Crystal: An Emulation Framework for Practical Peer-to-Peer Multimedia Streaming Systems

Mea Wang, Hassan Shojania, Baochun Li
Department of Electrical and Computer Engineering
University of Toronto
$\{$*mea, hassan, bli*$\}$*@eecg.toronto.edu*

Abstract

To rapidly evolve new designs of peer-to-peer (P2P) multimedia streaming systems, it is highly desirable to test and troubleshoot them in a controlled and repeatable experimental environment in a local cluster of servers, as it is risky to integrate untested protocols in live production and mission-critical peer-to-peer sessions, such as live P2P streaming. Though it is possible to construct such controlled experiments with virtual machine monitors, there are a number of challenges and roadblocks: (1) The deployment of such resource-hungry virtual machine environments are complicated and time-consuming for researchers without prior systems expertise; (2) The system designer needs to implement many basic streaming elements, such as playback buffers and message switches. In this paper, we seek to address these challenges by introducing Crystal*, an emulation framework for practical P2P multimedia streaming systems, which provides support for developing, testing, and troubleshooting new streaming system designs in a controlled server cluster environment. It is our imperative design objective that* Crystal *offers ease of use, rapid experimental turnaround, and the capability of emulating realistic P2P environments.*

Keywords: Peer-to-peer multimedia streaming, emulation framework, development toolkit.

I. Introduction

Peer-to-peer (P2P) multimedia streaming has received extensive attention in recent research. The essence of the P2P paradigm is a shift of algorithmic intelligence and bandwidth burden from dedicated servers to the end systems (*i.e.,* peers) at the edge of the Internet. P2P streaming systems have also been shown to offer higher performance, better scalability, as well as superb resilience to peer failures and departures. A large number of new P2P streaming systems has been designed, simulated, and implemented, from academia (*e.g.,* [1]) and industry (*e.g.,* PPLive) alike.

As P2P streaming systems are deployed in real-world multimedia streaming applications, it has become increasingly important to test, troubleshoot, and evaluate a new P2P streaming system design under practical network settings, before large-scale deployment. However, the complexity of implementing and testing a new P2P streaming system is non-trivial, and becomes the main obstacle that drives most academic researchers away from assessing new protocols in realistic networks. A testbed implemented to evaluate a new P2P streaming system design should involve a large number of bandwidth-limited peers behind home broadband connections, with arrival and departure dynamics, and *actual traffic* being relayed and transmitted. Any P2P streaming system needs to establish and manage TCP connections or UDP flows among the peers, use dedicated tracking servers to bootstrap new participating peers, and maintain a playback buffer that consists of segments to be played in the immediate future. Peers in a streaming system may also need to discover one another, and to exchange buffer availability information. Most existing academic studies resort to simulation studies, which do not appropriately reflect the complexity of real-world streaming systems.

Ideally, the best route to evaluate new protocols is to actually implement and deploy them across the Internet, on real peers with home broadband connections. Such an approach, while realistic, may not offer sufficient scientific evidence from which conclusions can be drawn. *First*, due to the highly dynamic nature of peers in P2P networks, experimental results in this setting may not be analyzed and diagnosed design parameters are tuned. They are also not repeatable, and as such difficult for other researchers to independently reproduce and verify the results. *Second*, without a dedicated commercial launch, it is hard to include a large number of peers with DSL/cable Internet connections, since home peers are more dynamic than institutional users. *Third*, it may be difficult to collect vital statistics and logistics measured at each peer, as central logging servers are usually not scalable to a large number of peers in the session. *Finally*, CPU and bandwidth — the most important resources that lead to the advantage of the

P2P architecture — are heterogeneous and highly dynamic, as the availability of CPU cycles and bandwidth is subject to the fluctuating load of concurrent tasks on the same host. Some existing evaluations of P2P protocols made use of experimental testbeds such as PlanetLab [2] and Netbed [3], which are examples of testing implementations that suffer from lack of home users, lack of peer dynamics, lack of repeatability, and lack of scalability to a large number of peers. Development may also be complex and time-consuming, as there does not exist a framework to support basic elements common to P2P streaming systems.

In this paper, we present the design and implementation of *Crystal*, an emulation framework to support the implementation and evaluation of practical P2P multimedia streaming systems in a cluster of high-performance servers. Implemented from scratch in C++, *Crystal* provides a flexible framework to rapidly develop, test, tune, and troubleshoot new designs of P2P streaming systems in a server cluster. We design *Crystal* to offer ease of use, rapid experimental turnaround, scalability with respect to the number of emulated peers, and the capability of emulating realistic P2P environments. It includes basic elements common to P2P streaming systems, such as bootstrapping protocols, efficient message forwarding mechanisms, timed and periodic event schedulers, TCP and UDP network socket programming, multi-threaded programming, exception handling of failures and disconnections, as well as facilities to control, troubleshoot, and measure the performance metrics. *Crystal* is to be released as an open-source project to the P2P streaming research community.

The remainder of this paper is organized as follows. We discuss Crystal in light of related work (Sec. II). In the main body of the paper, we first present the architecture and design of Crystal in Sec. III, and then demonstrate the scalability and effectiveness of Crystal with case studies of implementing practical P2P streaming systems in Sec. IV. We conclude the paper in Sec. V.

II. Related Work

Before designing Crystal, we have been developing and evaluating our P2P protocols using iOverlay [4], which was designed to simplify protocol development and to facilitate PlanetLab deployment. Over time, we discovered several noteworthy challenges with iOverlay. First, due to its internal design, iOverlay does not scale well when implementing computational expensive P2P streaming protocols, such as those involving network coding and fountain codes. Second, the results from PlanetLab are not repeatable due to usually overloaded PlanetLab slices. Third, iOverlay was not able to emulate peer dynamics, which is required to evaluate the resilience of our new protocols. Finally, iOverlay does not provide basic elements for P2P streaming. With the new design in Crystal, we have shifted our experimental platform to server clusters, and have completely re-implemented the new design from scratch, with better scalability in the same cluster node, and fully automated deployment in server clusters.

There exist previous work on using virtual machines (such as VMWare, Xen, or User-Mode Linux). The main objective was to support the deployment of full-fledged applications over a virtual network (*e.g.*, [5]), or in emulation testbeds and environments to test network protocols in a virtualized and sandboxed environment (*e.g.*, Netbed [3] and ModelNet [6]). In particular, ModelNet [6] has introduced a set of ModelNet core nodes that serve as virtualized kernel-level packet switches with emulated bandwidth, latency and loss rates. Crystal has similar objectives, but is designed to be simpler to deploy, more scalable on a single physical cluster node, and much easier to develop with. Crystal supports emulating hundreds of peers on a single physical cluster node, and is implemented entirely in user space beyond the abstraction of sockets. To deploy and test a new P2P streaming system using Crystal, a researcher does not need to configure the system with multiple virtual machines, or to patch and recompile the kernel. Crystal is cross-platform as well, readily deployable on not only major UNIX variants, but also Microsoft Windows (under Cygwin).

Mace [7], and its predecessor called MACEDON [8], featured new domain-specific languages to describe the behavior of an overlay protocol, from which actual code can be generated using a code generator. As a result, they allow protocol designers to focus their attention on the semantics of the protocol itself, and less on tedious implementation details. Crystal, however, is based on a drastically different design philosophy. Mace and MACEDON attempt to minimize the lines of code to be developed by the protocol developer, by using new language extensions to specify the characteristics of a specific category of P2P protocols, including DHT search and application-layer multicast. In contrast, Crystal represents a more traditional framework design that seeks to maximize the freedom and flexibility of designing and implementing new P2P streaming protocols with C++, including computationally intensive tasks (such as network coding). As such, Crystal reflects a different spot in the tradeoff between having the fewest lines of code and allowing maximum flexibility.

III. Crystal: Architecture and Design

Crystal features the following highlights. First, Crystal provides a set of common elements required in any P2P streaming system, including multi-threading, message switching, timed and periodic event scheduling, network socket programming, and exception handling. These elements are organized into three layers: network, engine, and algorithm. Second, Crystal is custom-tailored for server clusters. As shown in Fig. 1, each instance of a Crystal stack corresponds to an emulated peer. A server can easily accommodate from one to hundreds of emulated peers, depending on available physical resources such as CPU and memory. Finally, any two emulated peers can establish TCP or UDP connections. Crystal addresses the lack of reality in simulation and the lack of controllability in real-world deployments, and is capable to *emulate* any peer upload and download capacities, end-to-end delays, as well as peer arrivals and departures. This framework crystallizes the past two years of our work towards implementing a framework for P2P network emulation, in 10279 LOC (lines of code including comments and excluding scripts).

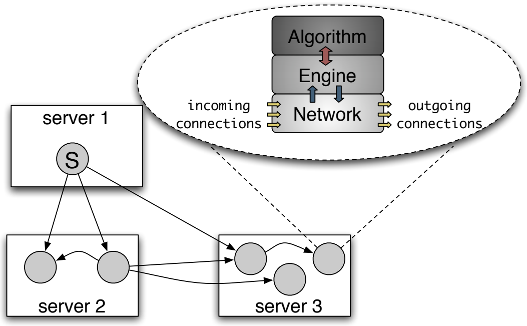

Fig. 1. The Crystal architecture.

A. The Core of Crystal

The design of Crystal is based on our previous work on *iOverlay* [4], a lightweight middleware framework for developing overlay applications over the Internet. The design of iOverlay employed a "thread-per-connection" concurrency model, using blocking socket operations and forking a new thread for each TCP connection. For each emulated peer, the number of threads is the same as the number of active connections, leading to excessive overhead of thread context switching. Though this model may be appropriate when building overlay applications on actual end hosts, it is not the most scalable way to build emulated peers in a server cluster with limited CPU resources. In contrast, Crystal employs two threads for each peer, the *network thread* and the *engine thread*. With fewer threads, the threading overhead is significantly decreased, which makes Crystal more scalable than alternative VMM-based solutions when emulating a large number of peers on a single server.

The network thread, referred to as *network* in Fig. 1, is responsible for handling queues of new incoming and outgoing messages, emulating bandwidth and delay, monitoring socket status, as well as detecting arrivals and departures of peers. The engine thread, including both *engine* and *algorithm* in Fig. 1, is mainly responsible for processing incoming messages, performing protocol-specific logic, emulating peer arrivals and departures, as well as handling all periodic or timed events.

The engine and the network naturally form a consumer-producer relationship with respect to messages. When an upstream peer sends a message to peer p, the network first detects the incoming traffic and receives the message into the corresponding queue. The engine then takes the message and passes it to the appropriate message handler, implemented in either the engine or the algorithm. In this case, the network is the producer, and the engine is the consumer. To send a message from peer p, the message is usually created by the algorithm, and is then queued into the network via the engine. The engine does not do any processing in this case, except looking up for the appropriate queue in the network. The network then sends messages to the downstream peer. Now the network becomes the consumer, whereas the engine is the producer.

Overall, the network thread is dedicated to manage network-level events, while the engine thread processes and produces messages. This design leads to very fast responses to socket events, and subsequently makes emulating high-throughput peers possible.

The Network

The network thread provides low-level network I/O services in Crystal. It handles basic sockets-level tasks related to new incoming connections, exception handling related to broken connections, as well as the actual communication (send and receive operations) for all active connections. The network thread supports both connection-oriented stream sockets (TCP) and connectionless datagram sockets (UDP). As implied in Fig. 2, each TCP connection from an upstream peer or to a downstream peer is associated with a queue, implemented as a circular buffer of messages. UDP traffic has its own dedicated incoming and outgoing queues: we maintain one pair for UDP data messages, and another pair for UDP control messages. Similar to the default *reactor* in the Twisted event-driven networking engine (written in Python) and in some CORBA implementations (such as ORBacus written in C++), all incoming and outgoing network traffic are monitored by a single `select()` call with a specific timeout value. The `select()` call releases when one or more sockets from the active list become ready for I/O operations, or when the prescribed timeout expires. The network thread then proceeds to process these sockets. All TCP I/O operations are non-blocking, and the send or receive operation of a message in a queue does not necessarily finish in one iteration of the network loop.

The Engine

Fig. 3 shows that the engine consists of four parts: (1) the message handler for processing incoming messages; (2) the event timer for scheduling timed or periodic events; (3) the logger for facilitating a thread-safe logging sys-

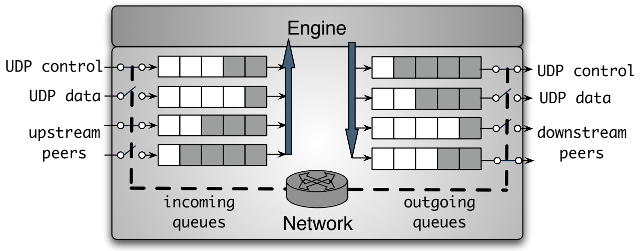

Fig. 2. The design of the network thread.

tem; and (4) the API for algorithm implementation. The engine's main responsibility is to retrieve messages from the incoming queues of the network, and to route them to the appropriate message handlers. Some messages are handled by the engine itself, and the remaining messages are forwarded to the algorithm.

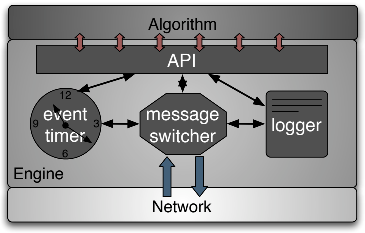

Fig. 3. The design of the engine thread.

The engine provides support for timed and periodic events by maintaining an event list. Each event in the list is associated with a timeout value and a callback function. The events are sorted according to their timeout values. To add an event, the algorithm or the engine itself simply inserts the event to the list. The engine periodically examines the list and triggers events. When an event is due, the engine invokes the event handler function and removes the event from the list. In the case of a periodic event, the event is inserted back to the list according to its next timeout after executing the event handler function.

The Algorithm

The algorithm in the engine thread is where new P2P streaming protocols are to be developed. To minimize development time, the algorithm includes a collection of basic elements for multimedia streaming, including playback buffers and simple protocols. The collection grows as new protocols are designed and implemented by system designers. The algorithm is usually implemented as an instance of an application-specific C++ class, within the engine thread. The application-specific class is derived from the `iAlgorithm` class, which defines the Crystal API between the engine and the algorithm. Using this API, the engine routes messages from all upstream peers to the algorithm for processing; and the algorithm sends messages — via the engine and the network — to the appropriate downstream peers.

When the incoming queues are filled with unprocessed messages and the outgoing queues are empty, the network thread is idle, as the engine thread is busy switching messages at the rate of receiving them. When an algorithm involves lengthy operations, it can block the engine thread from processing incoming queues and timed events. For this reason, we allow an algorithm to launch its own private thread(s). Similar to any other multi-threaded application with potential access to a set of shared data by multiple threads, each algorithm thread has to use proper synchronization construct to prevent potential race conditions. Crystal provides high-level synchronization constructs in its library, which can be easily used. While allowing algorithm-specific threads adds to the complexity of algorithm development, it gives developers the flexibility in designing efficient and sophisticated algorithms.

B. Design Objectives Revisited

There are no limitations in the Crystal design that preclude emulating more than one peer on each server; in fact, such a way of running emulated peers is encouraged. Emulated peers do not need to periodically contact a central server for logistics or authentication. All logs are written to local file systems, and are then collected and analyzed by prescribed scripts after each experiment. By minimizing the footprint of each emulated peer, we are able to run hundreds of peers on one server. Let us now revisit the original design objectives, and note how Crystal fulfills these requirements.

1) Emulating bandwidth and delay: In a realistic P2P network, peers are connected to the Internet using home (*e.g.,* cable/DSL) broadband connections. Crystal supports the emulation of peer upload and download bandwidth limits, peer total bandwidth limits, as well as per-connection bandwidth limits for TCP connections. More than one limit can be enforced concurrently on a peer. In implementing such bandwidth limits, our objective is to minimize the demand for CPU cycles. The engine incurs lighter CPU load as the bandwidth limit decreases, allowing more bandwidth-emulated peers on each server.

The basic idea is to use a timer in the network thread to limit peer upload and download bandwidth, and an individual timer to enforce per-connection bandwidth limits. The timers are implemented through the `select()` call that is already used for monitoring active sockets in the network. The timeout value of the `select()` call is tuned dynamically according to the network traffic and available bandwidth, and is critical to the correctness and scalability of the bandwidth/delay emulation. The basic idea is as follows. In the example of per-connection upload (download) bandwidth emulation, the message queue is delayed for n/b seconds after sending (receiving) a message, where n is the number of bytes in the message, and b is the bandwidth limit of this connection. The descriptor associated with this connection is not added to `fd_set` until n/b seconds later. In other words, we dynamically determine the set of "ready" message queues, which are allowed to transmit at the current time. By dynamically changing the membership of `fd_set` based on predefined bandwidth limits and the current time, multiple bandwidth limits can be enforced simultaneously with little overhead.

To implement delays on a connection in Crystal, we add a timestamp at the time a new message is created or received, and at the time of sending the message, we delay the corresponding message queue for $n/b+(d-d')$, where d is the link delay and `d'` is the difference between the current time and the timestamp of the message.

2) Emulating streaming servers and actual traffic: Crystal provides a testing data source to facilitate the emulation of real-world streaming servers in a live session. The testing data source is able to produce data messages from a regular file, the standard input, or a stream of randomly generated bytes. These data messages are stored in a dedicated message queue that is treated by the engine in the same way as other message queues. The emulated streaming servers send data messages via TCP or UDP connections, according to a specified streaming rate. This design leads to minimal changes in the implementation of the streaming servers, *i.e.*, they share the same architecture as a regular peer, with a special data generator and a source message queue. Though a testing data source is provided, algorithm designers may still choose to implement their own data sources.

3) Emulating peer dynamics: In a realistic P2P network, peers may present a significant level of dynamics by joining and departing at any time. To emulate peer dynamics, we have implemented a log-driven facility. Upon the startup of each experiment, each peer parses the events file for all events that are associated with its IP address and port number, and registers them in the event timer managed by the engine. The use of such an event-driven facility relaxes the necessity of contacting centralized servers, which may lead to a considerable amount of TCP traffic that may affect the precision of experiments.

The *events* file specifies the time that the event should occur in an experiment and the type of the event. Typical events include the *birth* and *die* times of a peer in the network, as well as the *join* and *leave* times of a peer in a session. There are two events that are dedicated to the streaming server: *deploy* and *terminate*. The deploy event specifies the streaming rate in bytes/second and the time that the server should start to produce streaming content, whereas the terminate event specifies the time when the source stops producing data. We also allow optional parameters associated with each event. For example, to specify the arrival of a peer in a session, the event in the log has parameters such as the session identifier, streaming server and streaming rate, as well as the arrival time.

Crystal provides scripts to generate events in the log according to a probability distribution. These scripts are able to generate the following distributions of events: (1) The *birth* events following the Poisson distribution with a PDF $f(k;\lambda) = \frac{e^{-\lambda}\lambda^k}{k!}$, where $1/\lambda$ is the inter-arrival time. (2) The *join* events may follow a Weibull distribution with PDF $f(x;k,\lambda) = \frac{k}{\lambda}(\frac{x}{\lambda})^{k-1}e^{-(x/\lambda)^k}$, where k is the shape parameter and λ is the scale parameter. (3) The lifetime of a peer (the difference between *join* time and *leave* time) may follow either a Weibull distribution or a Log-Normal distribution with a PDF $f(x;\mu,\sigma) = \frac{1}{x\sigma\sqrt{2\pi}}e^{-(lnx-\mu)^2/2\sigma^2}$, where μ and σ are the mean and standard deviation, respectively. Naturally, these distributions can be modified by algorithm developers at any time by modifying the prescribed scripts.

4) Bootstrapping new peers: Upon the creation of a peer, the peer needs to start with a set of live peers in the network. We have implemented a centralized *tracker* to provide this first-level bootstrap support by providing newly joined peers a random subset of existing peers. The tracker can either actively probe each peer for aliveness or passively wait for reports from each peer, depending on the configuration. The *topology* file provides an alternative solution for bootstrapping peers. For each peer, this file specifies a small number of peers that are alive at the *birth* time of this peer. The use of this file relaxes the necessity to contact the tracker, which may lead to a considerable amount of TCP traffic.

5) The playback buffer: For each streaming session, a peer maintains a *playback buffer* in which segments (small units of streaming content) are ordered according to their playback time. In Crystal, the playback buffer is internally implemented as a *circular queue*. The playback buffer registers a periodical event with the engine to emulate segment playback. The period of this event is determined by the size of each segment, *i.e.*, the number of seconds of playback represented by a segment.

6) Developing a streaming algorithm: Crystal provides an API to the developer of a new streaming algorithm. With the Crystal API, the algorithm developer does not have to be concerned with thread safety, in that all algorithm-specific code is executed in the *algorithm* component of the engine thread. Such a design allows the designers to focus on protocol-specific details, without worrying about the internal data structure and thread safety issues in the engine. However, it is the developer's responsibility to handle race conditions and synchronization issues in threads created by the algorithm outside Crystal.

A new algorithm should be developed as a derived class of `iAlgorithm`, a base skeleton class implemented by Crystal, defining the Crystal API. It includes a few member functions that the new algorithm must implement. They

TABLE I. The API of Crystal

Member Function of `iAlgorithm`	Explanations
`bootstrap()`	This function needs to be implemented to bootstrap a new peer, and initialize its algorithm-specific parameters.
`joinSession()` `leaveSession()`	The engine calls these functions when a peer joins a session or leaves a session. When a session is removed by the source, `leaveSession()` is also called by the engine. A session is uniquely identified by its session identifier (a short type integer).
`process()`	The engine calls this function when a control message is to be processed.
`processData()`	The engine calls this function when a data message is to be processed.
`peerDeparted()`	The engine calls this function when an existing upstream or downstream peer has departed. The engine encapsulates all exception handling mechanisms, and these functions are called when exceptions occur.
`removePeer()`	The algorithm calls this function to explicitly tear down the existing TCP connection to an upstream or downstream peer.
`send()`	The algorithm calls this function to send a message to a peer. A message can be sent via either TCP or UDP, specified by one of the parameters.

are listed and explained in Table I. The nature of such an algorithm is *reactive*, *i.e.*, it reacts to messages received by the engine from its upstream peers, by implementing `process()` and `processData()`, which is called by the engine. The algorithm calls `send()` to send one or more messages to downstream peers.

IV. Crystal: Scalability and Cases Studies

We first present benchmark measurements of Crystal in terms of CPU usage and maximum achievable throughput to show its scalability. We studied three performance benchmarks: (1) Cumulative streaming rate (MB/sec), the sum of the streaming rates achieved by all peers. (2) Per-peer streaming rate (MB/sec), the average streaming rate from all peers in the network. (3) CPU usage, the percentage of utilized CPU cycles, which signifies the scalability of Crystal in terms of CPU load. All experiments are carried out in a server cluster of 50 dual-CPU servers (Pentium 4 Xeon 3.6 GHz and AMD Opteron 2.4 GHz), interconnected by Gigabit Ethernet.

To evaluate the scalability of Crystal, we implemented a simple P2P relaying protocol. In this protocol, a peer simply relays incoming data messages to all neighboring peers. A streaming server and a tracking server are hosted on a separate machine from the remainder of the peers, so that they do not compete for CPU cycles. the streaming server generates data as fast as the CPU and TCP connections allow. The tracking server is configured to bootstraps peers in a way so that the peers form multiple chains of up to 10 peers that are stemmed from the streaming servers.

A. Scalability

We first examine the maximum achievable streaming rate as the number of emulated peers increases on a single cluster server. In this experiment, we enforced no bandwidth limits or link delays so that the engine switches messages as fast as possible, *i.e.*, both CPUs on each server are 100% saturated. The message payload is set to 1 KB. As shown in Fig. 4, the cumulative streaming rate gradually decreases as the number of peers scales up. Despite the decrease, Crystal is able to achieve more than 12 MB/sec in terms of the cumulative streaming rate. Fig. 5 further shows that the per-peer streaming rate quickly converges to 50 KB/sec, based on which we predict that a single server can support at most 200 - 300 peers at this rate.

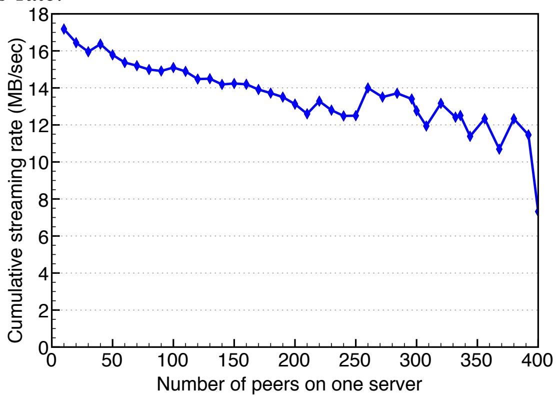

Fig. 4. The cumulative streaming rate from all peers on a single server.

The above observation turns our attention to the CPU performance when tuning the per-link upload bandwidth limit and the number of peers. In this experiment, we first fix the number of peers to 10 and increase the per-link upload bandwidth limit from 50 KB/sec to 1.7 MB/sec. As the bandwidth limit increases, the engine needs to switch more messages every second, *i.e.*, the CPU spends more time on message switching. As illustrated in Fig. 6, the CPU load decreases steadily as the bandwidth limit decreases, thus allowing more peers to be hosted on one

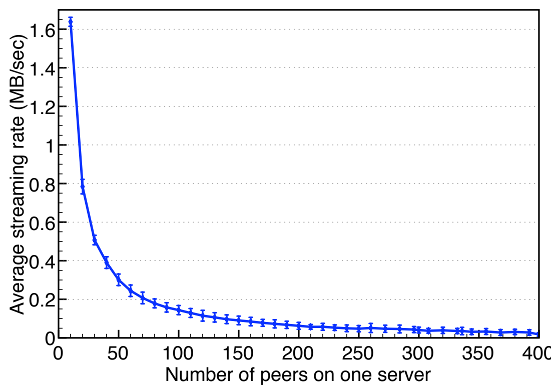

Fig. 5. The average streaming rate per peer on a single server, with standard deviation.

server. Without traffic, there is no load on CPU, and without any bandwidth limits, Crystal achieves the same TCP throughput as any other application. The CPU utilization grows slowly when the bandwidth limit is less than 600 KB/sec, and then linearly increases as the bandwidth limit grows. In real-world P2P networks, most peers have DSL-like connections with less than 300 KB/sec upload and download bandwidth. The results shown in Fig. 6 indicate that Crystal can easily support a few dozens of DSL-like peers on a single server with a light CPU footprint.

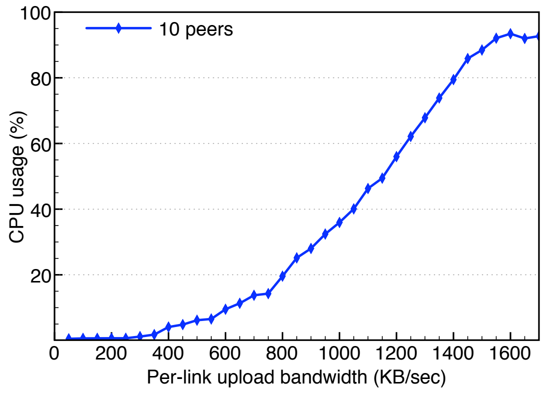

Fig. 6. CPU usage as the per-link upload bandwidth increases.

We then fix the bandwidth limit to 50 KB/sec, a typical DSL-like peer upload bandwidth, and increase the number of peers from 10 to 250. As shown in Fig. 7, the CPU load decreases steadily as the number peers decreases. We have observed a noticeable throughput drop on the peers that were far away from the streaming server when the network consisted of more than 200 peers. This means that the message switch in the engine could not keep up with the arrival rate of incoming messages, which confirms the observation from Fig. 5 that a single server can support at most 200 peers under this setting.

To further show the scalability of Crystal on multiple servers, we scale the network size across 10 different cluster servers. The average per-peer streaming rate in Fig. 8 follows the same pattern as in Fig. 5. However, for

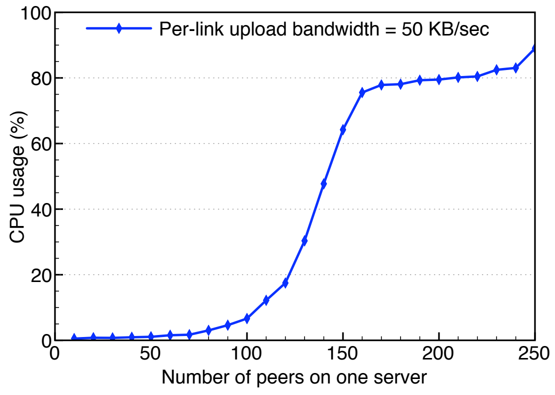

Fig. 7. The CPU usage as the number of peers increases.

the same network size, Crystal offers higher streaming rate on 10 servers than it does on a single server. Moreover, the streaming rate decreases at a slower rate in Fig. 8. Hence, Crystal scales better when it is deployed on multiple servers.

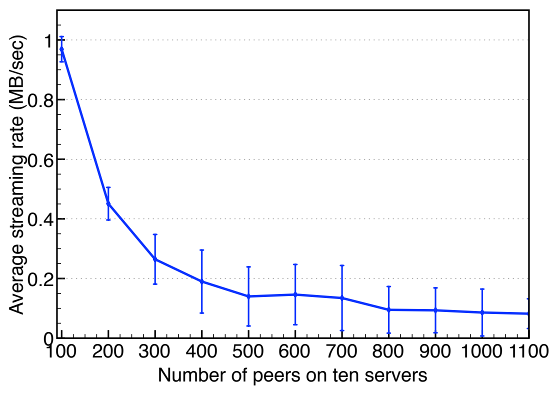

Fig. 8. The average streaming rate per peer on 10 servers, with standard deviation.

The number of messages to be switched by the engine is determined by not only the bandwidth, but also the message size. We conduct another set of experiments to determine the maximum achievable streaming rate as we vary the size of messages from 1 KB to 40 KB. The same experiment is repeated in three different networks consisting of 10, 100, and 200 peers. Fig. 9 shows that the cumulative streaming rate grows almost linearly as the message size increases. Thus, to emulate a network with higher bandwidth limits, one can either increase the number of servers or increase the message size.

B. Case Studies

To evaluate the effectiveness of Crystal, we have first implemented a peer-to-peer streaming protocol that uses conventional pull-based mesh topologies. In this protocol, referred to as *Vanilla*, a peer requests a missing segment from other peers (or the source) according to their *buffer maps*, exchanged among neighboring peers periodically.

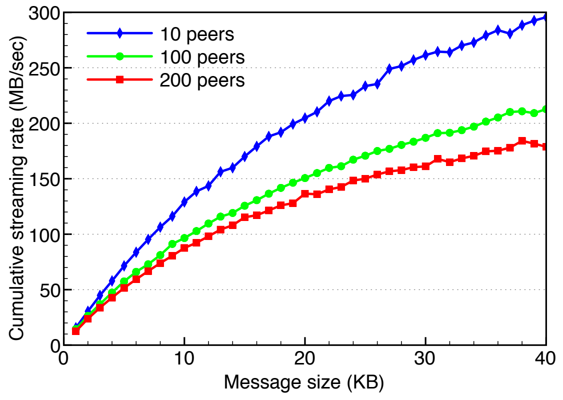

Fig. 9. The maximum sustainable streaming rate as the message size increases.

Most features of Crystal are used when Vanilla is developed, allowing us to focus on its protocol design, rather than implementation details. Our streaming protocol reacts to messages received from peers, by implementing functions for processing protocol-specific messages, which is called by the Crystal engine. Vanilla also takes full advantage of the Crystal support for periodic events (*e.g.*, to periodically exchange buffer maps and playback buffered segments). As a result of using Crystal, we are able to finish the implementation with only 2102 LOC (including comments), with one researcher, and two weeks of development time.

We have deployed Vanilla in our server cluster. With the assistance of Crystal, we were also able to conduct a detailed analysis of the system performance using Vanilla. We modified each peer to log their buffer status as the session progresses, and then collect the results using a script. More detailed results of Vanilla can be found in our previous work [9].

While advantages of network coding have been better understood and tested in scenarios of P2P content distribution, we are curious to know whether the same benefits apply to P2P streaming. Existing streaming systems are not explicitly designed to utilize network coding. We believe a complete redesign of the streaming system is necessary to take better advantage of network coding. In our previous work [10], we have designed R^2, a new P2P streaming protocol that uses network coding to improve streaming performance. Despite the computational complexity of network coding, with the scalability of Crystal, we are able to emulate more than 500 peers in total (around 12 peers on each cluster node), with each peer performing network coding in real time as the experiments progress. The success of implementing R^2 using Crystal shows that Crystal is sufficiently flexible to accommodate a wide variety of design objectives, including a design that uses network coding.

V. Concluding Remarks

This paper is written to present our motivation, design and experiences with *Crystal*, an emulation framework for practical P2P multimedia streaming systems. Crystal is designed to address the challenges of rapidly prototyping, testing, and evaluating new P2P streaming systems, in a more realistic scenario than simulations, but a more controllable and scalable environment than real-world experiments. Our implementation of *Crystal* crystallizes about two years of research and development, with about ten thousand lines of code in C++, and now ready to be released in an open-source form to the research community. It includes the *engine* at the core to handle networking and message switching functions, as well as a clearly defined API between the engine and the algorithm. Our hope is that Crystal can help to minimize the turnaround time required to design, implement, and evaluate new P2P streaming protocols.

References

[1] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum, “CoolStreaming/DONet: A Data-Driven Overlay Network for Peer-to-Peer Live Media Streaming,” in *IEEE INFOCOM*, March 2005.

[2] L. Peterson, T. Anderson, D. Culler, and T. Roscoe, “A Blueprint for Introducing Disruptive Technology into the Internet,” in *Proc. of the First Workshop on Hot Topics in Networks (HotNets-I)*, October 2002.

[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,” in *Proc. of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002), to appear*, December 2002.

[4] B. Li, J. Guo, and M. Wang, “iOverlay: A Lightweight Middleware Infrastructure for Overlay Application Implementations,” in *Proc. of the Fifth ACM/IFIP/USENIX International Middleware Conference (Middleware 2004)*, October 2004.

[5] X. Jiang and D. Xu, “vBET: a VM-Based Emulation Testbed,” in *Proc. of ACM Workshop on Models, Methods and Tools for Reproducible Network Research (MoMeTools 2003)*, August 2003.

[6] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker, “Scalability and Accuracy in a Large-Scale Network Emulator,” in *Proc. of 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, December 2002.

[7] C. Killian, J. Anderson, R. Braud, R. Jhala, and A. Vahdat, “Mace: Language support for building distributed systems,” in *Proc. the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, June 2007.

[8] A. Rodriguez, C. Killian, S. Bhat, D. Kostic, and A. Vahdat, “MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks,” in *Proc. of the USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)*, March 2004.

[9] M. Wang and B. Li, “Network Coding in Live Peer-to-Peer Streaming,” in *IEEE Transactions on Multimedia, Special Issue on Content Storage and Delivery in Peer-to-Peer Network*, 2007.

[10] M. Wang and B. Li, “R^2: Random Push with Random Network Coding in Live Peer- to-Peer Streaming,” in *IEEE Journal on Selected Areas in Communications, Special Issue on Advances in Peer-to-Peer Streaming Systems*, 2007.