Distilling the Ingredients of P2P Live Streaming Systems

Roy Friedman[*], **Alexander Libov**[*]
[*]Computer Science Department
Technion - Israel Institute of Technology
Haifa 32000, Israel
{roy,alibov}@cs.technion.ac.il

Ymir Vigfusson[†‡]
[†]Dept. of Math&CS
Emory University
Atlanta GA 30322, USA
[‡]School of Computer Science
Reykjavik University
Reykjavik, Iceland
ymir@mathcs.emory.edu

***Abstract*—**

Peer-to-peer live streaming systems involve complex engineering and are difficult to test and to deploy. To cut through the complexity, we advocate such systems be designed by composing *ingredients*: a novel abstraction denoting the smallest interoperable units of code that each express a single design choice. We present a system, StreamAid, that provides tools for designing protocols in terms of ingredients, systematically testing the impact of every design decision in a simulator, and deploying them in a wide-area testbed such as PlanetLab for evaluation. We show how to decompose popular P2P live streaming systems, such as CoolStreaming, BitTorrent Live and others, into ingredients and how StreamAid can help optimize and adapt these protocols. By experimenting with the essential building blocks of which P2P live streaming protocols are comprised, we gain a unique vantage point of their relative quality, their bottlenecks and their potential for future improvement.

I. Introduction

Online video services will be used by nearly two billion Internet users by 2018 according to projections, consuming over 79% of all Internet traffic [1]. While video-on-demand is the major constituent of Internet video, the popularity of live video streams has been growing rapidly, with the number of global IPTV subscribers now exceeding 100 million [2], and live streaming platforms such as *Twitch.tv* attracting more than 45 million users per month [3]. Unfortunately, the technology for Internet live streaming is still far from reaching the maturity of traditional broadcast media, such as cable TV. Massive outages show that online broadcasts from major events such as the FIFA World Cup and the Oscars shake the foundations of current live streaming architectures [4], [5], where scalability is typically afforded through expensive provisioning of large content-distribution networks (CDNs) [6], [7].

Researchers have long advocated that leveraging peer-to-peer (P2P) networks for delivering live streams can significantly ameliorate scalability concerns by reducing burden on the broadcast source [7]–[11]. Involving end-users in the multiplexing of live streams offers alluring bandwidth cost savings for businesses, yet relatively few companies have successfully incorporated the edge in content delivery – CoolStreaming [9], [12], PPLive [13] and Akamai NetSession [14] are prominent examples despite the first two having been discontinued due to legal concerns. Improved scalability has come at the price of increased complexity; live streaming deployments have been fraught with challenges ranging from unpredictable quality-of-service such as playback buffering [15] to concerns about upload rates. Significant research has been conducted with the aim of improving performance and quality of scalable live streaming systems [11], [16], [17].

We have identified two stumbling blocks that impede the development of practical live streaming systems. On one hand, *live streaming systems are complex, involving sophisticated design decisions and significant engineering effort*. To illustrate, an overlay must be formed to address scalability and must be maintained despite failures and peer churn; the source encodes content into chunks which must be forwarded by overlay members according to some strategy so that they transitively reach all interested parties; the chunks must arrive by their deadlines to prevent buffering and provide high quality playback. Furthermore, the peers may be heterogeneous and require different provisioning.

On the other hand, *testing live streaming systems is often ad-hoc and non-repeatable.* The live streaming community has converged on a set of yardsticks against which to measure live streaming systems, such as the continuity index [9], but no standard benchmarks or workloads are used to evaluate these metrics, with custom generated workloads or exclusive traces being commonly used [8]. The source code behind published live streaming systems is often unavailable, further preventing apples-to-apples scientific comparison between different approaches. Microbenchmarks of systems frequently fail to reveal how each design decision impacts the composite systems, how system components influence one another or how system parameters should be calibrated in new environments.

Motivated to address these challenges, our paper focuses on ***providing abstractions for understanding the impact of design decisions in live streaming systems.*** We introduce the concept of *ingredients*: atomic composable building blocks of code with minimal uniform interfaces that each specifies a partial functionality of a protocol at the level of a single design decision. As a concept, casting live streaming protocols as ingredients has several advantages:

- **Simplicity**. Each ingredient is a simple, well-defined programming task.
- **Modularity**. Developers of new protocols are motivated to make all design choices explicit and

to minimize coupling within code as much as possible.
- **Extensibility**. Additional ingredients can be added into the system when they are identified without touching any parts of the code that are not directly involved in the interaction.
- **Optimizability**. Each ingredient can be tuned and subjected to scientific tests, for instance by varying an independent parameter while keeping others constant.

We built a framework based on the ingredient abstraction, STREAMAID, that provides an environment for developing and improving upon practical live streaming systems. Aimed to help overcome the stumbling blocks identified above, STREAMAID leverages ingredients to dislodge the complexity of live streaming protocols and facilitate testing, and provides seamless deployment on PlanetLab for large-scale evaluation.

We surveyed seven prominent P2P live streaming systems [9], [18]–[23], identified the fundamental pieces of functionality and design choices made by each of them, and expressed them as ingredients within our system. The unified implementation of all of these systems in a single framework enables apples-to-apples measurements and testing. Further, STREAMAID lets us replace any basic design choice of a given system and evaluate the impact on the attained performance or other metrics simply by changing the corresponding ingredients, which is done through an XML configuration file. We have performed several such experiments and our paper highlights the insights we gained from them, underscoring the usefulness of the abstraction. We believe the framework can help identify shortcomings in less effective live streaming protocols to enhance their performance, and to characterize precisely what design choices constitute the best P2P streaming protocol for a given environment.

Contributions. First, we present the ingredient abstraction to disentangle the often complex components of live streaming systems into fundamental, atomic building blocks. This granular abstraction gives designers and implementers a rigorous methodology for creating, presenting, optimizing and evaluating individual design decisions of their protocols.

Second, we implement the STREAMAID system which supports the ingredient abstraction in two ways. First, STREAMAID provides a Java API by which developers can program new ingredients for live streaming protocols, and an XML configuration file that allows ingredients to be blended. Further, we provide an extensive framework to rigorously test and calibrate the resulting protocols, to evaluate them through local simulation and to deploy them on real distributed testbeds such as PlanetLab [24]. Unlike prior work, the framework allows experimentation on individual ingredients while keeping all others constant, providing a scientific way for testing and optimizing a live streaming system.

Finally, we show how several popular live streaming protocols can be ported to use the ingredient abstraction, including BitTorrent Live [25], CoolStreaming [9] and others. We found that many existing protocols share ingredients, with several published protocols having identical functionality except for a single ingredient. We report on experiments that evaluate individual ingredients of these protocols, and systems that have been composed of several existing ingredients. Our results show how STREAMAID helps expose and balance the impact of different trade-offs in live streaming protocols.

STREAMAID is free software available online at *https://github.com/alibov/StreamAid*.

Roadmap. The rest of this paper is organized as follows: We describe our concept of ingredients in Section II. We continue with the breakdown of P2P live streaming protocols to their basic design decisions in Section III. We evaluate several design decisions in Section IV. We survey related work in Section V before concluding in Section VI.

II. MODULES AND INGREDIENTS

Distributed systems are commonly engineered as a stack of micro-protocol layers that each serve a well-defined function [26], [27]. In such systems, a message sent by a specific layer L goes through every underlying layer at the sender, for example, as depicted in Fig. 1.

The micro-layer design is ideal when messages are relevant for all underlying layers, since messages will always flow through them. P2P live streaming protocols fit this pattern. The design is also effective when building general-purpose communication middleware for accommodating a large variety of communication patterns and systems. When these conditions fail to hold, however, the micro-layer approach imposes overhead and can make implementations awkward.

A. Ingredients of P2P Live Streaming

Our work is focused on finding thin yet informative micro-layers for P2P live streaming systems, which we call *ingredients*. The idea is for the aggregate functionality of a P2P live streaming system to be divided into both coarse-grained *modules* that are then further subdivided into these micro-layers ingredients. A module can be viewed as a composite set of ingredients along with a so-called *core*, as illustrated in Figure 2 and explained below. Within a single network node, modules may interact with one another through method invocation

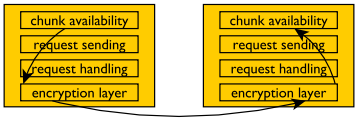


Figure 1: ***Micro-Layers****: a model where messages pass through a stack of thin layers for transmission and reception on the destination.*

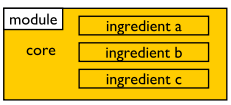
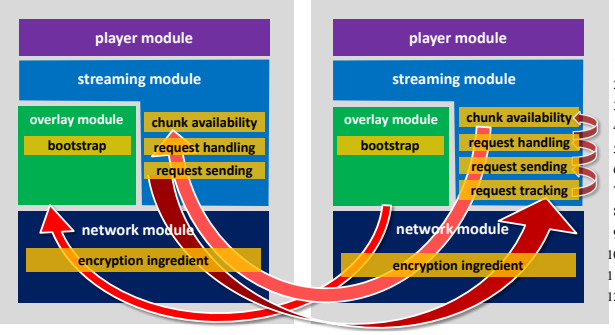


Figure 2: **STREAMAID** ***Module****: A module is composed of a set of ingredients in* STREAMAID.



Listing 1: *CoolStreaming [9] built using ingredients.*

```
<streamingAlgorithm algorithm="PullAlgorithm" size="200">
    <overlayAlgorithm H="200" M="4" c="1" amountToSend="6"
        exploreRound="30" gossipTimeout="6"
            algorithm="CoolStreamingOverlay">
            <ingredient name="NeighborChunkAvailabilityIngredient"
                operationMode="updateEveryRound"/>
    </overlayAlgorithm>
    <ingredient name="SourcePushIngredient"/>
    <ingredient name="EarliestContinuousChunkVSInitIngredient"/>
    <ingredient name="HandleChunkRequestsOnArrivalIngredient"/>
    <ingredient name="CoolstreamingChunkRequestIngredient"/>
</streamingAlgorithm>
```

Figure 3: STREAMAID ***Architecture****: Each independent module contains a queue of ingredients, and the corresponding XML configuration elements to instantiate the ingredients.*

having a fixed interface. Different implementations of modules can also be mixed and matched, similarly to our earlier work in this area [28].

This paper investigates *how modules can be broken down into a single core and multiple ingredients*. The core contains the "essential" uniquely identifying functionality of a given protocol for implementing that module, whereas ingredients include reusable pieces of functionality that could be used by several protocols, even to enhance the protocol. For maximal modularity, the core should be as small as possible, ideally even empty. Yet, sometimes certain pieces of functionality uniquely characterized a protocol and we found no reasonable way to augment them or include in other protocols. In those cases, we kept the functionality as part of the core, as this helped produce more simple and readable code. As discussed below, in modules for some P2P live streaming protocols we managed to obtain an almost empty core and organize almost all functionality into ingredients, while in others we were less successful.

Architecture. STREAMAID comprises four modules: a *player* module, an *overlay* module, a *streaming* module and a *network* module (Figure 3). The *network* module handles all communication aspects, providing a simple-to-use API with a SENDMESSAGE method and a RECEIVEMESSAGE upcall. The *overlay* module uses the network module to construct the actual P2P overlay. There may be multiple initiations of this module running in parallel. The *streaming* module queries an overlay module for neighboring connections and initiates video chunk exchange with the neighbors according to its protocol. Finally, the *player* module handles all buffering, encoding, decoding and playback of the video content. This approach allows the overlay modules for one known system to be mixed and matched with streaming modules corresponding to other systems.

For clarity of presentation, Figure 3 only exhibits a partial view of all possible ingredients. Each of the ingredients encapsulates *a specific design decision* in implementing the module's functionality. We explore the details in Section III below. Here, we focus on discussing conceptual aspects of ingredients and their communication and interaction model.

Communication model. Messages generated by a certain module on a given peer can only be received by the same module on the other peer. The only exception is network module, which is involved in passing every message sent between peers. Such a message can either be addressed to a specific ingredient on the receiving peer, or to all ingredients of the corresponding module on that peer as well as the core of that module. Sending a message to all ingredients is essential to allow easy addition and switching of ingredients, since the sending ingredient does not need to know in advance the exact ingredient that would handle the message. Further, a message can only be addressed to the ingredients of the same module in which they were generated. When such a message is addressed to all ingredients of a given module, the message is passed to these ingredients one after the other in the order these ingredients appear in the configuration file. All sent and received messages pass through the network module, which only contains ingredients relevant to all messages in the system.

At any event, passing messages across modules is not allowed. The only way to pass information between modules, and only between modules of the same peer, is through the defined interface for such invocations.

The actual set of ingredients to be invoked is specified in a configuration file that is parsed at runtime. The appropriate ingredients listed in the configuration file are instantiated when the framework is started.

Example. Consider the example ingredients shown in Figure 3. Here, the overlay module sends and receives messages from the corresponding overlay module on the remote peer. The "chunk availability" ingredient sends and receives messages from its remote counterpart "chunk availability" ingredient. The "request sending" ingredient sends a message to the corresponding streaming module, so that all associated ingredients of that module can receive the chunk request message. All messages in Figure 3 pass through the encryption ingredient embedded inside the network module.

The STREAMAID communication model maintains a clear separation between modules, thus ensuring that modules do not interfere with the tasks of one another. In contrast, ingredients contained in the same module require a more flexible communication model as an ingredient may not know at development time which ingredients will be instantiated alongside it during runtime to handle messaging. Further, different nodes may have different ingredients instantiated.

B. Discussion

Our approach has several benefits: independent modules allow for encapsulation and a loosely coupled protocol structure, while the intra-module ingredients allow messages to be handled according to a chain-of-responsibility and enable seamless addition of fine-grained functionality. Further, ingredients in the network module mimic the micro-layer architecture, allowing us to define system-wide ingredients that affect all sent and received messages. For example, the operation of a pull-based streaming module can be partitioned into several ingredients, as elaborated further below. One such ingredient can handle chunk requests. Different nodes may implement different ingredients for handling chunk requests. Thus, the chunk request message would be sent to the streaming module to be disseminated to all ingredients. Further, an optional reputation ingredient could be added to the streaming module to track some statistics, all without modifying any other part of the protocol. This example is illustrated in Figure 3.

Should the protocol designer decide to experiment with a different implementation for a specific design decision, they would only need to implement that specific ingredient and replace the previous implementation with the new one. An ingredient can easily be changed via a configuration file setting, making testing different design decisions a simple and fast process. For example, the CoolStreaming [9] algorithm was decomposed in MOLStream to a streaming module and an overlay module. Here, we have further decomposed these modules of CoolStreaming to separate ingredients. The results are shown in Listing 1, which depicts the portion of the STREAMAID configuration file that defines the P2P live streaming algorithm to be used.

C. Ingredients in STREAMAID

Under the STREAMAID model, the support for ingredients enables fine grained control of the protocols produced. An ingredient in STREAMAID is a class that implements two basic methods:

- The NEXTCYCLE method is called periodically and can be used to do routine checks and to invoke actions such as sending messages.
- The HANDLEMESSAGE method handles messages received by this ingredient, or the underlying module of the ingredient. When sending messages, an ingredient may choose to send the message to the ingredient layer on another node or to the entire corresponding module including all the ingredients associated with the module.

This simple interface proved enough to accommodate all the functionality needed to express a plethora of protocols; in particular every ingredient mentioned in Section III as well as ingredients we implemented but do not discuss due to a lack of space.

III. EXPRESSING DESIGN DECISIONS

We surveyed a large number of P2P live streaming systems and identified how their behavior could be encapsulated with the ingredient abstraction. By examining the range of systems in detail, we were able to identify several common design decisions involved in building these systems. We discuss several ingredients corresponding to these choices, and omit a number of others due to space constraints.

Recall that we view P2P live streaming as comprising four major modules: network interface, overlay maintenance, stream dissemination, and player issues.

A. Player Module

The player module is a consumer that displays the chunks received by the streaming module and as such does not send out any messages. However, as shown in Section IV, there are several important decisions to be made which affect the overall performance of the streaming protocol.

a) ***Player Initialization Time Ingredient****:* The streaming module chooses when to initialize the player module. Upon initialization, the player module waits (buffers incoming chunks) for a predefined amount of time (a parameter) and then begins playback. This buffer time is an important design decision by itself, but the point in time when the player module is initialized also impacts overall performance. One option is to initialize the player module on startup; another is to wait for a first bitmap or chunk to be received.

b) ***Player Initialization Position Ingredient****:* When the player module is initialized, the streaming module also sets the chunk from which the playback starts. If the player is initialized when a chunk is received, the playback can start from that chunk. However, if the player module is initialized when a bitmap is received, there are several possibilities for the playback starting point. This is not a trivial problem [12] since playback should start sometime between the first available chunk and the last one.

c) ***Skip Chunks Ingredient****:* After the playback has started, the player module can reach a state where the next chunk to be played is missing. The decision of what to do when it happens is usually treated as a binary one - either skip the missing chunk or wait for it. In fact, BitTorrent Live proposed to combine the two by waiting for some time and then skipping. However, throughout the run of a P2P streaming protocol, the decision whether to wait or skip a missing chunk is circumstantial. The algorithm ought to wait while the buffer window is empty, but if only the next or a few chunks are missing they may be skipped.

d) ***Adaptive Playout Ingredient****:* Several articles mention the possibility to use *Adaptive Playout*: increase the playback continuity by slightly changing the playback speed while fixing the sound pitch so that the change would be unnoticed by users [29]. Streaming algorithms can slow down the playback to increase the buffer window, increasing resilience and continuity at the cost of increased latency. If the system is stable enough and the buffer window is too large, a faster playback can be used to decrease latency and window

size. Adaptive Playout can also be used to decrease startup delay: a streaming protocol can start with a small buffer window, minimizing the startup buffer time, and increase the window by slower playback to get to the desired window size.

B. Streaming Module

In most streaming algorithms, chunks are only exchanged with neighbors, which by definition relies on an underlying overlay module. Pull-based streaming modules can normally work with any overlay, whereas push-based streaming modules usually work with tree-based overlays.

1) Push-based Streaming: Push-based streaming is simple: whenever a chunk is received, forward it to all child nodes. The overlay module is responsible for distinguishing child nodes. Multi-tree streaming operates in a similar fashion, except that the overlay maintains t spanning trees, each of which spans all peers, where t is a parameter.

The underlying overlay masks neighbor failures, so the only remaining concern for a streaming protocol is to recover the chunks that were missed while the parent node was being replaced. One option is to recover nothing, which is legitimate if another protocol running in parallel will handle chunk recovery, or if missing chunks are skipped during playback. Another option is to request the new parent node to continue sending chunks from where the old parent left off while also catching up if bandwidth allows.

BitTorrent Live [23], [25] takes a Multi-Tree approach, where each tree is called a *club*, allowing multiple parents in the same club. The choice increases the continuity of the stream at the cost of upload bandwidth consumed by redundant chunks being sent by multiple parents in the same club. Our decomposition of the BitTorrent Live protocol is illustrated in Fig. 5.

2) Pull-based Streaming: Pull-based streaming requires more communication and decision points. All pull-based streaming algorithms that we surveyed, however, made the same types of decisions, each of which we were able to successfully encapsulate as an ingredient. Accordingly, the core of the pull-based streaming module was virtually empty.

a) ***Chunk Availability Ingredient****:* When pull based streaming is involved, neighboring peers exchange *bitmaps* containing information on which chunks each peer has. The bitmap exchange can be done periodically (as in CoolStreaming [9] and PULSE [21]) or a bitmap can be sent for every new chunk received (as in Chainsaw [30]). As chunk availability exchange only occurs between neighboring peers, this ingredient is always tied to a specific overlay.

b) ***Request Sending Ingredient****:* Another decision in the pull-based streaming module is a choice of what chunks to send. That is, using the neighbor chunk availability and potentially additional parameters (such as latency, chunk time to deadline, requests already sent to neighbor, history with neighbor, *etc.*), the algorithm decides which chunks to request and from which neighbors. CoolStreaming sends requests based on rarest first but only to peers that can send the chunk before the deadline. Chainsaw sends out requests randomly, but limits the amount of requests sent to each neighbor, whereas PULSE prioritizes peers using the exchange history and buffer window overlap as parameters.

c) ***Request Handling Ingredient****:* While CoolStreaming and Chainsaw handle requests right on arrival (except for some special handling by the source in Chainsaw), PULSE prioritizes request handling using similar parameters as the request sending ingredient.

d) ***Source Push Ingredient****:* Finally, as in the overlay module, the stream's source node may employ an algorithm different from other nodes. For instance, in an effort to reduce latencies, the source node can push the newly generated chunks to its neighbors regardless of the protocol run by other nodes. While other ingredients shown in this section are mandatory and are required for the operation of the module, Source Push Ingredient is optional and can be turned off at will.

3) Push-Pull Hybrid Streaming: Push and pull algorithms can be combined by running them in parallel. Here, chunks closer to the playback deadline are requested by the pull algorithm while chunks further ahead should be received by the push algorithm. The pull and push algorithms may use the same overlay or different overlays. For example, in mTreeBone [22], the pull algorithm is the CoolStreaming streaming module using the CoolStreaming overlay, while the push algorithm uses the simple push-based streaming module described in Section III-B1, leveraging a tree overlay described in their paper.

C. Overlay Module

A fundamental building block of a P2P live streaming system is the underlying overlay. The overlay maintains and constantly updates two lists of nodes: the set of *known* nodes and the set of *neighbor* nodes. Known nodes are populated either by querying an agreed upon *tracker* node [18], [19], through *gossiping* [9], [20], or using neighbors of another underlying overlay [9], [21]. Nodes may be chosen from the known list to be included in the neighbors list. Several factors influence the choice of electing node p to be a neighbor:

- Local state: the local state of the node may include the current number of neighbors the node has, the average latency of the node, the chunks the node has, the total uptime of the node, and possibly the total upload bandwidth of the node.
- Exchange history: the chunks sent to p and received from p in total or in a recent time window.
- Neighbor state: the same properties as in the local state but of the potential neighbor p. Any such property that is used in the decision must be specifically sent by p and kept up to date. Updates can be either scheduled or as soon as the change in the state occurs.

Table I: *Classification of overlay algorithms.*

Overlay	Type	Known List	Neighbor List Dependencies	Request reply
SCAMP [18]	Non-symmetric	Tracker, forward messages	num. of neighbors	N/A
Coolstreaming [9]	Symmetric	SCAMP, gossip	num. of neighbors, exchange history	num. of neighbors
Araneola [20]	Symmetric	Tracker, gossip, forward messages	num. of neighbors	num. of neighbors, num. of neighbors of requesting node
Pulse [21]	Non-symmetric	SCAMP	num. of neighbors, recent exchange history, average latency, average latency of potential neighbor	N/A
Prime [19]	Symmetric multi-tree based	Tracker	num. of parent neighbors, download bandwidth, stream bitrate	num. of child neighbors, upload bandwidth, stream bitrate
BitTorrent Live [23]	Symmetric multi-tree based	Tracker, gossip	num. of parent neighbors, num. of children, num. of children of potential neighbor	Always positive
mTreeBone [22]	Symmetric tree based	Tracker, tree ancestors	Existence of parent neighbor, upload bandwidth of potential neighbor, distance from source of potential neighbor, stream bitrate	num. of child neighbors, upload bandwidth, stream bitrate

- Link to potential neighbor: latency or throughput between the node and p.

If the protocol dictates symmetry between neighbors (*i.e.*, if p is a neighbor of q then q must be a neighbor of p), then neighbor requests must be approved by the potential neighbor. The choice of accepting a neighbor can now involve the aforementioned factors.

Sometimes properties of the node requesting the connection differ from those of the node approving the request, for instance in tree-based overlays. Here, the requesting node is usually referred to as a *child* and the node receiving and approving the request is referred to as a *parent*. Hence, the list of neighbors is divided into two separate lists: a list of child nodes and a list of parent nodes with possible different treatment for different states of these lists.

The source node of a given live stream can employ a different algorithm than the other nodes. For instance, to battle free-riding and spread chunks more evenly, the source node may periodically switch neighbors to a random subset of the known nodes [21].

Table I classifies several popular overlay algorithms. The column "Known List" summarizes the sources from which a node learns about other nodes in the system and stores them in the known list. Next, "Neighbor List Dependencies" shows the parameters that affect the decision to include a node from the known list in the neighbor list. Finally, symmetric overlays have parameters for accepting neighboring requests. We state them in column "Request Reply".

As overlay construction algorithms differ significantly, most of the logic resides in the core of the overlay module. Nevertheless, we were able to extract some general ingredients applicable to many overlay construction algorithms. These are deferred to a full version of the paper.

IV. Evaluation

We evaluate how the StreamAid framework works in practice and the extent to which ingredients provide modularity, extensibility and optimizability by experimenting with protocols expressed within the framework. We use the CoolStreaming and BitTorrent Live protocols as case studies.

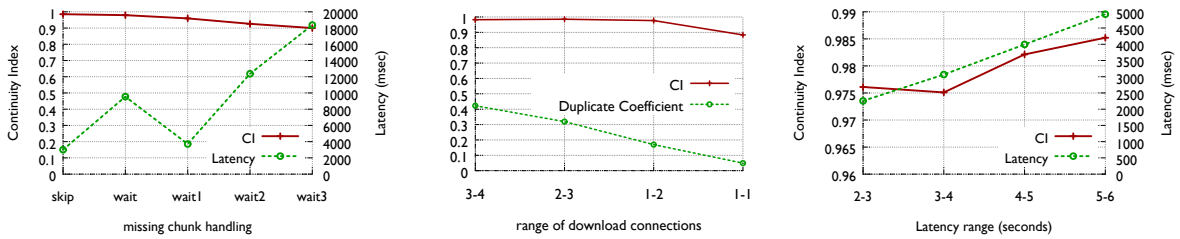
A. Setup and Measurements

The metrics we adopt concern the viewing experience of end-users: **(i)** the average time it takes from login to the playback of the first chunk (*Startup Delay*); **(ii)** the average time from chunk generation to chunk playback (*Latency*), **(iii)** the average fraction of chunks played of those that are playable (*Continuity Index*) (CI) and **(iv)** the percentage of users for which the Continuity Index is perfect (*Perfect Continuity Index%*, PCI%). We measure the CI and PCI% only for users who actually played chunks. We also define the *Zero Playback%* metric as the percentage of users who played no chunks at all. We ran tests in StreamAid using both the PeerSim simulator [31] and deployment on Planetlab.

In PeerSim, message latencies are distributed uniformly between 200 and 400 ms. All tests on PeerSim use LogNormal($\mu = 4.29, \sigma^2 = 1.28$) distribution of session lengths [32] and failed peers are immediately replenished. Each peer has upload bandwidth limit of 5.6 Mbps and the source's upload limit is 16.8 Mbps. Each test simulates 300 seconds of a 300 Kbps stream on 300 peers, while stream generation starts at second 10. Each shown result is an average of 10 runs with different initial random seeds. We test the popular CoolStreaming streaming algorithm with two overlays: the original overlay used by CoolStreaming (with $M = 4$) and the Araneola Overlay (with $L = 3$ or 4). The L and M parameters of the overlays were chosen so that the overlays would have a similar node degree.

On PlanetLab, we tested on 200 nodes and ran each test 5 times. We did not induce extra churn. However, PlanetLab nodes exhibit highly heterogeneous latencies and responsiveness, producing behavior reminiscent of actual churn. On PlanetLab, we tested our ingredient-based implementation of the BitTorrent Live protocol as described by Cohen *et al*. [23].

For each of the following sections, we pick a



(a) Handling of Missing Chunks (b) Number of Download Connections (c) Latency Range ingredient

Figure 4: *Testing our implementation of BitTorrent Live with startup buffer of 2 sec. Figure 4(a) shows different missing chunk handling methods. In 4(b) we use the skip ingredient for handling missing chunks and test various download connection ranges. In 4(c) we use the skip ingredient for handling missing chunks, 2-3 parent nodes and test different latency ranges for the Latency Range adaptive playout ingredient.*

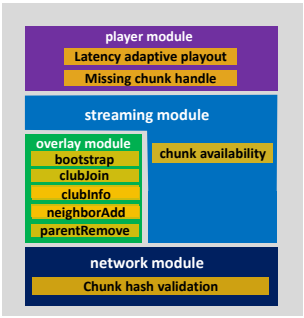


Figure 5: ***BitTorrent Live built from ingredients****. The BitTorrent Live protocol decomposed into ingredients as implemented in* STREAMAID.

single design decision and assess its effect on the aforementioned metrics. Each such design decision is implemented as an ingredient in STREAMAID.

B. Case Study: BitTorrent Live

BitTorrent Live was a highly anticipated decentralized live streaming protocol from Cohen *et al.*, the author of BitTorrent [23], and which to the best our knowledge has not been implemented independently before. Figure 5 shows our decomposition of the BitTorrent Live protocol into ingredients. In the overlay module, a newly joined peer contacts the tracker and receives a partial list of online peers in the *bootstrap* ingredient. The peer also receives number of clubs to join and decides to join them or choose different clubs to join in the *ClubJoin* ingredient. For every club, the peer gathers information on other peers in that club in an instance of the *ClubInfo* ingredient, and forms connections with peers in the club in the *NeighborAdd* ingredient, which also preserves upper and lower bounds on in-club and outer-club connections. Parent nodes that fail to send chunks on time are removed in the *ParentRemove* ingredient. In the streaming module, in order to reduce the amount of redundant chunks being sent by multiple parents, children update their in-club parents upon receiving a chunk in each club in the *Chunk Availability* ingredient. In the player module, the speed of the playback to adjusted to keep the latency bounded in the *Latency Adaptive Playout* ingredient and different options of handling missing chunks (skip, wait, wait and then skip) are also implemented as ingredients. Chunk authenticity is validated by the network module upon reception by the *chunk hash validation* ingredient.

1) Handling of Missing Chunks: When the player cannot play the next chunk because it is missing it will halt. The BitTorrent Live patent proposes several options of handling this case: Wait for the missing chunk, skip the missing chunk or wait for some time and then skip. Each of these options was implemented in STREAMAID as an ingredient. As can be seen from Figure 4(a) (where wait$X$ means waiting for X seconds and then skipping), skip and wait have the best continuity index and waiting and then skipping only hurts the continuity index while also increasing latency. It should be noted that the latency is marginally increased when waiting for missing chunks, but not as much as waiting for more than 1 second and then skipping.

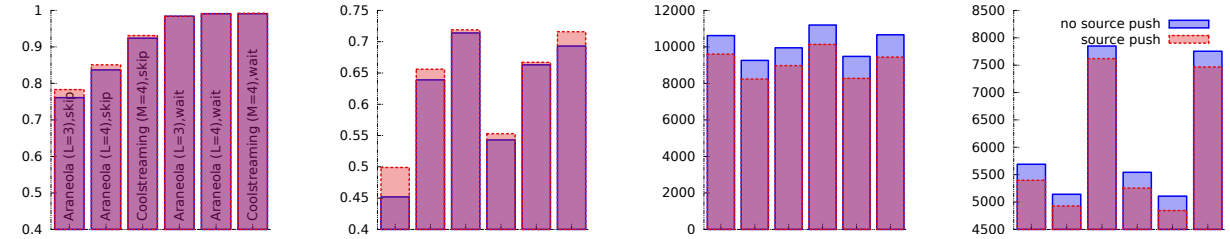
2) Number of In-Club Parent Connections: Another interesting parameter of BitTorrent Live is the number of in-club download connections. As shown in Fig. 4(b), as the number of download connections grow, the Continuity grows, but so does the percentage of duplicated chunks that are sent. We can see that in these settings, having two parents is enough for good continuity index and only yields about 14% duplicates. Note that even when there is only one parent there is a small amount of duplicate chunks due to parent switching.

3) Latency Adaptive Playout Ingredient: As suggested in the BitTorrent Live patent, we have added an ingredient that slows down or speeds up the playback in order to reach some set latency range. As was expected, and evidenced on Fig. 4(c), higher ranges improve the CI, but, increase the latency.

C. Case Study: Pull-based Streaming

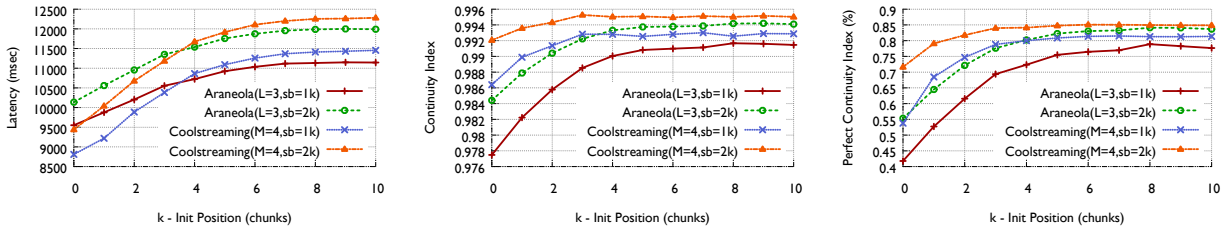
As discussed earlier, a decomposition of a pull-based algorithm into ingredients is shown in Figure 3. Our focus is on variants of CoolStreaming [9] and Araneola [20].

1) Source Push Ingredient: The source node can push newly generated chunks to its neighbors by running a push-based protocol regardless of the streaming protocol run by other nodes. We test this ingredient for several different settings. Fig. 6 shows that **enabling source push always improves the results**. In all settings tested, enabling source push lowered the Startup Delay by 0.2-0.3 seconds and Latency by 1.2 seconds on average and always improved both the CI and the PCI%. We enabled source push in all remaining experiments.



(a) Continuity Index (b) Perfect Continuity Index (c) Latency (msec) (d) StartupDelay (msec)

Figure 6: ***CoolStreaming source push.*** *Source push evaluation of CoolStreaming using the Araneola Overlay ($L = 3$ and 4) and the original CoolStreaming overlay ($M = 4$) waiting for missing chunks and skipping missing chunks. All algorithms use startup buffer of 2 seconds and start buffering from the first bitmap or chunk.* ***Enabling the source push ingredient can only improve performance.***



(a) Latency (b) Continuity Index (c) Perfect Continuity Index%

Figure 7: ***CoolStreaming streaming module*** *using the Araneola ($L = 3$) and CoolStreaming overlays ($M = 4$) with startup buffers of 1 and 2 sec and varying values of k (initial position). All algorithms start buffering from the first bitmap or chunk, use source push and wait if there is no chunk to play.* ***Initializing the player to start at a chunk earlier than the last one available strikes a trade-off between the Perfect Continuity Index% and the Latency while keeping the Startup Delay fixed.***

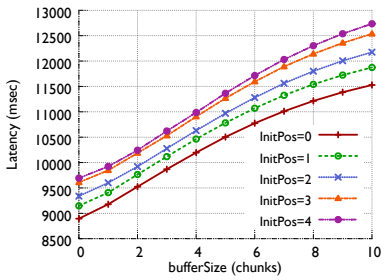
2) Player Initialization Position Ingredient: When the player module is initialized by the streaming module, the streaming module must first choose a chunk from which the playback would start, while also striving to keep latency low. We achieve better performance when initializing the player module only upon receiving a bitmap or chunk. When the player is initialized after receiving a bitmap, there are several possible playback starting positions, which we explore in an experiment. We propose a parameterized ingredient that begins the playback at most k chunks before the most recent available chunk reported in the bitmap. In order to also increase the likelihood of continuous playback, the ingredient initializes the play module to the beginning of the longest consecutive sequence of recent chunks, bounded by k. However, since the first bitmap may be received from a peer who lags behind, we limit the maximum allowed latency of the initialization position.

The Startup Delay is unaffected by the setting of k since the initialization of the player happens at the same time regardless of k. However, other metrics are greatly affected. In Figure 7(a), we see how the latency grows logarithmically with each added chunk. Also, Figures 7(b) and 7(c) show logarithmic improvement of the CI and the PCI%. Although the improvement of the CI is marginal (even with $k = 0$ the index is already at 0.978), the PCI% grows dramatically (Figure 7(c)). We observe that for both the CI and the PCI% the improvement stops for k larger than 3-5 (depending on the overlay), while the latency keeps growing, albeit at a slower pace. We conclude that initializing the player to start at a chunk earlier than the last one available strikes a trade-off between the PCI% and the Latency while keeping the Startup Delay constant. In other words, **assuming a hit in latency is tolerable, increasing k boosts Continuity, and greatly increases the percentage of users with Perfect Continuity.**

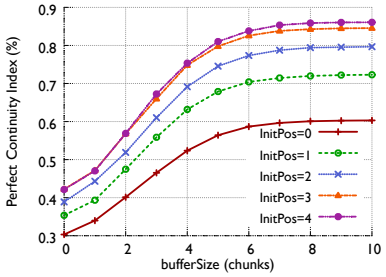
3) Adaptive Playout Ingredient: We discussed in Section III-A how Adaptive Playout can be used to decrease Latencies and Startup Delays. A simple Adaptive Playout ingredient implementation can try to reach a specific pull window size defined by a parameter. More sophisticated ones may leverage other available information. We propose an Adaptive Playout ingredient which tries to measure the minimal window size required for maximal resilience.

Our algorithm works as follows. We measure how long before the playback deadline missing chunks get before they are received. These measurements are being averaged with exponential weights using a sliding window. If chunks are not received before the deadline, then the buffer is too small. Otherwise, if chunks are received long before the deadline, the buffer can be shrunk without losing continuity. We set a target buffer and change the playback speed to reach the target buffer size. The speed change is limited to 10% to minimize annoyance for the user.

In Figure 8, we test the CoolStreaming streaming module with Araneola overlay module ($L = 3$) using startup buffer of 1 sec, and for several player module initialization position values k. We found that **increasing the target buffer behaves in a similar fashion as increasing k: the latencies grow linearly while the Continuity Index and the Perfect Continuity Index% grow logarithmically** reaching a plateau at roughly 7 chunks (graph omitted). The methods can be combined to reach very high Perfect Continuity Index%.



(a) Latency



(b) Perfect Continuity Index%

Figure 8: ***CoolStreaming/Araneola Adaptive Playout****. Latency and Perfect Continuity Index% of CoolStreaming streaming module using Araneola Overlay ($L = 3$) with startup buffers of 1 sec.All algorithms start buffering from the first bitmap or chunk, use source push and wait if there is no chunk to play.*

For instance, with $k = 4$ and the target buffer of 7 chunks, the PCI% hovers above 85% with an average latency of 12 seconds.

V. Related work

Ingredients. Coarse grained decomposition of complex functionality to provide modularity and extensibility has been considered before in different settings, such as micro-protocol layers in replication systems [26], [27], Portable Interceptors in OMG's CORBA communication broker [33], and RPC style remote method invocations augmented with channels in Microsoft's Windows Communication Foundation (WCF) [34]. Our ingredient abstraction is inspired by these works, while cast in the P2P live streaming environment. MOLStream is our previous system primarily designed for the rapid deployment and evaluating of P2P live streaming protocols, which also sought to modularize these protocols [28]. While we leverage the system for deployment, the built-in MOLStream modules are coarse-grained and mostly lack intercompatibility, whereas the finer-grained ingredients we proposed in StreamAid are more atomic, intercompatible and shed light on the design decisions involved.

P2P Live Streaming. There are shrewd surveys of P2P live streaming systems and principles [10], [11], [16], [17], [35]. Most of these works classify systems into tree-based or mesh-based [16], [32], while Zhang *et al.* [11] provide a taxonomy for classifying P2P live streaming protocols. ShadowStream [8] introduces methods for transparently embedding a live streaming protocol to be evaluated into large-scale live streams without affecting the quality for viewers, but relies on access to a production Internet live streaming network.

Chunk scheduling. Many works focus on the chunk scheduling aspect of P2P live streaming. Zhou *et al.* [15] give an analytic evaluation of the Rarest-First and Greedy strategies for chunk request scheduling using their own stochastic model and propose a new mixed strategy that achieves the best of both worlds. Shakkottai *et al.* [36] also evaluate these strategies for minimizing the buffer size and propose a hybrid policy that reduces the required buffer size to ensure high probability of chunk playout. Zhao *et al.* [37] propose a general and unified mathematical framework to analyze a large class of chunk selection policies. Other works propose and evaluate various chunk scheduling algorithms in different settings. Liang *et al.* [38] test five chunk scheduling algorithms in a variety of settings such as different source upload bandwidth, buffer delays, source chunk scheduling algorithms and node degrees. Our work corroborates many of their findings, and expands to other design decisions made by P2P streaming algorithms.

Overlays. Other works cover the overlay building aspect of P2P live streaming. Liu *et al.* [39] analytically derive a new overlay for push-based dissemination; Zhang *et al.* [40] evaluate two different overlay construction strategies: a Random overlay choice where a peer selects neighbors without considering their network locations, and a Nearby-Overlay where a peer only neighbors with nearby peers. In some systems, upload bandwidth affects the number of neighboring peers [41].

Our work encapsulates the concerns discussed in the literature about chunk scheduling, buffer delay, source chunk scheduling and overlay construction into ingredients, allowing them to be evaluated, improved and reasoned about while keeping other aspects of the P2P live streaming system fixed. By systematically applying our abstraction, we also identify several other concerns that we feel have been generally overlooked, such as the player module initialization time and position, and show how they affect the overall performance of the P2P live streaming protocol.

VI. Conclusions

Our large-scale experiments illustrate the power of the abstraction and the flexibility of StreamAid. We show how continuity can be traded for latency or duplicate chunks in the BitTorrent Live protocol; how pull-based streaming protocols always benefit from the source constantly pushing fresh chunks to its neighbors; how choosing the first chunk in playback is an implicit trade-off between latency and playback continuity; how adaptive playout can be used for the same trade-off, and how seemingly minor design decisions significantly impact the overall performance of the live stream. We believe the ingredients abstraction and our open-source framework can help accelerate development and discovery for future P2P live streaming systems.

Acknowledgements: We thank the anonymous reviewers for useful comments and insights. This work is partially supported by

the Technion Hasso Platner Institute (HPI) Research School, grant #152620-051 from the Icelandic Research Fund and funds from Emory University.

REFERENCES

[1] Cisco, "Visual networking index: forecast and methodology, 2013-2018," http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/VNI_Hyperconnectivity_WP.html, 2014.

[2] Point Topic, "Global broadband statistics." http://point-topic.com/free-analysis/global-iptv-subscriber-numbers-q1-2014/.

[3] P. H. O'Neill, "Twitch dominated streaming in 2013, and here are the numbers to prove it," *The Daily Dot*, Jan. 2014.

[4] T. Spangler, "ESPN Live Stream Crashes During USA-Germany World Cup Match," *Variety*, Jun. 2014.

[5] D. F. Amol Sharma, "ABC's Oscars Streaming Outage Shows Web Limitations for TV Networks," *Wall Street Journal*, 2014.

[6] E. Nygren, R. K. Sitaraman, and J. Sun, "The Akamai network: a platform for high-performance internet applications," *ACM SIGOPS Op. Sys. Rev.*, vol. 44, no. 3, pp. 2–19, 2010.

[7] H. Yin, X. Liu, T. Zhan, V. Sekar, F. Qiu, C. Lin, H. Zhang, and B. Li, "Design and deployment of a hybrid CDN-P2P system for live video streaming: experiences with LiveSky," in *Proceedings of the 17th ACM International conference on Multimedia*. ACM, 2009, pp. 25–34.

[8] C. Tian, R. Alimi, Y. R. Yang, and D. Zhang, "Shadowstream: performance evaluation as a capability in production internet live streaming networks," in *ACM SIGCOMM*. ACM, 2012, pp. 347–358.

[9] X. Zhang, J. Liu, B. Li, and Y. Yum, "CoolStreaming/DONet: a data-driven overlay network for peer-to-peer live media streaming," in *Proc. of the 24th IEEE INFOCOM 2005*, vol. 3, pp. 2102–2111.

[10] Y. Liu, Y. Guo, and C. Liang, "A survey on peer-to-peer video streaming systems," *Peer-to-peer Networking and Applications*, vol. 1, no. 1, pp. 18–28, 2008.

[11] X. Zhang and H. Hassanein, "A survey of peer-to-peer live video streaming schemes–an algorithmic perspective," *Computer Networks*, vol. 56, no. 15, pp. 3548–3579, 2012.

[12] B. Li, S. Xie, Y. Qu, G. Y. Keung, C. Lin, J. Liu, and X. Zhang, "Inside the new coolstreaming: Principles, measurements and performance implications," in *INFOCOM 2008*. IEEE.

[13] X. Hei, C. Liang, J. Liang, Y. Liu, and K. Ross, "A measurement study of a large-scale P2P IPTV system," *IEEE Trans. on Multimedia*, vol. 9, no. 8, pp. 1672 –1687, 2007.

[14] M. Zhao, P. Aditya, A. Chen, Y. Lin, A. Haeberlen, P. Druschel, B. Maggs, B. Wishon, and M. Ponec, "Peer-assisted content distribution in Akamai Netsession," in *Proc. of the ACM Conference on Internet Measurement Conference (IMC)*, 2013, pp. 31–42.

[15] Y. Zhou, D.-M. Chiu, and J. C. Lui, "A simple model for chunk-scheduling strategies in P2P streaming," *IEEE/ACM Trans. on Networking*, vol. 19, no. 1, pp. 42–54, 2011.

[16] X. Hei, Y. Liu, and K. W. Ross, "IPTV over P2P streaming networks: the mesh-pull approach," *Communications Magazine, IEEE*, vol. 46, no. 2, pp. 86–92, 2008.

[17] N. Ramzan, H. Park, and E. Izquierdo, "Video streaming over P2P networks: Challenges and opportunities," *Signal Processing: Image Communication*, vol. 27, no. 5, pp. 401–411, 2012.

[18] A. Ganesh, A. Kermarrec, and L. Massoulié, "Peer-to-peer membership management for gossip-based protocols," *Computers, IEEE Transactions on*, vol. 52, no. 2, pp. 139–149, 2003.

[19] N. Magharei and R. Rejaie, "Prime: Peer-to-peer receiver-driven mesh-based streaming," in *INFOCOM 2007*. IEEE, pp. 1415–1423.

[20] R. Melamed and I. Keidar, "Araneola: A scalable reliable multicast system for dynamic environments," in *Third IEEE International Symposium on Network Computing and Applications (NCA).*, 2004, pp. 5–14.

[21] F. Pianese, D. Perino, J. Keller, and E. Biersack, "PULSE: an adaptive, incentive-based, unstructured P2P live streaming system," *Multimedia, IEEE Transactions on*, vol. 9, no. 8, pp. 1645–1660, 2007.

[22] F. Wang, Y. Xiong, and J. Liu, "mtreebone: A hybrid tree/mesh overlay for application-layer live video multicast," in *Distributed Computing Systems, 2007. ICDCS. 27th International Conference on*, pp. 49–49.

[23] B. Cohen, "Peer-to-peer live streaming," September 2012, US Patent App. 13/603,395.

[24] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: an overlay testbed for broad-coverage services," *ACM SIGCOMM*, vol. 33, no. 3, pp. 3–12, 2003.

[25] J. Ruckert, T. Knierim, and D. Hausheer, "Clubbing with the peers: A measurement study of BitTorrent Live," in *P2P*, 2014, pp. 1–10.

[26] H. Miranda, A. Pinto, and L. Rodrigues, "Appia, a flexible protocol kernel supporting multiple coordinated channels," in *Distributed Computing Systems. 21st Int. Conference on.* IEEE, 2001, pp. 707–710.

[27] R. van Renesse, K. Birman, and S. Maffeis, "Horus: A Flexible Group Communication System," *CACM*, vol. 39, no. 4, pp. 76–83, 1996.

[28] R. Friedman, A. Libov, and Y. Vigfusson, "MOLStream: A modular rapid development and evaluation framework for live P2P streaming," in *ICDCS*, 2014.

[29] E. Steinbach, N. Farber, and B. Girod, "Adaptive playout for low latency video streaming," in *Int. Conf. on Image Processing*, vol. 1. IEEE, 2001, pp. 962–965.

[30] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. Mohr, "Chainsaw: Eliminating trees from overlay multicast," *Peer-to-peer systems IV*, pp. 127–140, 2005.

[31] A. Montresor and M. Jelasity, "PeerSim: A scalable P2P simulator," in *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P)*, 2009.

[32] N. Magharei, R. Rejaie, and Y. Guo, "Mesh or multiple-tree: A comparative study of live P2P streaming approaches," in *INFOCOM*. IEEE, 2007, pp. 1424–1432.

[33] "CORBA Portable Interceptors," http://www.omg.org/cgi-bin/doc?formal/04-03-08.

[34] S. Klein, *Professional WCF programming:. NET development with the Windows communication foundation*. John Wiley & Sons, 2007.

[35] A. Sentinelli, G. Marfia, M. Gerla, S. Tewari, and L. Kleinrock, "Will IPTV ride the peer-to-peer stream?" *IEEE Communications Magazine*, vol. 45, no. 6, p. 86, 2007.

[36] S. Shakkottai, R. Srikant, and L. Ying, "The asymptotic behavior of minimum buffer size requirements in large P2P streaming networks," *Selected Areas in Communications, IEEE Journal on*, vol. 29, no. 5, pp. 928–937, 2011.

[37] B. Q. Zhao, J. C.-S. Lui, and D.-M. Chiu, "Exploring the optimal chunk selection policy for data-driven P2P streaming systems," in *Peer-to-Peer Computing, 2009. P2P. IEEE 9th Int. Conference on*, pp. 271–280.

[38] C. Liang, Y. Guo, and Y. Liu, "Investigating the scheduling sensitivity of P2P video streaming: an experimental study," *Multimedia, IEEE Transactions on*, vol. 11, no. 3, pp. 348–360, 2009.

[39] Y. Liu, "On the minimum delay peer-to-peer video streaming: how realtime can it be?" in *Proceedings of the 15th international conference on Multimedia*. ACM, 2007, pp. 127–136.

[40] X. Zhang and H. Hassanein, "Understanding the impact of neighboring strategy in peer-to-peer multimedia streaming applications," *Computer Communications*, vol. 35, no. 15, pp. 1893–1901, 2012.

[41] A. P. C. da Silva, E. Leonardi, M. Mellia, and M. Meo, "A bandwidth-aware scheduling strategy for P2P-TV systems," in *Peer-to-Peer Computing, 2008. P2P. 8th Int. Conference on*. IEEE, pp. 279–288.