MyP2PWorld: Highly Reproducible Application-level Emulation of P2P Systems

Roberto Roverso[1,2], Mohammed Al-Aggan[1], Amgad Naiem[1], Andreas Dahlstrom[1],
Sameh El-Ansary[1,3], Mohammed El-Beltagy[1,4] & Seif Haridi[2]
[1]Peerialism Inc., Sweden,
[2] The Royal Institute of Tech. (KTH), Sweden,
[3]Nile University, Egypt,
[4]Cairo University, Egypt
{roberto,sameh}@peerialism.com

Abstract

In this paper, we describe an application-level emulator for P2P systems with a special focus on high reproducibility. We achieve reproduciblity by taking control over the scheduling of concurrent events from the operating system. We accomplish that for inter- and intra- peer concurrency. The development of the system was driven by the need to enhance the testing process of an already-developed industrial product. Therefore, we were constrained by the architecture of the overlying application. However, we managed to provide highly transparent emulation by wrapping standard/widely-used networking and concurrency APIs. The resulting environment has proven to be useful in a production environment. At this stage, it started to be general enough to be used in the testing process of applications other than the one it was created to test.

1 Introduction

The Case. MyP2PWorld is an application-level emulator with a focus on high reproducibility and simple integration with production code. The need for yet-another emulation/simulation package arose from the fact that we needed to provide an environment for debugging, testing, and evaluation of an *already-developed* product. Thus MyP2PWorld had to conform to the application rather than the converse. Existing emulators either did not provide enough features for our needs or required major re-engineering of the existing product. Our approach was to adapt an expressive-enough Discrete Event Simulator (DES) that was initially used in the algorithm design phase and develop a translation layer that enables the production code to run on top of it.

The Product Under Test. Peerialism's product is a content distribution platform which performs audio and video streaming directly to the customer's home computer. It does that by building an ad-hoc overlay network between all hosts requesting a certain stream. This network is organized in such a way that the load of the content distribution is shared among all the participating peers. The main entities in the system are:

- The Clients, which are the peers where Peerialism's client application has been installed, i.e. the customers home computers. The installed application requests audio and video streams according to the input received from the customer. It then receives streams from other peers, delivers them to the local media player and streams them once more to other customers.
- The Source. It represents a host which has all data of a certain stream. The Source itself is a Peer. A Peer becomes a source for a specific stream when it has received all the data of that same stream.
- The Tracker. It is the central coordinator of the system. It is not part of the overlay network but it organizes it. It receives requests from the clients, forwards them to an optimization engine and issues directions to the peers once the request has been satisfied.
- The Optimization Engine. It receives the forwarded requests from the tracker and performs decisions according to the overall state of the network. In addition, it periodically redefines the structure of the overlay network to normalize the load of the delivery among the peers.

2 Our Requirements

Our requirements for a testing environment are:

- **Single code base**. This is a widely sought-after goal in P2P systems research, mainly, due to the fact that initial design of algorithms and parameter trade-offs are studied on a discrete-event simulator, which uses a totally separate code base from the production code. The need for a single code base has even more value in an industrial context where people who design and simulate the protocol (Researchers), are different from those who deliver the production-quality software (Developers). The main issue, while scientifically unprovable but anecdotally evident, is that when one designs a protocol and specifies it for others to implement, some intuitive or based-on-trial/error design decisions are implicit. When given to another person the question of "Why don't we do it the other way?" always becomes an issue and there is no fast way to answer that, except rapid prototyping, especially when it comes to non-obvious second-order effects. A single code-base is a valuable catalyst for the rapid prototyping process.

- **High reproducibility.** We need to be able to execute the same experiment many times while preserving the same sequence of events and the same output every single time. This is mainly for debugging and inspection purposes rather than evaluation purposes.

- **Ease of deployment.** The ability to use the testing tool on every development and testing machine. That is, we want to avoid the slow cycle of develop-deploy-inspect using different development and deployment machines. Especially, if the deployment infrastructure needs to be shared among many developers.

- **Minimal changes.** We are testing a software that was already developed, therefore we are constrained by the way it was built. That is, whatever tool we choose, we want it to have a minimal impact (preferably none) on the present software architecture.

Having explained our requirements and our constraints, we will show, in the next section, that despite the abundance of existing tools, we were not able to find one which can simultaneously address our requirements and constraints.

3 Existing Tools

The testing of P2P systems production-code (ideally the same code as the simulation code) has been the motivation behind many tools in the research community. We enumerate here some of these tools and explain their desirable properties as well as their shortcomings.

TestBeds. The prominent example in this category is the Planet-Lab testbed [9]. It is one the most-widely used tools and an indispensable one. It is probably as close as one can get to a real P2P deployment. The main problem is the difficulty of debugging due to the lack of reproducibility. The problem is also exacerbated by the huge fluctuation of connectivity and computational resources. A testbed like Planet-Lab can not be replaced by other tools however there is a strong need to complement it.

Kernel-Level Emulators. Examples include systems like Modelnet[10] and NCTUns[11]. The main idea is to use the kernel to intercept network traffic and manipulate it to emulate the conditions of a physical topology. Total transparency to the overlying application is one of the strongest advantages of this approach. The main disadvantages are: i) A rather involved deployment process and the need to have a dedicated infrastructure for it, ii) While the emulated network behavior is repeatable in terms of delay, congestion, and packet loss etc., the fact that each Peer lives in a separate (and most likely multi-threaded) OS process violates the high reproducibility requirement.

Application-Level Emulators. Examples include systems like EmuSocket[1] and WiDS [8]. The main idea here is similar to Kernel-level emulators. Interception of network events is accomplished by providing to the application an interface that resembles the standard network APIs. Thus, transparency is partial due to the need for slight modification of the application code. The approach retains the lack of reproducibility property due to the same reasons as Kernel-Level Emulators, namely the control of the operating system on the concurrency. However, deployment is much easier and does not need any dedicated infrastructure.

Replay Debugging. Such tools tackle the issue of reproducibility by recording the execution of all network and concurrency events. The recorded events can be replayed in a deterministic way thus enabling complete reproducibility. The way of achieving this may vary. For instance, in Liblog[3] call to `libc` are intercepted and recorded in a causality preserving fashion. In that way Liblog, it's a perfect complement to Planet-Lab for recording and replaying a specific test run. Thus, it cannot be used for replaying the same experiment in different network conditions after code changes. In [7], an internal Microsoft software, real code is generated from a model written using a specification language. Executions of the generated code could then be recorded and replayed as in the case of Liblog. However, adopting it would require a complete re-writing of the application using the WiDS model, which is not a feasible solution in our case. Moreover, the main disadvantage of both is that they are restricted to the C/C++ programming language.

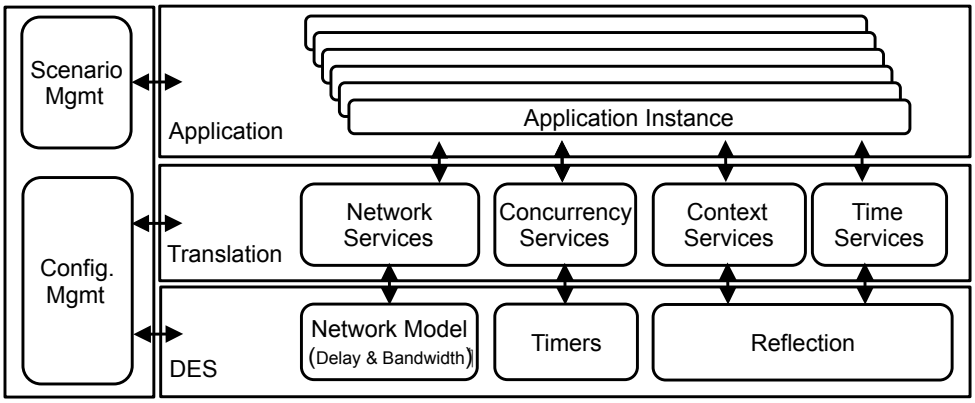

Figure 1. MyP2PWorld Architecture

4 Our Approach

Our approach resolves the lack of reproducibility problem of application-level emulation. The main novelty is that we do not stop at emulating the network behavior but we go further into taking control over concurrency and system time. The main idea is that the same code could be executed in emulated and real mode. Real mode means network events are sent to a real network, concurrency and time are provided by the OS. Emulated mode means that network events, concurrency and time are all controlled by a discrete-event simulator. To explain how we realized our approach we have to briefly outline how network communication, concurrency and system time are realized by the already-existing production code and what is needed to create a corresponding emulation environment.

Network Communication. The application depends on Apache MINA[5], a high-performance Java networking framework. It provides an event-driven API on top of Java non-blocking I/O libraries. It has many advantages such as filter chains and decoupling of marshaling formats from communication logic among other things. It provides a threading model to control the number of threads dedicated for network I/O. Creating a corresponding emulation environment requires that we keep all application code that depends on MINA interface intact while providing an alternative implementation. This is very similar in nature to the typical case of providing an emulated socket implementation except that it is done on the level of MINA rather than on the level of TCP/UDP sockets.

Concurrency. Aside from MINA threads, the application has a number of threads for scheduling of periodic activities and timeouts. To emulate concurrency, we need to preserve the programming interface for creating and running threads while redirecting their scheduling to the discrete-event simulator.

System Time During emulation mode, time is measured in simulated time units. However, the application code, defines time quantities such as the length of a timeout period in real time units. Therefore, again, for transparency's sake, care has to be taken to provide a proper correspondence between simulated and real time units.

5 System Architecture

MyP2PWorld is organized into four layers:

Discrete-Event Simulation (DES) Layer: Provides simulation time and network model and is not visible to the real application.

Translation Layer: Provides to the real application an interface that looks like real network/OS services however that get routed to the DES instead of being routed to the corresponding network/OS services.

Real Application Under Test: Multiple instances of the real application that got minimally modified to use the translation layer.

Scenario Management Layer: The main execution entry point. Responsible for taking as input a scenario file and configures all layers such as forking and killing instances of the peers at specified times, configure network behavior etc.

5.1 Discrete-Event Simulator

This layer could be (and in fact has been) used on its own as a traditional simulator. Every simulated node has access to a timer abstraction where it can schedule events in the future. For the network model, the most important feature is

the bandwidth model because it is crucial to studying content distribution protocols. Our work has mainly been inspired by BitTorrent simulators such as [2] and [12]. However, we have worked on providing a compact, explicitly-specified model with an efficient implementation. We won't delve into the traditional details of the DES, however we will briefly describe our bandwidth model.

5.1.1 Bandwidth Model

Given a peer, we assume that its upload and download bandwidths are independent. Consequently, we logically split the peer into two separate entities: a sender S which controls the upload bandwidth and a receiver R that controls the download bandwidth. Once the sender starts sending a block of data, the network should try to send the block at the maximum possible speed between the two parties. While the piece is in transit we say that S and are R have an ongoing "transfer". Naturally, the transfer of a certain block is affected by other transfers taking place between S or R and any other third party. The main quantities needed for the description of the model are: β the maximum bandwidth of a party, α the available (free) bandwidth of a party, and τ set of ongoing transfers of party.

Bandwidth allocation Each time a block is sent, i.e. a new transfer t is started, the amount of bandwidth $bw(t)$ that is given to the new transfer is equal to:

$$bw(t) = min \begin{pmatrix} max\left(\alpha_S, \frac{\beta_S}{|\tau_S|+1}\right), \\ max\left(\alpha_R, \frac{\beta_R}{|\tau_R|+1}\right) \end{pmatrix} \quad (1)$$

Having determined $bw(t)$, allocating it might require the "squeezing" of ongoing transfers at either/neither/one of the sides. At a given side where squeezing is needed, there is a certain amount of bandwidth $\pi = bw(t) - \alpha$ that need to be collectively deducted from the ongoing transfers to make room for the new connection t. A transfer gets a deduction only if it is using more than its fair share $f = \beta/(|\tau| + 1)$. Note that $f = bw(t)$ for at least one side, but might not be true for the other side. Let $\tau' = \{x \in \tau : bw(x) > f\}$ be the set of transfers that are taking more than their fair share. We only deduct from transfers in τ'. However we need to figure out how to collectively deduct π from members of τ'. Let $e_x = bw(x) - f, \forall x \in \tau'$ be the extra amount of bandwidth that a connection x is taking beyond its fair share. Let $bw'(x)$ be the new bandwidth of a transfer x after deduction, $bw'(x) = bw(x) - (e_x/\Sigma_i e_i)\pi$. That is, the connections are squeezed proportional to their extra bandwidth, which guarantees that no connection is squeezed to less than its fair share.

Needless to say, when the bandwidth of a transfer is squeezed, the delivery time of the transferred block is rescheduled to a later point in time proportional to the amount of squeezed bandwidth and the duration of time the block stayed in transit before the squeeze occurred.

The effect of allocating a new transfer goes beyond the two involved parties, because a transitive chain of readjustment is triggered. A squeeze of a transfer on the sender or the receiver frees some bandwidth on some third-party. Consequently, the third party would experience an effect similar to transfer deallocation because some bandwidth was freed on its end. This would result in its turn in the boosting of some of its ongoing transfers which would affect a fourth party and so forth. This process can take some iterations to converge. Ultimately, all bandwidth that could be utilized (respecting the nodes configurations) will be allocated. However, the process can suffer from the fact that the adjustments are very small quantities. Accepting a low threshold of as low as 2% of unutilized bandwitdth usually results in quick convergence.

5.2 Translation Layer

This layer is actually the core layer of MyP2PWorld and provides three core functionalities:

5.2.1 Network Services

Apache MINA is situated between the application and the Java NIO network APIs. It exposes to the application an event-driven interface. We preserved this style of interaction with the application and redirected all interactions to/from the real network that we initially passing through Java NIO to the DES layer instead.

Listing 1 shows the skeleton of a minimal TCP server. As we can see, the changes are limited to modifying the import line from the original to the modified version of the MINA APIs.

Listing 1. MINA TCP server with minimal changes that enable switching between real and emulated modes with minimal code changes

```
import org.apache.mina.common.IoAcceptor;
import ~~org.apache.mina.transport.nio.SocketAcceptor~~ com.peerialism.simpipe.SocketAcceptor;
import java.net.SocketAddress;
....
SocketAddress serverAddress = new SocketAddress(''localhost'', 1234);
IoAcceptor    acceptor = SocketAcceptor();
acceptor.bind(serverAddress, new IoHandlerAdapter(){
   public void messageReceived(IoSession session, Object message){...}
   public void messageSent(IoSession session, Object message){...}
   public void sessionClosed(IoSession session){...}
   public void sessionCreated(IoSession session) {...}
   public void sessionIdle(IoSession session, IdleStatus status){...}
       ...
   });
....
```

5.2.2 Concurrency Services

The main issue with taking control over concurrency is to eliminate all OS threads while in emulation mode, without changing the production code of the already-developed application. The approach to support this requirement is to have all concurrent events as atomic non-blocking actions. This style is already supported and advocated in Java since version 1.5 by using futures and executors. Futures are abstractions for representing the results of an asynchronous operation. For instance, instead of writing a periodic activity as loop in a blocking thread, one uses a future and schedules its execution using an executor after a certain delay. The executor itself can incorporate a single thread or a thread pool. This means if one has n periodic activities, instead of having n threads, one can use a single executor which can incorporate one or more threads. We have wrapped the Java Futures and Executors classes to provide support for transparent switching between real and emulation modes. Listing 2 outlines this style of programming pattern and shows the minimal change needed by substituting the original import line with an import line that loads our wrapped future and executor.

Having said that, we also have to report that not all developers were adopting this style in the production code. So in fact a bit of refactoring was necessary. However this style was embraced by the development team and was regarded as an improvement rather than imposing an unnecessary change just to support emulation. Using this the style provided a cleaner code and simplified among other things the process of tuning the number of threads dedicated to periodic activities and timeouts.

Listing 2. Wrapping of the Future, Executor and System Time

```
import java.lang.Runnable;
import ~~java.lang.System~~ com.peerialism.SimulableSystem;
import ~~java.util.concurrent.ScheduledFuture~~ com.peerialism.ScheduledFuture;
import ~~java.util.concurrent.ScheduledThreadPoolExecutor~~ com.peerialism.ScheduledExecutor;
...
class SomeActivity implements Runnable {
   public void run() {
     // Activity
   }
}
....
SchedulingExecutor executor = new SchedulingExecutor();
SomeActivity activity=new SomeActivity();
....
// Periodic Activity
long delay;
long period;

ScheduledFuture periodicFuture = executor.scheduleAtFixedRate(
        activity, delay, period, TimeUnit.MILLISECONDS);

// Timeout
ScheduledFuture timeoutFuture = executor.schedule(...
        activity, delay, TimeUnit.MILLISECONDS);
....
```

5.2.3 System Time Services

As mentioned earlier, in emulation mode, events happen on the simulated time scale. A simulated time unit models a millisecond. We have wrapped the `System.currentTimeMillisecs()` to provide a transparent support for working with system time. Listing 2 shows that the specification of time units is transparent to the mode of operation, again by importing the wrapped libraries.

5.2.4 Context Services

Unfortunately controlling threads inside the application is not sufficient for providing high reproducibility. The main problem is that our application (like most other P2P applications) was not designed for many nodes to run in the same OS process. Global data structures like singletons and loggers are examples of major issues in this category. For that, we had to introduce to the DES layer the concept of a "context", i.e. when a node is created, it has to request from the DES layer the creation of a context labeled by a unique id of the node. When the time comes for an event to be fired, the scheduler switches to the context of the executing node and we expose to the application the service of querying the emulation layer about the current context. Using the context services, singletons and loggers of all nodes were able to coexist in the same OS process as described below.

A singleton, in real mode, stores one instance of an object. In emulated mode, singletons were made to store sets of objects indexed by context ids, every time a singleton is requested to return an instance, it calls the scheduler to know in which context it is running and returns the corresponding instance. The above was a quick solution which does not satisfy the requirement of transparency, however we are working on a better solution using the Java Class loaders.

For logging, the product was using the Slf4j[6] package whose purpose is to provide a standard logging interface to the application. Different implementations of this interface may be used. We produced our own context-aware implementation. Therefore, that was a totally transparent change from an application point of view.

Other minor issues like port numbers, file locations, etc. were solved using configuration parameters.

6 Related Work

As we explained in section 3, the main approach in application-level emulation is to focus on taking control over network communication and leaving concurrency and system time in the hands of the operating system. The exception of this was the work in RealPeer[4] which, independently, raised the issue for the need to take control over

concurrency and system time. The difference between our work and RealPeer is that we try to achieve this goal be wrapping APIs that are either standard or already widely-used by developers, while RealPeer tries to achieve the same goal by requiring the application to use their framework which has been designed to be comprehensive and generally-applicable as much as possible. One can argue that both approaches have their merits depending the conditions of each project.

7 Conclusion & Future work

In this work, we have provided a case study summarizing our experience with improving the testing/evaluation process of an already-developed P2P application. Our requirements for such an environment were: minimal changes of the production code, ease of deployment, and high reproducibility. By inspecting the state of the art, we found that we could not find a tool that simultaneously satisfies all the requirements. Therefore, we created our own. Our approach was to adopt application level-emulation but with ensuring high reproducibility by controlling concurrency and system time. The resulting environment entitled "MyP2PWorld" has been used for a number of months for testing Peerialism's P2P live streaming solution. MyP2PWorld has resulted in huge improvements in product quality and bug discovery rate and became an integral part of the testing process. While we have initially developed MyP2PWorld as a tool to complement the testing and evaluation process of a particular product at Peerialism Inc., we are now trying to provide MyP2PWorld as an open source tool on its own that could be used in other projects.

We are currently in the process of adding the following features: achieving complete transparency for the context services, improving performance by adding a parallel scheduler, augmenting our bandwidth model to provide an enhanced behavior for UDP communication, adding recording and selective replay features.

8 Acknowledgments

We would like to thank Nils Franzen and Magnus Hedbeck who were the primary users of our work and who have given valuable input that was indispensable for making MyP2PWorld a usable tool in a production environment.

References

[1] Marco Avvenuti and Alessio Vecchio. Application-level network emulation: the emusocket toolkit. J. Network and Computer Applications, 29(4):343–360, 2006.

[2] Ashwin R. Bharambe, Cormac Herley, and Venkata N. Padmanabhan. Analyzing and improving a bittorrent networks performance mechanisms. In INFOCOM. IEEE, 2006.

[3] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference, pages 27–27, Berkeley, CA, USA, 2006. USENIX Association.

[4] Dieter Hildebrandt, Ludger Bischofs, and Wilhelm Hasselbring. Realpeer–a framework for simulation-based development of peer-to-peer systems. In PDP '07: Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, pages 490–497, Washington, DC, USA, 2007. IEEE Computer Society.

[5] The Apache Mina Java Netwokring Library. http://mina.apache.org.

[6] The Slf4j Java Logging Library. http://www.slf4j.org.

[7] Shiding Lin, Aimin Pan, Zheng Zhang, Rui Guo, and Zhenyu Guo. Wids: an integrated toolkit for distributed system development. In HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems, pages 17–17, Berkeley, CA, USA, 2005. USENIX Association.

[8] Kazuyuki Shudo, Yoshio Tanaka, and Satoshi Sekiguchi. Overlay weaver: An overlay construction toolkit. Computer Communications, 31(2):402–412, 2008.

[9] The Planet-Lab Testbed. http://www.planet-lab.org.

[10] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeffrey S. Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. In OSDI, 2002.

[11] S. Y. Wang, C. L. Chou, and C. C. Lin. The design and implementation of the nctuns network simulation engine. Simulation Modelling Practice and Theory, 15(1):57–81, 2007.

[12] Weishuai Yang and Nael B. Abu-Ghazaleh. Gps: A general peer-to-peer simulator and its use for modeling bittorrent. In MASCOTS, pages 425–434. IEEE Computer Society, 2005.