OSSim: A Generic Simulation Framework for Overlay Streaming

Giang Nguyen, Mathias Fischer, Thorsten Strufe
Technische Universität Darmstadt
nguyen@cs.tu-darmstadt.de, mathias.fischer@cased.de, strufe@cs.tu-darmstadt.de

Keywords: Discrete event simulation, overlay streaming systems, OMNeT++

Abstract

Overlay streaming systems have recently been favored by the academic community as a viable approach for IPTV. Over the last years, a multitude of different overlay streaming approaches have been proposed. Most of them, however, have been evaluated individually. The lack of a common simulation framework makes it difficult to compare the properties of the different systems with each other. To bridge this gap, we introduce *OSSim*, a general-purpose simulation framework that allows the instantiation of different overlay streaming protocols. For this purpose, it provides a generic and modular structure, and several membership management and overlay streaming protocols as well. Our simulation results indicate that the framework is accurate and flexible to simulate different overlay streaming systems.

1. INTRODUCTION

With the increasing popularity of high-speed broadband Internet connections among consumers, the Internet Protocol Television (IPTV) has been becoming a more viable alternative to the traditional broadcast television. The classical client-server IPTV reveals a bottleneck at the server side. Its alternative, the overlay (or Peer-to-Peer) streaming, has emerged in recent years as a cost-efficient and scalable form of content distribution. Overlay streaming incorporates resources of participating peers to distribute video contents, and thus reduces the bandwidth demands at video servers.

Overlay streaming approaches can be classified according to how they construct and maintain the overlay topology, and how video packets are disseminated between peers in the overlay. Usually, mesh-pull, tree-push, and hybrid overlay streaming approaches are distinguished [7, 10]. The mesh-pull one has shown to use upload bandwidths more efficiently. Furthermore, the reactive nature makes mesh-pull systems more resilient to network dynamics. However, their overhead and latency are significantly high. In another sense, the tree-push (both single and multiple trees) approach has a good performance in terms of latency. Moreover, multi-tree systems are also theoretically resilient against attacks [4]. However, the main drawback of this approach is the inefficient use of peers' upload bandwidths. The hybrid overlay streaming, e.g., mTreebone [15], combines the efficient stream delivery of the tree-push approach with the high utilization of bandwidths and the increased resilience of the mesh-pull one.

For each of the three classes, a multitude of different systems and overlay streaming methods exist. However, most of them have been evaluated individually as they usually share no common code basis, such as in [4, 17], that would enable a meaningful comparison. Most available simulators are specialized, and mostly focus only on one class of overlay streaming systems. However, to fairly evaluate and to compare the different approaches against each other, a generic simulation framework is required. It should support the assessment of overlay streaming systems regarding their efficiency, robustness, and resilience against attacks.

Our main contribution in this paper is the introduction of the Overlay Streaming Simulator (*OSSim*), which is a general-purpose and generic simulation framework for overlay streaming. It is based on the OMNeT++ [14] discrete event simulator and the INET [13] framework. *OSSim* supports the simulation of mesh-pull, tree-push, and hybrid overlay streaming. Thus, it allows to compare different approaches with each other by providing a modular structure that can be easily extended. Currently, *OSSim* contains models of different membership management schemes such as SCAMP [6] and Newscast [8], and the overlay streaming protocols such as DONet [18] and Coolstreaming [9].

The rest of the paper is organized as follows: Section 2. presents requirements to a general purpose simulation framework for overlay streaming. Section 3. discusses the state of the art in this area and Section 4. describes the components of overlay streaming systems. Section 5. presents the design of our framework, while Section 6. shows the validation results. Finally, we conclude the paper in Section 7.

2. REQUIREMENTS

To evaluate the performance and resilience of overlay streaming systems in a fair and comparable way, we identify the need for a flexible simulation framework. We consequently provide a simulation framework. It comprises of abstract components that relate to the general functionality needed within all these systems. Additionally, the components are simple to adapt to specifications of various systems. The requirements for such a simulation framework can be summarized as follows:

Validity: Its modules are verified for correctness, and are potentially used by a large community of researchers.

Generic design: It enables to simulate a large set of overlay streaming systems. Therefore, its design should not be tightly coupled with a specific system. Common functionalities have to be modeled by generic modules, instead.

Flexibility: The framework contains interchangeable and configurable modules to harbor variety of protocols with the least additional modifications.

Extensibility: The module interfaces need to be well defined to derive new modules easily. This consequently facilitates the simulation of different overlay streaming protocols.

Minimum Overhead: Given an overlay streaming system, the framework should be able to simulate it with negligible overhead. It should also consume a reasonable amount of resources (e.g., CPU, RAM, and storage) when simulating a large number of nodes in a reasonable amount of time.

3. STATE OF THE ART

Various simulation frameworks for overlay streaming were developed. This section surveys the existing frameworks, and the most remarkable ones are classified into general purpose simulators and tailor-made ones.

General purpose Simulators: *OverSim* [2] is a sophisticated simulation framework. It focuses mainly on structured Peer-to-Peer networks, but not on overlay streaming. Even though it is possible to modify and adapt *OverSim* for that purpose, it would require significant modifications from the original design. This would also considerably enlarge the code base of *OverSim*, which would violate the minimal overhead requirement. *Denacast* [12] and *Layeredcast* [11] are tailored from *OverSim* to separately simulate individual classes of systems. They are developed independently, and therefore, integrating them might not be promising.

Tailor-made Simulators: The first simulator in this class is *p2pstrmsim* [16]. It was developed to study the performance of mesh-pull and hybrid push-pull systems. The simplified underlay network (i.e. without the TCP/IP protocol stack) allows to simulate tens of thousands of nodes within reasonable time. Other examples are *p2ptvsim* [3] and *SSSim* [1]. *p2ptvsim* is an event-driven simulator, and abstracts the underlay network by a module that is responsible for calculating transmission delays of video packets. The other simulator, *SSSim*, is a round-based because it assumes that peers are synchronized. Therefore, the simulator compromises on accuracy. Moreover, internal states of peers are available globally for access. This design decision limits the extensibility of the simulator since it potentially produces conflicts when the code basis grows. The tailored designs of this class of simulators, however, hinder the possibility to extend them towards a general purpose framework.

To sum up, there is no generic and extensible simulation framework for overlay streaming so far that allows to compare different classes of overlay streaming systems in a fair manner. Nonetheless, there is still a need for selecting the best system under the same condition of workload and network dynamics. The lack of such framework does not allow us to provide a meaningful conclusion. For this reason, we are going to provide our own simulation framework in Section 5.

The task, however, is not straight forward. On the one hand, the framework has to harbor various classes of overlay streaming systems. On the other hand, it must be generic enough to produce common functionality modules to avoid implementing them again in different systems. In the next section, we analyze classes of overlay streaming systems. Their common components are extracted, and specific ones are also identified.

4. OVERLAY STREAMING COMPONENTS

Abstractedly, an overlay streaming system consists of one or several *Channel Servers*, one or several *Streaming Servers*, and a multitude of *Peers* [5, 7]. A *channel server* maintains a list of channels. Besides, it can also provide a bootstrapping service. Hence, it additionally holds a list of active peers (and streaming servers) per channel. The list is used to bootstrap joining peers with knowledge about one or several other peers in the respective channel. For each channel, the *streaming server* distributes a continuous stream of video packets. The stream might be further divided into multiple sub-streams or stripes in many systems. Moreover, the server participates in the streaming overlay as a normal peer, except that it does not request video packets. A *peer* runs an application that follows a specific overlay streaming protocol. Through the application, a peer establishes connections with others. Hence, they form an overlay network on top of physical networks. Moreover, peers also exchange video packets directly with each other.

Overlay streaming systems can be classified using the following two criteria: *a*) The way peers are organized to maintain the overlay topology, which is either mesh or tree, and *b*) The way the video content is disseminated among peers, which is either pull or push. Main approaches are mesh-pull, tree-push, and hybrid push-pull. However, independent from the specific classification, each overlay streaming system consists of the following three basic layers:

- **Discovery Service:** It provides peers with information about resources (streams or stripes). Moreover, it equips peers with knowledge about other peers in the overlay.
- **Topology Management:** It establishes connection between peers. Moreover, it also attempts to balance the load in the streaming overlay and provides measures to increase the efficiency and robustness of the overlay.
- **Media Management:** This includes the local storage of video packets in a buffer and the forwarding of packets

from the buffer. At peers the layer contains functionalities to playback a stream. Whereas at video sources, it is also responsible for video packet generation.

In the remainder of this section, we discuss the necessary components for all the three different classes of overlay streaming along the afore-mentioned generic layers. This discussion reveals the key components that need to be reflected to support the simulation of the classes within one unifying simulation framework. This serves as the input for the design of our simulation framework in Section 5.

4.1. Mesh-pull systems

In mesh-pull systems, peers are loosely coupled in a mesh overlay and need to request video packets from other peers well in advance of the playback. A partnership management mechanism ensures that the established mesh is connected and that each peer has a sufficient number of neighbors in the overlay. Each peer periodically sends Buffer Map (BM) packets to its neighbors. A BM contains the information about the sequence of video packets (or so-called *Chunks*) that are available at that peer. In addition, the mechanism selects the neighbors from whom the packets are actually pulled. Packet pulling is planned via a load balancing (or so-called *chunk scheduling*) algorithm that considers the available packets at neighbors. It can be reduced to the parallel machine scheduling problem and thus is NP-hard [18]. For this reason, heuristics (e.g., the Rarest-First in DONet [18]) need to be used. Moreover, the partnership management ensures a replacement of unreliable peers that deliver insufficient QoS, e.g., high delay or high packet loss. The reactive nature of mesh-pull systems renders them resilient to network dynamics. However, the overhead and latency are significant.

Following the afore-mentioned three generic layers of an overlay streaming application, the specific building blocks of a peer in a mesh-pull system are as follows:

Discovery Service

Membership Management is implemented at participating peers and the video source as well. It supplies peers with a partial view of active peers. To realize this functionality, there are two approaches: centralized or distributed. In a centralized approach, a tracker is used, and peers periodically report their status to the tracker. However, this approach is not scalable, and presents a single-point-of-failure. The distributed approach recently leverages gossiping protocols such as SCAMP [6] and Newscast [8].

Topology Management

Partnership management is in charge of establishing and maintaining partnership connections with other peers. For this purpose it sends partnership requests, and answers requests from other peers. Besides, it exchanges BMs with other partners.

Load Balancing is a key component of mesh-pull systems. At requesting side, it periodically monitors the local buffer to identify missing packets, and looks for their availability in the BMs of its partners, and requests them if necessary. While at the responding side, it sends back the requested packets.

Media Management

Packet Generator is located only at streaming servers. It creates and sends out video packets.

Video Buffer stores video packets at peers.

Player sequentially obtains packets from the *Video Buffer* for playback.

4.2. Tree-push Systems

Tree-push streaming protocols usually establish one or several preferably inner-node disjoint trees spanning all peers respectively. The stream is usually divided into several *stripes*, and each stripe is distributed via one spanning tree. When only a single tree is used, the system is not robust to peer churn. In deed, failing peers cause disruptions in the distribution for all other peers dependent on them, until the tree is restored. Using stripes and inner-node disjoint spanning trees, peer failures can be tolerated to a certain extent, e.g., as in the construction given in [4]. The main components of a tree-push system are as follows:

Discovery Service

Bootstrapping is responsible to providing nodes with one or several active peers in the desired stripes.

Topology Management

Topology Optimization is responsible to optimizing the neighborhood of participating peers and the overall topology.

Media Management

The media management components of tree-push systems re-use the *packet generator*, the *video buffer*, and the *player* from mesh-pull systems. The components differ only in small implementation details.

4.3. Hybrid push-pull systems

Hybrid overlay streaming systems [9, 17] combine the best of both worlds, namely the increased resilience of pull-based overlay streaming with the more efficient content distribution of push-based one. For this purpose, a mesh-based content distribution is combined with an additional push-based overlay. Similar to the stripe concept in tree-push approach, the stream is divided into sub-streams. When a peer joins the mesh, it starts to request BMs, and video packets from its partners (pulling phase). After the successful reception of the first few packets of a sub-stream, the pull-based connections are “frozen”, and all subsequent packets are pushed from the respective partners (pushing phase). If packets are lost, e.g., because of transmission errors or network congestion, peers can request missing packets from other partners. Consequently, hybrid overlay streaming systems make use of all

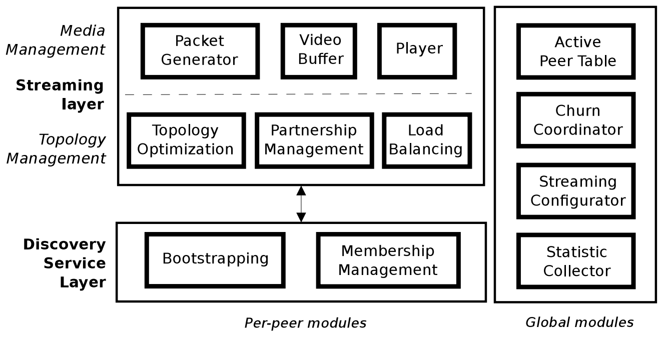

Figure 1. Architecture of the OSSim framework

components identified in Sections 4.1. and 4.2. In hybrid systems the components only differ slightly in implementation details.

5. DESIGN OF THE OSSIM FRAMEWORK

In this section we present the overall architecture of the framework, and then introduce per-node components, including the dispatcher and message hierarchy, and other corresponding layers.

5.1. OSSim Architecture

Taking into account the modeling of the various overlay streaming systems in the previous section, we produce an architecture as illustrated in Figure 1. Functionality modules of a peer are classified into two separate layers. The *Discovery Service* layer is responsible for bootstrapping and membership management. The *Streaming* layer consists of two sub-layers: The *Topology Management* harbors functionalities, such as the topology optimization for tree-push systems and partnership management for mesh-pull systems. In the *Media Management* sub-layer, video packets are generated, stored, and forwarded. Besides, the framework also consists of global modules such as the *Active Peer Table* to keep track of the currently active peers, the *Churn Coordinator* to control the join and leave operations of peers, *Streaming Configurator* and *Statistic Collector*. This architecture implies a *dispatcher* which connects modules of different layers.

Our simulation framework is based on OMNeT++. By doing so, we leverage available and stably built modules that are widely used and tested by the community. With this design choice, it is possible to keep our simulation framework unchanged while still being able to include other underlying network models such as cross traffic and mobility.

Figure 2(a) illustrates a node which follows the ISO/OSI model and bases solely on the UDP Transport layer. We simplify the StandardHost in the INET framework by removing TCP and TCPApp modules, and replace the UDPApp module by our own application module. We then model the streaming application at a peer node as illustrated in Figure 2(b). As can be seen from the figure, a generic peer application consists of a *dispatcher* module for message classification and modules implementing the functionalities of different layers. Specifically, the *memMan* module represents the membership management in the *Discovery Service* layer, while the other modules belong to the *Streaming* layer. The *partnerMan* module implements the *Partnership Management* functionality. The *forwarder* acts as a communication interface of the *videoBuffer* module. The model of a streaming server follows a similar design, except that a packet generator replaces the player module. Other differences in the behavior of the video server remain in small implementation details only.

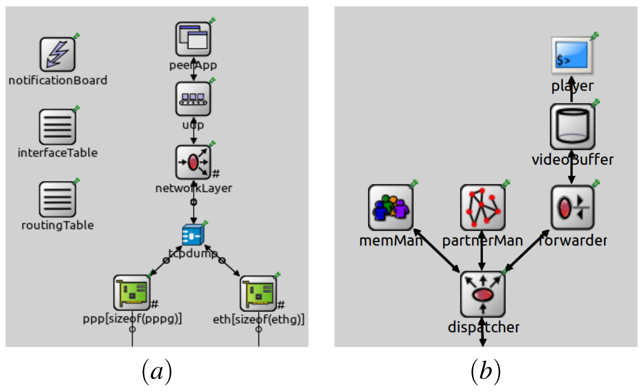

Figure 2. Model of (a) a peer node and (b) a peer application module in OMNeT++

5.2. Helper Modules

With the layered design in mind, we introduce a *Dispatcher* module and a *Message Hierarchy*. Moreover, we produce a set of global modules, whose tasks range from setting up and coordinating the simulation to collecting simulation results.

Dispatcher: This module acts as a gateway between the lower Transport layer which is UDP in our framework and functional modules belonging to our layers as illustrated in Figure 2. Even though there would be overhead due to the operation of the *dispatcher*, this design makes the model more flexible when it has to adapt to, or include other Transport layers such as TCP.

Messages Hierarchy: To correctly and efficiently forward messages to different modules, we introduce a message hierarchy as illustrated in Figure 3. Messages belonging to modules of the same layer or sub-layer in Figure 1 are grouped together using the same group identifier. When a message of a particular group is forwarded to the respective layer, it is up to the layer to either process the message directly or forward it to a subsequent module of the same layer.

Churn Coordinator: This module calculates and provides join and leave times of peers up on request. To scramble the

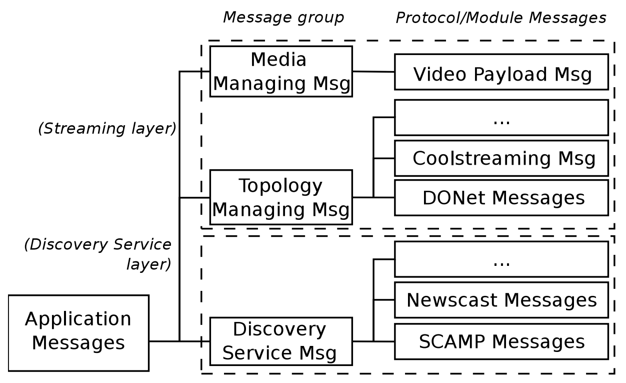

Figure 3. Hierarchical structure of Application messages

order of peers joining the network, we apply a two-phase arrival assignment process. In the first phase, peers are assigned at random order. In the second phase, peers actually query the *Churn Coordinator* to get their arrival and departure times. It is possible to extend this module to harbor additional actions of peer such as switching between channels. The selective API of this module includes:

getJoinTime(): To get joining time

getDepartureTime(): To get departure time

Statistics Collector: This module collects data, and calculates system-wide metrics. Depending on the protocol of interest, suitable APIs are implemented accordingly.

Streaming Configurator: This module calculates common parameters for all other the modules. It ensure the consistency of parameter values. The selective API of this module is:

getStreamingBitRate(): returns the streaming bit-rate

getVideoPacketSize(): returns the size of a video packet

getBufferSize(): returns the size of the Video Buffer

5.3. Discovery Service Layer

The main purpose of this layer is to provide the peer sampling service for the upper streaming layer. By maintaining a partial and frequently refreshed view on the active peers, it should provide a random list of active peers on demand. In tree-push systems, it bootstraps a newly joining peer to subscribe to the overlay. In mesh-pull systems, it also assists peers to find new partners besides the bootstrapping functionality. The central operation of the this layer has the following API:

getARandPeer(): returns a random peer address from the set of active peers in the system

Deployed systems, recently, realize this service by gossiping protocols since they are lightweight, scalable, and resilient against network dynamics. In event-driven simulations, however, those protocols slow down the execution time significantly because gossiping peers exchange many messages with each other. Therefore, we introduce a *Dummy* module to reduce simulation time, especially in the development phase. While the *Dummy* module resembles the same API, it keeps and updates a centralized data structure of active peers. Even though this is unrealistic, it speeds up of the simulation significantly.

5.4. Topology Management Layer

Partnership Management: This is the core of mesh-pull and hybrid systems. It is in charge of joining the streaming overlay, handling external messages and timers, sending periodic messages (e.g., Buffer Maps), and activating other modules such as *Load Balancing* and *Player*.

Load Balancing: The requesting part of this functionality is activated periodically. It depends on specific algorithms (e.g., *Random* or *Rarest-First*) applied in the overlay streaming protocol. The direct outcome of this functionality is a set of messages to request video packets. On the other hand, the responding part simply waits for request messages from other peers and replies accordingly.

Topology Optimizer: This module is the core of a tree-push system. It establishes and maintains the parent-children relations with other peers. It also sends *Heartbeat* messages to parent nodes to assist the failure recovery process. One important functionality of this module is to switch parents to improve the QoS of individual peers and to optimize the overall topology.

5.5. Streaming Management Layer

This layer consists of *Video Buffer*, *Forwarder*, *Packet Generator*, and *Player* modules.

Video Buffer: It stores received video packets. This implies a sliding window whose size equals the size of the buffer (in video packets). Since the sliding operation is executed quite often, the *Video Buffer* should be carefully designed and implemented. We realize it by a cyclic array containing video packets. Deciding where to insert video packets into the buffer will be done by modulo operations on packet identifiers. This avoids sliding operations, and consequently improves the overall performance of the framework. The *Video Buffer* module provides the following services:

isInBuffer(): checks whether a packet is in the *Video Buffer*

getPercentFill(): returns the percentage of available packets in the *Video Buffer*

Forwarder: It is the gateway of the *Media Management* sub-layer. The *Forwarder* receives video packets and stores them in the *Video Buffer*. It also picks up video packets from the *Video Buffer*, and sends them to other partners upon request. The *Forwarder* provides the following main service:

sendVideoPacket(): sends a video packet specified by its sequence number to a remote peer

Packet Generator: This module is active at streaming servers only. It periodically generates video packets, and then inserts them to the *Video Buffer* of a streaming server.

Player: This module has a playback pointer that specifies the currently required video packet. Based on the pointer, the Player periodically obtains from the *Video Buffer* the video packet. The pointer moves forward consistently with the playback rate. To maintain a smooth playback in live streaming, each packet should be available before a strict deadline, or it must be ignored by the player, otherwise. We implemented two types of players. They correspond to two playback strategies: *Simple_Skip* and *Skip_Stall*. The *Simple_Skip* player simply moves its pointer forward continuously and skips all unavailable packets. This player is simple and unrealistic, but it provides a plain estimation of the streaming protocol. The second player, *Skip_Stall* whose Finite State Machine is given in Figure 4, applies a more sophisticated strategy. It allows certain thresholds of skipped packets. Besides, the player can even be stalled to wait until late packets arrive, or a rebuffering operation completes. The player's selective API is following:

activate(): activates the player
stopPlayer(): stops the player
getPlaybackPoint(): returns the current playback point
getPlayerState(): returns the current state of the player

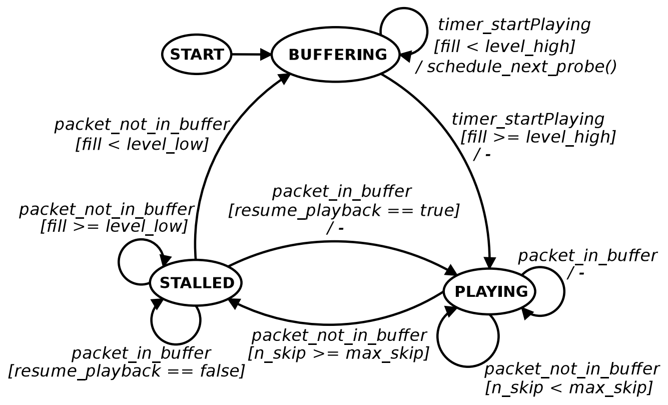

Figure 4. The Finite State Machine of the *Skip_Stall* player

5.6. Summary of the Design

To sum up, we already presented the generic modules and their selective APIs. We instantiated those modules for specific protocols. Key instantiations and their parameters are summarized in Table 1. Compared to the requirements presented in Section 2, our generic framework is flexible to harbor different overlay streaming and membership management protocols. Moreover, the layered and modular design also eases the extension of the framework using configurable components.

Table 1. Selective instantiations and their parameters

Component	Instantiation	Parameters
Churn Coordinator	*Uniform*	lower & upper bounds
	Exponential	joining & leaving rates
	Pareto	scale and shape
Membership Management	*SCAMP*	duplication factor
	Newscast	cache size
	Dummy	-
Topology Management	*DONet*	-
	Coolstreaming	-
Load Balancing	*Random*	-
	Rarest First	-
Player	*Simple_Skip*	percent fill to start
	Skip_Stall	thresholds of skipped packets, buffer fill, etc.

6. VALIDATION OF OSSIM

In this section, we describe different experiments for the validation of our simulation model. Due to space constraints, we selectively present the validations of a single representative protocol in each layer individually: the membership management protocol *SCAMP* in the discovery service layer and the mesh-pull protocol *DONet* in the streaming layer. Moreover, we conducted experiments with *Coolstreaming*, a hybrid push-pull protocol, running on top of the *Newscast* gossiping protocol to test the interaction between the layers as a whole. We based our validation on published results because none of the above protocols is available as an open-source or publicly available software.

Validating SCAMP To validate the *SCAMP* protocol alone, we used the average out-degree metric (or so-called the *partial-view-size*). In this experiment, 50000 peers joined the system one after another and stayed in the system until the end of the simulation. Their inter-arrival times followed an exponential distribution. The duplication factor (c) was set to zero. The total simulation duration was 1000 simulated seconds. The out-degrees of peers were recorded at the end of the simulation.

Figure 5 plots the histogram of the out-degrees. The shape of the histogram resembles closely the published results in [6]. The calculated average out-degree is 11.1 which is also close to the theoretical value of 10.8 (or $log(50000)$). The results reveal that the implemented protocol is correct, and our framework can simulate a significant system size.

Validating *DONet* We conducted an experiment with the *DONet* implementation in *OSSim*. In this experiment, the *Dummy* module was used to provide the discovery service, which significantly reduced the simulation time. To quantify the results, we selected the *Continuity Index (CI)* metric that

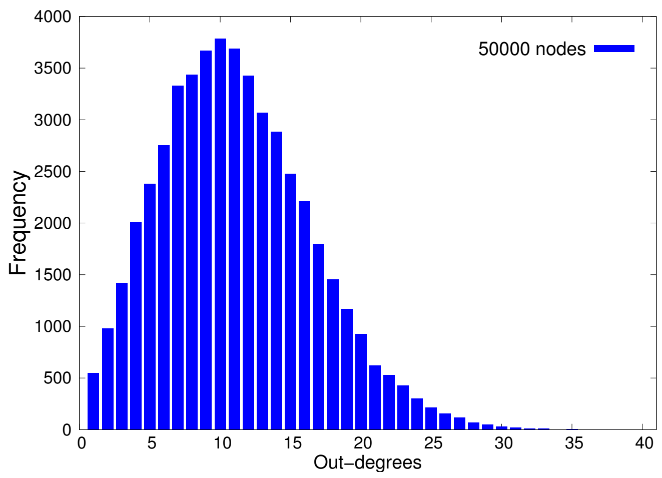

Figure 5. Validation of the SCAMP gossiping protocol: Histogram of the out-degree of peers

Table 2. Common parameters and their values in the experiment with overlay streaming protocols

Parameters	Values
Streaming Server's upload bandwidth	2 Mbps
Streaming Server's number of partners	10
Video packet size	1250 B
Video Buffer capacity	60 s

has commonly been used in the literature. The CI is the ratio of the number of video packets arriving before their deadlines (N_{hit}) and the total number of required video packets (N_{total}). The *Simple_Skip* player was used at peers to record the two statistics and to periodically send them to the global statistic module, which calculated the metric for the whole system. The parameter setting is shown in Table 2.

In this experiment, the streaming server first joined the streaming overlay. Then, 1000 homogeneous peers joined the overlay, and did not leave the overlay until the end of the simulation. Their inter-arrival times followed a uniform distribution. The upload bandwidth of peers was set to 1200 kbps. The *number-of-partners* parameter of peers was fixed at five. The streaming rate was varied from 100 kbps to 500 kbps. Each simulation stopped after reaching 1000 simulated seconds, even though the results consistently reached their steady-states after around 500 simulated seconds. We repeated each setting ten times with different seeds of the random number generator. Figure 6 plots the CI metric with respect to different streaming rates.

As can be seen in Figure 6, the CI decreases gradually when the streaming rate increases from 100 kbps to 500 kbps. The CI also reveals the same trend as in the published results in [18] whose experiments were conducted in a practical deployment. Both results are relatively close to each other.

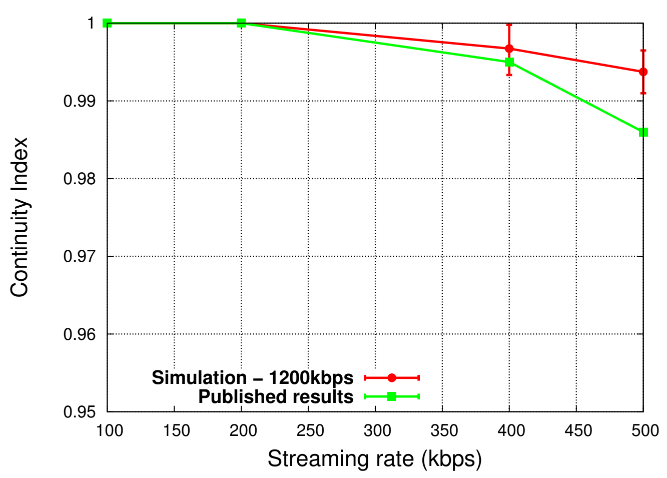

Figure 6. Validation of the DONet streaming protocol: The Continuity Index versus the streaming rate

Validating *Coolstreaming* In this experiment, we validated both Coolstreaming and Newscast. The *Newscast* gossiping protocol was used to manage the membership for the Coolstreaming overlay streaming protocol. The cache size of *Newscast* was set to 20 entries. The streaming rate was fixed at 400 kbps. The number of sub-streams was set to ten. The network included a source node and 1000 homogeneous peer nodes. Peers had upload bandwidths of 500 kbps. They joined the system and did not leave until the end of the simulation. Their inter-arrival times followed an exponential distribution. We repeated each setting ten times with different seeds of the random number generator. Simulations were run for 1000 simulated seconds.

The CI metric was calculated for the whole network. Figure 6. plots the CI versus the number of partners (from 10 to 20). As can be seen, the CI increases linearly when the number of partners increases even to a large value. Our results resemble quite well the published simulation results [9]. The slightly higher CI in our simulation can be a result of the smaller system size in our setting which might be more sensitive to the same increase in the number of partners.

7. CONCLUSION

In this paper, we present *OSSim*, a simulation framework in OMNeT++ to simulate overlay streaming systems. *OSSim* is characterized by a modular and generic layered design that we derived from an analysis of the three major overlay streaming classes. This design eases the implementation of additional overlay streaming protocols. Our simulation results indicate the validity and accuracy of our framework for SCAMP, DONet, and Coolstreaming. The source code of OSSim is publicly available at https://github.com/ntrgiang/ossim.

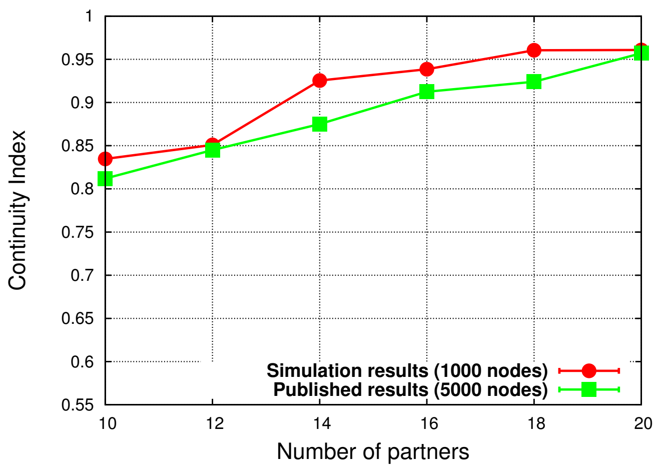

Figure 7. Validation of the Coolstreaming protocol: The Continuity Index versus the number of partners

Future work will comprise the implementation of more protocols in the streaming layer, such as tree-push systems, and comparison studies of efficiency and resilience properties of different overlay streaming classes. Moreover, we propose to extend our simulation framework by multiple channels.

8. ACKNOWLEDGMENT

This work has been done as part of the research project supported by the German Academic Exchange Service (DAAD), Grant number A/09/97565.

REFERENCES

[1] L. Abeni, C. Kiraly, and R. Lo Cigno. Sssim: a simple and scalable simulator for p2p streaming systems. In *IEEE 14th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks*, pages 1–6, 2009.

[2] I. Baumgart, B. Heep, and S. Krause. Oversim: A flexible overlay network simulation framework. In *IEEE Global Internet Symposium, 2007*, pages 79 –84, 2007.

[3] A. Biernacki. Simulation of p2p tv system using omnet++. In *EuroView*, 2010.

[4] M. Brinkmeier, G. Schafer, and T. Strufe. Optimally dos resistant p2p topologies for live multimedia streaming. *IEEE Transactions on Parallel and Distributed Systems*, 20(6):831 –844, june 2009.

[5] Y. Feng and B. Li. Peer-assisted media streaming: A holistic review. In *Intelligent Multimedia Communication: Techniques and Applications*, volume 280, pages 317–340. Springer, 2010.

[6] A. Ganesh, A. M. Kermarrec, and L. Massoulie. Peer-to-peer membership management for gossip-based protocols. *IEEE Trans. on Comp.*, 52(2):139–149, 2003.

[7] X. Hei, Y. Liu, and K. Ross. Iptv over p2p streaming networks: the mesh-pull approach. *IEEE Communications Magazine*, 46(2):86–92, Feb. 2008.

[8] M. Jelasity, W. Kowalczyk, and M. van Steen. Newscast computing. Technical Report IR-CS-006, Vrije Universiteit Amsterdam, Department of Computer Science, Amsterdam, The Neitherlands, Nov. 2003.

[9] B. Li, S. Xie, Y. Qu, G. Keung, C. Lin, J. Liu, and X. Zhang. Inside the new coolstreaming: Principles, measurements and performance implications. In *IEEE INFOCOM 2008.*, pages 1031 –1039, april 2008.

[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] M. Moshref, R. Motamedi, H. Rabiee, and M. Khansari. Layeredcast - a hybrid peer-to-peer live layered video streaming protocol. In *5th International Symposium on Telecommunications (IST)*, pages 663 –668, dec. 2010.

[12] Y. Seyyedi. Denacast: A p2p video streaming simulator. [Online]. Available: http://denacast.org/, 2010.

[13] A. Varga. The inet framework. [Online]. Available: http://inet.omnetpp.org, 2012.

[14] A. Varga. Omnet++ network simulation framework. [Online]. Available: http://www.omnetpp.org, 2013.

[15] F. Wang, Y. Xiong, and J. Liu. mtreebone: A collaborative tree-mesh overlay network for multicast video streaming. *Parallel and Distributed Systems, IEEE Transactions on*, 21(3):379–392, March 2010.

[16] M. Zhang. p2pstrmsim: Peer-to-peer streaming simulator. [Online]. Available: http://media.cs.tsinghua.edu.cn/∼zhangm, 2009.

[17] M. Zhang, Q. Zhang, L. Sun, and S. Yang. Understanding the power of pull-based streaming protocol: Can we do better? *IEEE Journal on Selected Areas in Communications*, 25:1678–1694, 2007.

[18] X. Zhang, J. Liu, B. Li, and Y.-S. Yum. Coolstreaming/donet: a data-driven overlay network for peer-to-peer live media streaming. In *IEEE INFOCOM 2005*, volume 3, pages 2102–2111, march 2005.