MOLStream: A Modular Rapid Development and Evaluation Framework for Live P2P Streaming

Roy Friedman* Alexander Libov*
*Computer Science Department
Technion - Israel Institute of Technology
Haifa 32000, Israel

Ymir Vigfusson†
†School of Computer Science & CRESS
Reykjavik University
Reykjavik 101, Iceland

***Abstract*—**

We present MOLSTREAM, a modular framework for rapid development and evaluation of P2P live streaming systems. MOLSTREAM allows P2P streaming protocols to be decomposed into basic blocks, each associated with a standard functional specification. By exposing structural commonalities between these components, MOLSTREAM enables specific implementations of these building blocks to be combined in order to devise, refine and evaluate new P2P live streaming protocols. Our approach offers several benefits. First, block encapsulation entails that more advanced individual components, *e.g.*, the overlay, can seamlessly replace existing ones without affecting the rest of the system. As a case study, we show how MOLSTREAM can seamlessly substitute the overlay used by DONet/Coolstreaming, a popular P2P live streaming implementation, for an improved version. Second, MOLSTREAM facilitates the comparison between various protocols over local clusters or wide-area testbeds such as PlanetLab. The combination of rapid prototyping and minimum effort evaluation enables researchers and students to faster understand how various design choices at different levels impact the performance and scalability of the protocol, as shown through several examples in this paper. MOLSTREAM is written in Java and is freely available as an open-source project at `https://sourceforge.net/projects/molstream/`.

I. INTRODUCTION

Developing a large-scale peer-to-peer (P2P) live-streaming system is a time-consuming, complex and error-prone endeavor. Such systems typically require a myriad of design decisions, such as the choice of P2P overlay structure, view maintenance algorithms and failure recovery mechanisms, each of which requires substantial effort to evaluate [1]. The complexity is further compounded by the wide range of objectives and metrics used to assess a live-streaming system, including minimizing costs, latency and bandwidth use while maximizing the quality of experience and playback continuity. In particular, some of these concerns are at odds with one another.

Part of the difficulty is the lack of convenient frameworks for rapid prototyping, deployment and evaluation of new algorithms and ideas in real settings. Quick prototyping in a custom simulator runs counter to the goal of real-world evaluation or deployment, and significant effort is required to move from simulation to a proper evaluation of a P2P live-streaming service. Moreover, the effort invested may be spent on parts of the system that are immaterial to proving the efficacy of an idea, such as debugging video codecs or calibrating third-party overlays. Evaluation typically involves a reimplementation of the simulation protocol with improved error handling, and often requires a plethora of metrics to be implemented, measured and assessed on a distributed testbed such as PlanetLab [2] or Emulab [3]. Effects that may have been visible in a synchronous discrete event-based simulator may fade away when effects of real testbeds (such as packet loss, churn or asynchrony) enter the picture, invalidating significant investment of work [4], [1].

To facilitate progress in the field of P2P live streaming, a development and evaluation framework satisfying the following goals is needed.

- **Generality.** The framework should be designed to support a broad family of live streaming constructs, such as interfaces for tree-based and mesh-based overlays, support for centralized and decentralized bootstrap services, implementation of diverse evaluation metrics, and so forth.
- **Modularity.** A P2P live streaming system should be loosely coupled with easily replaceable components [5]. Encapsulation enables isolated parts to be gradually improved, such as optimizing the dissemination overlay, without impact on other components being a concern.
- **Usability.** Coding and implementation should be facilitated by natural, well-defined interfaces. The transition from simulation to large-scale experimental evaluation should be seamless.
- **Measurability.** The framework should be capable of evaluating and reporting common performance metrics and statistics for P2P live streaming, such as end-to-end latency, bandwidth, playback continuity and lag.

While some frameworks, such as PeerSim [6], OverSim [7] and PlanetSim [8], have been developed to address some of these needs in the context of general P2P systems, they lack evaluation capabilities and functionality specific to live streaming applications. A domain-specific solution is required.

In this paper, we aim to disentangle the complexity of building, improving and evaluating P2P live streaming systems by introducing a novel framework MOLSTREAM that satisfies the aforementioned goals. The primary contributions of this paper are the following.

- We design and implement a modular, general framework MOLSTREAM to facilitate rapid prototyping and evaluation of P2P live-streaming systems.
- We demonstrate how several existing protocols can be modularized with MOLSTREAM and how this modularity accelerates improvements to these protocols.

- We evaluate MOLSTREAM by using it to run simulations and experiments on a mash-up of components from existing protocols under several metrics.
- We demonstrate that the same code can be evaluated with MOLSTREAM on both PeerSim [6], a real deployment over a cluster of machines as well as on PlanetLab [2] deployment. Moreover, we show that the results of these runs are consistent across these environments.

Roadmap. The remainder of the paper is organized as follows. We discuss various related works (§II), and explain the basic terminology and assumptions (§III). We describe the design of MOLSTREAM (§IV) and provide more detailed explanation of specific components we have already implemented (§V). We then illustrate the use and benefits of MOLSTREAM through several case studies where we reconstruct and calibrate known protocols using the MOLSTREAM framework (§VI), and offer concluding thoughts (§VII).

II. Related work

The goals of MOLSTREAM of generality, modularity and usability are shared with a number of frameworks that have been developed for general distributed systems and algorithms. To give specific examples, Weevil [9] is a programmable tool to help automate evaluation of distributed systems with focus on workload generation and experimental execution, as opposed to the development process. The SPLAY project [10] enables developers to specify distributed algorithms in the Lua scripting language and experiment with them directly on PlanetLab [2], or other testbeds. SPLAY provides libraries to facilitate development including third-party C/C++ libraries such as video transcoding to experiment with adaptive streaming. ProtoPeer [11] is a Java-based toolkit for prototyping and evaluating P2P systems that can transparently transition between an event-driven simulation and a live network deployment. In a similar vein, Kompics [12] is another generic component model coded in Java that allows mash-ups of event-driven modules to be created, but without imposing a hierarchical component structure like ProtoPeer. To the best of our knowledge, none of these projects have been specifically customized, or used to develop or experiment with P2P live streaming protocols.

A number of event-based simulators for P2P systems have been built to ameliorate research in the field [6], [7], [8]. Nevertheless, the lack of domain-specific features for live streaming may deter their use by researchers: A review of 287 papers in the P2P literature showed that over 62% of simulation-based papers used custom-made simulators, hindering the repeatability of results [13]. MOLSTREAM interacts with these simulators internally, such as by allowing experiments on PeerSim [6] and PlanetLab [2], but the developer is exposed only to a modular interface that pertains specifically to P2P live streaming concerns.

From the literature on ad-hoc P2P networks, MANETKit [14] is an example of a composable modular framework for developing MANET applications. While MANET applications are also a form of P2P systems, they differ from Internet based P2P in the fact that MANETs must rely on geographical proximity, forwarding, routing, and frequent network disconnections.

To achieve realism and scalability that experiments on academic testbeds, such as PlanetLab and Emulab [3] cannot achieve, ShadowStream [1] allows experimental algorithms to be embedded in production live streaming systems without risking performance failure or playback disruptions. Whereas MOLSTREAM is focused on quick and early development of live streaming ideas, ShadowStream is tailored for last-stage evaluation of modules that are nearly ready for production.

Finally, in the area of group communication, well-known examples of modular frameworks include, e.g., Horus [15], Ensemble [16], JGroups [17], Appia [18], QuickSilver [19] and Quilt [20]. Neko [21] is another example that is geared more broadly for consensus protocols and similar replication based systems. These frameworks are designed mostly for state machine replication in clusters and cloud systems and are optimized for such environments. They lack many P2P specific features and support for live streaming, which is the focus of our work.

III. Background and Model

Broadly speaking, our work seeks to accelerate progress in the growing area of P2P live streaming, which we now define more concretely. A P2P live streaming system consists of a set of end user machines that act as *peers* in the system and who interact with the system through a *client application* running on each of their machines. Hereafter, we use the term peer to represent both the donated machine and its user interchangeably.

Live streaming content is offered to the network from one of the peers, known as the *source* for that stream. A peer invokes a *join* operation to begin viewing a given live stream, and subsequently starts receiving a series of *chunks* from the stream. The client application at the peer may decide if and when to play these chunks. Chunks can only be played in the order they were generated by the source, but a peer may opt to play only part of the chunks in a non-consecutive manner.

We assume that peers have limited *bandwidth capacity*, imposed either on the upload link or total link capacity. Capacity constraints limit the aggregate exchange of content that can occur at each time unit between peers.

Latency is defined as the duration of time that passes from the generation of a chunk until the chunk is played at a peer. Streaming systems have different strategies for keeping latency low, with some trying to minimize the *average latency* over all the peers while others trying to minimize the *maximal latency*.

Due to capacity constraints, many P2P live streaming systems parallelize and pipeline the delivery of chunks by forming and maintaining an *overlay* that enables chunks to be transmitted by the source to its overlay neighbors and then ricocheted between neighboring peers in the communication structure. The details of the overlay and forwarding protocol are the primary features that differentiate P2P live streaming systems. For instance, each chunk in an overlay traverses multiple overlay *hops* before reaching a given peer, so the latency of a system can also be measured in terms of hop distances (average or maximal) from the source of the stream.

Another way to define the bandwidth constraints is to define some *cost function* for every connection between two

peers. The cost function would define the cost of using each connection as a function of the data transfer rate between the two peers. Systems minimize the overall cost of the stream dissemination over the P2P network. Typically, the cost function is a linear function that is equal for all the connections (then, the problem translates to the Steiner Tree Problem [22]). This method can be utilized to minimize latencies when the cost function is based on the latency between the two peers.

Another concern that P2P systems should worry about is the communication *overhead* they impose. The overhead can be expressed either in terms of messages or in terms of bandwidth. The former can be computed as the average of total number of messages received by each node versus the number of chunks this node has received. In the latter case, we take the total number of bytes transmitted in the system per peer versus the total number of bytes in all the streams' chunks.

Often, the clients are heterogeneous, i.e., every client can have different requirements for content quality. We can define a *utility function* that for each client would define the utility for that client for every content quality that client is receiving. Different Multiple Descriptor Coding (MDC) [23] schemes can be used to achieve different qualities for different users.

Some clients may wish to consume the content without interruptions at the cost of higher latency. For example, in Coolstreaming [24] peers buffer content for 10 seconds, then play the chunks continuously, skipping content if a chunk is not available. They define the *continuity index* as the number of chunks played out of the total possible chunks that could be played during the session of a peer.

Real P2P systems must handle *churn*, characterized by the rate at which new peers enter the system and existing peers drop out of the system. A successful P2P system should be able to continue its service despite churn and minimize the impact of churn on the performance parameters mentioned above.

Finally, for the success of a P2P network, peers need to be cooperative and execute the protocol as specified. We can distinguish between *altruistic* peers, who execute the protocol as prescribed even if they do not gain anything from doing so, vs. *selfish* peers who are willing to cooperate only if they benefit from this. In particular, peers that only consume services from the system but do not help others are known as *free riders*. *Incentive* mechanisms reward peers for their contribution to the system and greatly limit the ability of free-riders to get service from the system. This way, selfish peers are motivated to participate in the protocol rather than becoming free-riders. Incentives have been studied extensively and are not at the focus of this paper.

IV. MOLSTREAM ARCHITECTURE

The MOLSTREAM framework consists of roughly 10,000 lines of Java code. As seen in Figure 1, the system architecture is modular and consists of several generic components that we describe below. Each component has a well-defined, minimal interface and may be instantiated by different implementations. The bindings to specific implementations occur at run-time based on configuration parameters. Multiple instantiations of the same component type may be executed concurrently within the same system, and one may even invoke methods of the other, as we explain later.

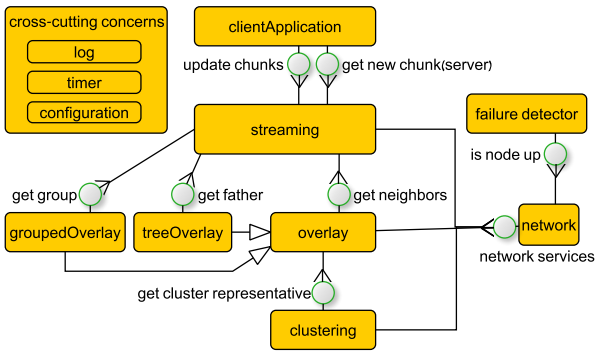


Figure 1. **Architecture.** An illustration of the components present in the MOLSTREAM implementation and the interfaces between them.

Cross-cutting services. As shown in Figure 1, MOLSTREAM includes several generic services which permeate all other components. These include CONFIGURATION, TIMER, and LOGGING. The CONFIGURATION provides access to the systems' configuration parameters, as we describe later in §V-C. These parameters are made persistent to an XML configuration file. The TIMER component allows other components to register a listener method called `nextCycle`. The listener is invoked periodically at a frequency controlled by a configuration parameter. The LOGGING service records errors, debug information, and performance data in log files. In particular, it records various performance counters and statistics that enable viewing and exploration of many performance metrics through an accompanying REPORTING application (not shown here).

Component layers. Next, we give a bottom-up description of the main modules that form the bulk of the streaming system. The lowest layer is the NETWORK component which deals with all networking related issues. The component binds to the necessary interfaces for networking, which could be an external library such as when running the PeerSim simulator, or the standard UNIX socket library when running on a real IP network, as specified in the configuration file. The layer also handles NAT and firewall traversal in Internet-wide deployments using the STUNT protocol [25]. The interface to the NETWORK component includes a best-effort `sendMessage` method. It also includes a listener (upcall) `handleMessage` method as well as procedures for exploring and influencing various network characteristics such as upload bandwidth.

The FAILUREDETECTOR component relies on the NETWORK component for detecting failures of other peers. It exposes an `isUp` method that returns true if the peer seems to be up and responding and false otherwise. The method can be invoked by any other component in the system. The accuracy of the failure detector response depends on the actual operational environment [26].

The role of the CLUSTERING component is to divide the network into clusters according to a metric, e.g., locality. The programming interface includes a *getSource* method that returns the identity of the stream's source peer within the cluster. While the current implementation of the CLUSTERING component treats all peers as members of a single cluster, we plan to include more sophisticated mechanisms in future

versions of MOLSTREAM.

Further up is the OVERLAY component, which manipulates a `Neighbors` object to form and maintain an overlay. The neighbor selection process can depend on a multitude of parameters. Incentive mechanisms that affect the neighbor selection are implemented in this component. Changes in the overlay are triggered by the `nextCycle` and `handleMessage` listener methods. OVERLAY exposes a `getNeighbors` method. As noted in Araneola [27] discussion below, an implementation of OVERLAY may invoke methods of another concurrently running implementation of OVERLAY, which removed any redundancy from our Araneola implementation [27]. Also, there are two kinds of components that inherit from OVERLAY: a TREEOVERLAY and GROUPEDOVERLAY. The former adds the `getParent` and `getChildren` methods; the latter splits neighbors into multiple groups that can be retrieved using the `getGroup` method.

Next is the STREAMING component, which implements the actual chunk dissemination protocol. Typical examples include *push*, *pull*, and combined *push-pull* protocols over the overlay's edges, but other mechanisms can also be used. This component also implements incentive mechanisms that affect the actual chunk exchange between neighbors. Also, STREAMING may use multiple concurrently running OVERLAY components, for instance push chunks over a tree overlay while pulling missing chunks over a mesh for robustness, as discussed for mTreeBone in §V-A below.

At the highest level is the CLIENTAPPLICATION component, which is responsible for generating the stream's chunks at the source and for playing the stream's chunks at other peers. This is also where all UI and codec issues are handled (or delegated to other modules [1], currently outside the scope of our framework). The main interface between CLIENTAPPLICATION and STREAMING is through obtaining generated chunks from a `ServerVideoStream` object at the source and passing chunks to a `VideoStream` object as other peers. The default implementation of CLIENTAPPLICATION includes buffering chunks for a period of time whose length is a configuration parameter as well as a policy whether to wait or skip missing chunks that is also a configuration parameter. An important role of the STREAMING component is to decide when the playback starts. For the case when a missing chunk is skipped, this decision actually sets the latency of the whole playback. This could also lead to dangerous situations when the playback has started with too short latency and the streaming algorithm is unable to deliver the required chunks in time before the playback deadline (for example, a node was preempted in a tree overlay). To that end, the STREAMING component can intentionally add delay to the playback, increasing the continuity at the cost of increased latency. If the CLIENTAPPLICATION waits for a chunk when it is missing, adding delays is unnecessary. However, one missing chunk could lead to big delays in extreme cases.

V. LIBRARIES

To facilitate the development of new P2P streaming systems, MOLSTREAM provides an extensible library of implementations for several popular protocols that tend to show up as building blocks for other, more complicated, protocols. We briefly discuss the main components.

Table I. **LIBRARY MODULES**. THE LINES OF CODE (LOC) OF THE MOLSTREAM JAVA IMPLEMENTATION FOR EACH COMPONENT, INCLUDING ERROR HANDLING.

Module	Type	LOC
Coolstreaming [24]	Streaming algorithm	278
Prime [28]	Streaming algorithm	305
mTreeBone [29]	Streaming algorithm	35
TreePush	Streaming algorithm	46
Bootstrap: Random group	Overlay protocol	94
Bootstrap: Random node	Overlay protocol	73
General gossip-based overlay	Overlay protocol (gossip-based)	87
SCAMP [30]	Overlay protocol (gossip-based)	69
BSCAMP	Overlay protocol (gossip-based)	25
Prime [28] overlay	Overlay protocol (group-based)	57
TreeBone [29] overlay	Overlay protocol (tree-based)	298
Coolstreaming [24] overlay	Overlay protocol (gossip-based)	182
Araneola [27], [31]	Overlay protocol (gossip-based)	197

A. Overlay modules

The MOLSTREAM library contains a number of overlay protocols that can be used as-is or extended as needed. The implementation for each overlay protocol is between 25 to 298 lines of Java code, as shown in Table I.

Each overlay protocol is further subdivided into the following three constructs. The *Bootstrap* logic is responsible for finding an initial set of peers when only a single peer is known. The *Neighborhood manager* determines and handles connections formed between pairs of peers. Finally, the protocol should implement a *Failure recovery* mechanism to deal with unresponsive neighbors. These are specified in the OVERLAY template in MOLSTREAM.

SCAMP. A number of overlay algorithms have been implemented and tested in MOLStream. The popular DONet/-Coolstreaming system [24] defines an overlay protocol which relies on an underlying overlay protocol called *SCAMP* [30] to disseminate membership messages. SCAMP maintains a unidirectional overlay designed to keep the average number of neighbors for each node at $\log(n)(1 + c)$, for a constant $c \geq 0$, where n is the size of the network. In the MOLSTREAM OVERLAY template, our SCAMP implementation bootstraps each new peer by sending a random seed peer from the bootstrap node. Whenever a connection request is received, the peer forwards it to its neighbors along with c additional forward copies of the connection request. Upon receiving a forwarded connection request, the peer adds the new connection with some probability. To recover from failures, SCAMP peers that stop receiving messages restart the entire protocol using the bootstrap node or one of their neighbors. We have also added a bidirectional version of SCAMP called BSCAMP, where all neighboring connections are mutual.

Coolstreaming. For the overlay structure itself, Coolstreaming defines a protocol that is designed to keep a stable amount of neighbors defined by a custom system parameter M. As in SCAMP, a new peer begins its tenure by receiving a seed peer (called a "deputy" [24]) from the bootstrap node. The new peer sends a request to the seed (deputy) peer for additional nodes. The peers then use SCAMP to disseminate membership messages. To manage the neighborhood, peers store a cache of known peers from the information received from the membership messages, and bias the cache towards storing those nodes with whom the peer has exchanged a large number of chunks of the stream. When the number of

Listing 1. **Sample code.** General gossiping protocol implemented in MOLSTREAM. The extract shows the main source code to implement the Araneola membership overlay [27], except for failure handling cases.

```
@override public void handleMessage(final Message message) {
  if (message instanceof PartialMembershipViewMessage) {
    final List<NodeSpecificImpl> neighborsList =
        ((PartialMembershipViewMessage) message).neighborList;
    // add the neighbors received in the message
    for (final NodeSpecificImpl newNeighbor : neighborsList) {
      addNeighbor(newNeighbor);
    }
    addNeighbor(message.sourceId);
    // prune random neighbors if needed
    while (getNeighbors().size() > groupSize) {
      final List<NodeSpecificImpl> nList =
          new ArrayList<NodeSpecificImpl>(getNeighbors());
      removeNeighbor(nList.get(random.nextInt(nList.size())));
    }
  }
}
@override public void nextCycle() {
  super.nextCycle();
  if (currentDelay-- == 0) {
    currentDelay = gossipDelay;
    final Set<NodeSpecificImpl> neighbors = getNeighbors();
    final List<NodeSpecificImpl> neighborsList =
        new ArrayList<NodeSpecificImpl>(neighbors);
    for (final NodeSpecificImpl neighbor : neighbors) {
      neighborsList.remove(neighbor);
      Collections.shuffle(neighborsList, random);
      final List<NodeSpecificImpl> sublist =
          neighborsList.subList(0,
          Math.min(amountToSend, neighborsList.size()));
      node.send(new PartialMembershipViewMessage(getMessageTag(),
       node.getImpl(), neighbor, sublist));
      neighborsList.add(neighbor);
    }
  }
}
```

neighbors drops below M, a peer contacts a random peer in its cache with a connection request.

Araneola membership. MOLSTREAM also implements a scalable randomized membership protocol similar to [31] used by Araneola [27]. Araneola's membership service begins by contacting a random peer group received from the bootstrap node. Members of the service gossip once in a while with their known neighbors, one random peer per round. Upon receiving new neighbors, the membership service adds them all as neighbors and then discards random neighbors if it exceeds a maximal neighborhood size. The maximal neighbors size is a parameter of the algorithm. When a peer has no more live neighbors left or stops receiving messages, it restarts the protocol using the bootstrap node.

The two main routines behind MOLSTREAM's implementation of Araneola's membership service are shown in Listing 1, except failure handling code which was removed for clarity. The figure demonstrates MOLSTREAM's interfaces in practice.

Araneola overlay. Araneola uses a separate overlay protocol for tracking its membership view [27]. Based on the Araneola membership service above, Araneola maintains an overlay that approximates a random regular graph. To bootstrap the Araneola overlay, nodes piggyback on the membership protocol by connecting to nodes in the current view. Nodes continually make connections and disconnect while striving to ensure that each node has exactly L (a configurable parameter) or $L+1$ neighbors. Each node is aware of the degree of each of its neighbors through a periodic exchange of information. When a peer is unable to accept a new connection because its degree is already $L+1$, it responds to the connect request with a NACK along with a list of its least loaded neighbor nodes as a hint to the connecting node. To prevent disconnections in the overlay, Araneola nodes actively connects to other peers when their degree drops below L.

TreeBone. The TreeBone overlay is used by mTreebone [29]. Initially, nodes choose a random node and become its children. A node accepts another node request only if it has enough upload bandwidth to maintain a streaming connection at the desired rate. Nodes with high uptime are viewed as stable and will gradually start joining other stable nodes near to the root of the tree. Stable nodes also perform transformations that decrease the maximal or average depth of the tree. Should the parent of a node fail, the peer will look for a new parent.

Prime Overlay. Finally, MOLSTREAM also supports the overlay protocol used by the Prime system [28]. In Prime, all neighbor nodes are defined as either parent nodes or child nodes. The service is bootstrapped by requesting a random group of peers from the bootstrap node and sending these peers a connection request. A node will send a connection request only if it has enough download bandwidth, and nodes will accept a connection request only if they have enough upload bandwidth. When all the parents of a peer fail, the peer restarts the algorithm. Prime Overlay implements the groupedOverlay interface providing a parents group and children group.

B. Streaming Protocols

Each of the topologies listed so far can be spliced with a number of streaming protocols that have been implemented. We will list the main ones here.

Coolstreaming. The live streaming mechanism of Coolstreaming [24] utilizes the Coolstreaming overlay protocol mentioned above. In this service, nodes periodically exchange data availability information with their neighbors, retrieve unavailable data from one or more neighbors, or supply available data to their neighbors. Each node continuously exchanges availability bitmaps of its segment with the neighbors, and then schedules which segment should then be fetched from what neighbor. The scheduling heuristic first calculates the number of potential suppliers for each segment. The algorithm then determines the supplier of each segment by starting with those that have only one potential supplier. When multiple potential suppliers could be chosen, Coolstreaming selects the one with the highest bandwidth and most ample time before the playback deadline. Even though the implementation of Coolstreaming in [24] relies on the Coolstreaming overlay, notice that any other overlay protocol could have been used as well. We will explore this further, along with the modularity of modern streaming services, in a case study in §VI.

mTreebone. We treat mTreebone [29] as a general push-pull protocol that uses a tree overlay protocol for disseminating messages together with a pull protocol that has a limited exchange window if the tree-push protocol failed to deliver chunks nearing their deadline. To implement mTreebone as in the original paper [29], we use TreePush with the TreeBone overlay for regular message dissemination and then use Coolstreaming with its Coolstreaming overlay as the pull

protocol. Note that any combination of a tree and other overlay component could be used instead and may produce better results for different scenarios.

TreePush. As a baseline, we implemented a simple streaming protocol that uses a tree overlay called TreePush. In TreePush, the source node waits for chunks to become available and then sends chunks to each child. Every node that receives content from its parent node immediately forwards the content to its children.

Prime. PRIME [28] groups peers into levels based on their shortest distance from the source. Chunk dissemination consists of 2 phases: a "diffusion phase" in which all participating peers should receive a data unit (a single description when Multi Description Coding is used) of the chunk as fast as possible and a "swarming phase" in which peers exchange their data units with each other until receiving their desired quality of the chunk. Prime requires a grouped overlay that has a parents group and a children group (such as the Prime Overlay component).

C. System Parameters

Having described the main libraries provided by MOLSTREAM, we now elaborate on some of the system configuration parameters.

1) Playback settings: The handling of missed chunks is an important design decision for live streaming systems. The system can wait for the missing chunk to arrive, possibly for some time, or skip it and proceed to the next available chunk. The decision affects latency and continuity and thus the viewing experience for end users.

Another configuration option is the buffering time which is exposed as two parameters in MOLSTREAM. The first is the time that passes after the system starts and before the playback is started, called *serverStartup*. The second is the duration which peer wait before starting playback, called *startupBuffering*. A larger *serverStartup* time gives additional opportunity for the overlay to organize and stabilize before the actual playback starts. *startupBuffering* increases the latency of stream, but can improve the continuity of the playback.

2) Network interface: The framework choice is orthogonal to the actual implementation of the framework components. The NETWORK component has a `NetworkNode` abstract class that interfaces with the underlying network. `NetworkNode` defines these methods.

- `NodeSpecificImplementation getImpl()` Returns a container for the implementation of the node. `NodeSpecificImplementation` encapsulates the unique identity and address of a node that can later be used to send messages measure latency etc.
- `boolean send(Message msg)` Sends a message. The destination `NodeSpecificImplementation` is already in the message.
- `long getUploadBandwidth()` Returns the upload bandwidth of the node in bits per second.
- `long getEstimatedLatency( NodeSpecificImplementation key)` Returns the estimated latency to a specific node.
- `NodeSpecificImplementation getServerNode()` Returns the address of the server node. Used by the clustering algorithm.
- `Send Info` The amount of data sent and received as well as the amount of useful data (content) that is sent and received from each neighbor is stored for use by different algorithms. The data can be used for peer selection.

More classes can be developed that extend the `NetworkNode` abstract class.

3) Deployment: In addition to being designed for quick prototyping, MOLSTREAM permits protocols to be deployed easily in simulated environments and on real test-beds. Currently, MOLSTREAM supports both running over the PeerSim simulator [6] as well as running over a real IP network using UDP. The framework is extensible and support for other simulation platforms can be added. Each simulation is fully reproducible which simplifies the debugging of new components compared to non-deterministic experiments over real networks. Experiments, on the other hand, produce more realistic results and trigger corner cases that are not always reachable by simulation. To enhance the realism of the simulations, we support node failures, constraints on upload bandwidth, drop rates and latencies.

4) Bitrate and upload bandwidth: The bitrate of the stream, upload bandwidth of the source and the upload bandwidth distribution can be configured. The effect of these parameters is twofold. First, some algorithms use the bitrate of the stream together with the upload bandwidth to define the maximum amount of neighbors or data to be sent. Second, algorithms that do not account for the bitrate or upload bandwidth may attempt to send too much data which may hamper latency.

5) Churn model: Churn experiments are critical to the analysis of any P2P protocol. We have implemented several churn models in MOLSTREAM that can be employed.

- **none** No churn, the number of peers remains constant throughout the run.
- **sessionLengthInterArrival** Allows distributions for the session length of nodes and the inter-arrival time for any two new nodes to be specified. Whenever a session ends, a node fails. The addition of new nodes is independent of the failures.
- **sessionLengthAddOnFailure** A distribution is specified for the session length of a node. When a session ends, the node fails and a new one joins at the same instant.
- **sessionLengthOffLength** In this model, a distribution for the session length of a node and a distribution of cool-off times. After a session ends, the node fails and a new one joins after the cool-off period.
- **availabilityFile** To support trace-driven simulation, this model parses a file describing the arrival and departure times of nodes. The nodes in the system behave as described in the file.
- **massiveChurn** Under this model, a large number of nodes will leave the system at a particular instant (simulating a massive correlated failure) and return later. The number of nodes and delays are parameters.

D. Supported Performance Metrics

MOLSTREAM logs a vast array of performance counters and statistics during each run. Some of the counters are time based, e.g., *startup delay* - the average time it takes for a new peer to start receiving content, *average latency* - the average time from the chunk generation until the chunk playback and *lag* – the difference in latency between the chunks played last and first. Other counters measure the data consumption such as total amount of data sent by a protocol, or amount of data sent by the nodes during each second of the playback. Last, there are counters that are chunk related such as the number of chunks played each second and the number of peers that have played each chunk. Some statistics are available based on the distance of the node from the source, for example the average latency per hop distance.

VI. CASE STUDIES

To demonstrate the effectiveness of MOLSTREAM, we investigate the effort taken to implement, test and deploy a popular P2P live streaming service in the literature: Coolstreaming [24]. The first goal is to replicate the implementation and results from the paper by Zhang *et al.* and then show how MOLSTREAM facilitates modular improvements.

1) Assumptions: In the following, we show usage examples of MOLSTREAM for different cases. If not stated otherwise, we have used the PeerSim simulator. If a chunk is unavailable when the playback time is reached, the chunk is skipped. The server startup time is set to 10 seconds. When using PeerSim, we run each experiment five times with a different random seed, and average the results.

2) Coolstreaming: We have implemented Coolstreaming inside MOLSTREAM in only 278 lines of Java code plus 184 LOC for the modified SCAMP overlay used in the paper. We experimented the implementation with a similar settings to the ones reported in [24] and obtained similar results as in that paper. For our deployment, we used the PeerSim simulator with 10 to 200 nodes, with playback of 10 minutes of 500Kbps, 10 second *startup buffer* and no churn. In Figure 3(a), we can see the continuity index for different network sizes and different values of M – the overlay parameter of Coolstreaming. Figure 3(b) shows the overhead incurred by the different M settings. In the graph, the playback continuity index improves with higher M values. However, after $M = 4$, the improvement becomes marginal. The graphs strongly resemble the graphs in the Coolstreaming paper [24] as can be seen in Figure 3, suggesting that MOLStream enabled the prototyping of a relatively complicated protocol in less than 500 LOC of Java in addition to facilitating experimentation.

3) Coolstreaming with different overlays: We have tested the Coolstreaming streaming algorithm with different overlays other than the original Coolstreaming overlay. We have used a startup buffer of 5 seconds and a network of 200 nodes with playback time of 12 minutes and no churn. Each node's upload bandwidth is set to 5.56 Mbps[1], while, the server's upload bandwidth is set to 16.68 Mbps (3 times the average). We have chosen the parameters of the different overlays so that the average node degree would be as similar as possible.

[1]Average upload bandwidth taken from http://www.netindex.com/upload/.

Table II. CONTINUITY INDEX, PLAYBACK LATENCY AND CONTROL OVERHEAD MEASURED FOR DIFFERENT OVERLAYS WITH THE COOLSTREAMING STREAMING COMPONENT

Overlay	Continuity Index	Playback Latency (ms)	Control Overhead
Araneola ($L = 3$)	0.997	9893	0.008
BSCAMP	0.996	9771	0.008
Coolstreaming ($M = 4$)	0.97	9239	0.01
Prime	0.925	8624	0.006

Figure 2 shows the average degree as a function of the uptime of a node. Recall that Coolstreaming overlay periodically drops the lowest scoring partner. We have set the parameter of Coolstreaming overlay to drop the lowest scoring neighbor every 30 seconds. As can be seen in Figure 2, indeed there is a drop in the Coolstreaming Overlay degree every 30 seconds. Table II summarizes the results. In this setting, Araneola and BSCAMP (Bidirectional SCAMP) obtain a high continuity index. Nevertheless, their playback latency is also high. On the other end of the spectrum, the Prime overlay has the lowest continuity index, but obtains the lowest latency and control overhead. Prime does not make any action after the initial bootstrap. Prime manages to get such low latency and overhead only because there are no failures in the setting. It is important to note that Prime is a hierarchical group-based overlay, whereas Coolstreaming treats all neighbors of Prime equally and makes no distinction based on group membership.

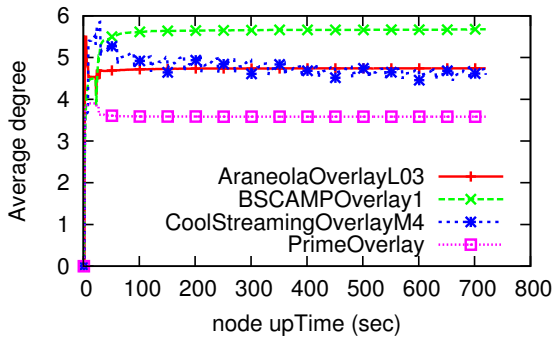
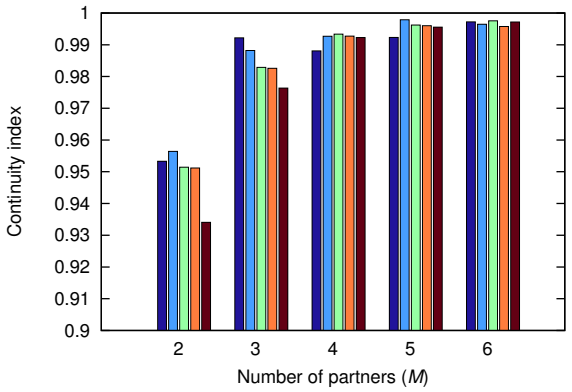
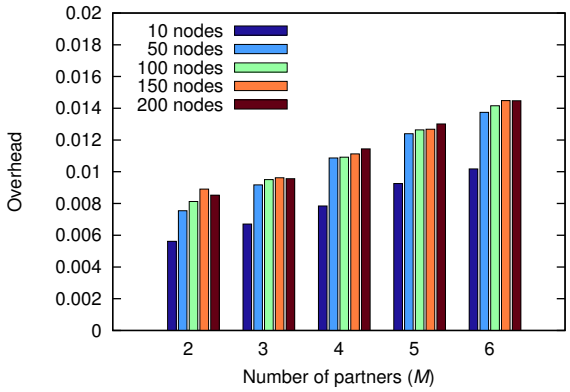


Figure 2. Average degree of Coolstreaming with different overlays

4) Araneola parameter tuning: To demonstrate how MOLSTREAM can be used to tune the parameters of the different algorithms, we tested Coolstreaming with the Araneola overlay in a network of 512 nodes simulated by PeerSim. The simulation length is 10 minutes. We use the *sessionLengthAddOnFailure* churn model: the session length of the nodes is log-normally distributed with *mean* $= 4.29$ and *variance* $= 1.28$ (following Magherei et al. [32]). When a node fails, a new one is added. Each node had an upload bandwidth of 5.56 Mbps, while the simulated stream bitrate was 450 Kbps. In these settings, each peer can maintain roughly 12 neighbors to which it could send one chunk of the stream each second ($5560/450 \simeq 12$). In Figure 5(c), we see that the minimal average latency for the Araneola algorithm is reached when the L parameter is exactly 12. This is because Araneola tries to achieve exactly L or $L+1$ neighbors for every node. The same behavior is evident in Figure 5 which portrays the average degree over uptime. As can be seen in Figure 5(b), Araneola takes roughly 5 seconds to amass neighbors and reaches L in



(a) MOLSTREAM Coolstreaming Playback continuity index.



(b) MOLSTREAM Coolstreaming overhead.

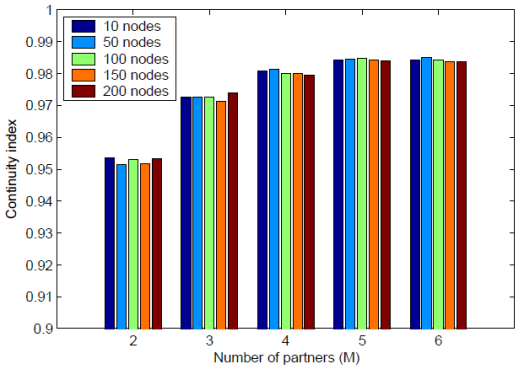


Fig. 8. Continuity index as a function of the number of partners.

(c) Original Coolstreaming continuity index graph (from [24]).

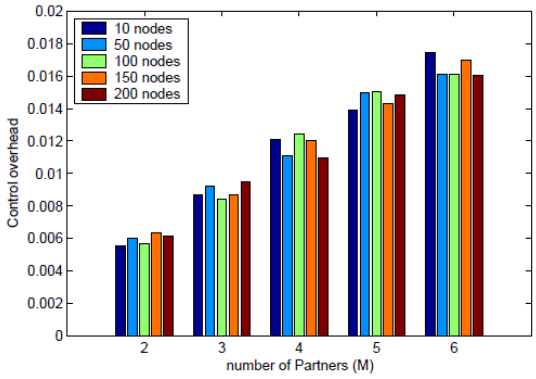


Fig. 7. Control overhead as a function of the number of partners for different overlay sizes. (Control overhead= Control traffic volume/Video traffic volume at each node).

(d) Original Coolstreaming overhead graph (from [24])

Figure 3. **Case study of Coolstreaming.** The continuity index and the overhead of a MOLSTREAM Coolstreaming implementation protocol in the MOLSTREAM compared to the original results of the Coolstreaming protocol.

roughly 25 seconds. The degree is maintained throughout the whole run as evidenced in Figure 5(a).

5) MOLSTREAM *network framework:* We have used a cluster of 31 machines to test the network framework of MOLSTREAM. To that end, we have ran the original Coolstreaming streaming algorithm with the Coolstreaming overlay with the parameter $M = 4$. We ran a stream of 500Kbps for 10 minutes with no churn. We compare these results to an identical settings using the PeerSim simulator and PlanetLab.

As shown in Figure 6, the results are comparable. The PeerSim simulation has a slightly higher latency since the simulated latencies are probably higher than the actual latencies in the cluster. In PlanetLab the latencies are naturally higher.

We have also tested on PlanetLab with up to 180 nodes. Figure 4 shows that the results scale up with a slight improvement of the continuity index as the system grows.

6) mTreeBone transformations: We have implemented the mTreebone algorithm [29]. In this experiment, we use the standard Coolstreaming streaming protocol (with $M = 4$) as the fallback streaming protocol of mTreebone. Also, we simulate churn using the *sessionLengthOffLength* churn model.

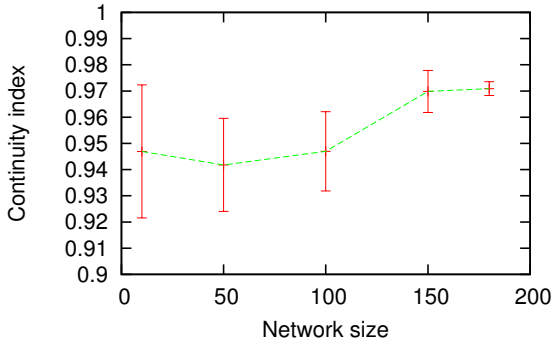
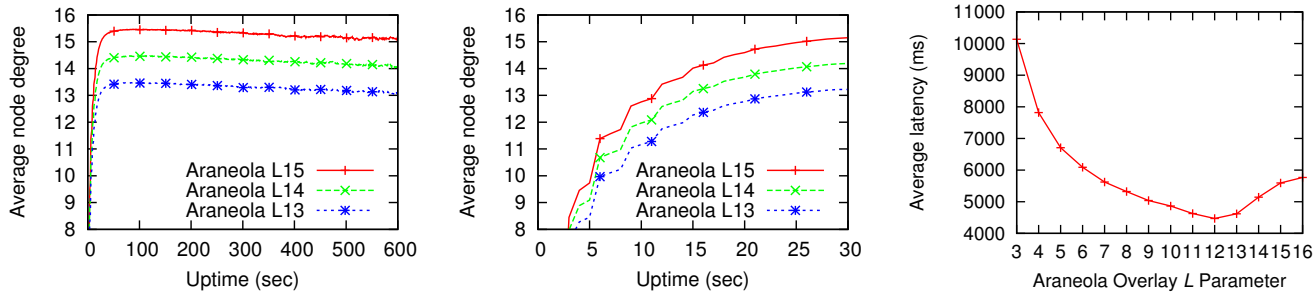


Figure 4. Continuity index and one SD for various sizes of networks of PlanetLab nodes

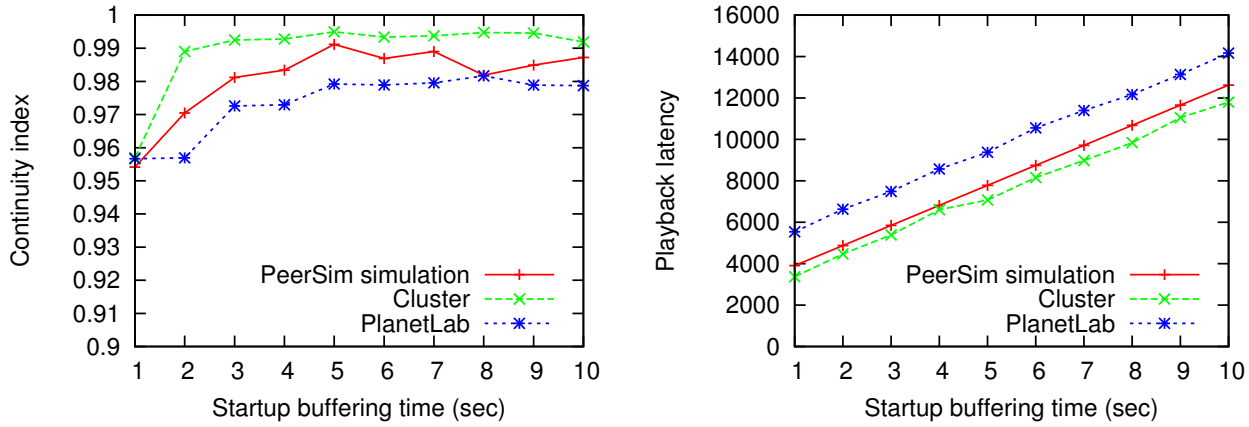
In which, the session length is exponentially distributed with mean of 50 seconds and then the node waits a period that is also exponentially distributed with mean of 50 seconds before rejoining the network (as was done in [24]). We have set the startup buffering time to one second.

We have tested the effect of the stable coefficient on the



(a) Node degree (b) Node degree during warmup (c) Average latency

Figure 5. **Araneola overlay. (a) and (b)** Average node degree as a function of uptime of a node **(c)** Average latency of different L settings.



(a) Continuity index (b) Latency

Figure 6. Comparison of Coolstreaming using PeerSim simulator, Cluster and Planetlab testbed

performance of the protocol. The stable coefficient defines when a node becomes stable, and thus able to perform transformations, with a coefficient of zero implying that all nodes are always stable and a coefficient of one implying that no node is ever stable (except the source node). We found that the continuity index is roughly the same for all the settings. In contrast, higher coefficient means less overhead and lower latency, yet above a value of 0.2 the differences are marginal. We omit the graph due to lack of space.

VII. Discussion and Conclusion

We have described MOLSTREAM, an open source modular framework for rapid prototyping, testing and performance tuning of P2P live streaming protocols. As demonstrated, MOLSTREAM facilitates comparing between different protocols, and in particular, between the various aspects of the protocols such as overlay maintenance and streaming. In particular, MOLSTREAM includes a built-in logging and reporting mechanism that records and generates statistics on common performance metrics we have found in various research papers on the subject, including, for instance, latency, throughput, communication overhead, continuity index and lag. Engineers can use our framework to establish which solution works best for their settings, while developers of new protocols and researchers benefit from MOLSTREAM as it enables them to focus only on the parts they wish to study and improve.

The ability to run the same code on both a simulator (PeerSim) and in a real deployment facilitates the transition from a proof-of-concept simulation to a real-world experiment, and from communication over emulated environments to physical networks. PeerSim is a mature and realistic simulator that we use by default; changing to a different or home-brewed simulator within MOLSTREAM is simple to do as simulation code is encapsulated behind a simple interface in the NETWORK component.

We have shown that various known protocols have been implemented and tested in MOLSTREAM. Relatively little effort is required: the implementations for each of the various components of MOLSTREAM, for example, comprise only between 35-305 lines of Java code. As mentioned earlier, MOLSTREAM is available at `https://sourceforge.net/projects/molstream/`.

In future versions of our framework, we plan to include additional protocols and in particular to experiment with various novel combinations of overlay and streaming protocols. The modularity and isolation MOLSTREAM affords further opens doors for innovation. In particular, many protocols that appear in the literature include or can be subjected to a wide range of optimizations and heuristics. Some of these ideas have been

mentioned by other authors but often without satisfactory evaluation. We believe the componentization of MOLSTREAM can help determine and separate out the performance implications of such ideas and optimizations, enabling new insights to be obtained in a breeze.

Acknowledgements: We would like to thank the anonymous reviewers for their useful comments and insights. Also, we thank Ido Gonen and Dor Hovav for implementing the Prime streaming and overlay components. This work is partially supported by ISF grant 1247/09, grant #120032012 from the Icelandic Research Fund, and the Technion Hasso Platner Institute (HPI) Research School.

REFERENCES

[1] C. Tian, R. Alimi, Y. R. Yang, and D. Zhang, "ShadowStream: performance evaluation as a capability in production Internet live streaming networks," in *Proc. of the ACM SIGCOMM conference on applications, technologies, architectures, and protocols for computer communication*, 2012, pp. 347–358.

[2] 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 Computer Communication Review*, vol. 33, no. 3, pp. 3–12, 2003.

[3] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 255–270, 2002.

[4] C. Wu, B. Li, and S. Zhao, "Diagnosing network-wide P2P live streaming inefficiencies," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 8, no. 1S, pp. 1–19, Feb. 2012.

[5] 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.

[6] A. Montresor and M. Jelasity, "PeerSim: A scalable P2P simulator," in *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P)*, Seattle, WA, Sep. 2009, pp. 99–100.

[7] I. Baumgart, B. Heep, and S. Krause, "OverSim: A Flexible Overlay Network Simulation Framework," in *IEEE Global Internet Symposium*, 2007, pp. 79–84.

[8] P. García, C. Pairot, R. Mondéjar, J. Pujol, H. Tejedor, and R. Rallo, "Planetsim: A new overlay network simulation framework," in *Software engineering and middleware*. Springer, 2005, pp. 123–136.

[9] Y. Wang, A. Carzaniga, and A. L. Wolf, "Four enhancements to automated distributed system experimentation methods," in *Proceedings of the 30th International Conference on Software Engineering (ICSE'30)*. ACM, 2008, pp. 491–500.

[10] L. Leonini, É. Rivière, and P. Felber, "SPLAY: distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze)," in *Proc. of the 6th USENIX symposium on Networked Systems Design and Implementation (NSDI)*, 2009, pp. 185–198.

[11] W. Galuba, K. Aberer, Z. Despotovic, and W. Kellerer, "ProtoPeer: a P2P toolkit bridging the gap between simulation and live deployement," in *Proc. of the 2nd International Conference on Simulation Tools and Techniques*, ser. Simutools. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009, pp. 60:1–60:9.

[12] C. Arad, J. Dowling, and S. Haridi, "Building and evaluating P2P systems using the Kompics component framework," in *IEEE 9th International Conference on Peer-to-Peer Computing (P2P)*, 2009, pp. 93–94.

[13] S. Naicken, B. Livingston, A. Basu, S. Rodhetbhai, I. Wakeman, and D. Chalmers, "The state of peer-to-peer simulators and simulations," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 2, pp. 95–98, Mar. 2007.

[14] R. Ramdhany, P. Grace, G. Coulson, and D. Hutchison, "MANETKit: Supporting the Dynamic Deployment and Reconfiguration of Ad-Hoc Routing Protocols," in *ACM/IFIP/USENIX Middleware*. Springer, 2009, pp. 1–20.

[15] R. van Renesse, K. Birman, and S. Maffeis, "Horus: A Flexible Group Communication System," *Communications of the ACM*, vol. 39, no. 4, pp. 76–83, April 1996.

[16] M. Hayden, "The Ensemble System," Department of Computer Science, Cornell University, Tech. Rep. TR98-1662, January 1998.

[17] "The JGroups Project," http://www.javagroups.org.

[18] H. Miranda, A. Pinto, and L. Rodrigues, "Appia: A Flexible Protocol Kernel Supporting Multiple Coordinated Channels," in *Proc. of the 21st IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, 2001.

[19] K. Ostrowski, K. Birman, and D. Dolev, "QuickSilver Scalable Multicast (QSM)," in *Proc. of the 7th IEEE International Symposium on Network Computing and Applications (NCA)*, July 2008.

[20] Q. Huang, Y. Vigfusson, K. Birman, and H. Li, "Quilt: a patchwork of multicast regions," in *Proceedings of the 4th ACM Int. Conference on Distributed Event-Based Systems*, 2010, pp. 184–195.

[21] P. Urban, X. Defago, and A. Schiper, "Neko: A single environment to simulate and prototype distributed algorithms," in *In Proc. of the 15th Intl Conf. on Information Networking (ICOIN)*, 2002, pp. 503–511.

[22] F. K. Hwang and D. S. Richards, "Steiner tree problems," *Networks*, vol. 22, no. 1, pp. 55–89, 1992.

[23] V. Goyal, "Multiple description coding: Compression meets the network," *Signal Processing Magazine, IEEE*, vol. 18, no. 5, pp. 74–93, 2001.

[24] X. Zhang, J. Liu, B. Li, and Y. Yum, "Coolstreaming/donet: a data-driven overlay network for peer-to-peer live media streaming," in *24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, vol. 3, 2005, pp. 2102–2111.

[25] S. Guha and P. Francis, "An end-middle-end approach to connection establishment," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, pp. 193–204, Aug. 2007.

[26] T. Chandra and S. Toueg, "Unreliable Failure Detectors for Asynchronous Systems," *Journal of the ACM*, vol. 43, no. 4, pp. 685–722, July 1996.

[27] 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.

[28] N. Magharei and R. Rejaie, "Prime: Peer-to-peer receiver-driven mesh-based streaming," in *Proc. of the 26th IEEE International Conference on Computer Communications (INFOCOM)*, 2007, pp. 1415–1423.

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

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

[31] K. Shen, "Structure management for scalable overlay service construction," in *Proc. of NSDI*, 2004, pp. 281–294.

[32] N. Magharei, R. Rejaie, and Y. Guo, "Mesh or multiple-tree: A comparative study of live p2p streaming approaches," in *Proc. of the 26th IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 2007, pp. 1424–1432.