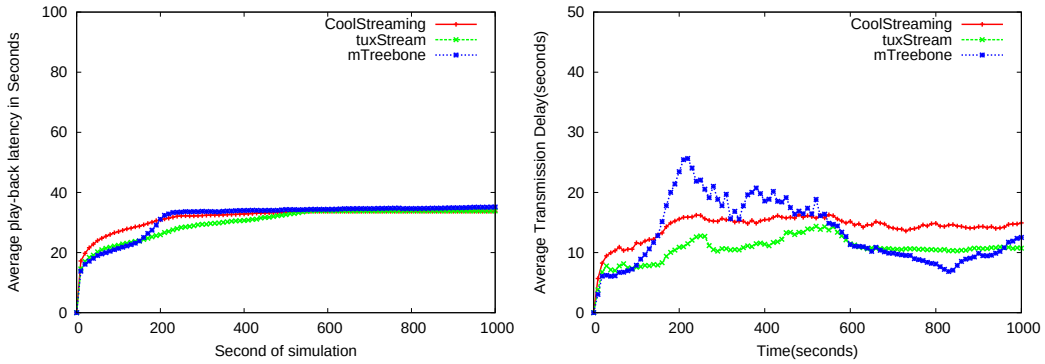
Continuity-index and Playback-latency

As shown in the figure 4.18a, all of the three systems handle the node dynamics in the system well. The difference between solutions performance is nearly similar to the join only scenario.

In tuxStream, as the system is mainly pull-based and as the failure of nodes in the mesh does not have an outstanding negative effect on data delivery by other nodes we observe a good playback continuity. Also, as the probability of failure of a super-node is low, it is unlikely that the tree of super-nodes needs reconstruction. But if it does, affected nodes use their global-view to keep their playback continuity high.

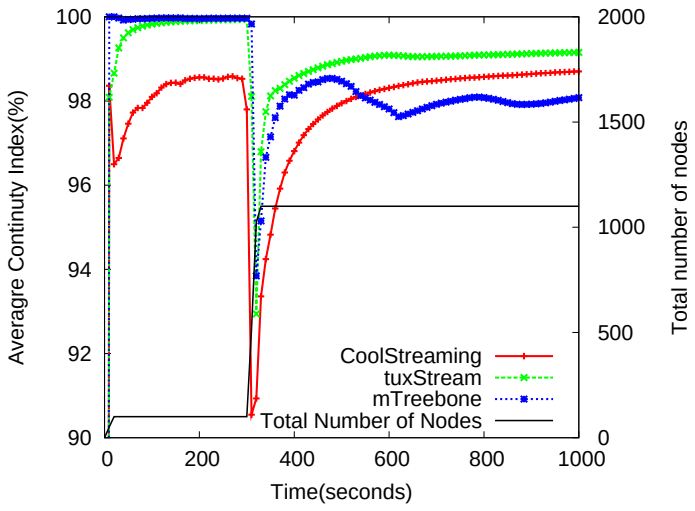
CoolStreaming, as expected, manages to keep a good playback continuity in presence of churn.

We can see that mTreebone also handles churn well. The reason is that, nodes in the tree disable their tree push-pointer whenever they observe a problem in receiving data chunks. Consequently, tree nodes do not observe severe data loss even if a node in the upstream of the tree fails. On the other hand, as treebone is made of stable nodes, it is less probable that a tree node



(a) Playback latency (b) Transmission delay

Figure 4.19: Playback-latency and Transmission-delay in Churn Scenario

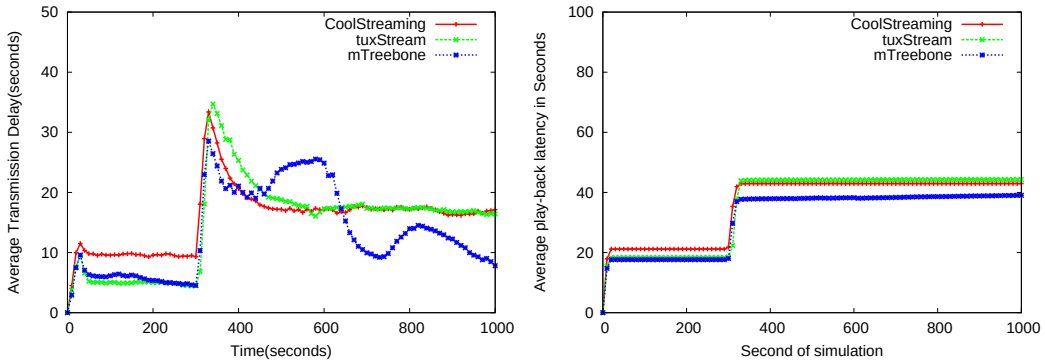


(a) Continuity index

Figure 4.20: Continuity-index in Flash-crowd Scenario

fails.

The explanation for the playback-latency, shown in figure 4.19a observed in this scenario is the same as churn scenario.



(a) Transmission delay (b) Playback latency

Figure 4.21: Playback-latency and Transmission-delay in Flash-crowd Scenario

4.4.4 Flash-crowd Scenario

Continuity index

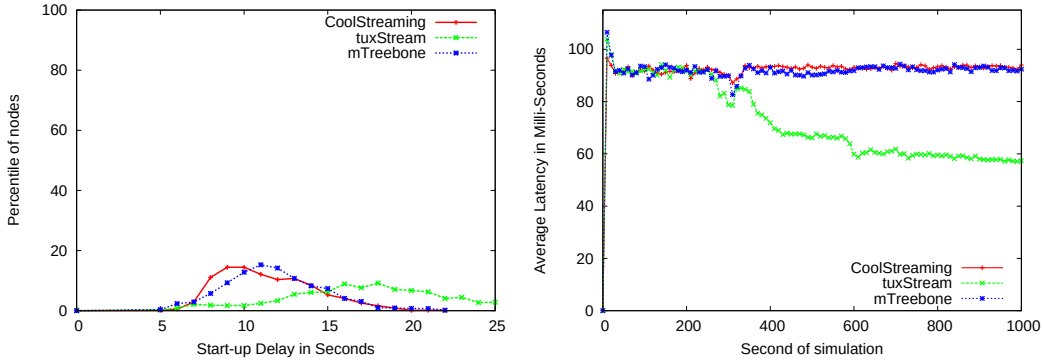
We can see that in this scenario, until the 300th second of simulation as the number of nodes in the system is low (100 nodes) both mTreebone and tuxStream perform well. In coolStreaming, as suggested by [8], continuity indexes are closely correlated with client population. There fore, with increase in the number of participating nodes we observe an increase in the playback continuity in the system. The behavior of tuxStream and mTreebone are the same, as they both use their mesh structures to pull data chunks from their neighbors.

Playback latency and Transmission delay

As shown in figure 4.21b, for all three solutions the value of the playback-latency increases to twice as its value in normal join scenario. After formation of treebone, transmission-delay (figure 4.21a) for mTreebone decreases to the least of all three solutions.

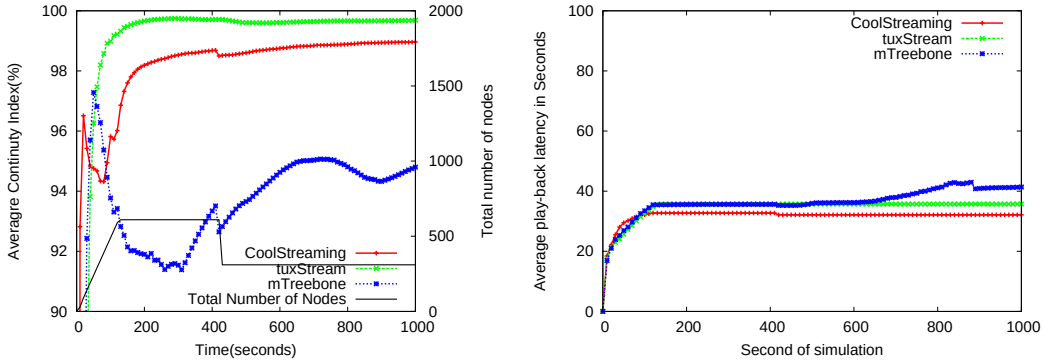
Network Latency

As shown in figure 4.22b and considering that we have the burst in joining of the nodes in the 300th second of simulation, we can see that it takes more than 200 seconds for nodes to find the optimal cluster and begin to fetch data chunks from their local neighbors.



(a) Startup delay (b) Network latency

Figure 4.22: Network-latency and Startup-delay in Flash-crowd Scenario



(a) Continuity index (b) Playback latency

Figure 4.23: Playback-latency and Continuity-index in Catastrophic-failure Scenario

Startup delay

In this scenario we can see a general increase in value of startup-delay (figure 4.22a), nearly twice the value of startup-delay in the join scenario.

4.4.5 Catastrophic-failure Scenario

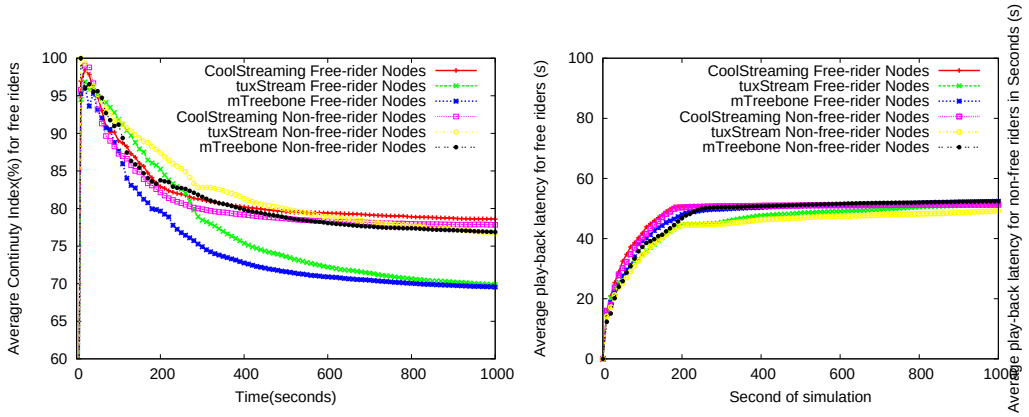
Continuity-index and Playback-latency

As shown in the figure 4.23a, catastrophic failure has the least effect on tuxStream, but all protocols are handling it and their general behavior is similar to that of churn and join.

We can see an increase in the continuity index of mTreebone and coolStreaming, which is the result of using pareto distribution for failure of the nodes. Based on that, nodes who have joined the system more recently have higher probability of failure. The figure shows an increase because the average continuity index is held back by those new nodes, and as they are killed, the average is increased in a noticeable jump.

4.4.6 Free-rider Scenario

In this section we show the behavior of all three systems in presence of free-rider nodes. As half of the nodes are free-riders and they do not dedicate any upload bandwidth to the system, the overall average of upload bandwidth in the system is decreased compared to situations when we do not have any free-riders in the system. This results in a decrease in the playback-continuity of all participating nodes.



(a) Continuity index (b) Playback latency

Figure 4.24: Continuity index and Playback-latency in Free-rider Scenario

Continuity index

As shown in figure 4.24a, tuxStream differentiates most between free-rider nodes and non-free-rider nodes. Free-rider nodes do not starve, but their playback-continuity decreases gradually.

Playback-continuity for free-rider and non-free-rider nodes are the same in coolStreaming. This is expected, as neither in partner refresh nor in chunk distribution algorithms, the amount of upload from partners are not considered as a deciding factor.

In is also expected that in mTreebone we observe no difference between free-rider and non-free-rider nodes. This is not happening and the reason is

that free-rider nodes are placed in the leaves of the treebone. As explained, mTreebone uses two optimization algorithms to placed the nodes with more children closer to the source of media stream. Consequently, free-rider nodes are repeatedly replaced by non-free-rider nodes and they must rejoin the treebone. This iterative connect/dis-connect from the treebone results in lower playback continuity for free-rider nodes.

Playback latency

As shown in figure 4.24b, the playback latency for mTreebone and coolStreaming is the same. In tuxStream, as free-rider nodes face higher values of data loss their playback-latency increases.

Chapter 5

Future Work

In tuxStream we proposed innovative idea of introducing locality through multiple overlapping overlays, and grouping of nodes into clusters. This new idea opens a lot of new fields for more research and experiments. In this thesis, we have tried to put together a thin layer of research on all components of our system to provide a working project, but each of these topics can be extended in future research projects.

We can categorize the topics that can be extended and have more room to experiment about tuxStream into these items:

1. Estimation of optimal size of clusters based on properties of the network and participating nodes.

2. Dynamic qualification factors for super-nodes. For example, in a live media streaming session, some nodes might be located in autonomous system A whose average bandwidth is much lower than autonomous system B. With predefined qualification factors, there will be no or very few super nodes in A compared to number of super nodes in B.

3. Applying aggregation functions in a cluster to find best startup position for requesting data chunks.

4. Currently the only parameter in shift-up process is bandwidth of super-node, i.e super-nodes with higher bandwidth take place of nodes with lower bandwidth that are closer to the source. Using an aggregation function to estimate the size of the cluster of each super-node, we can experience on placing nodes with bigger clusters higher in the tree of super-nodes.

Chapter 6

Conclusion

In this thesis, we presented tuxStream, a push-pull peer-to-peer live media streaming protocol. tuxStream features include locality-awareness, robustness to churn and extensive use of bandwidth of all nodes. The innovative use of multiple mesh structures, one for maintaining the system as whole and multiple smaller structures for locality-awareness, lets the system utilize network resources more efficiently while maintaining resiliency to churn.

The protocol leverages the benefits of both tree and mesh overlays, by using a tree of super-nodes to push data with low latency, and distributing data chunks in clusters using mesh. Experiments show that this protocol can achieve low network latency and transmission delay while maintaining high playback continuity and resiliency to churn. On the whole, we introduce locality through multiple mesh structures with a flexible fall-back method to prevent any loss due to locality awareness policies, in other words "locality-aware when it's calm, no locality-awareness when in crisis".

Bibliography

[1] PPLive homepage. [Online]. Available: *http://www.pplive.com/*

[2] Kunwadee Sripanidkulchai and Aditya Ganjam and Bruce Maggs and Hui Zhang. The Feasibility of Supporting Large-Scale Live Streaming Applications with Dynamic Application End-Points. In *Proceedings of ACM SIGCOMM pages 107-120*, Apr. 2004.

[3] K.C. Almeroth and M.H. Ammar, Collecting and Modeling the Join/Leave Behavior of Multicast Group Members in the MBone, In *Proc. IEEE Intl Symp. High Performance Distributed Computing (HPDC), p. 209*, Aug. 1996.

[4] F. Wang, Y. Q. Xiong, and J. C. Liu. mTreebone: A Hybrid Tree/Mesh Overlay for Application-Layer Live Video Multicast. In *IEEE INFOCOM'08*, Apr. 2008.

[5] X. Zhang,J. Liu, and Y.Yum. CoolStreaming/DONet: a Data-driven Overlay Network for Peer-to-Peer Live Media Streaming. In *IEEE INFOCOM'05*, Mar. 2005.

[6] K. Sripanidkulchai, B. Maggs, and H. Zhang. An Analysis of Live Streaming Workloads on the Internet. In em Proc. ACM Internet Measurement Conf. (IMC), pp. 41-54, Oct. 2004.

[7] Y. Chu, S. Rao and H. Zhang.A case for end system multicast. In *Proc. of ACM Sigmetrics*, June 2000.

[8] 1. Zhang X, Liu J, Li B. On Large Scale Peer-To-Peer Live Video Distribution : CoolStreaming and Its Prelimianry Experimental Results. *Science And Technology.2-5.*

[9] J. Liu, S. G. Rao, B. Li, and H. Zhang. Opportunities and Challenges of Peer-to-Peer Internet Video Broadcast. In *Proc. of the IEEE*, 2007.

[10] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron and A. Singh. SplitStream: High-bandwidth multicast in cooperative environments. In *Proc. ACM SOSP 03*, October 2003.

[11] V. Pai, K. Tamilmani, V. Sambamurthy, K. Kumar, and A. Mohr. Chainsaw: Eliminating trees from overlay multicast. In *Proc. The 4th International Workshop on Peer-to-Peer Systems (IPTPS)*, February 2005.

[12] M. Zhang, J.-G. Luo, L. Zhao, and S.-Q. Yang. A peer-to-peer network for live media streaming - using a push-pull approach. In *Proc. ACM Multimedia 05* , November 2005.

[13] V. Venkataraman, P. Francis, and J. Calandrino. ChunkySpread: Multitree unstructured peer-to-peer multicast. In *Proc. The 5th International Workshop on Peer-to-Peer Systems*, February 2006.

[14] D. A. Tran, K. A. Hua, and T. T. Do. A Peer-to-peer Architecture for Media Streaming. In *IEEE J. Selected Area in Communications*, 2004.

[15] M. Bishop, S. Rao, and K. Sripanidkulchai. Considering priority in overlay multicast protocols under heterogeneous environments. In *IEEE INFOCOM*, 2006.

[16] 1. Kostic D, Rodriguez A, Albrecht J, Vahdat A. Bullet: high bandwidth data dissemination using an overlay mesh. In: *SOSP 03 Proceedings of the nineteenth ACM symposium on Operating systems principles. Vol 37.* ACM Press; 2003:282-297.

[17] PPLive, http://www.pplive.com/, 2009.

[18] S. Asaduzzaman, Y. Qiao, and G. Bochmann. CliqueStream: an efficient and fault-resilient live streaming network on a clustered peer-to-peer overlay. In *Proc. of the 8th IEEE International Conference on P2P Computing*, September 2008.

[19] T. Locher, S. Schmid, and R. Wattenhofer. equus: A provably robust and locality-aware peer-to-peer system. In *Peer-to-Peer Computing, 2006. P2P 2006. Sixth IEEE International Conference* Sept. 2006.

[20] F. Wang, J. Liu, and Y. Xiong. Stable Peers: Existence, Importance, and Application in Peer-to-Peer Live Video Streaming. In *IEEE INFOCOM'08*, Phoenix, AZ, USA, April 15-17, 2008.

[21] S. Voulgaris, D. Gavidia, and M. van Steen. CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. In *J. Network and Systems Management*, June 2005.

[22] J. J. D. Mol, J. A. Pouwelse, M. Meulpolder, D. H. J. Epema, and H.J. Sips. Give-to-get: An algorithm for P2P video-on-demand. In P*roc. of-SPIE/ACM MMCN '08, San Jose, California*, Jan 2008

[23] 1. Saroiu S, Gummadi KP, Gribble SD. Measuring and analyzing the characteristics of Napster and Gnutella hosts. In Multimedia Systems. 2003;9(2):170-184. Available at:http://www.springerlink.com/Index/10.1007/s00530-003-0088-1.

[24] 1. Bustamante F, Qiao Y. Friendships that Last: Peer Lifespan and its Role in P2P Protocols. In: *Web Content Caching and Distribution.*; 2004:233-246. *Available at: cite seer.ist.psu.edu/bustamante03friendships.html*.

[25] 1. Wang F. Stable Peers : Existence , Importance , and Application in Peer-to-Peer Live Video Streaming. Communications Society. 2008.

[26] W. P. Ken Yiu, Xing Jin, and S. H. Gary Chan. Challenges and approaches in large-scale p2p media streaming. In *IEEE MultiMedia, 14(2):5059*, 2007.

[27] Cosmin Arad, Jim Dowling, and Seif Haridi. Developing, simulating, and deploying peer-to-peer systems using the kompics component model. In *COMSWARE 09: Proc. of Fourth International ICST Conf.*, New York, USA, 2009.

[28] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: Estimating latency between arbitrary Internet end hosts. In *SIGCOMM Internet Measurement Workshop*, 2002.

[29] Duc A. Tran Kien A. Hua Tai T. Do. ZIGZAG: An Efficient Peer-to-Peer Scheme for Media Streaming. In *University of Central Florida Technical Report*, 2002.

[30] Liang, J. Nahrstedt, K. DagStream: locality aware and failure resilient peer-to-peer streaming. In *PROCEEDINGS- SPIE THE INTERNATIONAL SOCIETY FOR OPTICAL ENGINEERING*, pages 6071, 2006.

[31] The Annotated Gnutella Protocol Specication. Jan-2008, Available at *http://gnutella.sourceforge.net/developer/stable/index.html*

[32] J. Liang and K. Nahrstedt. Randpeer: Membership management for QoS sensitive P2P applications. In *Tech. Rep. UIUCDCS-R-2005-2576, UIUC*, May 2005.

[33] 1. Castro M, Druschel P, Kermarrec AM, Rowstron AIT. Scribe: a large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*. 2002;20(8):1489-1499.

[34] Rowstron A, Druschel P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: *Lecture Notes in Computer Science. Vol 2218*. Springer; 2001:329-350.

[35] 1. Goyal VK. Multiple description coding: Compression meets the network. *IEEE Signal Processing Magazine*. 2001;18(5):74-93.

[36] 1. Hei X, Liang C, Liang J, Liu Y, Ross KW. A Measurement Study of a Large-Scale P2P IPTV System. In *IEEE Transactions on Multimedia*. 2007;9(8):1672-1687.

[37] Kostic, D., Rodriguez, A., Albrecht, J., Bhirud, A., and Vahda, A. M. (2003). Using Random Subsets to Build Scalable Network Services. In *USENIX*, p. 111-125

[38] Lehrieder, F., Oechsner, S., Hossfeld, T., Despotovic, Z., Kellerer, W., Michel, M.. Can P2P-Users Benefit from Locality-Awareness? In *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference*.

[39] 1. Chu Y, Rao SG, Seshan S, Zhang H. A case for end system multicast. IEEE Journal on Selected Areas in Communications. 2002;20(8):1456-1471. Available at: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1038577.