ABSTRACT
Flexible Group Data Delivery over Application-Layer Overlays
Ashish Vashishta
Mentor: Michael J. Donahoo, Ph.D.

Group applications demand specialized services, such as multicast, subcast and concast, to provide efficient network resource utilization and scalability. Large-scale deployment of these services at the network layer appears unlikely; consequently, recent work focuses on the provision of these services in overlay networks. Unfortunately, existing approaches focus on providing a single service tailored for a specific application. We propose the creation of a unified framework, providing customizable implementations of these services. We relax the strict requirements of the traditional layered solutions to enable overlay formation based on application semantics, instead of purely optimizing connectivity according to standard network metrics. We extend the use of overlay structure management to security by allowing application-driven admission control. Since the resulting overlay services are best-effort, we adapt several transport layer services to our framework to provide congestion and flow control, packet mux/demux, and relay services. Finally, we demonstrate the use of our framework by developing two bulk-data delivery applications.
Flexible Group Data Delivery over Application-Layer Overlays

A Thesis Submitted to the Faculty of
Baylor University
in Partial Fulfillment of the
Requirements for the Degree
of
Master of Science

By
Ashish Vashishta

Waco, Texas
December 2004
# TABLE OF CONTENTS

## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>v</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Data Delivery Services ........................................... 1

1.1.1 Multicast .................................................... 1

1.1.2 Subcast ....................................................... 2

1.1.3 Concast ....................................................... 2

1.2 Network-Layer Deployment ......................................... 3

1.3 Application-Layer Deployment ..................................... 4

1.4 Proposal Of Project ................................................ 5

1.5 Paper Organization ................................................ 7

## 2 Solution Exploration

2.1 Spanning Tree Creation ............................................. 8

2.1.1 Application Semantic Tree Formation .......................... 9

2.2 Secure Communication ............................................... 10

2.2.1 Admission Control ............................................. 10

2.2.2 Secure hop-by-hop tunnels .................................... 11

2.3 Flow/Error Control ................................................. 11

2.3.1 Session Organization Packet Scheduling ...................... 13

2.3.2 Partitioned Organization Packet Scheduling .................. 14

2.3.3 TCP ............................................................ 14

2.4 Peer Cooperation ................................................... 15

## 3 Design And Implementation

3.1 Network-Layer Services ............................................. 16

3.1.1 Application Semantics Based Tree Formation .................. 16
LIST OF FIGURES

1.1 Example: Data Delivery Services ........................................ 3
1.2 Example: Overlay Network .............................................. 5

3.1 Tree Refinement Overview ............................................. 18
3.2 YOID architecture .................................................. 22
3.3 Multiplexer-Demultiplexer ............................................ 24
3.4 UDP application class diagram ..................................... 27

4.1 Topology used ...................................................... 33
4.2 Performance UDP application ..................................... 33
4.3 Performance Comparison BitTorrent and Our Protocol .......... 34
CHAPTER ONE

Introduction

Internet resources have increased at a phenomenal rate over the last several years, but at the same time, application requirements have increased at a much greater pace. Many of these new applications can take advantage of specialized services that enable more efficient use of existing network resources. Applications involving group communication provide a primary example where services such as multicast, subcast, and concast can be beneficial. Such applications can benefit from a mechanism to efficiently replicate or aggregate messages in the network. Currently, the only ubiquitous data delivery service in the Internet is unicast. All other services must be approximated with point-to-point delivery.

1.1 Data Delivery Services

We consider three main data delivery services:

1.1.1 Multicast

Multicast is a service for distributing messages from one or more senders to many receivers. An efficient multicast service works by constructing a spanning tree connecting senders and receivers. Data is only replicated at branching point. Such a multicast service can provide a mechanism for shared data distribution that improves group-size scalability. Streaming audio, real-time video distribution and bulk-data distribution, such as newspaper and software distribution, are examples of these applications.
1.1.2 Subcast

Multicast alone provides a partial solution to group communication applications. In multicast, every message is delivered to every receiver in the group. This behavior is not always desirable. Consider a distributed search application where sender wants to send a query to members of the group. If only a small subset of the group members hold relevant data, multicasting the query to all members wastes network bandwidth. An optimal solution for such applications is to be able to send messages to a subset of members within the same multicast group. Subcast offers this service by allowing messages to be sent to a specified subset of group. Using subcast, messages can be sent to one, some or all members of the group. Multicast then becomes a special case of subcast where the subset is the entire group.

1.1.3 Concast

Some group communication applications require feedback from a set of participants. This feedback may be sent in response to an explicit request for feedback or when triggered by an application specific mechanism. As the group size increases, the number of feedback messages also tend to increase. A large number of feedback messages in a short time span may overwhelm the receiver and overburden the network, resulting in feedback implosion. Concast provides feedback aggregation services. Messages from multiple sources are aggregated in the network to deliver aggregated messages to the receiver. Again consider the distributed search application where the sender of the query requires feedback from the members. Each member sends its response to the query in a feedback message. The intermediate nodes between the feedback senders and the feedback requestor aggregate these messages to deliver an aggregated response to the query. A concast service thus prevents feedback implosion.

Figure 1.1 demonstrates these three data delivery services. The order of the
messages is indicated by step labels given in parenthesis. Assume that every node maintains a range of numbers and node A has a query and node D and C has the answer. Node A multicasts the query on the tree as shown by Step 1. Every member responds to the query by sending a list of values available to its upstream neighbor via the concast service. Step 2 in the Figure shows this. Once node A gets these replies, it knows which part of the tree can satisfy its query, so next time if A has same query, it subcasts it. Step 3 shows how this subcast query gets routed to appropriate nodes.

![Figure 1.1. Example: Data Delivery Services](image)

1.2 Network-Layer Deployment

Even though the provision of these services at the network-layer promises to provide increased scalability, large-scale deployment is yet to be seen. There are several concerns that have prevented this from happening. Narada (Chu, Rao, and Zhang 2000) discusses these issues with respect to multicast; however, the same issues apply to subcast and concast deployment. These issues are summarized as follows:

1. Network multicast requires routers to maintain per-group state. This violates the stateless architectural principle of the original design and also introduces high complexity.
(2) The current network multicast model allows for an arbitrary source to send data to an arbitrary group, thus making the network vulnerable to flooding attacks.

(3) Network multicast is a best-effort service and providing higher level features such as reliability, congestion control, flow control, and security requires application-specific solutions.

(4) Network multicast requires every group to dynamically obtain a globally unique address from the multicast address space, which is difficult to coordinate in a scalable, distributed fashion.

(5) Network multicast calls for changes at the infrastructural level, slowing down the pace of deployment.

1.3 Application-Layer Deployment

The unavailability of network multicast has led to the development of application-layer multicast or overlay multicast. An overlay network consists of a collection of end systems connected to each other via virtual links. The nodes participating in the overlay network establish connections among themselves and form a virtual tree. Figure 1.2 shows an overlay network on top of an existing network. Each of the four nodes is connected to the network via a physical link to the nearest router. Each node establishes a virtual link to one or more nodes and the resulting virtual network is called an overlay network. When node A wants to send a message to node D, it sends the message to either of its virtual neighbors – B or C, as a unicast message. Members participating in overlay networks send data to each other using the underlying network’s unicast delivery mechanism.

With multicast, subcast and concast providing efficient data distribution and application-layer deployment of these being feasible, it seems the basic approach is enough to have
the desired data distribution. However certain questions arise like:

(1) How do I provide reliability?

(2) How do I organize my tree in a manner such that nodes can benefit from their parents, instead of just source?

(3) How do I ensure peers cooperate among themselves, instead of just using the service?

(4) How do I control my flow, should the sender send at the rate acceptable to the slowest receiver or is there a way to group the receivers?

(5) If I can group receivers into different channels then how should I organize my data packets across these channel?

The next section discusses our solution which provide solutions to these problems.

1.4 Proposal Of Project

We propose a new protocol that enables efficient data distribution on top of an overlay network. Our solution creates a maximally flexible overlay tree. To achieve
this we identify the need for a set of services at the transport, network and the application layer.

At transport layer, we provide three new services:

**Flow/Error Control:** We use Forward Error Correction (FEC) and packet scheduling techniques discussed in (Vicisano 1997) and (Donahoo, Ammar, and Zegura 1999) to provide a flow and error control. With \((k,n)\) FEC\((k \leq n)\), we encode the \(k\) original data packets and create \(n\) packets such that we can still rebuild the original data using *any* \(k\) of \(n\) transmitted packets. This coupled with packet organization techniques provides an efficient reliable multicast delivery solution. The goal of packet organization techniques is to organize the data packets across multiple channels in such a way that receivers can join the multicast tree any time but be still able to get the data in the minimum receiving time possible.

**Multiplex/Demultiplex:** Certain applications, like streaming audio, multicasts both data and control information in the same stream to preserve synch. To achieve this functionality at the application-layer multicast we need a way to be able to multiplex the data and control information on same socket and then demultiplex this information on the receiving side.

**Rootcast:** In many overlays, the sender is at the root of the tree, but what if the non root node wants to multicast the data on the same tree? We propose a solution where such a node sends the data to the root of the tree and then the root on behalf of this node multicasts this message.

At network layer, we provide two new services:

**Application Semantic Tree Formation:** We want the overlay tree to be constructed in a way so that applications can benefit from the overlay tree struc-
ture. Thus we want our overlay tree not only to use network metric but also application semantics while making tree refinement decisions.

**Security:** We want to secure our overlay network from any unauthorized access. Not only this we also want to be able to provide secure point-to-point communication among the peers.

At application layer, we propose two new applications:

**UDP Application:** This application uses the FEC/Packet Organization techniques to demonstrate reliable multicast over an overlay network.

**TCP application:** In this application, peers cache the data as soon as they get it from the source and then these peers can act as the source for that particular piece. We develop a novel scheme, using subcast and concast services, where peers not only use the service but also provide service to the network and peers joining the tree is guided by topological information.

1.5 **Paper Organization**

This paper is organized as follows. In Chapter 2, we discuss the protocol requirements. Chapter 3 discusses the design and implementation of our protocol. Chapter 4 analyze the results. The final chapter summarizes our work.
CHAPTER TWO
Solution Exploration

Standard application-layer multicast (ALM) provides a minimal best-effort service. Our application needs more capabilities than that in order to provide flexible and efficient data delivery. This chapter describes previous work in data distribution. First we talk about spanning tree creation, followed by securing the overlay. We, then talk about the flow and error control. Finally we discuss cooperation among peers.

2.1 Spanning Tree Creation

In application layer multicast, participating hosts share responsibility for forwarding information to other hosts. This makes the issue of how we build our spanning tree from source to end-users critical. A poorly created spanning tree can lead to resource wastage and increased delays.

Traditional systems using ALM have focussed on using the overlay-network’s refinement mechanism to optimize the spanning tree. Overlay edge weights are typically inferred through the use of probes that measure various network metrics, such as round trip time or available bandwidth. Narada (Chu, Rao, and Zhang 2000) is one of the earliest application-layer multicast protocols based on the mesh-first approach. It first constructs a virtual mesh among all participants and then run a distance vector routing protocol to create multiple source specific trees. Narada optimizes its mesh using a utility function reflecting mesh quality. A member, i, in Narada computes the utility gain if the link is added to member, j, based on:

1. the number of members to which j improves the routing delay of i; and

2. how significant this improvement in delay is.

YOID (Francis, Pryadkin, Radoslavov, and Govindan 2002) is based on the
tree-first approach of constructing application layer multicast trees. It builds a single shared tree over which data is flooded. YOID uses loss and latency measurements to optimize its tree. Specifically, YOID’s approach relies on local observations of data loss and latency at each node. If a node observes high loss, or high latency, it unilaterally decides to correct the situation by switching parents.

In Content Addressable Network (Ratnasamy, Francis, Handley, Karp, and Shenker 2001) members form a virtual d-dimensional Cartesian coordinate space and each member owns a portion of this space. Key-value pairs are stored on individual nodes in (Ratnasamy, Francis, Handley, Karp, and Shenker 2001) by deterministically mapping a key to a point in the coordinate space using a uniform hashing function. Requests for a particular key are routed by intermediate CAN nodes towards the CAN node whose zone contains that key.

Scattercast (Chawathe 2003) uses delay as the routing cost and builds shortest path trees from data sources. Overcast (Jannotti, Gifford, Johnson, Kaashoek, and O’Toole 2000) explicitly measures available bandwidth on an end-to-end path and builds a multicast tree that maximizes the available bandwidth from the source to the receivers.

2.1.1 Application Semantic Tree Formation

Application semantics based refinement to the spanning tree uses application specific metrics, either in combination with network metrics or by itself.

1. Parallel Downloads - This is important class of applications that can benefit from application metrics. Here a participating node receives data from multiple sources, specifically peers that have already received the data from some other peer.

SplitStream (Castro, Druschel, Kermarrec, Nandi, Rowstron, and Singh 2003) is one such application. SplitStream builds a forest of multicast tree. The data content is split into k stripes and each stripe is distributed on a separate tree. Peers join as
many trees as there stripes they wish to receive. The challenge is to locate peers with both available bandwidth (network metric) and diversity in the set of received data items (application metric).

Random subsets (Kostic, Rodriguez, Albrecht, Bhirud, and Vahdat 2003) provides a mechanism for locating nearby peers that do not share same bottleneck link (and hence have a good chance of containing lost data). Random subsets achieve this by distributing a subset of participants to each node.

2. Content Distribution Networks (CDN) - CDNs can also benefit from application metric. In CDNs, objects are stored at multiple sites spread across the network. Important challenges from the client perspective include resource discovery (determining which replicas store which objects) and request routing (sending the request to the replica likely to serve the node and deliver the best performance under current load levels). Based on the above discussion we want our spanning tree to use a combination of network metrics and application specific metrics.

2.2 Secure Communication

The next solution requirement is securing the group communication. Not only do we want to secure the admission to group but also to provide secure hop-by-hop tunnels over which peers can talk.

2.2.1 Admission Control

Most overlay networks ignore the admission control issue, resulting in no restriction on who can connect to overlay nodes. In BitTorrent (Cohen 2003) there is no admission control. This can lead to a malicious node choking all the peers and be still able to download from other nodes. Such nodes can destroy the topological structure of overlay. For example in CAN (Ratnasamy, Francis, Handley, Karp, and Shenker 2001), a node can cause a routing loop by changing its routing table. We
want to have a scalable authorization method that assigns each member a credential.

In ATG (Cai and Donahoo 2004), when a node wants to join a group, it sends the join request to the Authorization Server. This server verifies the node and creates a credential if the node is valid. This credential, which also includes a time stamp of join, is signed by the authorization server. On successful creation of credential for this particular node, this node is allowed to join the group. The authorization server also broadcasts this information, that a new node has joined the network, to all the already members of the group. When a member leaves the group, the AS also generates a credential and broadcasts it in the overlay.

2.2.2 Secure hop-by-hop tunnels

Admission control is one aspect of secured overlays, the other being secure hop-by-hop data transmission. One way to do this is to encrypt group data with a secret key. Any node can receive such data but only nodes with the secret key can decrypt the data. The challenge is then how to distribute the group key securely within a group. After a node joins, it is easy for the server to distribute the new key encrypted by the old group key. But the challenge is when a node leaves the group.

Again in ATG (Cai and Donahoo 2004), after a node joins the overlay and thus becomes a member of the overlay, it maintains its own credentials via updates broadcast. A member receiving a connection request can verify the credential locally, if the member keeps its own credential fresh according to the update broadcasts. Since every credential includes a time stamp as well as group data, a member is able to make forwarding decisions with respect to its neighbor’s time stamp, which ensures the past and future secrecy.
2.3 Flow/Error Control

Due to heterogeneity in bandwidth available to peers, the sender can adapt to the rate of the least-capable receiver. In such a scenario the other receivers will suffer performance degradation. The other solution to handle receiver rate heterogeneity is to stream data on multiple channels and then each receiver subscribes to set of channels based on its own rate capabilities.

Once we have such a mechanism in place, we need to ensure reliable data delivery. To achieve reliability three solutions have been proposed (Saggi and Donahoo 2003) — ACK-based, NACK-based and FEC-based solutions. In ACK-based reliable multicast transport protocols, the acknowledgment is sent for each message received. We can reduce the acknowledgement messages sent by aggregating these messages as they progress toward the sender, reducing the number of messages at the sender (Chiu, Kadansky, and Wesley 1999). An alternative to this approach is the negative acknowledgment or NACK-based solution. In this scheme, receivers send a negative acknowledgment when they detect loss. If the sender is going to multicast all retransmission, it only needs to know that some receiver needs a particular packet transmitted; therefore, the aggregation at intermediate nodes simply eliminates duplicate NACKs.

In both the above solutions a single loss of data packet by a single receiver can trigger a multicast retransmission of that packet wasting network bandwidth. To overcome this Forward Error Correction (FEC) based solutions have been proposed. In FEC (Lin and Dostello 1983), the sender transmits redundant information so that the original data can be reconstructed at the receiving end without sender having to retransmit packets. In (n, k) FEC, we divide the data into groups of k packets and generate n − k redundancy packets for a total of n packets. Any k of the n packets can be used to reconstruct the original k packets. Some of the advantages of using FEC are (Donahoo 1998):
(1) Any of the n encoded packets can be used to repair data so different receivers with different losses can be repaired by the same packet.

(2) FEC can adapt to the network by increasing or decreasing the redundancy based on perception of error rates.

(3) Delay can be reduced by sending enough redundancy packets in the original transmission so that the data can be reconstructed without retransmissions.

To perform congestion control in multicast environment in a scalable way, receiver-based mechanisms and layered data transmission have been suggested. Among these Receiver-Driven Layered Multicast (S.McCanne, Jacobson, and Vetterli 1996) proposed a TCP-based, receiver-driven congestion control algorithm. Receivers with similar capabilities are grouped together and join the same channel or set of channels.

Once we have data encoding at source using FEC and congestion control mechanism using receiver-driven rate adaptation, we need to have a packet scheduling scheme so as to allow the receivers to join the multicast tree any time they want and still be able to get all the packets in a minimum receiving window. Design of a schedule to accommodate asynchronous requests from the clients is a challenging (Vicisano 1997). Here we discuss two such schemes, Session based and Partitioned Channels.

2.3.1 Session Organization Packet Scheduling

Session based scheme is discussed in (Vicisano 1997) in detail. A session organization (SO) schedule for $n_c$ channels uses a rate of $b_0$ for channel $C_0$ and a rate of $b_0 \cdot 2^{(i-1)}$ for channel $C_i$, $i > 0$, giving an exponential distribution of channel rates. The SO scheme specifies that each receiver joins a session, where session, $S_j$, consists of channels $C_0$ through $C_i$. The ratio of the data rates of the fastest channel to the slowest channel is denoted by $R = 2^y$ where $y = n_c - 1$. If the bandwidth is measured
in packets per second and \( b_0 \) (Channel 0 bandwidth) = \( 1/T \) where \( T \) is the inter-packet time in the slowest channel, \( C_0 \), then the minimum receiving window \( W_i = TP/2^i \), where \( P \) is the number of packets constituting the file being transmitted. This ensures that a receiver listening to a session \( S_i \) can complete its reception in time \( W_i \), no matter when it starts to receive, provided that it does not experience any packet loss.

The SO scheme operates on superblock level. A superblock is a set of \( B \times R \) packets, where \( B \) is the number of blocks in the file and \( R \) is the ratio of the fastest channel data rate to that of the slowest channel. A packet in a superblock is identified by \((p,b)\) where \( p \) is the packet index and \( b \) is the block index. The first superblock contains packets \((0,0)\) through \((R-1,B-1)\). The i-th superblock contains packets \(((i-1)R,0)\) through \((iR - 1,B - 1)\). The sender transmits all packets in the first superblock on the highest session before proceeding to the second superblock.

2.3.2 Partitioned Organization Packet Scheduling

Partition Organization (PO) is discussed in (Donahoo, Ammar, and Zegura 1999). In this scheme, data is partitioned into \( n_c \) equal parts, and each FEC encoded partition is sent over a multicast channel made up of two multicast subchannels, one for carrying original data packets and the other for carrying the redundant packets. The SO scheme does not distinguish between original and redundant packets, while PO does make this distinction. The original subchannel’s objective is to deliver the original data so that the decoding can begin as soon as possible; the redundancy subchannel’s objective is repair of packet losses. A receiver simply listens to the original channel until all of the original data has been sent once, after which it joins the redundancy channel until it receives enough packets such that each block has \( k \) unique packets.
2.3.3 TCP

Both the previously discussed packet organization techniques, work well for our application when used in conjunction to FEC by sending enough redundant packets along with source packets and then not caring about retransmissions. The other approach, to multicast a file, makes use of caching technique at each peer. We use TCP as data transport protocol for such applications to handle the issue of reliability.

Initially the source starts multicast transmission of the file. As the peers get pieces of file, they cache and forward the pieces to their children (Cohen 2003). With caching, the source does not have to worry about multicasting file pieces every time a new peer shows interest in the file. Since the file is cached at various nodes, source failure does not halt distribution.

2.4 Peer Cooperation

Cooperative applications are the ones that allocate a subset of its resources, typically processing, bandwidth and storage, for use by other peers in the application. A large class of applications, like media streaming, P2P applications, can all significantly benefit from a cooperative infrastructure. However such systems perform best if all the users do, in fact, cooperate and provide their share of resources to the system. Experiences with deployed systems, such as Gnutella, show that only a small subset of peers offer selfless service to the community, while the vast majority of users use the services offered by these generous peers (Adar and Huberman 2000). NICE (Lee, Sherwood, and Bhattacharjee 2003) is one work which aims to identify the cooperative users. Applications in NICE gain access to remote resources by bartering local resources. Transactions in NICE consist of secure exchanges of resource certifications. These certifications can be redeemed for the names (remote) resources. Non-cooperative users may gain “free” access to remote resources by issuing certificates that they eventually do not redeem.
BitTorrent (Cohen 2003) is a widely used P2P system that redistributes the cost of upload to downloaders, thus making hosting a file with a potentially unlimited number of downloaders affordable. Peers in BitTorrent download pieces of file from whoever they can and then deciding which peers to upload to via a variant of tit-for-tat. To cooperate, peers upload, and to not cooperate they choke peers. Choking is a temporary refusal to upload; it stops uploading, but downloading can still happen. Thus choking/unchoking a peer forms a basis of peer cooperation.
CHAPTER THREE

Design And Implementation

We propose a set of services at the network layer to create a maximally flexible overlay tree. Once we can construct a tree providing efficient, best-effort data distribution, we add transport layer services common to many applications. Finally we demonstrate the use of these new features by developing two new applications at the application layer.

Though our solution will work on any overlay network that constructs spanning trees, we choose YOID (Francis, Pryadkin, Radoslavov, and Govindan 2002) to demonstrate our approach for several reasons: 1) YOID shared trees are scalable to a large number of users, as compared to trees constructed by protocols like Narada and 2) the single shared tree constructed by YOID saves the overhead of storing information about all the participants of the multicast group at every member.

3.1 Network-Layer Services

3.1.1 Application Semantics Based Tree Formation

Once an overlay tree is formed, services like multicast, subcast and concast can be easily deployed on top. All these services rely on tree shape to provide efficient data distribution. Most of the overlay networks that exist today consider just network metric while refining the tree. If we look at YOID, it continually refines the tree based on network metrics of loss and latency. As discussed in Chapter 2, there are certain classes of applications that benefit from application specific metrics. In the current implementation of YOID, there is no mechanism for custom tree refinement.

In order to provide custom refinement, we identify the need for two such interfaces, one common interface both for the root and node and the other one for the node only. These interfaces are discussed below:
3.1.1.1 Common Interface for Root and Node

TreeRefinementAlgorithmInterface, is the interface that provides basic refinement functionality like starting and stopping the tree refinement. The following methods are provided in this interface:

**startRefinement** indicates that the node can start refining the tree

**stopRefinement** indicates that the node has to stop refining the tree

**processQuery** processes the query and returns the reply. A reply is a YoidMessage object and is null if no reply is generated.

**createContext** creates a context for this particular node. This context is used for further communication about this node.

3.1.1.2 Interface at Node

In addition to TreeRefinementAlgorithmInterface, node also needs some specific functionality before it can start the tree refinement process. The following methods are provided in this interface, NodeTreeRefinementAlgorithmInterface:

**connectToPPPARENTS** finds prospective parents and spawns a thread to maintain the connection to each prospective parent. Once these threads are spawned, they handle their connections on their own and return when there are no members in the member list.

**canStayConnectedToParent** makes a decision whether this node can stay connected to its current parent.

3.1.1.3 Data/Control Handling

We also identify the need for the separation of tree structure management methods from the YOID node’s functionality. Figure 3.1 shows the interactions between node,
tree structure and tree refinement objects in the new model. In the original YOID (Saggi and Donahoo 2003), all these functionalities were embedded in the node and hence no scope for custom tree refinement. As a result of this new interface-driven approach to tree refinement, it became necessary to refactor the data/control information handling. In original implementation the data/control is handled in a procedural way, where a node sends a message (data or control) and then receives the expected reply in a loop. We changed this and provide a new interface that registers with a node and receives events about packets being sent and received. The interface, YOIDPacketNotification provides methods like:

sendingDataToChild sends a data packet to the child

receivedDataFromChild callback to indicate a data packet was received from the child

sendingDataToNonRelative sends a data packet to nonrelative. A nonrelative is a node who is not your parent or child, so a nonrelative is essentially either a prospective parent or child.
receivedDataFromNonRelative callback to indicate that data packet was received from a nonrelative

processControlPacketFromChild callback to indicate that a control message from the child has to be processed

processControlPacketFromNonRelative callback to indicate that a control message is received from a nonrelative

getLevel gets the level of this listener. This has to be one of the five levels (LOWEST, LOWER, MEDIUM, HIGHER, HIGHEST). If the level is HIGHEST, the callback methods of that listener are invoked first. (No order is followed between two listeners of the same level.) It is ensured that once a HIGHER level listener is called, all the HIGHEST level listeners have already been invoked.

Some features of this new approach are:

1. event driven reception of messages
2. uses publisher-subscriber model
3. listeners have different priority, enabling processing phases

Now that we have application semantic-based tree refinement, the next service we provide at the network layer is the security.

3.1.2 Security

In our protocol, we not only want the admission control but also secure hop-by-hop communication tunnels. Once we have such a mechanism, the nodes in our overlay should be able to decide on their own whether to forward a data packet or not. We provide a modestly generic security framework.
We assume that there is an external server that authorizes every node whenever a node wants to join the overlay tree. Whenever this server authorizes the node, it broadcasts this information on overlay tree so that every existing node in tree knows about this new peer. Since this authorization server is not part of the existing overlay, the server unicasts this information to the root of tree using Rootcast service (discussed later) and then root on behalf of server multicasts the information on the tree. To achieve this functionality we provide a BroadcastCallback interface containing following method:

**broadcast**  broadcasts data from the authorization server to every node in the overlay tree

### 3.1.2.1 Client Side

On the client side, we have security manager that lists methods that any security module should provide. These methods include:

**initializeService**  initializes the security module

**authorizeMyself**  authorizes this node from an authorization server

**authNodePassively**  authorizes a node to accept a child. The node that receives a join request from some prospective child calls this method to authorize the requestor.

**authNodeActively**  authorizes a node to join some parent node. After server has authorized a node that node needs to join the overlay tree and for that it sends a join request to some other node and calls this method to get authorized.

**leave**  authorizes a node to leave the network. Whenever a node wants to leave the network call this method to let the authorization server know that it needs tell everyone in the overlay about this node leaving the tree.
processSecurityMessage updates security credentials maintained by node. Whenever a node gets a security message from its parent, call this method to process that message. A security message is received by a node whenever root multicasts packet on authorization server’s behalf.

3.1.2.2 ATG Security

To demonstrate the use of our security framework, we use ATG (Cai and Donahoo 2004). ATG provides a mechanism to secure overlay from any unauthorized access and also provides secure point-to-point communication between peers. We explain our protocol using an example. Say there are two nodes, N1 and N2, which have been authorized by the authorization server to join the overlay. N2 wants to join N1 as its child.

(1) N2 sends a join request to N1. N2 also hands its communication channel to its ATG layer and calls the authNodeActively method.

(2) N1 receives the join request from N2 and hands its communication channel to its ATG layer and calls the authNodePassively method.

(3) ATG performs the authorization and creates a secure channel.

(4) N1 and N2 are informed by their respective ATG layers that the other endpoint is authorized, and both N1 and N2 are given a channel endpoint, which they use for all future communication.

(5) N1 send join reply to N2.

3.1.2.3 Security Module Placement

Since we are using YOID to demonstrate our approach, if we look at the basic architecture of YOID, we notice there are two places where we can put our security
module see Figure 3.2 (Saggi and Donahoo 2003). We can have the module either at the node level or adaptor level. The subcast module relies on current application state to determine whether to forward the packet to each downstream node. Looking at the figure, we realize that this state table is maintained by the adaptor, thus we decided to put our security module at adaptor level.

![YOID architecture](image)

**Figure 3.2. YOID architecture**

The advantage of having security framework at the adaptor level is that, when a packet comes from upstream, the adaptor can make its forwarding decision based on the security state (whether downstream node is authorized to get the packet), which it gets from security module and state table.

If we had put the security module at the node level then forwarding decision must be made in two places. First adaptor decides based on its state table whether to forward or not and then the node makes that decision based on security state.

### 3.1.3 Subcast and Concast Services

The last of proposed services at the network layer are subcast and concast services. Subcast lets members of a group send messages to a subset of group members.
Every time a member receives a data packet, it forwards the packet to the subcast module. The subcast module relies on current application state to determine whether to forward the packet to each downstream neighbor. Application state is maintained in a table, which is maintained by the forwarding modules.

For concast service the root of the tree is the final receiver of all messages. Members send their messages toward the root of the tree by forwarding them to their upstream neighbor in the tree. Each member in the tree is configured with our concast module. This module contains a message aggregation and forwarding algorithm. Every time a member in the concast tree receives a message from a downstream neighbor, the message is forwarded to this module. The concast module updates the current state of the message sender in the state table for downstream neighbors on the basis of the message contents.

3.2 Transport-Layer Services

Now that we have constructed a flexible overlay tree, we adapt several transport layer services to our framework to provide packet mux/demux, flow and error control, and relay services.

3.2.1 Multiplexer/Demultiplexer

The multiplexer/demultiplexer (mux/demux) service lets the user to send both data and control information in the same stream to preserve synch. Figure 3.3 shows how we build a dummy socket for each port. Thus given a port we can send and receive data on different dummy sockets but on same actual socket. The mux/demux object is used for communication to and from dummy socket to the actual socket. We came up with an application that runs on top of YOID using our mux/demux implementation. This application is called launch.yoid and without a mux/demux service it would be difficult to run the application over the YOID because of application’s inherent need
to create control and data listening ports but on same socket. launch.yoid is used to stream wav files over the YOID overlay network.

![Diagram of Multiplexer-Demultiplexer](image)

Figure 3.3. Multiplexer-Demultiplexer

### 3.2.2 Rootcast

The application-layer multicast builds an overlay tree with the source as the root of the tree, allowing the source to multicast the data. When another, nonroot peer wants to multicast the data, it has to start a new multicast group as the root of the tree. This creates significant overhead if the nonroot node just wants to occasionally multicast. A simple solution for this nonroot node is to unicast the data to the root of the tree and have the root multicast the data on the tree on behalf of this node. Rootcast provides this functionality.

### 3.2.3 Flow/ Error Control

The last of our proposed services aims at providing flow and error control at the transport layer. Once we have such mechanism, the next thing is to provide reliable multicast data delivery. For this, we use Forward Error Correction and Packet scheduling. We encode the data at source using FEC and then have a packet schedul-
ing scheme so as to allow the receivers to join the multicast tree at any time and still be able to get all the packets in a minimum receiving window. In our protocol we provide a generic interface to support any scheduling mechanism. We demonstrate that using two different schemes, Session Organization (SO) and Partitioned Organization (PO).

3.2.3.1 FEC Interface

The FEC interface is modestly generic enough. The interface, IDataStriper, provides functionality to encode and decode data, these methods are:

- **encodeFile** creates an encoded file using any (k,n) FEC encoding
- **decodeFile** receives a vector of packets, the indexes of packets, and produces the correct vector as output

For our protocol we use FEC library based on (Rizzo 1997). This library is a Java implementation of Luigi Rizzo’s (Rizzo 1997) C code.

3.2.3.2 Packet Organization

Once we have encoded data packets using FEC, we want to schedule the data packets across multiple channels. For this, we provide two interfaces one for Sender and the other on the receiver side. The sender-side interface, LayerSender, is used to send the packets on the channel and has one method:

- **getPacketIndex** returns the packet index and the block index of the next packet to transmit. Each scheduling technique has its own algorithm to calculate this index.

The receiver-side interface, LayerReceiver, is used to get the packets from a channel. This interface also has following method:

- **havePacket** collects the packets received on a channel
In our protocol, we use two scheduling techniques, the SO and PO schemes, to demonstrate the use of scheduling.

3.3 Applications

Now that we have created flexible overlay tree and provided services at the transport layer, we demonstrate their usability through applications. We develop two bulk-data delivery applications. The idea in both these applications is to split the content into k stripes and to multicast each stripe using a separate tree. Peers join as many trees as there are stripes they wish to receive. The key is to construct forest of multicast trees such that the bandwidth constraints specified by the nodes are satisfied. Participating peers may receive a subset of the stripes, thus controlling their inbound bandwidth requirements in increments of B/k, where B is original content’s bandwidth requirement and k is number of stripes into which content is split. The following sections discuss these applications.

3.3.1 UDP application

The UDP application aims at providing reliable multicast over forest of multicast trees. This application uses the FEC/Packet Organization technique to provide reliability.

3.3.1.1 Class Diagram

Figure 3.4 shows the class diagram for this application. The key components of this application are:

(1) SuperNode: A super node runs on a single machine and consists of number YOID components. A YOID component is defined as either a root or a node. This class uses methods in IDATAstriper interface to encode/decode the data.

(2) SupercastSocket: This class provides methods that the user can use to send
and receive data over the YOID forest. Multiple YOID trees are referred to as YOID forest.

(3) SupercastTreeManager: This class is responsible for managing the YOID components that are part of a particular super node. It can have zero or more node components and/or 0,1 root components.

3.3.1.2 Configuration

To bootstrap the system the user has to download the configuration file, encoded in XML. This file contains:

(1) IP address: the bootstrapper’s IP address

(2) port: the bootstrapper’s listening port

The above entries are repeated for each tree participating in the multicast forest.

(3) length: length of the file in bytes
(4) name: the name of the file

(5) packet size: number of bytes in each packet

3.3.1.3 Protocol

The following steps explain how this protocol works:

(1) The nodes download the configuration file containing the file and bootstrapper information.

(2) The source does a FEC encoding on the data packets.

(3) The source starts the cyclic transmission of file using a packet organization technique.

(4) The other peers interested in the file join the forest. These peers may join a single tree or multiple trees, depending on their bandwidth constraints. Also, the peers can join any time and leave any time.

(5) The peers on receiving data packets, hand it to the decoder, once minimum number of packets required by the decoder, are received.

3.3.2 TCP application

The TCP application aims at solving two key issues:

(1) Random joining of peers: In YOID, a new peer joins the overlay network at a random place. With this application, we want the new peer to find a parent that is appropriate for it, a parent from whom this node can benefit.

(2) Cooperation among peers: We want to make sure that every node is not only using the overlay service but also is providing the service.
3.3.2.1 Configuration

The configuration file is a XML file and is similar to UDP application configuration file.

(1) IP address: the bootstrapper’s IP address

(2) port: the bootstrapper’s listening port

The above entries are repeated for each tree participating in the multicast forest.

(3) length: length of the file in bytes

(4) name: the name of the file

(5) piece size: number of bytes in each piece

3.3.2.2 Protocol

The following steps, explain the protocol working:

(1) The nodes download the configuration file, containing the file and bootstrapper information.

(2) The source starts multicasting file on trees. If there are n trees then source multicasts file on n-1 trees. The first tree is used as a control tree.

(3) Peers join the control tree and the data tree.

(4) Initially peers join anywhere in data tree, but then they use the control tree to find the best place to join the tree.

(5) Once control tree informs a peer of better place to join on a particular tree, a switch is made from current parent to new parent.
(6) Peers cache the data as they get the data from their parent. This allows peer to act as the source of data for the downstream nodes.

The purpose of control tree is two-fold:

(1) Help peers find the best place to join the tree. The best place is defined as the parent who can serve these new peers with data.

(2) Help peers elect new source to stream a piece on a particular tree. Voting mechanism is done using Concast service.

3.3.2.3 *Control Tree Working*

First we explain how control tree is used to find the best place to join the tree. The protocol is explained below with data tree 1 as reference but is same for every data tree:

(1) The source of the control tree multicasts the data packet asking for who needs data piece being transmitted on data tree 1.

(2) Everybody sends a concast response stating if they need the piece or not.

(3) The source on collecting these concast packets send a multicast packet containing information on who all needs the piece.

(4) The nodes send a concast packet containing if they can serve as a source for that piece.

(5) Source subcast this source information (Step 4) to all peers who needed that piece and then those peers connect to the sources.

The second use of the control tree is to encourage peer cooperation. This is done using a voting mechanism. As long as a peer is uploading a piece, that peer is providing service and not just using the service. Every peer maintains a counter,
which is incremented every time a piece is uploaded. This upload can be on any tree. Now there is a voting timer maintained by the source of the control tree on whose expiration a voting is done on control tree.

(1) After voting timer’s expiration, a vote request is multicast on the control tree.

(2) On getting this request everybody sends its counter value using concast.

(3) The source on getting these votes, pick a source with the least value.

(4) The source multicasts this information on the tree.

(5) The old source stops being the source and the new peer is selected the source.
CHAPTER FOUR
Experimental Results

This chapter details preliminary evaluation, demonstrating operation and performance, of our protocol. We measure the average download time for different sized files, using our protocol. Section 4.1 details the experiment setup. Section 4.2 discusses the UDP application-based experiments, and Section 4.3 details TCP application-based experiment.

4.1 Experimental Setup

The experiments described in the following sections were all performed on the Emulab testbed at the University of Utah. This testbed allows users to arrange Emulab nodes in desired topologies to perform experiments in a controlled environment. It also allows users to configure the links between the nodes according to the needs of the experiment. Figure 4.1 shows the topology we used for our experiments. In all experiments, Node 0 is the root node, meaning this node has the file that gets distributed over the overlay tree using our protocol. The solid lines are the network links labelled with link bandwidths.

4.2 UDP Application

We begin evaluation of our framework by means of a UDP application, discussed in Chapter 3. The experiment is done with three different files. In each case, node 0 starts multicasting the file, and soon after nodes 1 to 8 join the overlay in quick succession. Figure 4.2 shows experiment results. The columns with heading 1 and 2 are the rerun of the same experiment. The results for our protocol are shown in columns 4 - 9. The fourth and fifth columns show the results when the application is run using FEC and Session-based packet organization technique. The eight and
ninth columns show the results for the same application but when used in conjunction with a security mechanism, that provides a admission control and secure hop-by-hop channels. The sixth and seventh columns shows the results when the application is run using FEC and Partitioned Channels based packet organization technique.

We also run a BitTorrent protocol of (Cohen 2003), discussed in Chapter 2, on the same topology to show that our protocol performs as well or better than BitTorrent. The comparison of BitTorrent with security application (Columns 8 and 9) indicates BitTorrent performs better in all three cases; however the BitTorrent protocol does not build a secure network and our security application allows only authorized nodes in the network and thus building a secure overlay network.

4.3 TCP application

In this section, we use other application discussed in Chapter 3 to evaluate our protocol. The experiment compares BitTorrent with our application and shows how both respond to scattered joining of peers.

Again the node 0 is the peer with the file to distribute, and we run experiments
Figure 4.2. Performance UDP application

on three different file sizes. Node 0 starts the transmission. After 5 seconds nodes 1 and 2 join the network. Node 3 joins 15 seconds later. Node 1 leaves the network as soon as it completes the download. Nodes 4 and 5 join 20 seconds after the start of the experiment. After 30 seconds, node 6 joins while 0 leaves the network. Finally, 40 seconds from the start of experiment 7 and 8 join the network.

Figure 4.3 shows the results. The columns with heading 1 and 2 are the rerun of the same experiment. As the file size increases, the difference between downloading time of two approaches increase with our approach gaining on performance. This can be attributed to the fact that with small file sizes the control tree does not get enough time to guide the data tree shape.

<table>
<thead>
<tr>
<th>firstRun.dat</th>
<th>Size 47 Kb</th>
<th>BitTorrent</th>
<th>Our Protocol</th>
<th>using SO</th>
<th>using PO</th>
<th>using security</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Avg Time(sec)</td>
<td>11.5</td>
<td>11.6</td>
<td>11</td>
<td>11.5</td>
<td>10</td>
</tr>
<tr>
<td>node 1</td>
<td>13</td>
<td>13</td>
<td>14</td>
<td>13</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>node 2</td>
<td>12</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>node 3</td>
<td>11</td>
<td>11</td>
<td>15</td>
<td>15</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>node 4</td>
<td>10</td>
<td>12</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>node 5</td>
<td>15</td>
<td>14</td>
<td>12</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>node 6</td>
<td>10</td>
<td>9</td>
<td>11</td>
<td>12</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>node 7</td>
<td>10</td>
<td>11</td>
<td>11</td>
<td>9</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>node 8</td>
<td>11</td>
<td>12</td>
<td>8</td>
<td>10</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>secondRun.dat</th>
<th>Size 100 Kb</th>
<th>BitTorrent</th>
<th>Our Protocol</th>
<th>using SO</th>
<th>using PO</th>
<th>using security</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg Time(sec)</td>
<td>10</td>
<td>17.7</td>
<td>17</td>
<td>17.1</td>
<td>16</td>
</tr>
<tr>
<td>node 1</td>
<td>17</td>
<td>16</td>
<td>17</td>
<td>17</td>
<td>18</td>
<td>17</td>
</tr>
<tr>
<td>node 2</td>
<td>18</td>
<td>19</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>node 3</td>
<td>20</td>
<td>20</td>
<td>18</td>
<td>18</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>node 4</td>
<td>18</td>
<td>17</td>
<td>17</td>
<td>18</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>node 5</td>
<td>20</td>
<td>19</td>
<td>22</td>
<td>20</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>node 6</td>
<td>16</td>
<td>17</td>
<td>19</td>
<td>19</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>node 7</td>
<td>17</td>
<td>19</td>
<td>16</td>
<td>16</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>node 8</td>
<td>17</td>
<td>17</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>thirdRun.dat</th>
<th>Size 5.75 Mb</th>
<th>BitTorrent</th>
<th>Our Protocol</th>
<th>using SO</th>
<th>using PO</th>
<th>using security</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg Time(sec)</td>
<td>337.5</td>
<td>338</td>
<td>327</td>
<td>326</td>
<td>321</td>
</tr>
<tr>
<td>node 1</td>
<td>340</td>
<td>345</td>
<td>335</td>
<td>335</td>
<td>330</td>
<td>330</td>
</tr>
<tr>
<td>node 2</td>
<td>320</td>
<td>320</td>
<td>312</td>
<td>315</td>
<td>320</td>
<td>315</td>
</tr>
<tr>
<td>node 3</td>
<td>350</td>
<td>347</td>
<td>330</td>
<td>322</td>
<td>330</td>
<td>322</td>
</tr>
<tr>
<td>node 4</td>
<td>360</td>
<td>361</td>
<td>340</td>
<td>335</td>
<td>330</td>
<td>331</td>
</tr>
<tr>
<td>node 5</td>
<td>300</td>
<td>378</td>
<td>350</td>
<td>367</td>
<td>350</td>
<td>360</td>
</tr>
<tr>
<td>node 6</td>
<td>320</td>
<td>320</td>
<td>330</td>
<td>321</td>
<td>330</td>
<td>321</td>
</tr>
<tr>
<td>node 7</td>
<td>310</td>
<td>312</td>
<td>330</td>
<td>305</td>
<td>300</td>
<td>302</td>
</tr>
<tr>
<td>node 8</td>
<td>320</td>
<td>321</td>
<td>310</td>
<td>308</td>
<td>300</td>
<td>300</td>
</tr>
</tbody>
</table>

Figure 4.2. Performance UDP application.
**Figure 4.3. Performance Comparison BitTorrent and Our Protocol**

<table>
<thead>
<tr>
<th>firstRun.dat</th>
<th>Size 47 Kb</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BitTorrent</td>
<td>Our Protocol</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Avg Time(sec)</td>
<td>12.37</td>
<td>12.12</td>
<td>10.1</td>
</tr>
<tr>
<td>node 1</td>
<td>13</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>node 2</td>
<td>14</td>
<td>14</td>
<td>11</td>
</tr>
<tr>
<td>node 3</td>
<td>12</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>node 4</td>
<td>10</td>
<td>12</td>
<td>9</td>
</tr>
<tr>
<td>node 5</td>
<td>13</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>node 6</td>
<td>12</td>
<td>12</td>
<td>9</td>
</tr>
<tr>
<td>node 7</td>
<td>13</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>node 8</td>
<td>12</td>
<td>13</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>secondRun.dat</th>
<th>Size 100 Kb</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BitTorrent</td>
<td>Our Protocol</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Avg Time(sec)</td>
<td>17.5</td>
<td>17.75</td>
<td>14.3</td>
</tr>
<tr>
<td>node 1</td>
<td>17</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td>node 2</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>node 3</td>
<td>16</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>node 4</td>
<td>17</td>
<td>17</td>
<td>12</td>
</tr>
<tr>
<td>node 5</td>
<td>16</td>
<td>17</td>
<td>13</td>
</tr>
<tr>
<td>node 6</td>
<td>16</td>
<td>20</td>
<td>14</td>
</tr>
<tr>
<td>node 7</td>
<td>17</td>
<td>18</td>
<td>16</td>
</tr>
<tr>
<td>node 8</td>
<td>19</td>
<td>20</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>thirdRun.dat</th>
<th>Size 5.75 Mb</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BitTorrent</td>
<td>Our Protocol</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Avg Time(sec)</td>
<td>334.3</td>
<td>334.6</td>
<td>320</td>
</tr>
<tr>
<td>node 1</td>
<td>340</td>
<td>342</td>
<td>330</td>
</tr>
<tr>
<td>node 2</td>
<td>335</td>
<td>337</td>
<td>335</td>
</tr>
<tr>
<td>node 3</td>
<td>330</td>
<td>326</td>
<td>315</td>
</tr>
<tr>
<td>node 4</td>
<td>340</td>
<td>347</td>
<td>300</td>
</tr>
<tr>
<td>node 5</td>
<td>345</td>
<td>345</td>
<td>310</td>
</tr>
<tr>
<td>node 6</td>
<td>330</td>
<td>328</td>
<td>330</td>
</tr>
<tr>
<td>node 7</td>
<td>335</td>
<td>338</td>
<td>320</td>
</tr>
<tr>
<td>node 8</td>
<td>330</td>
<td>330</td>
<td>330</td>
</tr>
</tbody>
</table>
CHAPTER FIVE

Summary

Services such as multicast, subcast and concast, provide efficient network resource utilization and scalability for group-based applications. Existing approaches provide a single service tailored for a specific application. Our work focuses on creation of a unified framework to provide customizable implementations of these services. In addition, we propose overlay formation based on application semantics, not just standard network metrics. We restrict the overlay service usage to authorized nodes by providing admission control at the network layer. This module uses a generic interface, hence it can easily be integrated with existing security solutions, as we have demonstrated with ATG (Cai and Donahoo 2004).

We adapted several transport layer services to our framework to provide reliability, congestion and flow control, packet mux/demux and relay services. To show the usability of all these services, we developed two data delivery applications.

To evaluate our protocol, we performed experiments using various application models. We compared our results to that of BitTorrent (Cohen 2003). The results indicate our application gains on performance while addressing the same issues that BitTorrent address. The file transfer over our secure overlay slightly losses on performance when compared to unsecured BitTorrent network.
BIBLIOGRAPHY


