A Primer on Video Streaming over Peer-to-Peer Networks

Architectures, Challenges, and Security Issues

Ahmed Hamza (#301107073)
Adrian Kwok (#200136359)

CMPT 765 Final Project
December 2nd, 2010

Contents

1 Introduction **1**

2 P2P Streaming Architectures **1**
2.1 Live Streaming . . . 1
Tree-based . . . 1
Mesh-based . . . 2
2.2 Video-on-Demand . . . 3
Tree-based . . . 3
Mesh-based . . . 4
2.3 Hybrid Architectures . . . 5

3 Challenges and Open Issues **5**

4 Security and Threats **6**
4.1 Credit Models: Ripple-Stream . . . 6
Credit Management . . . 6
Overlay Management . . . 6
Discussion . . . 7
4.2 Bandwidth Throttling: Kantoku framework . . . 7
Creation of the Kantoku overlay . . . 7
Monitoring and throttling bandwidth using Kantoku nodes . . . 8
Discussion . . . 8
4.3 Data Integrity and Authentication: Hash Chaining . . . 8
Simple Hash Chaining: Zero Loss Tolerance . . . 9
Augmented Hash Chaining: Burst Loss Tolerant . . . 9
Star Chaining: Completely Loss Tolerant . . . 10
Tree Chaining: Completely Loss Tolerant . . . 11
Discussion . . . 11

5 Conclusions **12**

Bibliography **12**

1 Introduction

The main challenge facing Internet video streaming systems is scalability. The client-server model and a variation known as Content Distribution Networks (CDNs) are currently the main solutions for streaming and distributing video content. However, despite the fact that CDNs reduce startup delays by pushing the content to a set of content delivery servers closer to the clients, both models incur significant bandwidth costs for content providers and are incapable of scaling well in proportion to the client population.

Recently, peer-to-peer (P2P) emerged as a promising technique for providing large-scale media distribution systems over the Internet. The two key points that make P2P networks attractive are: (i) they are cost effective and easy to deploy because they do not require changes to the Internet infrastructure; and (ii) they are able to self-scale as the resources increase with the number of users.

In this report, we survey the various architectures used for developing live and video-on-demand P2P streaming systems with emphasis on the advantages and disadvantages of each architecture as well as the tradeoffs between them. We then briefly mention some open research issues. Finally, we focus on possible security attacks on these systems, and some of the solutions in the literature that attempt to mitigate these attacks.

2 P2P Streaming Architectures

Based on the overlay network structure, P2P streaming systems can be classified into two categories: tree-based systems and mesh-based systems. Both categories can support live streaming and video-on-demand streaming with various degrees of efficiency. Based on the type of streaming, each category has its advantages and disadvantages; live video streaming disseminates the content to all users participating in the system synchronously and in real-time, while video-on-demand (VoD) enables users to view the video content at their convenience and gives them the flexibility of starting the stream at different points in time.

2.1 Live Streaming

Tree-based

The initial approach to providing media streaming over the Internet was through IP multicast. In IP multicast, a multicast tree is constructed over the IP routers to which the source server and participating users are connected. However, up to this day, IP mutlicast deployment still remains limited due to the need for infrastructural changes and the overhead imposed on the routers in order to maintain per-group state. Thus, it was advocated that the multicast functionality should be pushed away from the routers and towards the end systems via overlay structures. Tree-based P2P systems achieve this by forming a multicast tree at the application-level between the participating peers.

However, tree-based streaming systems suffer from a major drawback – these systems are vulnerable to peer churn, i.e. the unpredictable and independent arrival and departure of peers. The departure of one internal peer in the tree would temporarily interrupt the delivery of the video to all peers in the sub-tree rooted at the departed peer. Moreover, recovering from such failures and performing tree maintenance is a costly process that takes time and requires the exchange of many messages between the peers.

Single-tree Streaming

Single-tree systems organize peers into a multicast tree in which there are well-defined parent-child relationships and content is pushed from a parent to its children. New peers join the tree at a certain level and receive the content from their parent. The main considerations involved when constructing the tree are the depth of the tree and the fanout of internal peers. Even though the fanout should be as wide as possible in order to reduce the depth (and thus reduce latency), the maximum fanout degree of a peer is limited by its upload bandwidth. Examples of single-tree systems include ESM [2] and Overcast [12], and PeerCast [6].

Multi-tree (Forest) Streaming

In single-tree P2P video streaming systems, a large fraction of the peers are leaf nodes which do not contribute their upload bandwidth to the system. In a balanced tree with a fanout f and height h, the number of internal peers is $(f^h - 1)/(f - 1)$ while the number of leaves is f^h. Thus, in a balanced binary tree, for example, more than half of the peers are leaves. This leads to inefficient bandwidth utilization. Multi-tree-based systems attempt to address this problem by constructing multiple trees, also known as a forest, between the peers, such that a peer has a different position in each tree. Thus, multi-tree systems provide two main advantages: (i) they improve the overall resiliency of the system; and (ii) the potential bandwidth of all nodes can be utilized, provided that a node is not a leaf in at least one tree.

The stream is divided into multiple sub-streams and each sub-stream flows over a separate tree. The sub-streams should have approximately the same bandwidth and the receiver should be able to reconstruct the original stream from any subset of these sub-streams, albeit with various degrees of quality. Thus, specialized coding techniques such as multiple description coding (MDC) are necessary to leverage the full benefits of this approach. One representative example of such systems is SplitStream [1], shown in Figure 1. To distribute the forwarding load efficiently over all participating peers and achieve high bandwidth utilization, the number of trees in which a peer is placed as an internal peer can be set in proportion to the peer's uploading capacity.

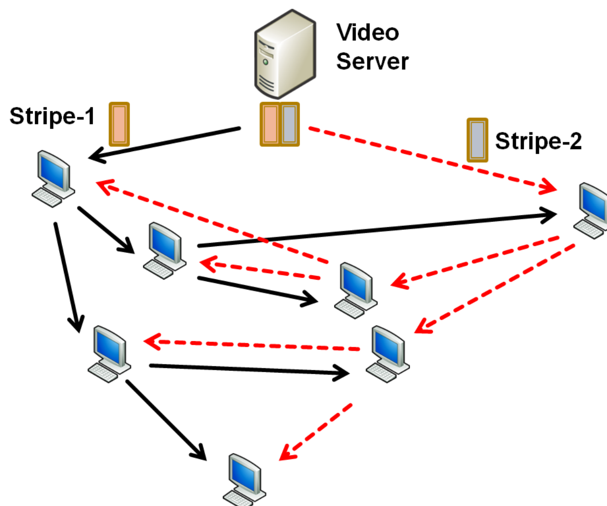

Figure 1: SplitStream

Mesh-based

Similar to P2P file sharing systems such as BitTorrent [3], peers in mesh-based streaming systems dynamically establish and terminate peering relationships. Thus, at any given time, a peer maintains peering relationships with a set of neighbors. Based on this relationship, a peer can either upload or download content to or from a number of its neighbors simultaneously. If a neighbor departs from the system, the peer can still obtain the content from other neighbors that have it. The peer can also seek new neighbors to maintain a certain degree of connectivity. This can be done by either contacting the tracker server which keeps track of active peers in the system, or through gossiping protocols that exchange neighbor sets between neighboring nodes.

Unlike tree-based live streaming systems, mesh-based systems do not have the notion of a continuous stream flow from the source to all other peers. Video content is divided into small video chunks that have unique sequence numbers. Because there is no static structure in a mesh network, each chunk may take a different path in the overlay network to reach the destination peer. Thus, chunks may arrive out of order and buffering is required at the receiver to reorder them before playback. Based on the chunk

exchange mechanism, there are two kinds of mesh-based streaming systems: *mesh-push* systems, and *mesh-pull* systems.

In mesh-push systems, peers actively push received chunks to neighbors that do not have them. Chunk push schedules should be carefully planned between the neighbors in order to avoid wasted upload bandwidth due to redundant pushes. To avoid such complications, peers in mesh-pull systems explicitly ask for chunks that they need from neighbors who have them. To facilitate this, each peer maintains a chunk availability map, called a *buffer map*, which it periodically distributes to its neighbors, shown in Figure 2. Based on the buffer maps collected from neighboring peers, a peer would then decide on a chunk pull schedule.

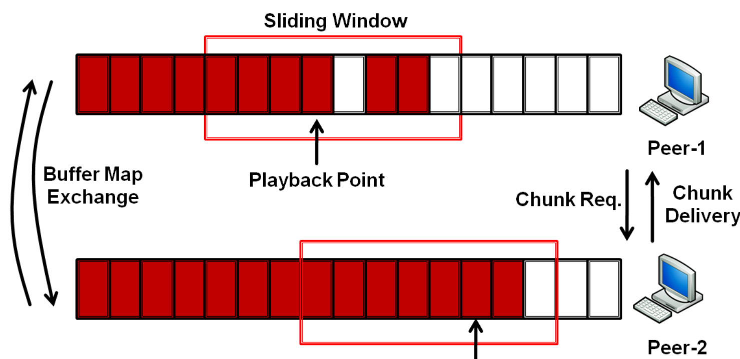

Figure 2: Buffer map exchange in mesh-based systems.

Even though mesh-based P2P streaming systems may appear very similar to P2P file sharing systems like BitTorrent, the main difference between the two is the real-time constraints required from streaming systems so that segments/chunks are guaranteed to arrive in a timely fashion in order to meet their playback deadlines. Efficient scheduling algorithms are thus of great importance and a key component in these systems in order to provide better playback quality. This sceduling problem is, however, a variation of the *parallel machine scheduling* problem which is known to be NP-hard. Thus, it is not easy to find optimal solutions, especially since any algorithm would need to do so quickly in order to adapt to network dynamics.

2.2 Video-on-Demand

Tree-based

Tree-based P2P systems were originally designed as an application layer replacement of IP multicast. Thus, it is more suitable for streaming sessions in which all users are synchronized and receive the content in order. However, there have been some proposals for adapting tree-based systems in order to support VoD functionality. One example is P2Cast [10], which was inspired by patching schemes that enable VoD support in IP multicast. In P2Cast, clients that arrive close in time within a predefined threshold constitute a session. Clients within a session form an overlay multicast tree with the server, which is called a base tree. The server streams the entire video over each base tree. A new client joins one of the base trees based on its arrival time and receives the stream from its parent in the tree.

However, the newly joined client needs to obtain the portion of the video that precedes the point in time in which it joined the network. In P2Cast, this portion is termed the patch. The patch can either be obtained from the server or from any other node in the tree (since all other nodes in the tree arrived earlier and have already cached part of the patch that the new client requires). This way, all nodes in the tree receive the base stream synchronously while patches are delivered asynchronously as a separate stream. Figure 3 shows a snapshot of P2Cast at time 40 using a time threshold of 10.

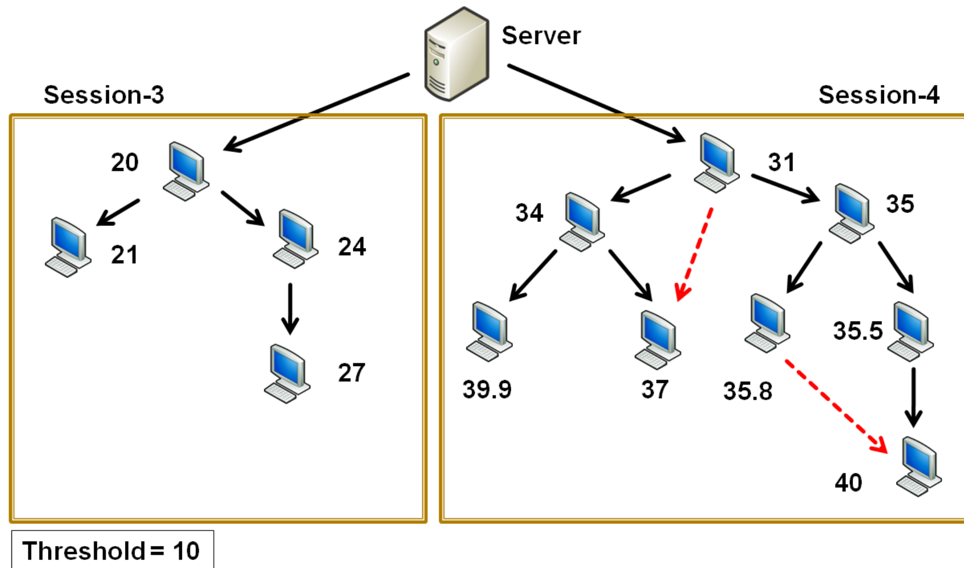

Figure 3: A snapshot of P2Cast at time 40.

Mesh-based

In order to achieve the highest possible download throughput in mesh-based P2P systems, the chunks available at different peers should be as different as possible. For streaming applications, this diversity requirement, however, conversely affects the smoothness of playback. This effect is greatly amplified by the fact that in a VoD service, the users are viewing the content at different points in time. Thus, it is required to balance the overall system efficiency with the sequential playback requirements for asynchronous users. Scheduling chunk downloads in order of playback is inefficient since new peers joining the system will not have anything to contribute to existing peers. In addition, the number of peers that will be able to contribute to this peer will decrease with time since peers that arrived earlier will finish watching and may leave the system.

BiToS [13] is a mesh-based P2P system which modifies the BitTorrent protocol in order to support VoD service. The main difference between BiToS and BitTorrent is in their scheduling (chunk selection) process. As shown in Figure 4, peers in BiToS maintains three sets: a received chunks set, a high priority set, and a remaining chunks set. The received chunks set is a buffer that stores all the video stream chunks that have been received so far. The high priority set contains the video chunks that have not been downloaded yet but are close to their playback deadline. The remaining chunks set contains the rest of the chunks that have not been downloaded.

Figure 4: BiToS and its three chunk sets.

When deciding on which chunks to download, a peer chooses with some probability p to download a chunk from the high priority set, and with probability $1 - p$ to download a piece from the remaining pieces set. If p is greater than 0.5, the chunks in the high priority set are favored for download earlier. The mechanism used to choose a chunk within both the high priority set and the remaining chunks set is the *rarest-first* mechanism, which is the original mechanism of BitTorrent. However, if two or more chunks have the same rareness, the one closer in meeting its deadline will be chosen.

2.3 Hybrid Architectures

Tree-based and mesh-based systems each have their advantages and disadvantages. There have been attempts to combine the two approaches to realize a hybrid overlay that provides the best of both worlds and is both efficient and robust. One possible direction is the tree-bone based approach represented by the mTreebone system [15]. In mTreebone, a chunk distribution tree, known as the *tree-bone*, is constructed between a set of nodes which are both stable and have high upload capacity. These nodes are referred to as *super nodes* and, because of their properties, video chunks can be pushed to them quickly from the video source.

The super nodes along with the rest of the nodes will further be organized as a mesh-pull-based overlay which will enable exploiting the available bandwidth in leaf nodes. Since super peers would contribute more upload capacity than other ordinary peers, the main question to ask would be: what are the incentives for a peer to become a super peer? One possible answer to this issue is to provide differentiated services based on the type of the peer. Thus, ordinary peers receive basic video playback quality while super peers enjoy an enhanced video quality.

3 Challenges and Open Issues

Even though there has been many research advances in the area of P2P streaming, there exists several techincal challenges and open issues that still need to be resolved. One main issue is that the quality of experience in current P2P streaming systems is still not comparable to traditional TV services. Users of P2P streaming systems still experience long startup delays as well as video/channel switching delays. In addition, there's always the possibility of playback time lag between peers. Moreover, current P2P streaming designs are not ISP-friendly. Unregulated P2P video exchange significantly increases traffic volumes on links within and between ISPs. P2P streaming systems also need to support heterogeneous receivers that differ in their processing capabilities and uplink capacities. This requires more intelligent media coding techniques as well as complex scheduling algorithms. Designing incentive mechanisms for peers is also a challenging issue that is more complicated in P2P streaming systems than in P2P file sharing systems due to real-time

requirements. Finally, P2P streaming systems are more vulnerable to quality of service (QoS) fluctuations, which makes them prone to various security threats.

4 Security and Threats

From a security perspective, P2P streaming systems are more challenging than other P2P applications. Streaming applications are in general more sensitive to delay and delay jitter. Thus, these applications are more vulnerable to QoS fluctuations. Malicious users can take advantage of these characteristics by advertising false information to its neighbours (e.g. round-trip time (RTT) cheating and advertising fake data availability)[16], exhausting network bandwidth (e.g. connecting to too many downstream/upstream peers)[4], corrupting/modifying media data[9][17][7], or disguising as authoritative figures. We will examine solutions to some of these problems and briefly discuss their advantages and disadvantages.

As an aside, P2P streaming networks are more prone to successful external distributed denial of service (DDoS) attacks since the number of data sources in the network is relatively small. However, this aspect does not exclusively pertain to streaming systems only [19] and thus will not be covered in this paper.

4.1 Credit Models: Ripple-Stream

As mentioned previously, malicious users can perform a variety of attacks on a network. In order to deter the majority of these attacks in general, Ripple-Stream [16] introduces the concept of trust; its framework is relatively simple – the main idea is that we wish to only allow trusted nodes to be near the data source, while pushing possibly malicious nodes to the outskirts of the network where they can do the least amount of harm. Ripple-Stream accomplishes this by explicitly defining how each node accumulates or loses credit, and also how the credit rating of a node should affect its position and connectivity in the network.

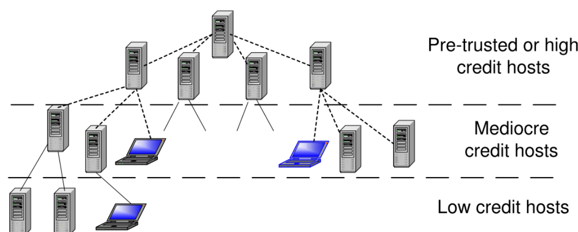

Figure 5: Ripple-Stream in Tree-Based Overlays [16]

Credit Management

Initially, all nodes in the network are assigned some base credit score. Depending on the behaviour of a node on the network, the management system then increments or decrements the node's credit rating. Nodes can accumulate credit by performing their streaming duties without issue; similarly, nodes that report suspicious behaviour from other peers are given positive credit – that is, it is in everybody's best interest to report malicious activities. Nodes that are discovered to be tampering data or reporting false information are given a heavy credit penalisation and are disconnected from their peers. Likewise, two connected peers with low data quality transfer between them are disconnected from each other and both peers' credit score is lowered; this dissuades malicious nodes from purposely disconnecting other peers, since it would negatively affect his own credit. The specific credit allotment amounts are based on a modification of the PeerTrust [18] credit distribution metric, which considers factors such as a peer's history, the severity of the malicious behaviour performed (if applicable), etc.

Overlay Management

When a new node enters the network, it is assigned a mediocre credit rating and given a list of all peers alongside their credit ratings. When a node wishes to connect to a new peer, it is given strict guidelines as

to who it can connect to; two nodes are only allowed to be neighbours if the difference of their credit score is below some predefined threshold. To assist nodes in finding other peers with similar credit score, the Ripple-Stream overlay provides a random node discovery mechanism. Periodically, to account for possible fluctuations in credit score, every node is required to check to make sure all of its connected peers are still within the predefined credit threshold; if not, those peers are disconnected. This, in turn, is why this system is named *Ripple-Stream*; malicious nodes are rippled to the outskirts of the P2P network, where they are not supposed to be able to affect the network in any perceivable way.

Discussion

Although Ripple-Stream has been shown to be effective by its authors [16], their experimental results rely on their malicious users performing simple and easily detectable attacks. For example, to verify that RTT cheating has occured, it is trivial for a node to just get the actual delay times from a suspected malicious peer, and report if there's a discrepancy between advertised and actual results. However, attacks that involve more than one peer simultaneously, such as bandwidth exhaustion, are extremely difficult to detect with this model as it requires global knowledge of the behaviour of nodes in the network.

Furthermore, this framework relies heavily on the notion that reporting nodes are truthful; this is a rather strong and dangerous assumption to make in a security model. A malicious user could easily report an honest peer as performing malicious deeds to either boost its own credit score or deflate the target user's credit score.

4.2 Bandwidth Throttling: Kantoku framework

To account for some of the issues discussed previously with Ripple-Stream, we introduce the Kantoku framework by Conner and Nahrstedt[4]. In a basic P2P network, each node has some predefined maximum download rate – either set by the P2P application or by some physical limitations of the node's connection – and the node is then able to monitor its own download rate. However, a malicious or selfish node could still request to download more than it is allowed to from its neighbours, and since the overall bandwidth in a P2P network in some scenarios is minimal (e.g. an initial flash crowd), this could have dire consequences on the overall availability of the stream to genuine users. Furthermore, a malicious node could request downstreams well below his predefined maximum from a large number of neighbours, with the aggregate far exceeding the maximum, but each of its singular neighbours would be unable to determine that a violation has occurred.

A naive solution to these issues would be to utilize a centralised server which tracks each node's aggregate download rate. However, aside from being a single point of failure, this is also computationally slow[5]. The Kantoku framework, on the other hand, addresses this in a decentralised manner by selecting trusted nodes, defined as *Kantoku nodes*, which monitor the aggregate download rate of every node in the network and throttles violators accordingly.

Creation of the Kantoku overlay

The framework assigns trusted nodes as Kantoku nodes based on predefined metrics from other systems in the literature. The authors also entertain the possibility of requiring trusted nodes to be held accountable (e.g. with trusted hardware, or with contracts)[5], thus greatly diminishing the likelihood that they would do something malicious. Kantoku nodes, like 'untrusted' nodes, have a common goal of retrieving data from the stream. Once these Kantoku nodes are defined, a DHT-based overlay network of these nodes is created on top of the underlying P2P network to aid the distributed storing and lookup of each node's download and upload rate. Untrusted nodes are assigned to the closest Kantoku node based on the lookup protocol used – for example, via closest node identifiers in Figure 6.

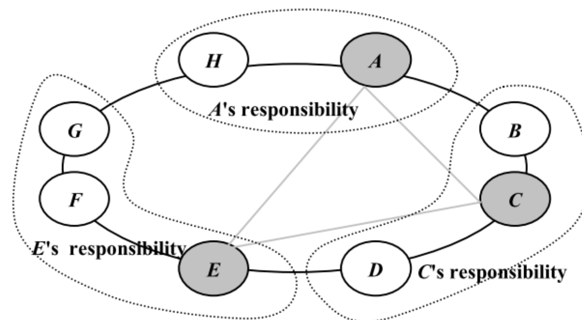

Figure 6: Kantoku Overlay[4]

Monitoring and throttling bandwidth using Kantoku nodes

When a node requests to download from a peer, instead of sending a message to the peer directly, it sends a control message to its associated Kantoku node. The Kantoku node then verifies whether this request is valid, and if so, digitally signs another control message and sends it to the requested peer to initiate uploading. All non-Kantoku signed requests to upload are ignored by all nodes.

The importance of the DHT-based overlay defined earlier is noted here: each node's current aggregate download rate and maximum download rate is stored in the overlay; a Kantoku node requires this information whenever it needs to verify the validity of a download request (e.g. that by allowing a request, the current aggregate download rate doesn't exceed the maximum download rate). As well, the overlay also keeps track of the peers a node is downloading from, in order to throttle a node later on.

However, note a possible scenario where a node requests to download from a peer but would exceed his maximum download rate by just 1 Kbps – it is not ideal to disallow this node from connecting entirely. In order to do something smarter than just disallowing a node from downloading from its peers if they would exceed their maximum download rate otherwise, the framework introduces the notion of throttling; that is, for each peer the node is downloading from, there's a probability that a packet will be dropped. When a node sends a download request control message to its Kantoku node, the Kantoku node calculates this probability – which increases the more the node exceeds its maximum download rate – and sends it to all of the node's current and potential uploaders. Whenever an uploader decides to send a packet to the downloading node, it drops the packet based on this probability.

Discussion

The selection of trusted nodes to assign as Kantoku nodes is very unclear; since Kantoku nodes are still normal nodes who also wish to receive streaming data, yet have to do significantly more work than 'untrusted' nodes, this may affect the quality of their streams. It is unlikely that normal users would volunteer themselves as Kantoku nodes because of this – and if the framework chooses nodes involuntarily, then it's in their best interests to rejoin the network in the hopes to regain 'untrusted' status. Perhaps a predefined incentive in becoming a Kantoku node (e.g. higher peer connectivity) would be ideal.

In addition, the authors have shown that this framework is very effective in throttling bandwidth in real-world scenarios; however, exceeding download bandwidth is only one of the aspects a user can perform to exhaust available network bandwidth. A malicious user could just as easily deny all upload requests even if there is a minimum upload bandwidth that all nodes are meant to upkeep. Furthermore, a user could collude with several other malicious users and request small streams from thousands of peers, effectively exhausting all available download slots for honest users. However, the framework is a step in the right direction, and should be adaptable to these scenarios with some additional work.

4.3 Data Integrity and Authentication: Hash Chaining

As noted earlier, a malicious user can modify the packets he receives (possibly with garbage) and resend them to other peers in an attempt to pollute the network: this is called a pollution attack. Similarly, malicious

users can disguise as the data source and send packets to its peers, attacking the authenticity of the data. The Ripple-Stream framework briefly touched upon these issues by moving malicious users to the outskirts of the network every time they would pollute, but due to the nature of P2P networks, a polluted packet is still able to propagate throughout the network very rapidly so the harm would be already done. Furthermore, requiring users to authenticate every packet (to report non authentic streams to the framework) is infeasible due to computational costs. As shown in an experiment by Dhungel et al.[7], a single malicious user was able to significantly degrade the streaming quality of the majority of users in a popular unprotected P2P streaming system, PPLive – that is, this single user was able to cause approximately 78% of the network's roughly 3250 peers to quit in frustration in less than an hour of active polluting.

The obvious solution to prevent these types of attacks is for the streaming source to sign every packet and have each node verify this signature whenever it receives a packet. However, the act of digitally signing a packet is an order of magnitude more computationally expensive than something simpler such as computing a one-way hash of a packet [9]; additionally, users may be streaming from mobile devices with low computational power, so it is imperative to utilize a solution that is both efficient in terms of bandwidth utilization (overhead created by these integrity checks) and computational requirements. We will now present a summarisation of some possible solutions, starting with simple techniques with low or nonexistent loss tolerance before moving onto more involved techniques with complete loss tolerance.

Simple Hash Chaining: Zero Loss Tolerance

Hash chaining, first described in 1997[8], was introduced as a way to sign digital streams in general, before the idea of P2P live streaming became popular. The technique itself is simple: as described by the authors themselves and summarised in [11], the source partitions chunks into blocks, and in each block every chunk includes a one-way hash of the next chunk in the block. The first chunk in the block is digitally signed to authenticate the whole block – otherwise, a malicious user could just discard the whole block and generate his own stream and send it to a receiver. When a receiver receives the first chunk in the block, he only needs to verify this signed chunk to authenticate the remaining chunks in the whole block. Furthermore, he can now use the stored hash in the first chunk to verify the integrity of the second chunk when it arrives, and then use the hash stored in the second chunk to verify the integrity of the next chunk, and so on. An illustrated example is shown in Figure 7.

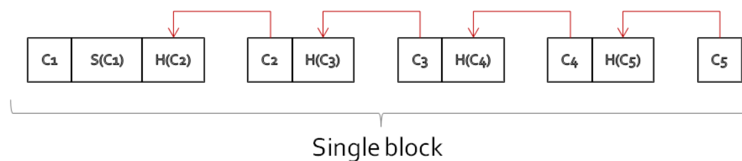

Figure 7: Simple Hash Chaining with 5 chunks

Augmented Hash Chaining: Burst Loss Tolerant

In hash chaining, in order to verify the authenticity of any particular chunk, all of its previous packets must have been verified successfully as well. Consider the scenario where some chunk is lost in transmission; this leads to issues since, as described in [17] and [11], a video streaming network's sensitivity to time delays will make waiting for the retransmission of lost chunks infeasible; either losses should be tolerated or error correction techniques should be used instead [17]. A relatively simple improvement, augmented hash chaining [9] is a technique that aims to improve the robustness of a stream to bursty losses. Similar to the simple hash chaining method, augmented hash chaining divides chunks into blocks. The general idea is that the hash of a chunk is stored in the immediate preceeding chunk and another chunk some user-defined number x chunks away. Thus, in the case where a succeeding chunk is lost in transmission, it is still possible to verify the integrity of the current chunk by using some future chunk possibly outside of the duration of the transmission loss burst (i.e. choosing a good value for x to improve this possibility). The technique itself also provides some optimizations to improve loss robustness by augmenting additional chunks between chains if the source allows buffering [9]. An illustrated example is shown in Figure 8.

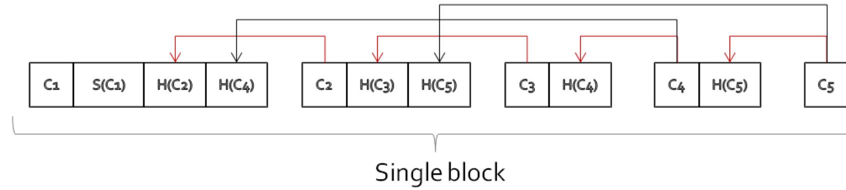

Figure 8: Augmented Hash Chaining with 5 chunks

Star Chaining: Completely Loss Tolerant

We now introduce verification techniques that are impervious to transmission loss; that is, a chunk is able to verify its own authenticity without requiring any prior or future chunks. The first technique, star chaining [17], has the source, as with previous techniques, partitions chunks into blocks. For every chunk in the block, a hash is created, and a hash of the ordered concatenation of all these singular chunk hashes is generated – coined the *block digest*. The source then signs this block digest. Furthermore, for each chunk in the block, the source adds a chunk signature containing the hash of every other chunk in the block, the current chunk's position in the block, as well as the signed block digest. An example is shown in Figure 9.

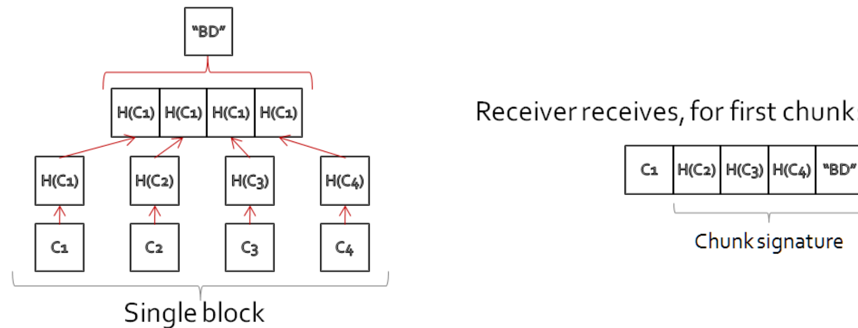

Figure 9: Star Chaining with 4 chunks

When a receiver receives a chunk with the chunk signature, he first verifies the authenticity of the block digest. Once that is verified, he hashes the information in the chunk less the chunk signature, and concatenates it with the hashes of all the other chunks (in a linear, ordered manner similar to the source's creation step). It then checks that this concatenated hash matches the verified block signature. Note that the receiver can utilize caching to save computational power: once the receiver has successfully verified one chunk in a block, it does not need to concatenate and hash again for future chunks in the block – it is sufficient to just check that the hash in the future chunk matches the hash of that chunk stored in the signature of the previously verified chunk. Examples of both of these scenarios are shown in Figure 10 and Figure 11.

Figure 10: Receiver verifying first chunk via star chaining

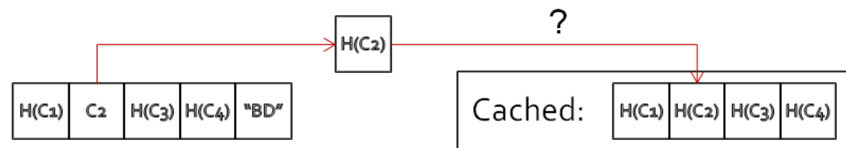

Figure 11: Receiver verifying second chunk via star chaining with caching

While this method is completely loss tolerant, it requires significant bandwidth overhead, since each chunk's chunk signature is not small. We now introduce another completely loss tolerant technique with much lower bandwidth overhead at an increased computational cost.

Tree Chaining: Completely Loss Tolerant

As a variation of star chaining, tree chaining requires more computation (i.e. hashing) for both the sender and receiver, but at a significantly reduced bandwidth overhead by reducing the number of chunk hashes each chunk has to store in its chunk signature. The sender, as in star chaining, partitions chunks into blocks. It then creates a tree in a bottom up fashion: the hash of every chunk in the block are leaves in the tree and is paired up with another chunk in sequential ordering, and each singular chunk is hashed. The parent of any two nodes in the tree is the hash of the concatenation of the nodes. The root of the whole tree is then the block digest, and the sender signs it as in star chaining. An example in Figure 12 is given to better illustrate the overall structure of the tree.

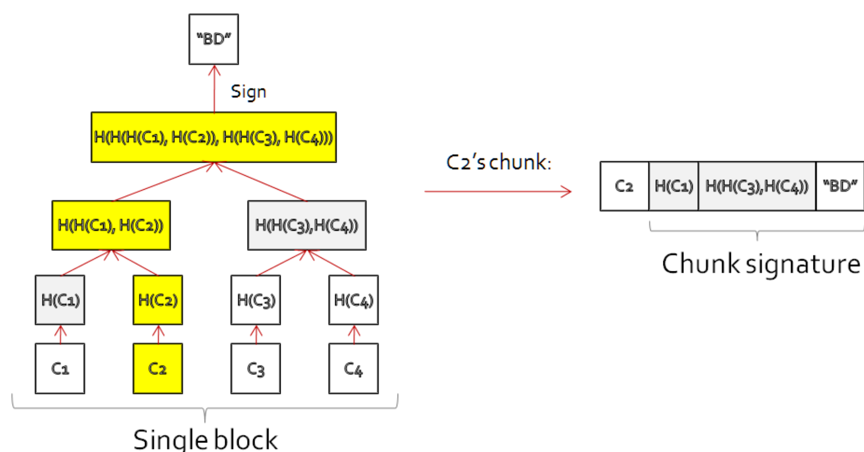

Figure 12: Tree Chaining with 4 chunks

As before, the chunk signature for each chunk in a block contains the current chunk's position in the block, as well as the signed block digest. However, instead of including all other chunks' hashes, it only includes the hashes of siblings of nodes in the path from the chunk to the root of the tree. For example, such nodes for chunk C2 are highlighted in grey in Figure 12. Thus, when the receiver receives a chunk, it verifies its integrity by hashing the concatenation the hash of itself with its sibling node (defined in the chunk signature). It would then concatenate this value with the next hash of the sibling node, and so forth. This corresponds to moving up the singular path in the overall tree from the chunk (leaf). If the chunks have not been tampered with, then this overall hash value should match the signed block digest provided in the signature.

Discussion

As described by Hefeeda et. al.[11] and Dhunge et. al.[7], the effectiveness of these solutions are dependent on the situation they're being used in. Hash chaining, by itself, is only useful in ideal networking situations where there will be zero projected dropped or polluted packets – infeasible in the real world. In networks with expected bursty losses, an augmented chain approach is ideal among these four techniques in terms of minimum overhead [7]. However, in networks where the packet loss is not predictable and the timely delivery of data is of uptmost importance (e.g. P2P streaming), techniques such as star chaining and tree chaining are preferred: star chaining would be preferable in situations where computational power is strained, and tree chaining in situations where bandwidth is limited. Research is still quite active in this area, and these techniques only skim the surface of the solutions available to prevent pollution and authentication attacks;

quantitative analyses [11] show that there are much more efficient yet complex approaches [14], built upon techniques covered in this paper, which adapt well to most real world situations.

5 Conclusions

We presented a general overview on the different methods to both facilitate the creation of peer-to-peer streaming systems and prevent possible attacks from malicious users on these systems. Specifically, we focused on tree and mesh-based systems for both live and on-demand streaming architectures, and discussed the advantages and pitfalls of each. We then introduced some possible attacks that could be performed on these systems, focusing on solutions that attempt to prevent a subset of simple attacks (Ripple-Stream), more complicated aggregate attacks (Kantoku framework), and attacks on the authenticity and integrity of data (hash-chaining techniques). The attacks and solutions covered in this paper only skim the surface of the literature available on the security of peer-to-peer streaming networks; our intent was not to provide an exhaustive survey, but to present a general introduction of the research being performed in this area.

Bibliography

[1] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: high-bandwidth multicast in cooperative environments. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 298–313, 2003.

[2] Y.-h. Chu, S. G. Rao, and H. Zhang. A case for end system multicast (keynote address). In *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'00)*, pages 1–12, 2000.

[3] B. Cohen. The BitTorrent protocol specification. `http://www.bittorrent.org/beps/bep_0003.html`.

[4] W. Conner and K. Nahrstedt. Securing peer-to-peer media streaming systems from selfish and malicious behavior. In *Proceedings of the 4th Middleware Doctoral Symposium (MDS'07)*, pages 13:1–13:6, 2007.

[5] W. Conner, K. Nahrstedt, and I. Gupta. Preventing DoS attacks in peer-to-peer media streaming systems. In *In MMCN*, 2006.

[6] H. Deshpande, M. Bawa, and H. Garcia-Molina. Streaming live media over peers. Technical Report 2002-21, Stanford InfoLab, 2002.

[7] P. Dhungel, X. Hei, K. W. Ross, and N. Saxena. The pollution attack in P2P live video streaming: measurement results and defenses. In *Proceedings of the 2007 Workshop on Peer-to-Peer Streaming and IP-TV (P2P-TV'07)*, pages 323–328, 2007.

[8] R. Gennaro and P. Rohatgi. How to sign digital streams. In *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*, pages 180–197, London, UK, 1997. Springer-Verlag.

[9] P. Golle and N. Modadugu. Authenticating streamed data in the presence of random packet loss (extended abstract). In *ISOC Network and Distributed System Security Symposium*, pages 13–22, 2001.

[10] Y. Guo, K. Suh, J. Kurose, and D. Towsley. P2Cast: peer-to-peer patching scheme for VoD service. In *Proceedings of the 12th International Conference on World Wide Web (WWW'03)*, pages 301–309, 2003.

[11] M. Hefeeda and K. Mokhtarian. Authentication schemes for multimedia streams: Quantitative analysis and comparison. *ACM Transactions on Multimedia Computing, Communications, and Applications*, 6:6:1–6:24, February 2010.

[12] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole, Jr. Overcast: reliable multicasting with on overlay network. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation (OSDI'00) - Volume 4*, pages 14–14, 2000.

[13] J. Lv, X. Cheng, Q. Jiang, J. Ye, T. Zhang, I. Lin, and L. Wang. LiveBT: Providing video-on-demand streaming service over BitTorrent systems. In *Proceedings of the 8th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'07)*, pages 501–508, 2007.

[14] A. Pannetrat and R. Molva. Efficient multicast packet authentication. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'03)*, 2003.

[15] F. Wang, Y. Xiong, and J. Liu. mTreebone: A collaborative tree-mesh overlay network for multicast video streaming. *IEEE Transactions on Parallel and Distributed Systems*, 21:379–392, 2010.

[16] W. Wang, Y. Xiong, Q. Zhang, and S. Jamin. Ripple-Stream: Safeguarding P2P streaming against DoS attacks. *IEEE International Conference on Multimedia and Expo*, pages 1417–1420, 2006.

[17] C. K. Wong and S. S. Lam. Digital signatures for flows and multicasts. *IEEE/ACM Transactions on Networking*, 7:502–513, August 1999.

[18] L. Xiong and L. Liu. PeerTrust: supporting reputation-based trust for peer-to-peer electronic communities. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):843 – 857, 2004.

[19] J. Yang, Y. Li, B. Huang, and J. Ming. Preventing DDoS attacks based on credit model for P2P streaming system. In *Proceedings of the 5th International Conference on Autonomic and Trusted Computing (ATC'08)*, pages 13–20, 2008.