Video Streaming in Unstructured Peer-to-Peer networks

by

Craig Furtado

A PROJECT REPORT

submitted to

Oregon State University

in partial fulfillment of

the requirements for the degree of

Master of Science

Presented December 2008

Commencement June 2009

AN ABSTRACT OF THE PROJECT OF

Craig Furtado for the degree of Master of Science in Computer Science presented on

_________.

Title: Video Streaming in Unstructured Peer-to-Peer networks

Abstract Approved:_______________________________

Dr Thinh Nguyen

With the increase in demand for streaming media capabilities across the Internet, the focus has shifted from traditional client-server to peer-to-peer approaches. Content Distribution Networks (CDNs) have also recently moved from web acceleration to media streaming. P2P CDNs can be used both as a delivery mechanism and as an independent network. However, media streaming poses different challenges from traditional content distribution, such as in-order distribution; and p2p networks use more traffic, and lack QoS control and measurement. In addition, constraints like a high churn rate and small upload bandwidths can affect the video playback at the peers. We find that certain strategies can be used to optimize the streaming experience at the receiving nodes, while also being scalable and robust to churn. This project presents the experimental results of MPEG-4 video streaming using different approaches in unstructured p2p networks.

TABLE OF CONTENTS

1. Introduction

Peer-to peer systems consist of multiple nodes spread across a public network and use the cumulative bandwidth of network participants rather than conventional centralized resources where a relatively low number of servers provide the core value to a service or application [6]. In recent years, several systems have emerged for purposes like file-sharing (BitTorrent, Kazaa), telephony (Skype), media-streaming (PULSE, PPLive, CoolStreaming), etc. Peer-to peer networks may also be classified according to their degree of centralization, which may be 'pure p2p', i.e. having no central server or router, where peers perform roles of both server and client; and 'hybrid p2p', where a central server keeps information on peers and responds to requests for information, [6]. All peers provide resources, including bandwidth, storage space, and computing power. However, challenges like scalability, bandwidth-awareness, resilience etc. frequently arise [2]. Thus, as nodes arrive and demand on the system increases, the total capacity of the system also increases. PULSE [3] is a related p2p system for unstructured networks, which places resource-rich nodes close to the source.

In live streaming for events like sporting events, live webcasts, etc. the transmission is characterized by large number of distributed clients, short ramp-up time between fewest connected nodes and most connected nodes, and quick network teardown at the end of the transmission. In such networks, we may either have a video server which contains the seed video to be distributed, or a set of one or more peers contributing the video. In hybrid or structured networks, we frequently need to distinguish between peers such as super-node [7]. Being a p2p network, the other features of peer-to-peer systems are also preserved: peers arrive and depart on demand; resource discovery is supported, etc. In constructing a p2p network supporting video streaming, we consider the case of an unstructured network for live streaming of mpeg4 video. We use the standard audio-video interleaved (avi format), and build our own protocols for distribution, peer-management and stream-management. To implement video playback at the peer, we use the IBM Toolkit for MPEG4 [5].

2. Design

The design goals for this project aimed to create a flexible, efficient unstructured network for video streaming. The p2p overlay network consists of all the participating peers as network nodes. The presence of edges between one node A and another node B is based on whether or not the node A is aware of node B, and vice versa. In an unstructured network, the edges are constructed arbitrarily; provided an incoming peer is aware of at least one node it can proceed to build its peer-list over time. Structured P2P network employs a globally consistent protocol to ensure that any node can efficiently route a search [6]. For this project we consider the case of unstructured networks. A big challenge in using unstructured networks is to efficiently transmit from source to receivers while maintaining transmission efficiency.

Some of the problems that can arise in p2p networks include freeloading, where users do not share resources, asymmetry and variability of bandwidth over time, jitter in the packet arrival times and churn. In structured systems, nodes are organized following a hierarchical tree structure to form an application-layer overlay network [2]. The benefits of this approach include easy analysis and an intuitive understanding of the data-flow. Disadvantages include the problem of finding successors for failed non-leaf nodes, bottlenecks in performance due to low bandwidths at non-leaf nodes and lack of contribution from leaf nodes. In our approach, we use an unstructured network which resembles a mesh like BitTorrent. These can be built and on-the-fly node departures have a lesser effect on the streaming performance since the streaming algorithm does not rely on the structure of the network.

Some of the assumptions that we make are: we do not discard chunks of video that have already been played. For streaming which may potentially have an extremely large bandwidth however, we can consider a sufficiently large window of time for preserving the packets within which *most* packet requests may be answered. We consider a single streaming session, however, multiple video sessions may be established. To demonstrate, some of the peers in our experiment may join late or leave early, with their exit handled gracefully by the overlay network. Peers do not need to recover from packet loss, since the buffer-time window is adjusted appropriately to facilitate retransmission of missing chunks.

2.1 General Architecture

The general architecture can be shown by the following block diagram:

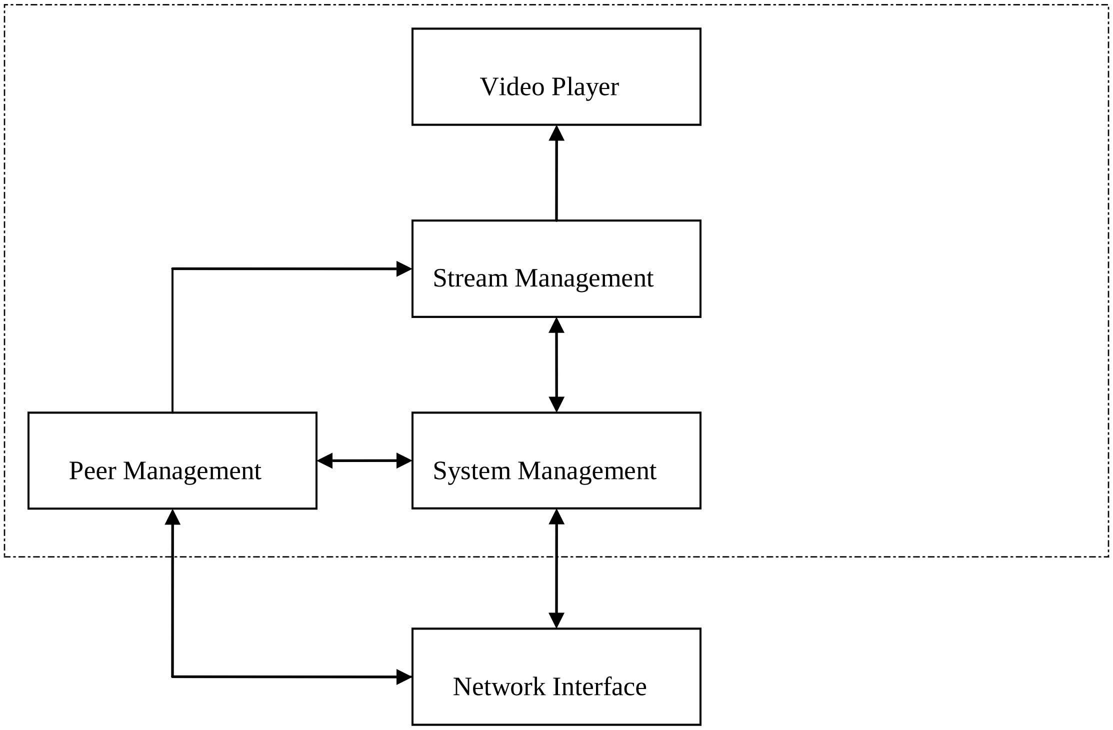


Fig 1. *General Architecture*

The system consists of four modules connected to a network interface:

Peer Management: This module is responsible for managing the peer's neighbors, through peer-join, exchange of buffermaps and node properties, messaging, etc. It is also responsible for satisfying requests of the connected peers. The peer-management function is closely tied to the system management.

Stream Management: This is the set of algorithms that ensures that the stream continues to function till the end of video-playback. While the video is essentially broken into chunks and transmitted via the overlay network, stream management is responsible for assembling the chunks into a single coherent stream. It ensures that the video is sent to the player in order, or that buffering occurs while waiting for chunks to arrive.

Video Player: Video Playback of MPEG-4 video is possible through the use of a set of class libraries.

System Management: This module acts as a bridge between the others. It is responsible for configuring parameters and beginning the streaming sessions.

2.2 Class Diagram

We explain some of the important classes used to construct the system beginning from the lowest level of detail.

Chunk: This class wraps a unit of video data within a payload field. Chunks are transmitted between peers during the session. It also contains a sequence number for the purpose of organizing the data chronologically, and node-data of the sending node.

Node-Data: Instances of the node-data class are passed in messages between peers during the streaming session. The object encapsulates the data about the owner-peer such as IP address, Peer-ID, number of connected nodes and Quality Score. These messages indicate both the presence and state of the peer, and can be used to detect the departure of peers.

Buffer Map: A buffer map of predefined size *BUFFERSIZE* is used to store a fixed amount of sequence numbers of chunks. These are the sequence numbers within a particular window of time. Buffer maps are periodically exchanged so that peers may know the chunks available at their neighbor peers.

Message: This is a generic class that can wrap any object e.g.: of class Node-Data, BufferMap, or Chunk according to the protocol. It can also wrap primitive types and each type is distinguished by means of a Message-ID field. Messages are exchanged between peers during the session.

Peer-Manager: This class is responsible for peer management functions such as peer-join and peer-leave, exchanging status messages and buffermaps. It maintains a list of hash-tables that store the sequence numbers of chunks of its connected peers. Incoming buffermaps from these peers are used to create and update the hash-tables. Peer-Manager also launches a separate multi-threaded TCP server that performs its peer-management functions.

Quality Score: This class maintains and computes the metrics for any streaming session which includes the arrival times, jitter, number of out-of-order packets, number of duplicate packets, peer-uptime, connected nodes at the beginning and end of the session, etc. The quality score of a peer is a weighted function of metrics. These are written to log files by separate threads.

Peer: This is the overall class that controls all the peer functions. It contains instances of the Peer-Manager and Quality Score classes.

A simplified view of the class diagram with some members is shown:

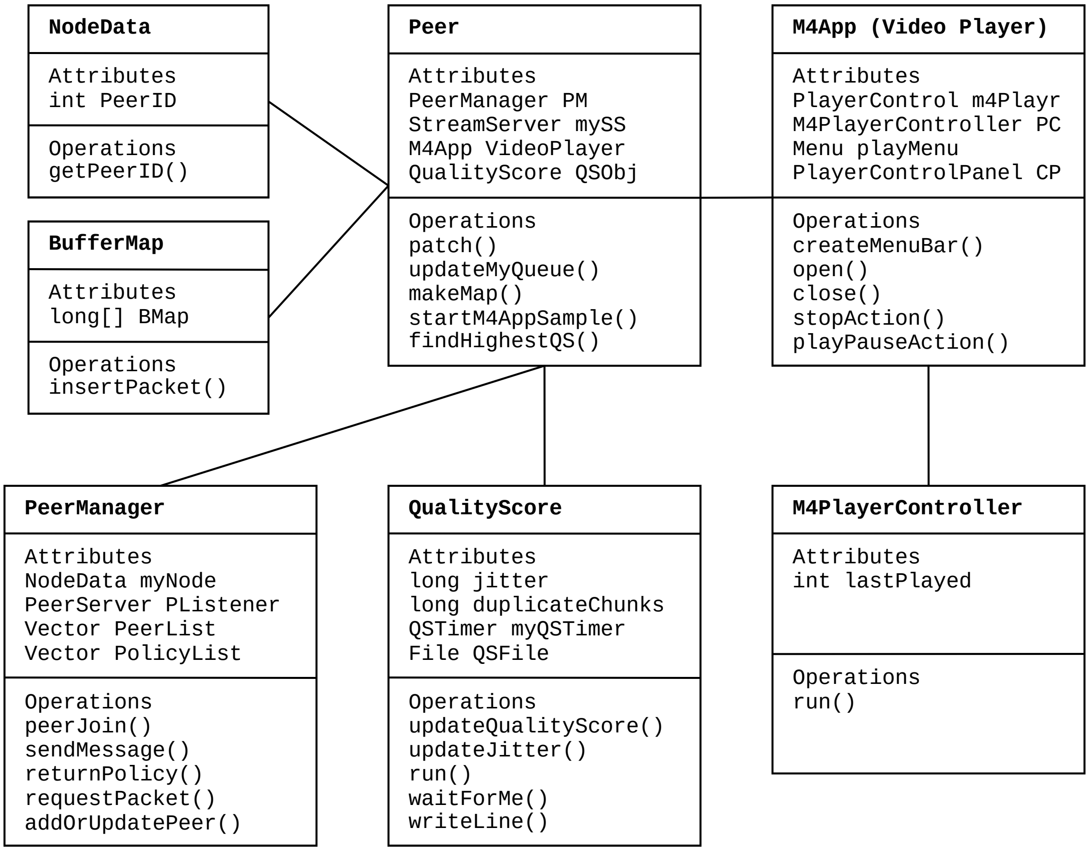


Fig 2. *Class Diagram.*

2.3 Knowledge Management

Knowledge management refers to the knowledge a peer has about the local network. In an unstructured network, a peer is unable to make assumptions about the entire network. However, we can consider the global network to be an aggregate of several local networks. Thus, every peer has a neighborhood which defines the peers with which it may carry out transactions. The peers to which a peer is connected may be divided into two groups, active peers and passive peers. While a peer is aware of both groups, it chooses to interact with only one group of peers due to the selection criteria like QualityScore. Peers may however, move between groups during the length of the session.

The information known to the peer includes:
-*Node-Data* of connected nodes.
-*Buffermaps* of connected nodes
-*QualityScore* measurements at connected nodes

Node-Data are periodically transferred between peers to ensure that the peers are still online. Since the *QualityScore* measurements are part of the *Node-Data,* they are also transferred periodically between peers. Every node has a history queue, in which a fixed number of chunk sequence numbers are stored. This queue is updated whenever a packet arrives at the peer. At intervals, the contents of the history queue are used to construct a buffermap which is sent to the connected peers. Every peer maintains a unique hash-table for each of the connected peers. When a buffermap from peer p_c arrives at peer p_t, the individual sequence numbers from the buffermap are updated in the hash-table for p_c stored at p_t. Thus p_t has an approximately close view of the state of the buffer at p_c, and similarly other connected nodes.

Message-passing: Messages, which are serializable objects, are passed between peers over TCP. Examples of messages include Node-Data request and response, BufferMap broadcast, Peer-Join request, accept and reject, chunk request and response, Peer-Info (which provides information about connected peers), Policy-Tokens etc. A Policy-Token for some node n_j is stored at n_i and determines how streaming from n_i to n_j will take place. The different types of Policy-Tokens include default, even, odd, and none. These designate all, even-numbered, odd-numbered and no chunks respectively. Policy-Tokens are normally initialized to default when nodes connect but can be changed by the requesting node. Consider the following case

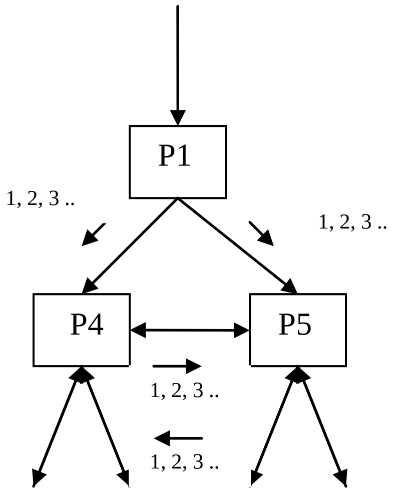


Fig 3a. *Without modification*

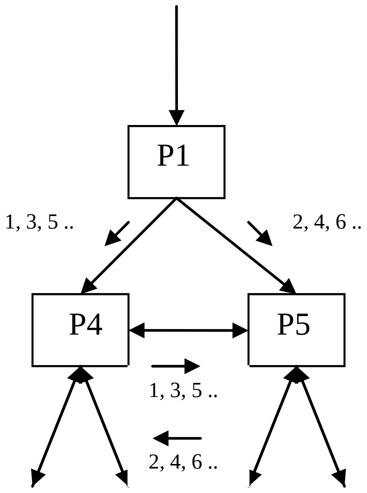


Fig 3b. *With modification*

Although we construct mesh networks, when using a single source the above situation (Fig 3.a) frequently arises. Assume there is a part of the network in which the following condition is satisfied: packets reaching P1 and P5 must be routed through P1. In that case, a stream containing chunks 1, 2, 3… would be duplicated to both P4 and P5. Assuming P4 and P5 are connected, this would be a waste of bandwidth. We implement *streaming policies* between P1 and P4 and between P1 and P5. Thus, P1 sends odd-numbered packets to P4 and even-numbered packets to P5 (Fig. 3b.). The video stream is reconstructed and provides good results (Section 4.4 Fig 1).

2.4 Quality Score and Measurements

One of the criticisms of current approaches to p2p video streaming is that p2p uses much more traffic to deliver the same asset [1], increasing the delivery costs for the network owners. Additionally, it has been claimed that there is a lack of Quality of Service and measurements. Also, most p2p video streaming applications are limited to the upstream bandwidth of the uploading peer. Using a multi-thread based model, we measure the performance of the peers during the streaming session, on the basis of which we make some useful observations characterizing the system.

During a streaming session, one of the most important characteristics is the timely arrival of chunks in playback order. We define jitter as the variation in the packet arrival times. A constant arrival rate would lead to no jitter, which is optimal. Jitter could be caused by network congestion, packet routing changes, or unbalanced loads in the network. We measure jitter for packet p_i as the un-weighted mean of the difference in arrival times of the previous 10 packets i.e. p_{i-1}, p_{i-2} … p_{i-10}. If each of these times is represented by $\Delta t_{i,}$ then jitter *J* is

$$J = \frac{1}{10}\sum_{i=1}^{10}\Delta t_i$$

Assume a new node joins the network. We need to construct the immediate node neighborhood from a set of peers. Given the set of peers to choose from, we need to select those peers with high bandwidth and a low churn rate. We also choose to reward Peers that have a long uptime, low jitter and have contributed more to the stream. Contributions can be calculated by *s*, the total number of chunks sent. Quality Score *Q.S.* is

$$Q.S. = w_1 r + w_2 s - w_3 J - w_4 d$$

Where$w_1, w_2, w_3, w_4$are weights and r, s, j, $c$are the running-time, number of contributed chunks, jitter and number of duplicate packets received respectively. The weights$w_i = 1, w_2 = w_3 = w_4 = 0.5$ chosen were maintained constant for all sessions.

The Quality Score factor can be used while selecting the node peers as well as while choosing peers to actively trade chunks with. The BitTorrent protocol uses a tit-for-tat strategy, which ensures clients send chunks of data back to those clients who contributed to them. However, strict policies like these can result in suboptimal situations, such as when new peers join, which have not yet contributed to the system. Thus, we consider the total number of contributed packets to the *network*, and not to any single peer.

2.5 An Overview of Threads

The system can also be conceptualized as a set of interacting threads of execution. The Java platform is designed to support concurrent programming, and the Java programming language provides basic concurrency support. With the increasing use of multi-core hardware and support for multi-threading in software, we can use concurrency to better utilize computing hardware resources. Since the Java Virtual machine runs as a single process, we can use multiple threads even on single-core systems. The following threads were designed to execute in parallel at any peer node:

1. Playback Thread: This thread is the back-end interface between the media stream and the media player. The playback thread is launched when the video is opened, and monitors the available video stream; if the stream is available it is played, else buffering time is computed. This thread can directly access the media player interface and requires no manual control once playback has begun.

2. Quality Score Thread: This thread maintains data about events such as packet arrivals, duplicate chunks, etc. and writes information to log files. It executes concurrently with other threads and updates statistics like Quality Score. It has an associated QSTimer thread class that provides timing information that can be written and read or re-read from log files, i.e. data is saved between different executions of the program. This models real-world streaming more accurately since churn occurs; peers arrive and depart from the same session while in progress.

3. Stream Server and Stream Server Thread: This is a threaded TCP server that accepts incoming chunks and processes them. For each incoming chunk, a new thread (Stream Server Thread) is launched that processes the chunk. Thus, multiple threads can be independently launched to handle several chunks at a time. Since this is peer-to-peer computing, this component performs functions of both client and server for other peers.

4. Pull and Check tasks: These are background tasks set up to automate the process of either pulling or checking for missing chunks, respectively.

5. Peer Server and Peer Server Thread: These threads are owned by the Peer Manager. They are analogous to the Stream server and Stream Server Threads, but used for the message-passing and knowledge-management functions.

3. Algorithms

3.1 Peer Management

For the experiment, we designate a single peer, called as video-server, as having both the seed video as well as being a bootstrap node. The video-server has almost equal capabilities as other peers but for its ability to begin network construction.
Peer management deals with the algorithms for peer-join and peer-leave.

A peer connects first to the video-server (Peer-ID = 0) and then to other peers. Joining the network is done in two phases:
Phase1: A new peer *p* connects to the video-server (well-known IP) with a peer-join message request. Every peer initially has a priority field (tunable parameter, initially priority = 1) that indicates the number of iterations of requests (a single iteration of requests is defined as all the peers to which it attempts to connect to with the same priority-value.
The video-server (VS) has minimum and maximum limits on the number of peers it can accept; if the request from *p* is within these limits it may be accepted. If it is less than this limit it may be accepted or rejected with equal probability. A peer-accept or peer-reject message is sent to *p* and if-accepted, VS adds *p* to its peer-list and *p* adds VS to its peer-list. VS sends a list of peers P_I to *p* which is a subset of the peers at VS.

Phase 2: The value of priority is incremented. While priority is less than some valueP_{Ma} (P_{Ma}>4), *p* attempts to connect to nodes in the listP_I. As in Phase 1, the request may be accepted or rejected with equal probability from each of the nodes. For every iteration of peer-requests, the priority value is incremented. In order to prevent every request of *p* from being rejected, we stipulate that no node may reject a priority = 4 request. This continues until *p* has sufficient number of neighbors.

Video Streaming Algorithm/ Modes of streaming:

There are various approaches to distribute the streaming media. These can range from pushing, to pulling, to a combination of pushing and pulling. In pushing, the seed peers actively disperse the video through the network. In pulling, the seed peer passively responds to requests for chunks from peers on the network. In the hybrid approach, we explain our results and observations when we permit both pushing and pulling in the streaming session. In this project, we aim to measure the performance of push-based streaming, pull-based streaming and a combination of the two.

3.2 Push-based streaming:

At peer-join time, peers establish some policy by which packets will be pushed to them from each incoming connection. This may be odd, even, default, or none. In the figure (fig 4a.) below, Peer P_1 has three incoming edges where odd-numbered, even-numbered and odd-numbered chunks arrive respectively. It also has outgoing edges where all, even-numbered, odd-numbered, even-numbered and all chunks are sent. Those policies correspond to the nodes to which the packets are incoming.

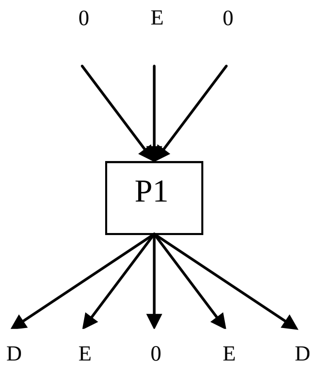


Fig 4a. *Streaming Policies*

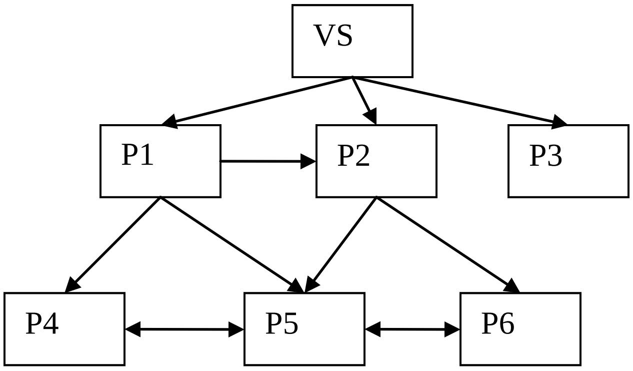


Fig 4b. *Example network*

In this mode, the video server pushes content to each of the nodes successively. The algorithm may be stated as follows:

Pushing algorithm at the video-server

In our approach, the video-server (VS) described has essentially the same functionality as the peers; however, it is also a bootstrap node and contains the seed data (video).

```
Given a video-file at the seed peer (VS) split that file into a series of distinct chunks $c_1, c_2, c_3 ...$
Set $i = 0$
For every chunk $c_i$ do
    Set $j = 0$
    For every peer $p_j$ in the peer-list $P_I$ of VS, do
        Extract the streaming-policy $P(p_j)$ from the policy-list $Pol_L$ at VS
        if P = odd and i%2 = 1 then
            Transmit $c_i$ to $p_j$
        else-if P = even and i%2 = 0 then
            Transmit $c_i$ to $p_j$
        else-if P = default
            Transmit $c_i$ to $p_j$
        end if
    end for
end for
```

At the peer, the following sequence of execution takes place

```
For every received chunk $c_i$ do
    Set $j = 0$
    For every peer $p_j$ in the peer-list $P_I$ of the peer, do
        Extract the streaming-policy $P(p_j)$ from the policy-list $Pol_L$ at the peer
        if P = odd and i%2 = 1 then
    Transmit $c_i$ to $p_j$
else-if P = even and i%2 = 0 then
    Transmit $c_i$ to $p_j$
else-if P = default
    Transmit $c_i$ to $p_j$
end if
end for
end for
```

A parallel sequence of execution at the video-server VS (not shown) handles incoming requests for packets from peers and fulfils them in serial order. In the push-based approach, a similar algorithm as explained above executes at the peers. While peers do not generate chunks, they forward incoming chunks to all other outgoing nodes depending on the streaming policy. In addition, a separate thread of execution periodically checks that the chunks are incoming continuously, else it begins to request chunks (e.g. to prevent excess buffering time due to missing chunks). Thus, the behavior of push in the case of retransmissions approximates pull. This happens as follows:

Algorithm to patch the video stream and request retransmission

To playback video, it is necessary to patch the individual chunks into a stream which can then be played back by the player. Given a set of chunks with sequence numbers, the player considers *lastPatched* as the sequence number of the chunk that can be played last (i.e. chunks with sequence numbers from *0* to *lastPatched* have been received. In contrast, *lastReceived* is the highest sequence numbered chunk received. (*lastPatched<=lastReceived*)

```
Set lastPatched = -1 (the sequence number up till which the video-stream has been reconstructed)
```

```
Let lastReceived be the sequence number of the last received unique chunk with highest sequence number
When the first chunk $c_($ arrives, invoke the following sequence of execution
    For every unit of time $\Delta t$, do:
    if lastPatched < lastReceived
        Determine the missing chunk
    if there exists a peer in the Peer-List $P_I$ which owns that chunk (based on the data contained in hash-tables) then
    Request that chunk from that peer
    else
        Determine the peer with highest Quality-Score and request from that peer
    end if
end if
end for
For every received chunk c with sequence number s, do:
if s > lastReceived, then
    lastReceived = s
end if
Attempt to reconstruct (patch) the video stream if and only if the all the chunks with sequence number S are present, where lastPatched < S < s
if patched, then
lastPatched = s
end if
end for
```

3.3 Pull-based streaming

The pull algorithm takes place with the video-server in a passive role. Peers connect at will and establish a mesh following peer-join. The algorithm for pull resembles the one for retransmission during push.

```
Set lastPatched = -1 (the sequence number up till which the video-stream has been reconstructed)
Let lastReceived be the sequence number of the last received unique chunk with highest sequence number
For every unit of time $\Delta t_1$ do:
    Find the next chunk sequence number = lastPatched+1
    if there exists a peer in the Peer-List $P_I$ which owns that chunk (based on the data contained in hash-tables) then
        Request that chunk from that peer
    else
        Determine the peer with highest Quality-Score and request from that peer
    end if
End for
For every received chunk with sequence number s, do:
if s > lastReceived, then
    lastReceived = s
end if
Attempt to reconstruct (patch) the video stream if and only if the all the chunks with sequence number S are present, where lastPatched < S < s
if patched, then
lastPatched = s
end if
end for
```

The following diagram is useful in understanding the patching of the stream. If we initially have chunks 5 and 6 missing in a sequence from 1 to 10, then we need to wait till chunks 5 and 6 are available before we can patch the stream till the end. Once chunk 6 arrives, it automatically patches the rest to have a complete stream from 1 to 10.

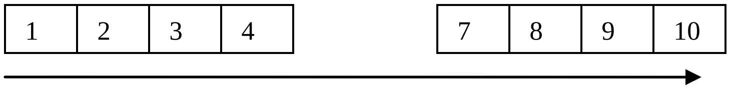


Fig 5a. Stream with chunks, 5, 6 missing. lastPatched = 4. lastReceived = 10

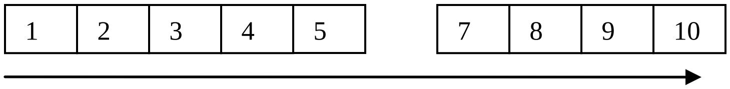


Fig 5b. Chunk 5 arrives. lastPatched = 5. lastReceived = 10

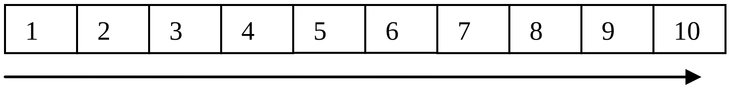


Fig 5c. Chunk 6 arrives. lastPatched = 10. lastReceived = 10

3.4 A Hybrid Approach

From our observations on push and pull strategies, we inferred that pushing may have performed worse for reasons like the large number of duplicate chunks or bandwidth wastage. Nodes having a larger round-trip time from the Video Server could also slow down the streaming for those with a shorter round-trip time. Also, the overall push strategy was not very efficient. We aim to determine if it is possible that a hybrid strategy between pushing and pulling will give results better than those of pull.

In this approach, there will be essentially two flows of media into peers:
1: The stream of chunks that is pushed by the video-server
2: The stream of chunks that is pulled by the individual peers from other peers or the video-server.

Our strategy in the hybrid approach is manifold:
1. To ensure that the bandwidth is utilized effectively, which implies zero or minimal overlap of chunks between streams in 1 and 2 above.
2. To reduce the overhead of pushing at the seed peer, this implies transmitting to some and not all peers, for some and not all chunks.
3. To limit the number of peers connected to any node; this would make our design more scalable.

As we observed the push approach to be slower than pull, at any time we aim to always push those chunks which have a sequence number *greater* than the highest sequence-numbered chunk pulled so far.

This approach has the following changes from the above two:
-To impose a hard-limit on the number of peers that can connect to a single peer
-To pull from multiple peers simultaneously and to pull multiple chunks at a time
-To push to some and not all peers. At the video-server, this is set in *limit,* which is the limit of the number of peers we may push a given chunk to. We push chunks 0 to 10 to

all peers, 11 to 100 to half of the peers, 101 to 1000 to one-third the number of peers, and so on. We do this in order to reduce the repetitive process of pushing.

-We also alternate pushing between sets of sequence numbers, e.g. we push 0, 20 to 30, 40 to 50, 60 to 70 etc.

Below we explain our algorithm. Let *i* denote the chunk sequence number. At the video-server:

```
Given a video-file at the seed peer (VS) split that file into a series of distinct chunks $c_1, c_2, c_3 ...$
Set $i = 0$
for every chunk $c_i$ do
    if $c_i$ does not lie in a transmission interval (e.g. 10-20, 30-40, 40-50, etc) then
        Set I = i+1 and skip transmission
    endif
    if $(0 \le i \le 10)$ then
        Set limit = size(Peer-List)
    else if $(11 \le i \le 100)$ then
        Set limit = 0.5 * size(Peer-List)
    else if $(101 \le i \le 1000)$ then
        Set limit = 0.33 * size(Peer-List)
    end-if
    Set $j = 0$
    for every peer $p_j$ in the subset (0, limit-1) of the peer-list $P_l$ of VS, do
        if $c_i$ has already been pulled then
            Set i = i+1 and skip transmission
        else Extract the streaming-policy $F(p_j)$ from the policy-list $Pol_L$ at VS
        if P = odd and i%2 = 1 then
            Transmit $c_i$ to $p_j$
        else-if P = even and i%2 = 0 then
    Transmit $c_i$ to $p_j$
else-if P = default
    Transmit $c_i$ to $p_j$
end if
end if
end for
end for
```

At the peer, we implement an extension of pull where we pull several chunks in each interval of time, as long as we do not already have them. The algorithm is as follows:

```
Set lastPatched = -1 (the sequence number up till which the video-stream has been reconstructed). Let lastReceived be the sequence number of the last received unique chunk with highest sequence number
For every unit of time $\Delta t_1$ do:
    Find the next missing chunk sequence number = lastPatched+1
For each value of i in $(lastPatched + 1 \leq i \leq lastPatched + 5)$
    if there exists a peer in the Peer-List $P_I$ which owns that missing chunk (based on
    the data contained in hash-tables) then
        Request that missing chunk from that peer
    else
        Determine the peer with highest Quality-Score and request from that peer
    end if
End for
End for
For every received chunk c with sequence number s, do:
if s > lastReceived, then
    lastReceived = s
end if
```

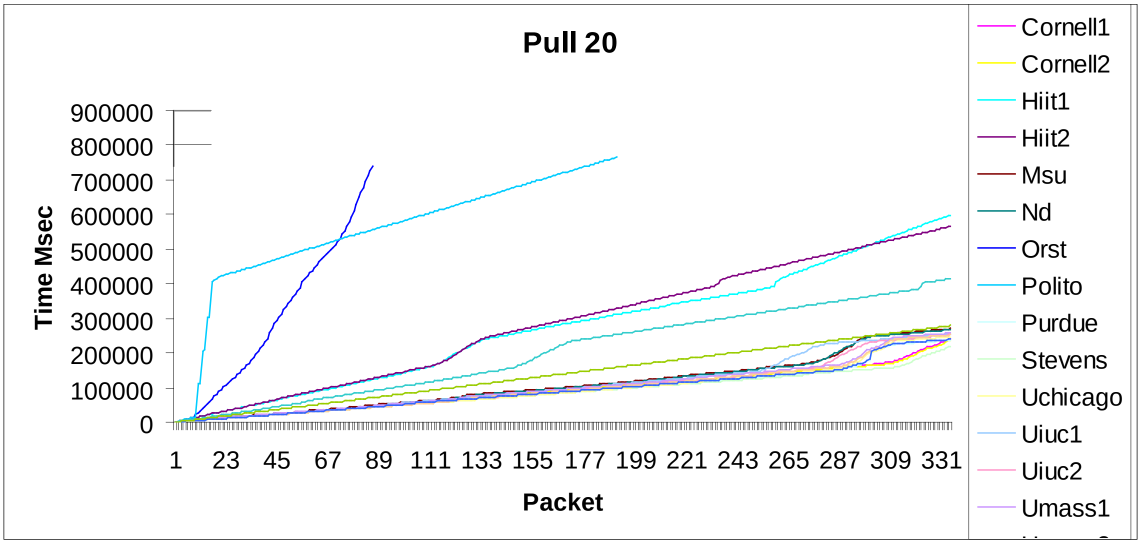
```
    Attempt to reconstruct (patch) the video stream if and only if the all the chunks with sequence number $S$ are present, where $lastPatched < S < s$
    if patched, then
        $lastPatched = s$
    end if
end for
```

4. Performance Evaluation

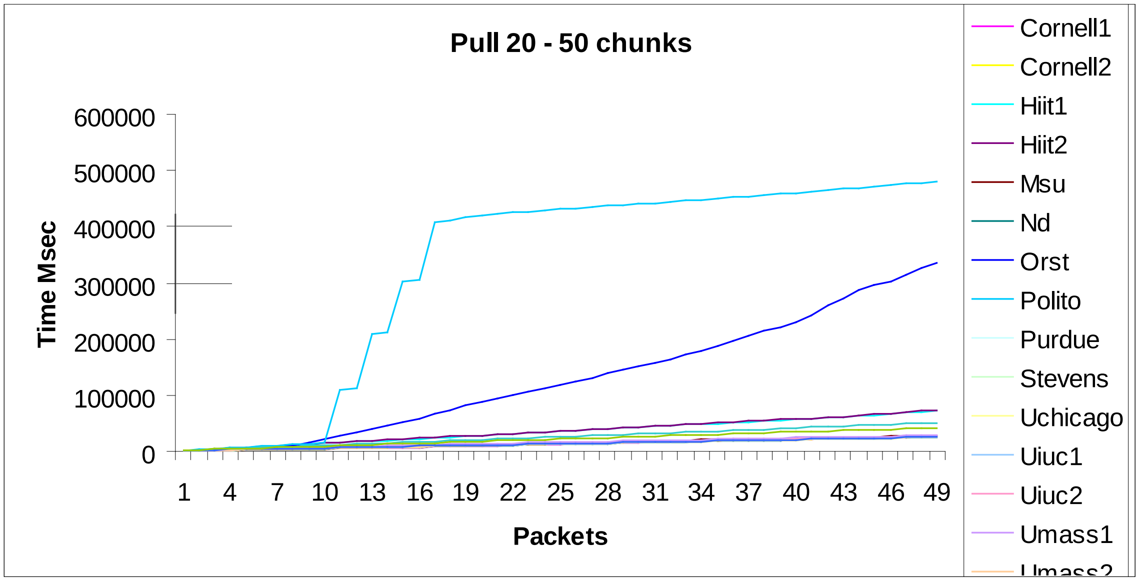
The system was built with a GUI which could be used to begin the video playback. Buffering time depends entirely on the availability of chunks to be played. We used a sample video with a 544 x 304 resolution, with 119 kbps MPEG1 Layer3 audio and 1162 kbit/s MPEG4 video streams, providing for a total data rate of about 1280 kbit/s. The selected chunk-size was 100,000 bytes, which for one chunk approximates to about (100,000*8)/1280,000 = 0.625 sec of playback. We deployed the system on PlanetLab, an open platform for developing, deploying and accessing services over the Internet. The software was run on 20 machines over the Internet repeatedly. We present our results below.

4.1 Pull Approach

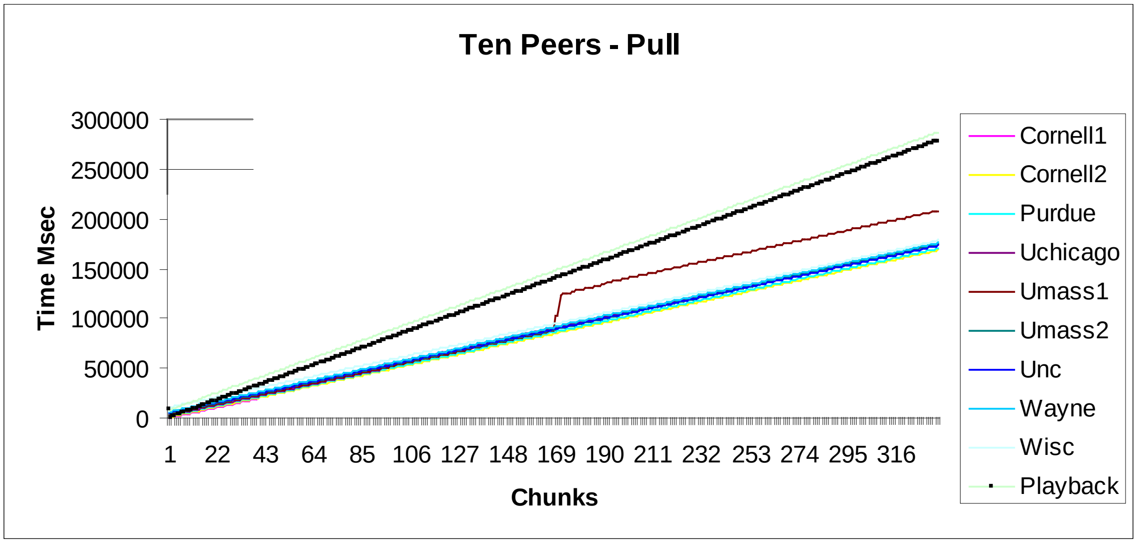
1. Results for the pull algorithm, with number of chunks vs. time in msec. Each of the nodes is labeled on the right. We note the slower performance of two of the nodes, while the rest complete streaming between 220 sec and 600 sec. Most of the nodes complete downloading the file before 300 seconds. The playback duration (not shown) is 225 sec.



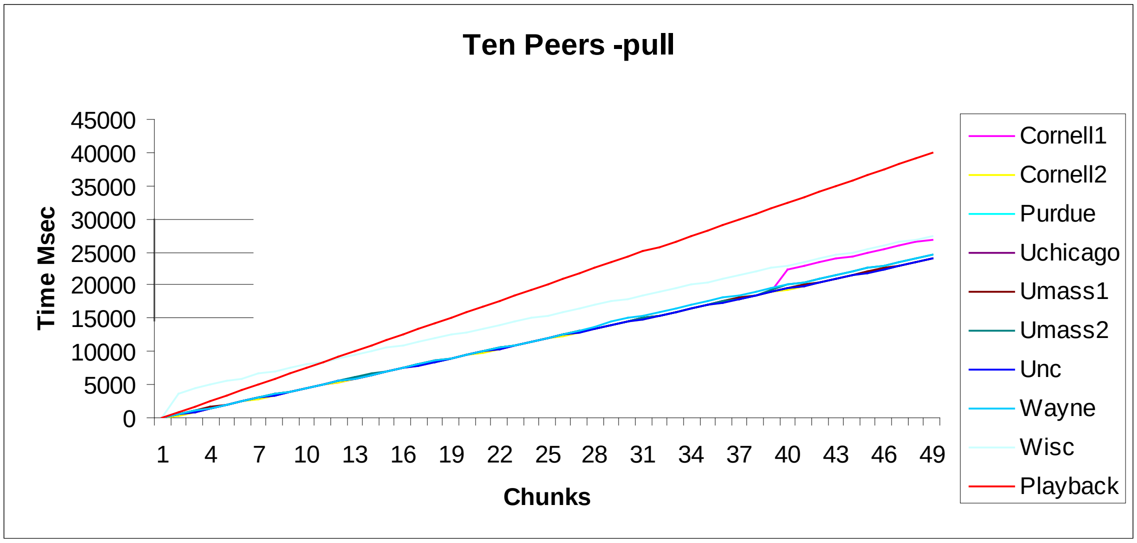
2. For the case of 20 peers, we observe the streaming for the first 50 chunks as in the figure below, for all peers.



3. Results for the pull algorithm for 10 peers. From top to bottom, the jagged line is the playback; the lines below indicate the streaming performance of the peers. The slowest peer completes at 170 sec. Looking at the difference in slope between the playback line and the peer-streaming lines, we conclude that buffering would not be required in this case.

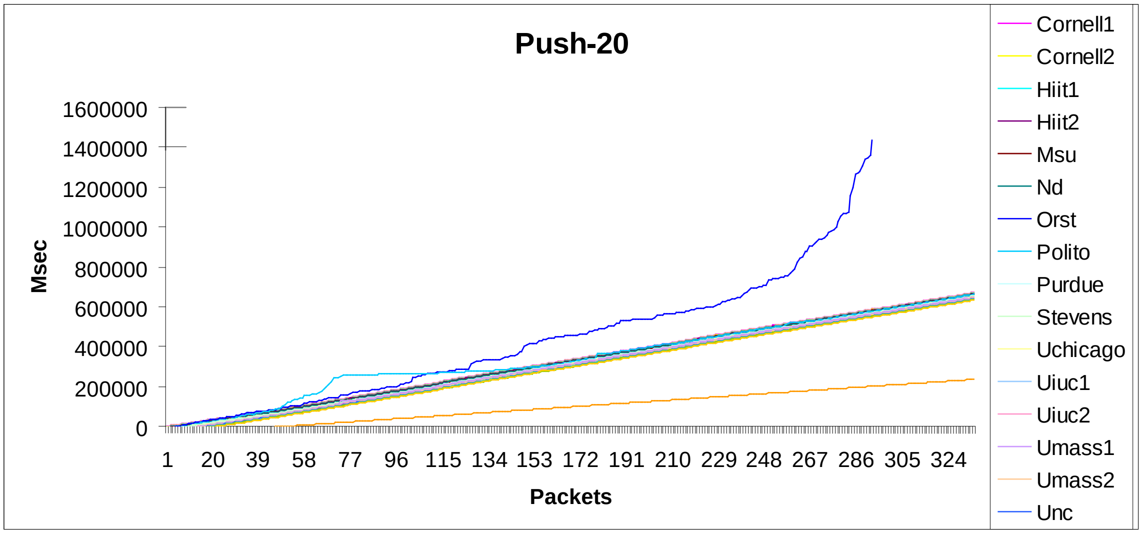


4. The first 50 chunks are blown up for the same algorithm.

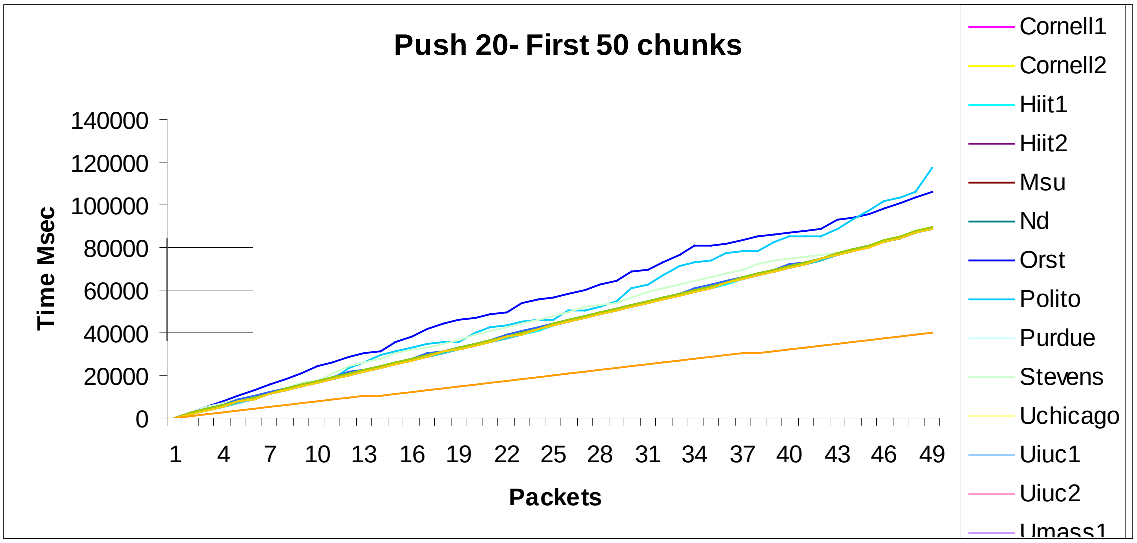


4.2 Push Approach

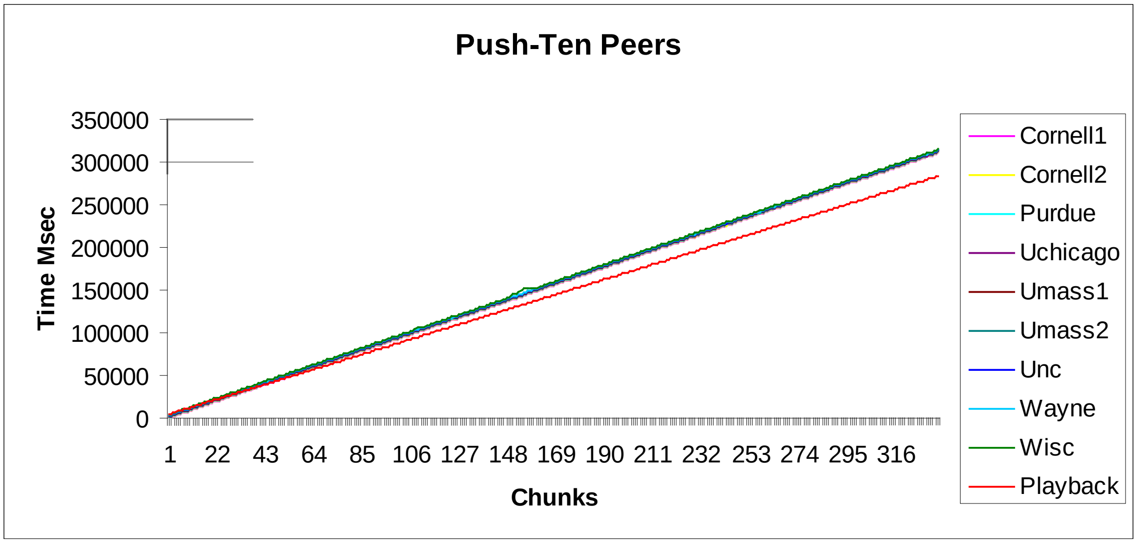
1. The Push Algorithm, for 20 peers, with Time vs. packets. From top to bottom, the first line represents een.orst.edu, which does not complete. We then have a set of lines representing all peers, which complete all at the same time (670 sec) approximately. The lowest line indicates the linear playback rate, which would have to be shifted up on the Y-axis to compensate for the delay (buffering).



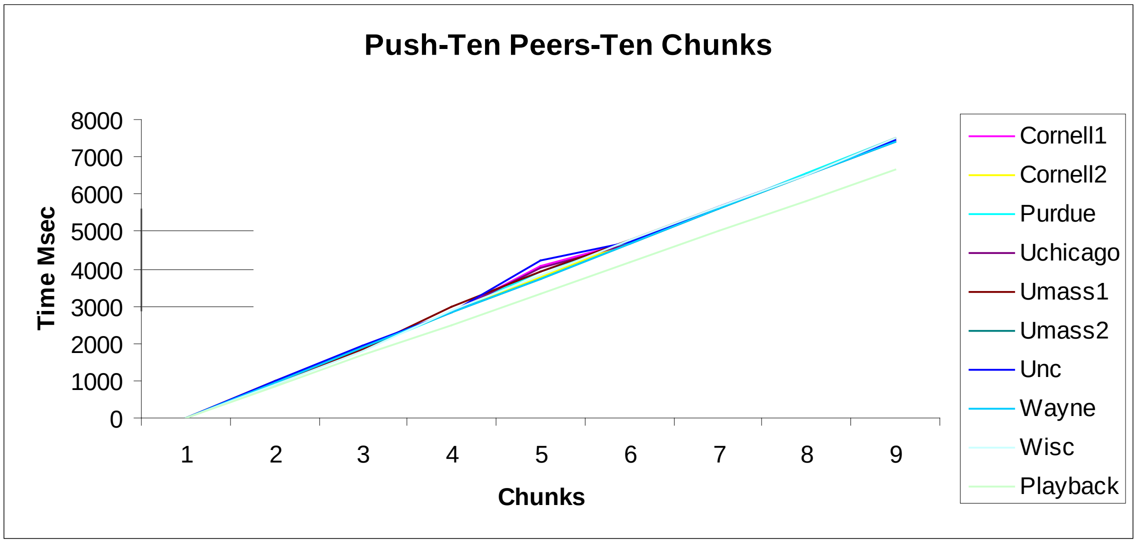
2. The streaming for the first 50 chunks is expanded below. The lowest (straight) line is the playback.



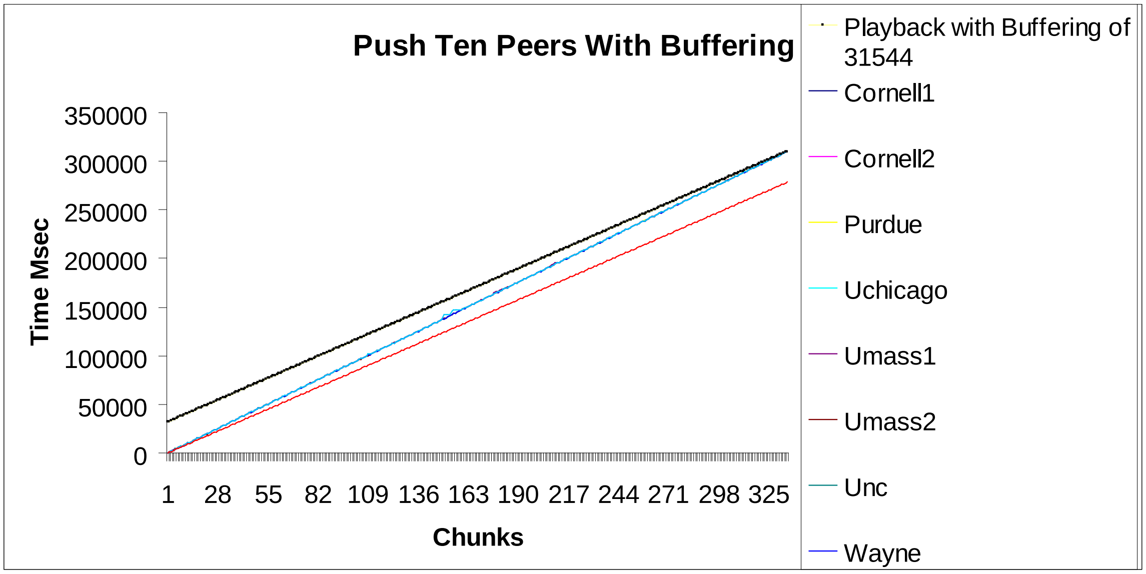
3. Results for the push algorithm for 10 peers. The first set of lines indicates all nodes download times. The lower line is the playback.



4. Only the first 10 chunks are shown, for the same algorithm.

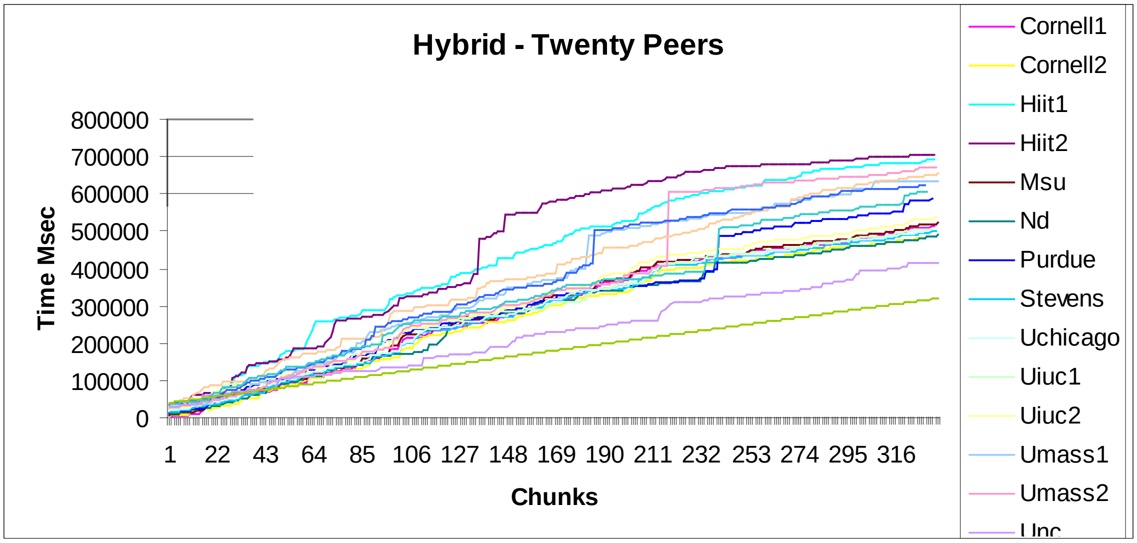


5. The same graph, with a buffering time now added (top black line).



4.3 Hybrid Approach

1. The results of the hybrid approach, for 20 peers, with Time vs. packets. The lowest line indicates the linear playback rate, which implies all peers would require buffering to view the video. We note that the number of chunks arriving out of order is large, and the jitter, or the difference in inter-arrival times, varies a lot. This is indicated by the jagged lines as compared to the smoother lines in the other approaches.



2. We observe interesting results for the case of five peers (below). The network has been graphed to display the connected nodes. In the figure, v.s. is the source node.

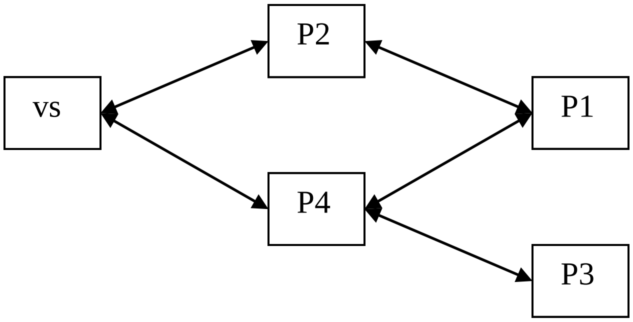
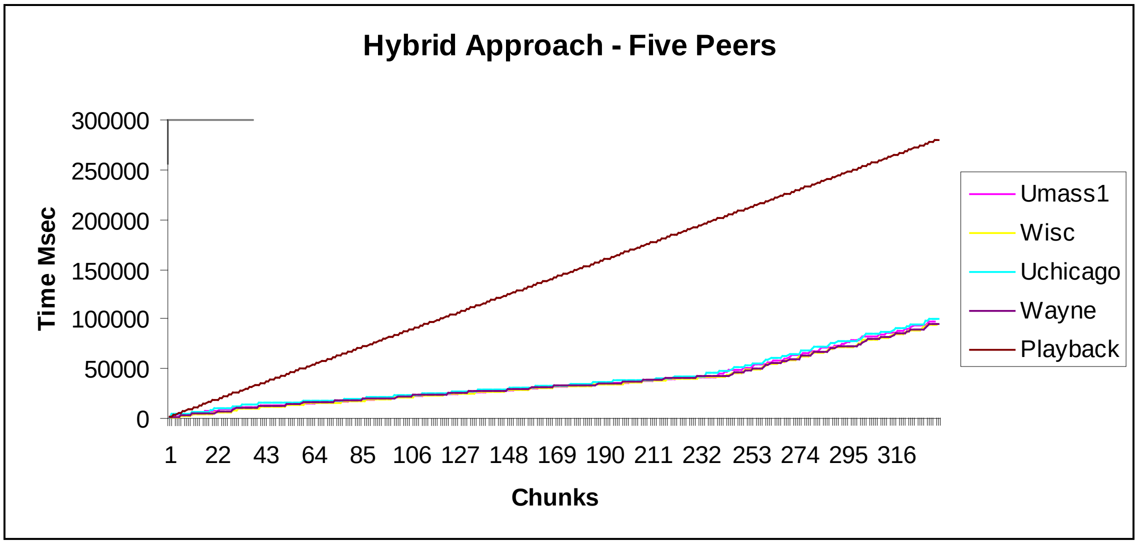


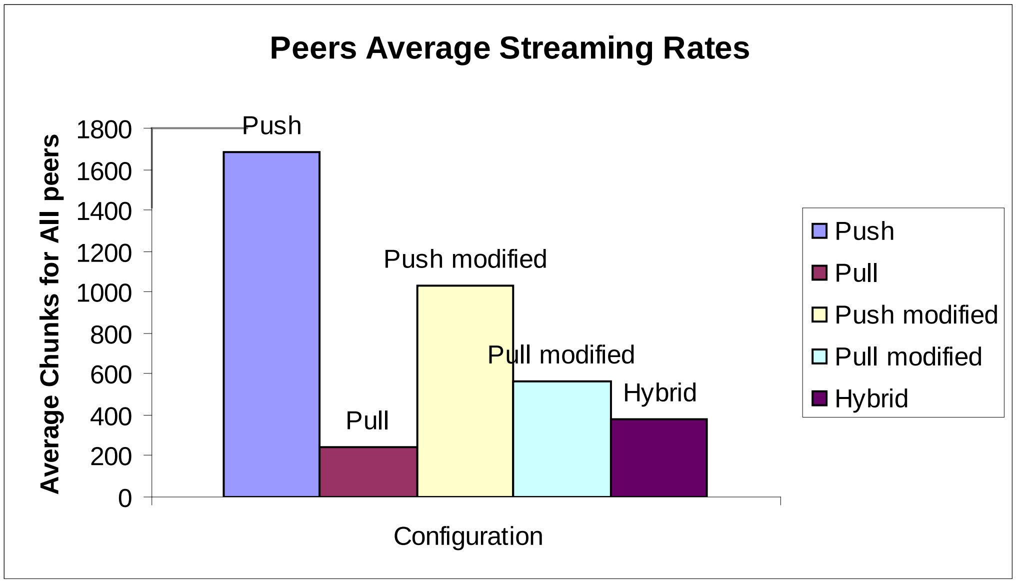
Fig 6. Network graph for 5 peers

3. For 5 peers, the streaming finishes much earlier than the playback time, which implies no buffering and a fast stream. The jitter is also less as compared to the case of 20 peers.

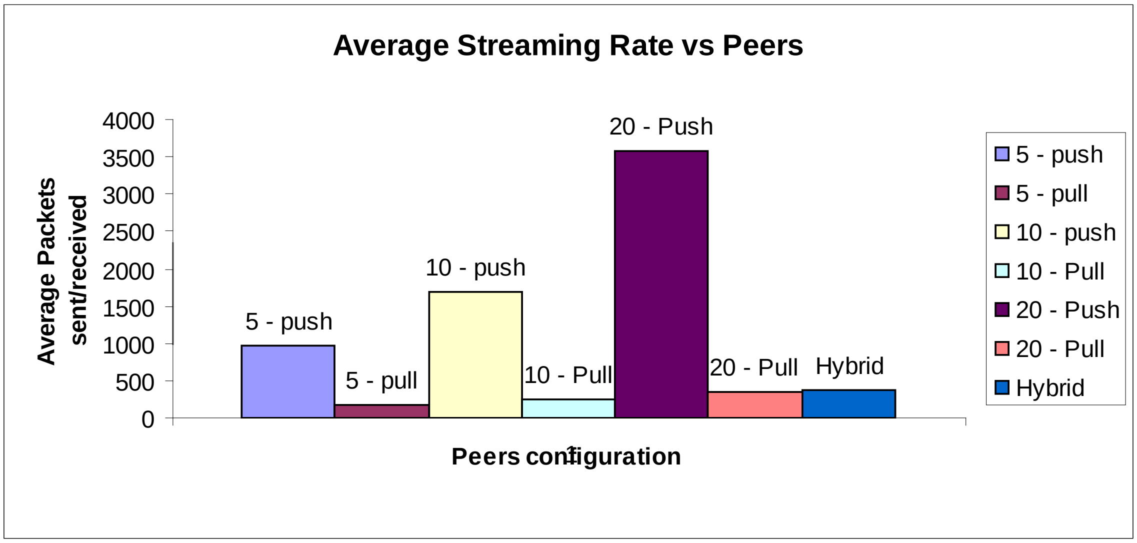


4.4 Other

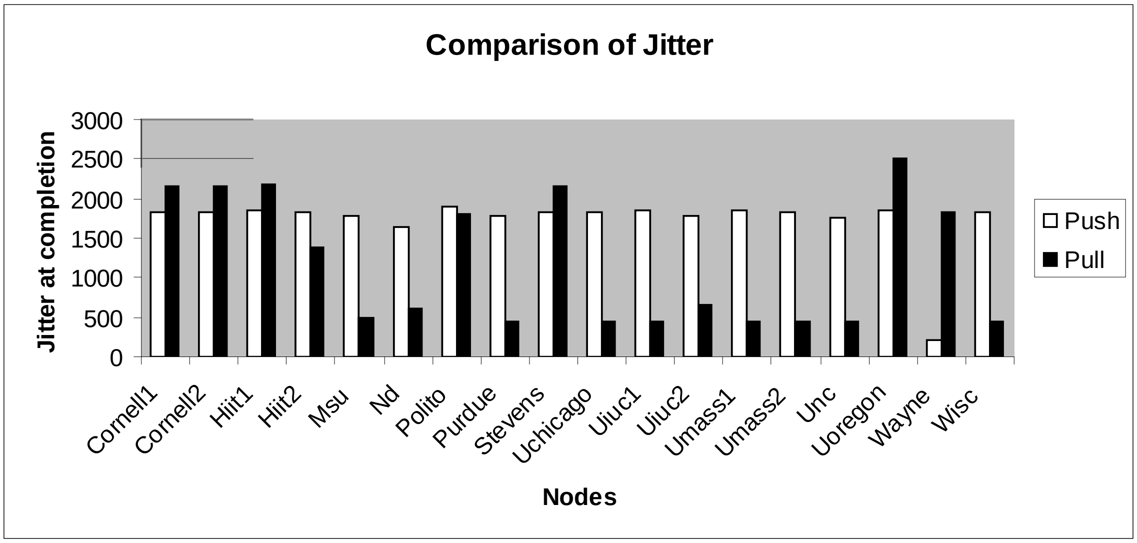
1. The average streaming rates (average number of chunks, both incoming and outgoing) for peers on PlanetLab in a streaming session. The modified algorithms (through the use of Policy tokens for streaming) indicate a more efficient streaming. The hybrid approach also shows a far lower number of packets transmitted on average.



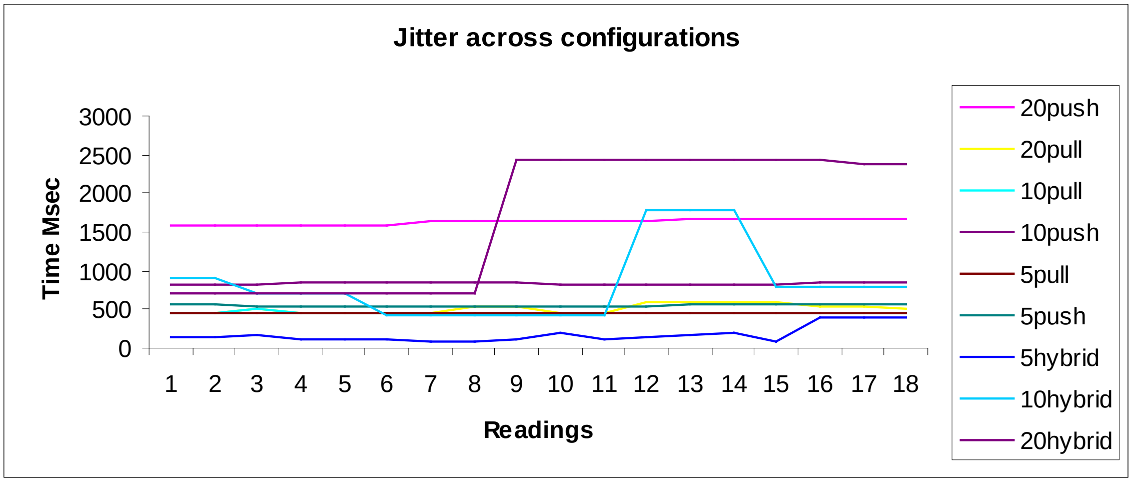
2. We plot the number of chunks sent/received with number of peers to see the effect of increasing the number of peers. The average number of chunks sent/received in each push or pull configuration, first with 5 peers, then 10, then 20 is shown. The pull algorithm was more efficient in terms of number of packets sent and received.



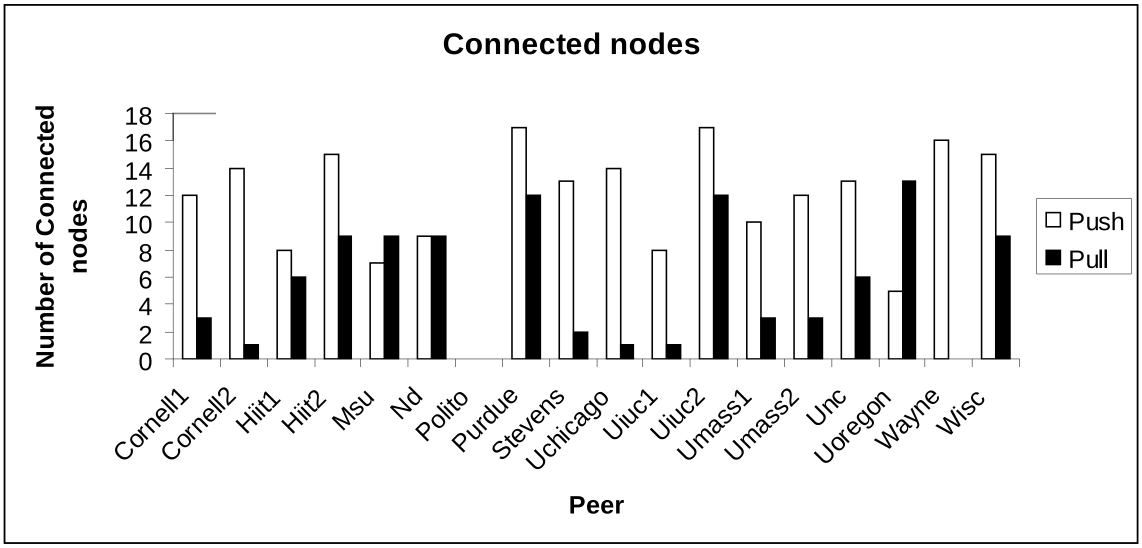
3. Jitter across nodes during a streaming session appears to be relatively constant for push and largely variable for pull.



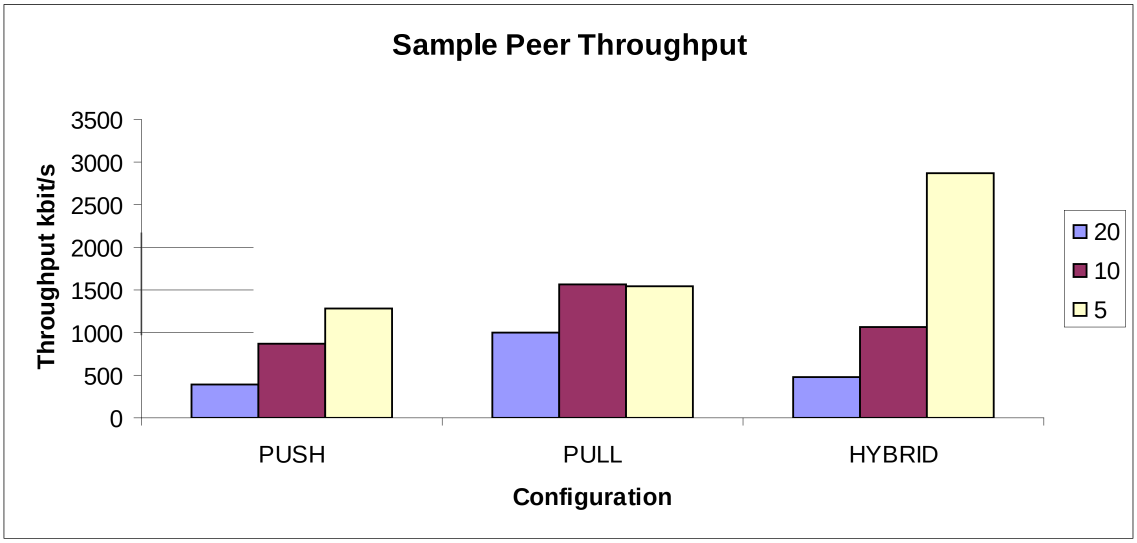
4. A sample peer’s values of jitter were taken below for each configuration. We observe that in general, there are higher values of jitter for push, and lower values for pull and the hybrid approach. Also, as the number of peers increase, we find that total jitter also increases.



5. The node neighbors were randomly generated. Sample number of connected nodes for a streaming session (each pair of values on x-axis represents a different machine).



6. The sample throughput at a single peer is shown below for the different tests. We find that the throughput increases as the number of peers decrease, and that it is largest in the case of hybrid approach (push and pull) for five peers. On average, pull had higher throughput than push.



Conclusions

We observe that in general, pulling appears to be more effective than pushing. In an unstructured peer-to-peer network, it is difficult for a peer to predict whether neighboring peers require a newly-received chunk. This increases the number of duplicate transmissions during a session. To mitigate that, we introduce the concept of streaming policies, and find that average number of transmissions is reduced. However, an advantage in unstructured networks is non-reliance on key nodes; any node may fail gracefully without affecting the remaining nodes. In terms of throughput, we find that pull again performs better on average. However, the throughput decreases as the number of peers increase. In the hybrid approach, we aimed to improve the efficiency of the pull algorithm by pushing selectively; while this was successful for five peers, it was not as successful for larger number of peers.

References

[1] *Content Delivery Network,* http://en.wikipedia.org/wiki/Content_Delivery_Network

[2] F. Pianese. "PULSE. An Adaptive, Practical Live Streaming System", Oct 2007

[3] F. Pianese, J. Keller, and E. W. Biersack. "PULSE, a Flexible P2P Live Streaming System".

[4] F. Pianese, D. Perino, J. Keller, and E.W. Biersack. "PULSE: An Adaptive, Incentive-Based, Unstructured P2P Live Streaming System*", IEEE Transactions on Multimedia, Vol 9., No. 8*, December 2007.

[5]. *IBM Toolkit for MPEG-4*, http://www.alphaworks.ibm.com/tech/tk4mpeg4

[6]. *Peer-to-Peer,* http://en.wikipedia.org/wiki/Peer-to-peer.

[7] T. Nguyen, K. Kolazhi, R. Kamath, S. Cheung. "Efficient Video Dissemination in Structured Hybrid P2P Networks", *IEEE International Conference of Multimedia and Exp.*, 2006.