ProtoPeer: A P2P Toolkit Bridging the Gap Between Simulation and Live Deployement

Wojciech Galuba, Karl Aberer
Ecole Polytechnique Fédérale de Lausanne
Lausannne, Switzerland
firstname.lastname@epfl.ch
`http://protopeer.epfl.ch`

Zoran Despotovic, Wolfgang Kellerer
DOCOMO Euro-Labs
Munich, Germany
lastname@docomolab-euro.com

ABSTRACT

Simulators are a commonly used tool in peer-to-peer systems research. However, they may not be able to capture all the details of a system operating in a live network. Transitioning from the simulation to the actual system implementation is a non-trivial and time-consuming task.

We present ProtoPeer, a peer-to-peer systems prototyping toolkit that allows for switching between the event-driven simulation and live network deployment without changing any of the application code. ProtoPeer defines a set of APIs for message passing, message queuing, timer operations as well as overlay routing and managing the overlay neighbors. Users can plug in their own custom implementations of most of the parts of ProtoPeer including custom network models for simulation and custom message passing over different network stacks.

ProtoPeer is not only a framework for building systems but also for evaluating them. It has a unified system-wide infrastructure for event injection, measurement logging, measurement aggregation and managing evaluation scenarios.

The simulator scales to tens of thousands of peers and gives accurate predictions closely matching the live network measurements.

Categories and Subject Descriptors

I.6.7 [**Simulation Support Systems**]: Environments—*event-driven simulation, network simulation*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*peer-to-peer systems, message passing*

Keywords

distributed systems, peer-to-peer systems, prototyping, toolkit, framework, simulator

1. INTRODUCTION

The three important research tools used by the P2P research community are the analytical models, simulations and experiments on the actual systems. Large-scale distributed systems are complex and accurately modeling them analytically is not an easy task. Most of the time the first iteration of an analytical model is not tractable and the model needs to be successively simplified to produce useful insights. This leads to models that describe the system at a very coarse-grained level and with many uniformity assumptions. Despite this, the analytical models are in most cases accurate, scale to an arbitrarily large system size and are an excellent tool for early feasibility assessment of a P2P solution. However, for a more complete and accurate evaluation under a wider range of conditions the researchers commonly turn to simulation or system evaluation on the actual networks.

Simulations cannot be scaled to an arbitrarily large size as the analytical models can, but allow for exact implementation of the message passing protocols. The protocols can then be evaluated under a wide range of conditions without being limited to the highly uniform cases used in the analytical models. With all their advantages simulators often make many simplifying assumptions about the underlying network model, which might affect the predictions obtained from the simulation in non-trivial ways.

Usually the simulators are purpose-built for specific applications or classes of applications. Few simulators are designed as more general tools for system building and evaluation. What is more, even though simulation is a widely used evaluation technique there is almost no simulator code sharing among the researchers and little standardization of the common practices. Naicken et. al [20] examine 287 papers on P2P systems. Out of the 141 papers that use some form of simulation 114 either use a custom-made simulator or do not specify what software was used. The other 27 papers use publicly available simulators, however these papers are either authored by researchers who developed the simulator or by researchers from the university in which the simulator was developed. The limited evaluation tool reuse motivated us to invest effort in sharing our framework, ProtoPeer, with the rest of the community. We have also made ProtoPeer modular such that it can be easily extended and the modules can be shared among the system builders. We further define the niche ProtoPeer fills and compare our framework to the others in §6.

While simulators are a popular tool, the most accurate system evaluation can only be achieved with a system implementation running in the real network. Switching from the simulation to the actual implementation often requires a considerable development effort. Network I/O implementation, measurement instrumentation, deployment automation and distributed debugging are among the most time consuming tasks that developers face. Moreover, large parts of the application code existing in the simulator are not

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5681
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5681

reused in the actual system. Bridging the gap between the simulation and the actual system deployment was the primary motivation for developing ProtoPeer.

In ProtoPeer the developer can switch between the simulation of a P2P system to its deployment on the actual network without changing a single line of code. This dramatically speeds up the implement-evaluate-reimplement cycle. Most of the major bugs and performance problems are caught early on during the simulation while the more time-consuming live deployment is used for the accurate evaluation of the final system on testbeds such as ModelNet [25] or PlanetLab [11].

ProtoPeer has an API for building arbitrary message passing systems not only the P2P systems. Most of the common tasks such as network I/O, message serialization and message queuing are handled by ProtoPeer. The user only needs to focus on implementing the message passing logic of the application without worrying about the underlying details. If need arises, the ProtoPeer API allows users to plug in their own implementations of most of the parts of the system. This can be for example used to implement a specialized message queuing discipline or to implement message passing over transports other than the default TCP or UDP.

Measurements are an important part of any system evaluation. ProtoPeer has tools and APIs covering most of the measurement pipeline: from measurement instrumentation through aggregation and computation of the basic statistics to plotting the measurements at the end. ProtoPeer also implements a general event injection mechanism, which is particularly useful for evaluating the system under various failure scenarios and modeling churn (i.e. peer departures and arrivals).

2. ARCHITECTURE

2.1 Message passing & timers

In ProtoPeer the system is composed of a set of peers that communicate with one another by passing messages. Each application[1] defines its set of messages and message handlers. An application typically also defines a set of timers and handlers for the timer expiration events. All the application logic in ProtoPeer is called from within the timer and message handlers.

2.2 Networking & time abstraction

One of the main goals of ProtoPeer is to be able to switch between simulation and live network deployment without changing any of the application's code. The key architectural feature that enables this are the abstract time and networking APIs[2]. The APIs allow for only a small number of basic operations: creation of timers for execution scheduling and creation of network interfaces for sending and receiving messages. These simple APIs serve as the key building block for the rest of the ProtoPeer and form the "waist" of the framework's hourglass architecture (Fig. 1).

[1]by "application" we will understand a distributed application implemented using ProtoPeer, there can be several applications running simultaneously in the system

[2]The reader is referred to the on-line ProtoPeer tutorial at `http://protopeer.epfl.ch/wiki/IntroTutorial` for numerous code examples. We have omitted them in the paper due to the lack of space.

When switching from the simulated run to the live run the simulated time and networking implementations are simply swapped with the implementations using real timers and TCP or UDP networking (see §4.1). The ProtoPeer user can also provide alternative time and networking implementations, using, for example, some other transport protocols or different message serialization. The new implementations can be plugged in without any changes to the application code using them. This allowed us, for example, to implement a separate network model for mobile ad-hoc networks (MANETs) using the JiST/SWANS framework [1]. The model was simply plugged into the simulator and the same message passing protocols that run in peer-to-peer systems could then be evaluated in MANETs. Besides confirming ProtoPeer's versatility these experiments have led to a number of new insights and improvements in the simulated protocols. We elaborate further on network modeling in §2.4.

2.3 Event-driven execution

Execution in ProtoPeer progresses by calling event handlers in response to time and networking events. For example, when a timer expires an appropriate handler is called for that event. The handler might send a message or schedule other timers, which subsequently trigger other handler calls and so on.

During simulation the events are stored in a single system-wide queue. The events are ordered according to their scheduled time of execution (the time is virtual). When a handler call finishes the next closest future event is dequeued and its corresponding handler is executed.

During live deployment handlers are asynchronously called as the various networking and timer events happen. The ProtoPeer's event-driven architecture does not forbid concurrent execution of two handlers in two different threads. The implementations of time and networking have complete freedom in allocating the different handler calls to the threads, which gives great flexibility in performance optimization. In the current version of ProtoPeer both time and networking implementations use thread pools with configurable size. We discuss the advantages of this approach in §4.1.

2.4 Network modeling

As mentioned in §2.2 switching between the simulation and the live run is as easy as swapping one networking implementation for the other. During simulation the network needs to subject the messages to realistic delay and loss. Loss and delay modelling are encapsulated in the `NetworkModel` interface in ProtoPeer. Users can provide their own implementations of that interface. There are several implementations already available, including simple uniformly distributed delay model, the Euclidean model (i.e. delay between nodes proportional to their distance in the Euclidean space) or the delay matrix model into which arbitrary delay matrices can be loaded (e.g. based on the King dataset[3]).

Messages in ProtoPeer can be of arbitrary size after serialization. The delay to which the message is subjected should be a function of its size and the available bandwidth. For simulating bandwidth allocation we are currently developing a MaxMin flow-based model. In that model each peer has a limited incoming and outgoing bandwidth. This model is especially useful for simulating bandwidth-bound applica-

[3]http://pdos.csail.mit.edu/p2psim/kingdata/

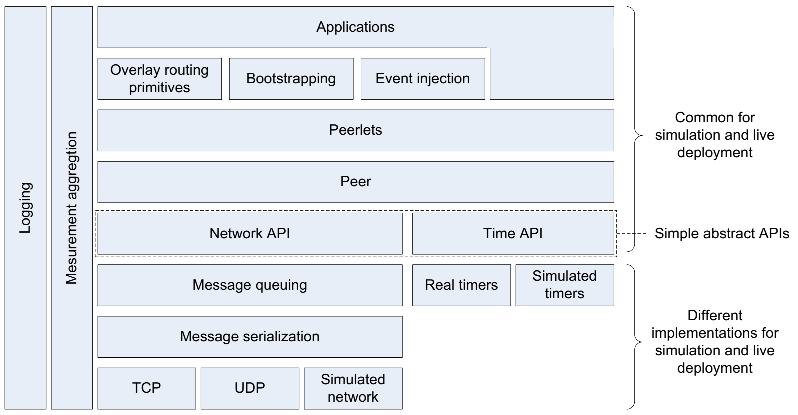


Figure 1: The ProtoPeer architecture. The time API and the networking API consist of only a few basic abstractions and methods and form the narrow "waist" of the ProtoPeer's hourglass architecture. The upper part of the hourglass are all the components that use the time and networking APIs. The peer provides the runtime context for the peerlets (§2.6) in which the various parts of the peer's functionality are encapsulated, e.g. bootstrapping logic, overlay message routing, event injection (§3.2) or the application-specific logic. The lower part of the hourglass are the concrete implementations of the abstract time and networking APIs, e.g. message passing over TCP or virtual timers for scheduling events during simulation. Switching from the simulated system to the actual system is as simple as switching from one time & networking implementation to another.

tions such as BitTorrent, while the other simpler and faster models mentioned in the previous paragraph can be used for delay-bound applications such as DHTs.

2.5 Overlay modeling

Research on peer-to-peer systems has produced many ways of constructing and maintaining overlays. Most overlays assign identifiers to the peers from some ID space. Each ID space typically has a distance metric associated with it. For example, in Chord [23] the ID space is the unit ring (0,1] and the metric is the distance between the IDs on the ring, while in Kademlia[18] peers take IDs from the set of 160-bit integers, the distance is measured as the numerical value of the bitwise XOR between two IDs. In fact, there exist overlays construction protocols for arbitrary identifier spaces and distance metrics[13]. To support the wide range of overlays, ProtoPeer defines an abstract `PeerIdentifier` that is used throughout the system. The different overlay implementations override it with concrete implementations of their ID space and the distance metric.

Most of the peer-to-peer systems define some form of logical links between peers that together form the overlay topology. This concept is so fundamental that it has been added at the core of ProtoPeer. Each peer has its neighbor set and exposes it in a uniform way to all applications. Each neighbor is stored as a pair of the neighbor's peer ID and the neighbor's network address. The neighbor set appears in many overlay implementations and is sometimes referred to as the routing table, finger table etc. The neighbor set is used not only by the overlays but also by the applications running on top of them such as DHTs, which is another reason for making access to the peer's neighbors uniform throughout the system.

2.6 Peerlets

A peer in the peer-to-peer system typically implements more than one piece of the message passing functionality. For example, a peer might need one protocol for contacting the bootstrap server and getting the initial neighbors, another protocol for maintaining the overlay during churn and yet another for DHT key replication. In ProtoPeer the message passing logic and state of each of the protocols is encapsulated in components called *peerlets*. Peers are constructed by putting several peerlets together. The peerlets can also be removed or added at runtime.

The peerlets, just as the applets or servlets, have the familiar init-start-stop lifecycle. The peer provides the execution context for all of the peerlet instances it contains. The peerlets can discover one another within that context and use one another's functionality. The peerlets have access to the peer's network interface through which they send and receive messages. Peerlets can also arbitrarily modify the peer's neighbor set.

The peerlet-based approach has all the advantages of any other modular design. Firstly, the message passing functionality is conveniently encapsulated in building blocks with well defined behavior. The blocks can be composed to achieve the desired peer functionality. Certain functionality can be easily enabled or disabled depending on the context (e.g. debug mode vs. evaluation mode). Secondly, peerlets can be reused across applications. Peerlets can export well defined interfaces e.g. a DHT interface, which can have several implementations that can be easily swapped one for another. Lastly, peerlets can be unit tested either in isolation or with other peerlets as mock objects.

2.7 Queuing

Message queues are an essential component of many peer-to-peer system designs. Queues buffer the messages during the transient periods when the rate of asynchronously arriving messages at the peer exceeds its capacity to process them. Very often queues are not explicitly implemented and

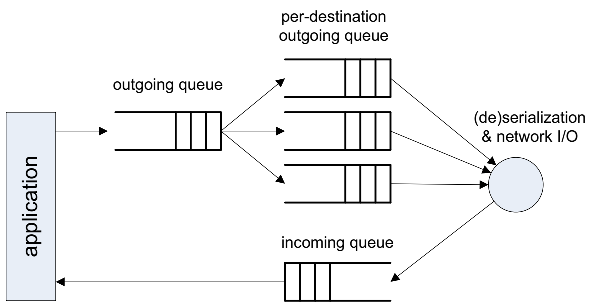

per-destination outgoing queue
outgoing queue
(de)serialization & network I/O
application
incoming queue

Figure 2: Message queueing in ProtoPeer. When an application sends a message it first passes through the main outgoing queue and then is placed on the appropriate per-destination outgoing queue. The networking layer picks the messages from the outgoing queue, serializes them and sends the raw binary data over the network. When data is received, the deserialized message is put on the incoming queue for the application to pick up. Each of the queues can be replaced by application-specific implementations. Queues can consist of from several queues linked together in various ways. Basic implementations of mux/demux queues, queue chains etc. are available.

the applications implicitly rely on operating system's socket buffers for message buffering. Even if the queues are implemented they are typically unbounded. This is a valid assumption for the applications with low message rates. However, queuing has to be more carefully designed in systems optimizing for high throughput or implementing congestion control schemes (for example [15, 12]).

In ProtoPeer all the peers have incoming and outgoing message queues (Fig. 2). Applications may add their own queue implementations. At runtime the applications have access to the queue state and can for example use it for congestion signalling. Queues are a versatile tool and can be used for many purposes: message prioritization, message flow rate-limiting or failure injection (§3.2) by dropping, mutating or delaying messages.

3. TOOLS

So far we have covered the basic architectural building blocks for constructing a peer-to-peer system. We now turn to facilities available in ProtoPeer for evaluating the peer-to-peer systems once they are built.

3.1 Measurement infrastructure

Obtaining reliable and accurate measurements is an important, if not the most important part of any peer-to-peer system evaluation. Measurements need to be instrumented in the application code, logged, aggregated from all the peers in the system, analyzed and optionally plotted. ProtoPeer's measurement infrastructure helps with all of these tasks. The measurements are instrumented by doing calls to the measurement API in the appropriate places in the application code. While the system is running the measurements can either be sent regularly to the measurement server for live monitoring of the network or they can be dumped to a local file at each peer and aggregated later.

```
//count messages per class
mlog.log("count", incomingMessage.getClass(),1);
//log the measured round-trip-times
mlog.log("rtt",rttValue);
//log the traffic flow for every link (in bytes)
mlog.log("outflow", sourceAddress, destinationAddress,
         outgoingMessage.getSize());

//get total number of DHTLookup messages
mlog.getAggregate("count", DHTLookup.class).getSum()
//get the 95th percentile of the RTTs
mlog.getAggregate("rtt").getPercentile(95);
//get the total number of bytes TXed from one IP to another
mlog.getAggregate("outflow",new NetworkAddress("62.35.31.65:3392"),
         new NetworkAddress("132.35.136.25:6342")).getSum();
```

Figure 3: Measurement API example. Measurement logging is done by calling `log()` on the `MeasuremenLogger` instance. The function takes as arguments an arbitrary number of tags and the measured floating point value. A tag can be any Java object as long as its `hash()` and `equals()` methods are well defined, which makes the tagging-based measurement a multi-purpose tool. The same tags used for logging are then used to obtain aggregates. Aggregates can also be computed for specific time windows, specific peers or system-wide. There are tools for fetching and merging measurement logs from all the peers as well as tools for log browsing and plotting.

While the measurements are accumulating the system computes the basic statistics on-the-fly: average, sum, variance etc. This allows for extremely compact representation of the measurements. Optionally all logged values can be kept which later on permits the computation of other statistics such as percentiles. The statistics can be computed at various aggregation levels: per peer, per time window and per measurement tag. Measurement tags are objects used to "mark" the values that are reported to the measurement logger. The user can request an aggregate performed over all the measurements that match a certain tag or a set of tags. Tags can be any object, in particular they can identify an individual peer, a link between peers, the message class or they are simply a string describing the nature of the measured value. We found that in practice the simple tagging-based measurement infrastructure covers the vast majority of needs (Fig. 3). Alternative solutions were necessary only in very obscure cases where complex objects had to be logged as the "measured values", which can be done using the existing logging frameworks [4]

ProtoPeer has several tools to support the measurement post-processing phase. Measurement log files coming from the different peers can be merged into a single system-wide log which can then analyzed separately. The access to the log content is programmatic so the users have complete freedom in outputting the processed log information in any format they desire. There is a basic tool that allows to browse the contents of the log files and plot measurements for the various aggregates types and tags.

Measurement logging can be disabled to minimize its impact on the system performance and for evaluating the "production" version of the system.

3.2 Event injection & scenarios

While evaluating the peer-to-peer system there is frequently

[4]http://logging.apache.org/log4j/

```
#set up the churn sequence
4       14.3    Peer.start()
3       15.1    Peer.stop()
1       36.9    Peer.start()
4       44.4    Peer.stop()

#inject a failure at 150s on 10 peers
0-9     150.0   Peer.Router.setDropMessages(true)
```

Figure 4: An example scenario file. Each scenario file consists of three columns. The first one specifies the peer index (or a range of indices) that uniquely identify the affected peer. The second column is the number of seconds since the beginning of the simulation when the event should be injected. The last column indicates the method to be called. Churn can be simply defined as a sequence of calls to the peer's start and stop methods. Calls can be made to any method of the peer or its peerlets. Users can define their own methods and call them in the scenario files, which makes event injection a multipurpose tool.

a need to test the system's response to various, often exogenous events, e.g. peer arrivals and departures (i.e. churn), user actions or, more commonly, failures. ProtoPeer provides a simple but general mechanism for event injection. The events are specified in triples consisting of time, the set of unique peer indices to be affected and the method to call. Despite its simplicity this way of describing events is expressive enough to cover most of the common use cases.

A set of events defines a *scenario*. Peers load the scenarios on startup and execute the events specified in them. Each scenario can be kept in a separate file (Fig. 4). For example, one scenario file can contain the precise churn model for the system specifying when the peers should go online or offline, while another scenario file can define the failures injected at the different moments in time. The scenario files can be generated, merged and filtered in various ways using the common text processing utilities. Scenarios are an important tool for systematizing the evaluation process and ensuring high experiment repeatability.

3.3 Unified randomness source

Another facility for ensuring repeatability in ProtoPeer is the unified source of randomness in the system. All random numbers are drawn via single system component managing the random number generators. The generators are seeded at the beginning. The same seed leads to the same sequence of events, which is particularly important during debugging. Naturally, randomness can only be completely controlled during simulation, however, if used correctly the unified randomness source also improves the repeatability of live runs. ProtoPeer uses the Mersenne twister [17] random number generator. Users can plug in their own generators if needed.

ProtoPeer manages several random generators at a time, each for a different purpose. This allows to, for example, keep the randomly generated overlay topology the same while changing the random sequence of message delays.

4. IMPLEMENTATION

ProtoPeer is developed in Java, which has been chosen for its popularity, ease-of-use and availability of libraries. We next cover the key implementation details.

4.1 Networking

In most of the peer-to-peer systems, messages arrive at the peers asynchronously. We have implemented both the time and networking event handling using the thread pools. The application puts the incoming messages on the queue, there is a pool of processing threads, whenever one of the threads becomes idle it dequeues the next message and processes it. We expose the queues to the application §2.7 so that the application can access their state or replace them with its own custom queue implementations.

Networking is implemented using Apache MINA[5], a high-performance networking framework. Its event-driven design and the use of non-blocking I/O fits well into ProtoPeer. Messages can be sent either over UDP or TCP. The choice which of the two to use can be made by the application on the per-message basis. All the complexity of opening sockets, maintaining them, handling the I/O errors and serializaing/deserializing messages is hidden from the application, which only receives callbacks for successfully sent messages and network exceptions.

4.2 Messages

Each message type in ProtoPeer is a separate Java class. The fields of the class correspond to the fields of the message. Messages can contain fields of any type, not only the primitive types, all of which are correctly passed between the peers.

During a live run when a peer sends a message it is serialized into bytes and then deserialized at the receiving peer. During the simulation the messages are efficiently cloned using the standard Java cloning mechanisms. Cloning is necessary to ensure that any subsequent modifications of the message instance at the source after the send call returns do not affect the message instances received at the destinations. Message classes that are guaranteed to be immutable can easily disable cloning. To measure bandwidth consumption during simulation, cloning can be optionally replaced with serialization, naturally, at a performance cost.

The standard Java serialization mechanism is extremely verbose and is primarily designed with persistent storage in mind not for transient data sent over the network. To address this, ProtoPeer uses its own *lightweight serialization* protocol which reduces the bandwidth consumption by a factor of 3 to 10 compared to the standard Java serialization. New message classes do not have to implement any serialization code, lightweight serialization is done through the Java Reflection API. Optionally, if message serialization becomes a performance bottleneck for some messages, they can define their own optimized binary serialization protocol. We are also considering dynamically injecting the serialization code into the Java classes using the Java bytecode rewriting to avoid the reflection overhead. Efficient serialization is still work in progress and we omit it in the evaluation section.

We have not yet considered the interoperability of ProtoPeer applications with other systems, however, the serialization is well separated from the rest of the framework and custom code can be added for serializing messages into a standard format, e.g. XML-based.

[5]http://mina.apache.org/

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5681
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5681

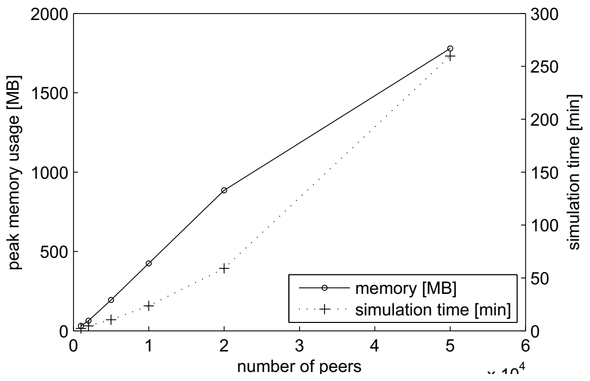


Figure 5: ProtoPeer simulator scaling. As the number of peers increases, the number of messages that are passed around in the system is increasing at least linearly as each added peer contributes its own lookup workload to the system. There are approximately 20 different measurement instrumentation points, the measurements are aggregated on-the-fly during the simulation. Logging is turned off. The system used was a 4-core Xeon 3.4GHz with 3GB of RAM running 2.6.8 Linux and Sun JDK 1.6.

5. PERFORMANCE

The goal of this section is to to evaluate the main performance characteristics of ProtoPeer. The results presented here should be treated as indicative, they are likely to change as the framework's codebase is evolving.

5.1 Scalability

We place the peers on a one-dimensional ring and wire them according to the Chord[**?**] rules. Each peer at 500-1500ms intervals picks a key from the ID space uniformly at random and performs a lookup on that key. The lookup is routed using the Chord algorithm. We simulate 300 seconds of the running system. The number of peers is gradually increased to test the scalability of the system (Fig. 5).

In our Chord routing test ProtoPeer reaches its scalability limit at approximately 50000 peers, when it hits the memory cap of 1.8GB, the maximum Java heap size the JDK 1.6 would allow us to allocate on a 3GB Linux system. The main memory consumer are the messages in the simulator's event queue scheduled to be delivered to their destinations. The main CPU bottleneck (approx. 20%) is message cloning when messages are passed from one peer to another. However, for most of the systems implemented in ProtoPeer that we worked with the CPU bottlenecks in the application code are far more common than in the ProtoPeer code.

5.2 Accuracy & realism

Simulation realism greatly depends on the network model used (§2.4). To validate our simulator's accuracy we have developed the *NetMapper* tool for generating network models that reproduce the exact delay and loss conditions occurring in a live network. We deployed NetMapper on 350 PlanetLab hosts. For 3 hours the hosts were pinging one another at one minute intervals. For each link the message loss and roundtrip latency were measured. Based on the measurements for each link we then generate a log-normally distributed delay model and compute the fixed loss probability based on the fraction of messages that got through on a given link. The delay and loss models are then plugged into ProtoPeer. Different delay and loss scenarios can then simply be generated by changing the random seed. This removes the need for capturing many PlanetLab traces and replaying them in the simulator.

To test the accuracy of ProtoPeer simulation and our delay-loss model we have re-run the same Chord routing test as in §5.1. We simulated a 350 peer Chord system and ran the same system live on PlanetLab. Figure 6 compares the measurements from both runs. The latency predictions from the simulator are very close to the ones obtained from the live network, especially considering the fact that the lookup delay accumulates over many peers and links on the routing path. On the other hand, the message loss is slightly overestimated in simulation, i.e. there are more lookup timeouts. Overall, however, the simulation results stay remarkably close to the ones from PlanetLab, which validates ProtoPeer as an accurate simulation tool. We have consistently observed similar accuracy in the wide range of other systems that we implemented with ProtoPeer. In general, when there is no network congestion on the links or CPU overload at the nodes, the simulation results from our network model are very close to the ones from PlanetLab. Our framework is open-ended and if needed, users can implement more sophisticated models that take congestion into account.

6. RELATED WORK

6.1 Simulators

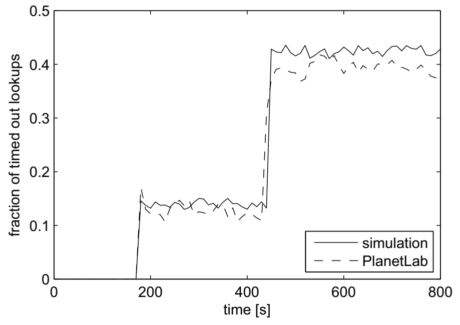
Naicken et al. [19] survey nine existing peer-to-peer system simulators. The simulators can be broadly classified into flow-level [2], message-level [5, 8, 7] and packet-level [6], in increasing level of simulation detail. ProtoPeer in its current implementation falls into the message-level category, which offers an acceptable level of accuracy for most peer-to-peer applications (§2.4).

Despite the lack of packet-level simulation detail, ProtoPeer provides an API for defining the models of the underlying network. Message loss and delay can be accurately simulated. This is in contrast to some of the existing simulators which either ignore the problem of delay and loss simulation or do not provide a way to customize the network model [7, 5, 3].

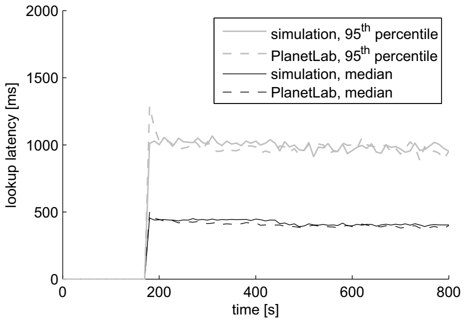
Most of the existing simulators do not provide any measurement facilities and those that do [6, 5, 7] offer only a default set of measurements, adding new ones requires a considerable development effort. ProtoPeer has a unified API and tools covering most of the measurement pipeline (§3.1)S: instrumentation, system-wide log merging and aggregate computation (e.g. mean, sum, percentiles).

A number of the existing simulators have the ability to script events in the system [6, 27]. Event injection is an important system evaluation tool. It can be used, for example, for injecting failures into specific system components or for simulating churn by specifying the peer arrival and departure events. ProtoPeer uses simple (but expressive) scenario files for event injection (§3.2), which help the user systematize the evaluation process.

6.2 System development tools



(a) Lookup timeouts



(b) Lookup latency

Figure 6: ProtoPeer simulator accuracy. The Chord routing test was run in a 350 peer PlanetLab deployment and in a 350 peer ProtoPeer simulation using the PlanetLab network model. All messages are sent over UDP. We measured the median and the 95^{th} percentile lookup latency as well as the fraction of lookups that timed out, i.e. when the destination did not respond back to the source. The timeout was set to 1500ms. The peers start issuing lookups at 180s. In addition at 450s we inject a failure (§3.2), 50 randomly chosen peers stop forwarding the lookup messages. This leads to an increase in the number of timeouts.

ProtoPeer is not only a simulator but also a tool for building systems running in real networks and we need to relate our framework to the existing ones.

Mace [14], a C++ language extension, defines a language for message passing protocol specification. Given the protocol specification Mace generates the C++ code which can then be deployed on the peers. One of the distinctive features of Mace is that it allows for automated protocol verification based on the specifications. Unlike ProtoPeer, Mace code cannot be easily run in a simulator, the more common practice with Mace is to use the emulators such as ModelNet [25], which limits the scalability.

In P2 [16] overlays are defined in a declarative language OverLog. OverLog allows for concise representation of protocols. However, OverLog, being a high-level language, hides most of the low-level implementation details which are delegated to the runtime.

In ProtoPeer we opted for the more traditional event-driven way of protocol specification (as in MACE) instead of the more concise declarative way with a steeper learning curve (as in P2). Unlike the two above frameworks we do not define a new language for specifying protocols, the application programmatically sets up the handlers for the various events occurring in the system (§2.2) and the execution progresses by calling these handlers (§2.3). The same handler setup is used both during the simulation and live deployment.

Neither P2 nor Mace allow for simulation of the system prior to deployment, which is their main disadvantage in comparison to ProtoPeer.

6.3 Testbeds & emulators

ProtoPeer requires the users to develop the application with the ProtoPeer API, there is no trivial way of taking the existing unmodified application and evaluating it with ProtoPeer. However, our framework was not designed for that, many testbeds and emulators have been developed with that goal in mind.

GODS [9], provides tools for automating the evaluation of distributed systems, but is not a library for developing them. Similarly to ProtoPeer, GODS has an infrastructure for instrumenting and aggregating measurements (§3.1) and for injecting events into the system (§3.2), e.g. peer departures and arrivals or link state changes. The framework also has tools for automating the evaluation and controlling the system lifecycle. GODS uses ModelNet [25] for IP layer emulation. In the MicroGrid [22], the application's networking calls are intercepted and mediated by the framework to emulate the network. MicroGrid uses a local scheduler that starts and stops the application processes according to a predetermined configuration.

Although the testbeds and emulators do not require any modifications to the application code, they have limited scalability since each application process is run separately and requires orders of magnitude more resources than application instances running in a simulator.

6.4 Develop once, deploy many times

ProtoPeer is designed from ground up to serve the dual role of a simulator and a tool for developing the live-deployable systems. There are several other frameworks capable of this. Most notably, Neko [24], uses the same message passing model as we do. Similarly to ProtoPeer, the messages can either be serialized and sent over TCP or UDP or passed between the peers in a simulator. However, Neko focuses primarily on the application layer and not the networking layer. The simulator has been designed for the clustered distributed systems communicating over LAN rather than for wide-area system communicating over the Internet, which is the target deployment environment for ProtoPeer.

GRAS [21] together with SimGrid [10] form a development and simulation framework. The framework was designed with Grid systems in mind and thus the focus is on bandwidth- and CPU-limited distributed applications while our ProtoPeer was designed for prototyping the delay-limited message passing systems. We are currently working on implementing a network model for ProtoPeer using the MaxMin bandwidth allocation, which would allow the simulation of bandwidth-limited applications such as BitTorrent.

Emulab (Netbed) [26] was originally designed as an emulator and a framework for network virtualization, but has been extended and now supports all three modes: simulation, emulation and live deployment. Just like ProtoPeer, Netbed has a wide range of tools for experimental control and event injection. However, Netbed's simulator uses *ns* [4], which is a packet-level simulator and is less scalable than our message-passing approach.

The focus of ProtoPeer is slightly different than that of the existing frameworks. We designed ProtoPeer primarily with the ease of use and rapid application prototyping in mind. Despite its simplicity, the framework is extensible both up, in the application complexity and down, in the simulation detail. This is how we see the ProtoPeer's development progressing in the future.

7. CONCLUSIONS

We have presented ProtoPeer, a P2P systems prototyping framework that bridges the gap between simulation and live system deployment. The applications are built on top of a simple time and networking abstraction which allows the user to switch from simulation to live deployment without any changes to the application code. This dramatically speeds up the implement-evaluate-reimplement cycle. All the low-level complexities of managing the network sockets, message serialization and queuing are hidden from the application developer. Applications in ProtoPeer can be modularized into peerlets which are reusable, unit-testable and can be composed together to achieve the desired peer functionality. Finally, ProtoPeer's measurement infrastructure, event injection and scenarios allow for systematic evaluation and performance tuning of the complete system.

8. ACKNOWLEDGMENTS

The authors would like to thank Julien Herzen and Jean Respen for their numerous contributions to the ProtoPeer source tree and the helping hand in testing the early versions of the framework. We would also like to thank Frank Lehrieder for the flow-based network model implementation that has become an important part of ProtoPeer.

9. REFERENCES

[1] Jist/swans. http://jist.ece.cornell.edu/, Mar 2008.

[2] Narses. http://sourceforge.net/projects/narses, Mar 2008.

[3] Neurogrid. http://www.neurogrid.net/, Mar 2008.

[4] ns2 network simulator. http://www.isi.edu/nsnam/ns/, Mar 2008.

[5] Overlay weaver. http://overlayweaver.sourceforge.net/, Mar 2008.

[6] P2psim. http://pdos.csail.mit.edu/p2psim, Mar 2008.

[7] Peersim. http://peersim.sourceforge.net/, Mar 2008.

[8] Planetsim. http://www.planetsim.net/, Mar 2008.

[9] C. Arad, O. Kafray, A. Ghodsi, S. Haridi, N. Finne, J. Eriksson, A. Dunkels, and T. Voigt. GODS: Global observatory for distributed systems. Technical Report Technical Report T2007-09, Swedish Institute of Computer Science.

[10] H. Casanova, A. Legrand, and M. Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In *10th IEEE International Conference on Computer Modeling and Simulation*, Mar. 2008.

[11] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, 2003.

[12] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *NSDI*, pages 85–98. USENIX, 2004.

[13] W. Galuba and K. Aberer. Generic emergent overlays in arbitrary peer identifier spaces. In *2nd International Workshop on Self-Organizing Systems (IWSOS 2007)*, volume 4725, pages 88–102, 2007.

[14] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. *SIGPLAN Not.*, 42(6):179–188, 2007.

[15] F. Klemm, J.-Y. Le Boudec, and K. Aberer. Congestion Control for Distributed Hash Tables. In *The 5th IEEE International Symposium on Network Computing and Applications (IEEE NCA06)*, 2006.

[16] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. *SIGOPS Oper. Syst. Rev.*, 39(5):75–90, 2005.

[17] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[18] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric, 2002.

[19] S. Naicken, B. Basu, A.and Livingston, and S. Rodhetbhai. A survey of peer-to-peer network simulators. In *Proceedings of The Seventh Annual Postgraduate Symposium*, 2006.

[20] 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.*, 37(2):95–98, 2007.

[21] M. Quinson. GRAS: A research & development framework for grid and P2P infrastructures. In *International Conference on Parallel and Distributed Computing and Systems*, 2006.

[22] H. Song. The MicroGrid: A scientific tool for modeling Computational Grids. *Scientific Programming*, 8(3):127–141, 2000.

[23] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM'01*, pages 149–160, 2001.

[24] P. Urban, X. Defago, and A. Schiper. Neko: a single environment to simulate and prototype distributedalgorithms. In *Information Networking, 2001. Proceedings. 15th International Conference on*, pages 503–511, 2001.

[25] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. *SIGOPS Oper. Syst. Rev.*, 36(SI):271–284, 2002.

[26] 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. pages 255–270. Boston, MA, Dec. 2002.

[27] W. Yang and N. Abu-Ghazaleh. Gps: a general peer-to-peer simulator and its use for modeling bittorrent. *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005.*, pages 425–432, 27-29 Sept. 2005.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5681
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5681