Beyond Simulation: Large-Scale Distributed Emulation of P2P Protocols

Nathan S. Evans
Technische Universität München, Munich, Germany
Email: evans@net.in.tum.de

Christian Grothoff
Technische Universität München, Munich, Germany
Email: grothoff@net.in.tum.de

Abstract

This paper presents details on the design and implementation of a scalable framework for evaluating peer-to-peer protocols. Unlike systems based on simulation, emulation-based systems enable the experimenter to obtain data that reflects directly on the concrete implementation in much greater detail. This paper argues that emulation is a better model for experiments with peer-to-peer protocols since it can provide scalability and high flexibility while eliminating the cost of moving from experimentation to deployment. We discuss our unique experience with large-scale emulation using the GNUnet peer-to-peer framework and provide experimental results to support these claims.

1 Introduction

The outcome of a network security experiment can vary significantly depending on whether the experiment was based on simulation or emulation [5]. While both methods can provide new insights, there is a dearth of scalable approaches for assessing large-scale peer-to-peer (P2P) networks using emulation. While some studies [9] have run attacks against deployed P2P networks, there is a clear benefit to being able to run large-scale experiments without potentially negatively impacting actual users.

This paper presents a design and implementation for large-scale experiments with P2P protocols using distributed emulation. The key insight behind this paper is that while simulators can achieve significant scalability by abstraction, emulators for P2P networks can achieve comparable scalability through parallelization and distribution. By distributing computations, a modest computer laboratory can achieve performance gains of an order of magnitude over a single machine for suitable problems. Similarly, due to the advent of many-core processors, local computational resources are often only limited by the amount of parallelism in the problem. While parallelization and distribution of simulators can be difficult, distribution is inherent in P2P networking, making it easier to create distributed P2P emulators.

Emulation has many advantages over simulation: the code used for an experiment can be the same code used for deployment, and programming an appropriate model with abstractions is unnecessary. The emulator can be used to easily evaluate the entire system, not just individual components; as a result, the experimental setup can be used to evaluate performance and security issues as well as serving as an integration testbed. Given problems on a deployed system, modifying experiments using emulation to reproduce, measure and evaluate the undesired behavior is generally easier than doing the same using experiments through simulation. In fact, depending on the abstractions chosen, a simulation may fail to model observed real-world problems. This is even more applicable for security assessments as abstraction eliminates implementation details, and thereby sources of vulnerabilities.

P2P simulators typically simulate tens of thousands to hundreds of thousands of peers [11,16,20,23], with some distributed simulators reportedly scaling up to 80 million peers [6]. While distributed emulation may be the natural choice for experiments with distributed systems, it is still not at all obvious that emulation actually would scale to the desired problem sizes. Previous work on emulation falls far short of the scale managed by simulations; the best-performing previous emulation setup that we are aware of has been reported to scale to only 4,096 peers [26].

This paper details our design and experiences in providing a scalable system for evaluating P2P protocols using emulation in the context of the GNUnet framework[1]. We did experiments running a 80,000 peer Kademlia-style DHT (with link-encrypted P2P communication) using a small cluster of 32 machines with 7 GB of memory

[1] `https://gnunet.org/`

each; our results indicate that with proper tuning, emulation can be scaled to much larger sizes than previously achieved without sacrificing the ability to run realistic experiments.

Section 2 describes our high-level design goals, delimiting the scope for our design. Key design properties of the GNUnet P2P framework are described in Section 3 including extensions made to GNUnet for large-scale emulation presented in Section 3.4, followed by the lessons we learned tuning the system in Section 4. Section 5 presents some of the results obtained from our experiments. Finally, Section 6 compares our setup with other emulators and simulators for P2P networks.

2 Design Goals

The primary requirement for our experimentation framework is that it must be distributed, taking advantage of the inherent properties of P2P networks to spread computational load. Additionally, we require the ability to run many peers on the same host, making use of multiple cores and taking into account the fact that a single peer would rarely use the entire computational or storage resources of an individual computer.

We run the original P2P software directly on top of the operating system. Instead of providing an operating system and network abstraction layer below the application, our design is to issue commands to the various layers of the P2P software to achieve the desired environmental constraints (Figure 1). In order to achieve reasonable performance, we run peers as independent processes and assume no control over when the operating system gives CPU time to peers. As a result, a major limitation of our design is that our emulator cannot produce timing-accurate results. Similarly, our current framework does not directly support emulation of network delays.

Given that an emulation is executed using a cluster (or single host) with universal connectivity, an important feature for realistic experiments is the ability to impose restrictions on which peers should be able to communicate directly. Existing P2P network simulators often only allow topology configuration at very low levels [17] such as configuring AS links or link delays. None of the DHT-capable simulators [1,2,20] that we have encountered allow user defined underlay topology restrictions, such as those which result from firewalls, network address translation (NAT) or topologies based on personal trust relationships, like those extracted from Facebook [30]. In contrast, our system can be configured to impose constraints on underlay topologies; we also provide various algorithms to construct common topologies such as cliques, Small-World networks and Internet-like graphs.

A primary goal of any experiment is to collect relevant data. Our system supports adding custom instrumentation to log results to a file or to a database. However, given a sufficiently large number of data points, such logging activities can become the bottleneck for an experiment. To mitigate this potential issue, we provide a scalable integrated facility to log, accumulate and store simple numeric values collected during an experiment using a single function call. Collection of these statistics can be performed in a distributed or centralized manner.

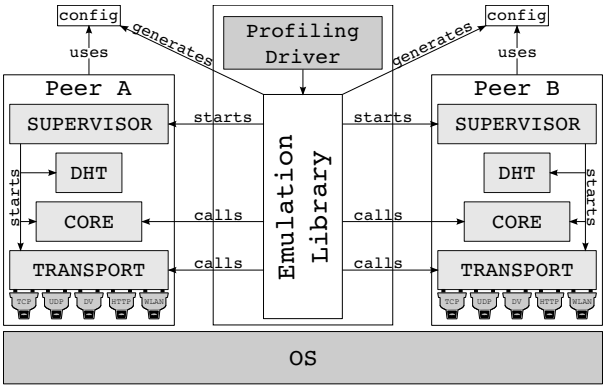


Figure 1: Design of the emulation system (key components only). A driver for the experiment uses the emulation library to issue commands to the various layers or the GNUnet framework to achieve the desired experimental setup and to obtain performance metrics. Many GNUnet peers run as independent process groups on the same host.

Finally, one common criticism of most P2P network simulators and emulators is their steep learning curve, which can lead to strange results. The authors of [21] note that two different simulators running the same experiment on a five node Chord network produced inexplicably different results. Using emulation shifts this problem from understanding the simulator to understanding the underlying implementation, written as part of the GNUnet P2P framework. The next section will describe key features of GNUnet, arguing why we believe that this will ease usability problems.

3 The GNUnet P2P Framework

The GNUnet framework is a GNU software package for secure P2P networking. The package integrates about a dozen different services for building P2P applications. A GNUnet service consists of a client API and a server process. The client API uses a service-specific protocol to access the server that keeps the global state for the service and provides the respective functions to many clients.

New services and P2P applications can be built by

either accessing the APIs of existing services or re-implementing the client-side of the respective service-specific protocol. Extensible services use user-defined protocol numbers to channel requests to client(s) responsible for the extension. Messages that no client has claimed responsibility for are discarded.

A result of this architecture is that each individual peer typically consists of roughly a dozen processes that make extensive use of interprocess messaging. Each peer's process group is coordinated using a supervisor service which is primarily responsible for starting and stopping other service processes. Services can be started automatically on-demand by the GNUnet supervisor service. As a result, functionality that is not needed for a particular application or experiment requires no run-time resources.

3.1 Services in GNUnet

Important general-purpose services in the GNUnet framework include the transport service (which provides direct, unencrypted P2P communication), the core service (which provides direct, encrypted P2P communication), the statistics service (which tracks performance metrics), the peer information service (which tracks peer addresses), and the distributed hash table service (for distributed storage of key-value pairs).

GNUnet provides some basic level of security for all applications using the core service. Each peer is identified via a public-private key pair (using 2048-bit RSA). Connections between peers are link-encrypted and authenticated using AES-256 and SHA-512. On the link-layer, the transport service signs network addresses, ensuring that peers do not send (encrypted) traffic to addresses other than those controlled by the intended destination.

The transport service also communicates and — as much as technically feasible — enforces bandwidth limitations as set by the user, and assigns bandwidth based on preferences as determined by the P2P applications using GNUnet. The transport service also allows applications (and the emulation library) to impose arbitrary restrictions on direct connections between peers. It is also possible to restrict connections between peers to particular transports, which is useful when emulating transport constraints on P2P communication. As a result, the transport service of the GNUnet framework allows our emulation library to exert control over P2P communications that other emulation designs typically enforce via virtual network interfaces and virtual machines.

3.2 The *gnunetutil* Library

All components of the framework build on top of the shared *gnunetutil* library which provides a portable event-loop as the key execution driver. Blocking operations are generally only allowed in dedicated processes; APIs typically provide asynchronous calls instead of blocking and the use of continuation passing is common. The *gnunetutil* library also provides portable hardened wrappers around common `libc` operations, common data structures, cryptographic primitives, logging operations, network and interprocess communication and configuration management functions.

3.3 Scalability Benefits of the GNUnet Architecture

Minimization of memory consumption is critical for our system to scale. GNUnet is implemented in C, avoiding the large memory footprint of systems written using managed languages. C code is compiled prior to execution; hence, the operating system can use the same pages in real memory for the respective read-only code and data segments of different processes. Because only the required service processes are started, it is typical that less than half of the binary code in the framework is loaded into memory at run-time. Furthermore, memory consumption for the heap is reduced by avoiding garbage collection overheads.

This architecture has several key advantages. First, it isolates faults within components, making it easier to diagnose problems. Second, each peer can make use of many cores. Given that we run thousands of peers per host resulting in tens of thousands of processes, this architecture is suited to modern-day many-core processors [24]. Finally, a major advantage of the multi-process architecture is that new components can, in theory, be written in other languages. While certain other languages may incur significant performance penalties for large-scale experiments, this facility may still be beneficial from a usability perspective.

3.4 The Emulation Library

We control large-scale emulation experiments using an experiment-specific driver that is linked against our GNUnet emulation library; the library is used by the driver to set up the testbed. The emulation library is accessed via a layered API. The low-level API provides functions to start and stop individual peers, to explicitly connect pairs of peers, and to change the configuration of running peers. For P2P security evaluations, the ability to dynamically reconfigure peers at runtime can be a valuable tool: in some of our experiments, we use it to configure a subset of the peers to become adversaries and start executing different attack vectors. The high-level API allows groups of peers to be started across multiple

hosts, automatic generation of a range of network topologies and induction of network-wide churn at configurable rates. The API also provides means for accessing the state of each individual peer, including access to the current network topology and statistics logged by peers.

The high-level API is given a configuration file as a template for generating an initial configuration for each emulated peer. The library itself primarily adjusts options such as port numbers that must be unique per peer. `ssh` is then used to copy configuration files to execution hosts and to start peers on those systems. Hostnames or IP addresses of hosts which are to be used for emulation are also specified in this high-level configuration. Peers run on these hosts use this hostname or IP address for connections; virtual networks are not supported by our framework. Of course, utilizing other virtual network environments underneath our emulation framework is possible.

3.4.1 Network Topologies

Our emulation library can connect peers using a diverse set of specific topologies. The library distinguishes between the underlay topology, which specifies transport-level connections, and the overlay topology, which specifies peer-level connections. Whitelisting connections (explicitly allowing particular peers to connect) and blacklisting connections (explicitly disallowing particular connections) is also supported. Finally, the emulation API allows specification of an initial set of connections that peers should have at the beginning of the experiment in lieu of executing the network's ordinary bootstrapping phase. In the emulation API, these different requirements on network topologies are reflected by allowing the user to specify the different graphs or constraints for each of these topologies.

Our framework supports specification of each of the topologies listed in Table 1. Our intent is to provide well-known topology generators for experiments involving both highly structured and random topologies. Note that some of the supported topology generation algorithms require specification of arguments to control their construction. For example, randomized graph construction requires an argument indicating the probability for establishing a link.

The "file" topology is the most expressive option supported. This topology requires an argument indicating the path to a topology specified in the METIS [15] file format. We allow topologies to be specified in these files both for easy import from other topology generators and for consistency in testing. The emulation library provides utility functions to export the topology of an active peer group to a file. This file can be used to recreate the exact same topology during subsequent experiments, or repetitions of the same experiment.

Table 1: Built-in generators that can be used for supported topologies.

Topology	**Description**
Clique	Connects all peers
Line	Connect peers in line
Ring	Connects peers in line with wraparound
2d-Torus	2d-grid with wraparound
Erdos-Renyi	Random graph construction
Small World (ring)	Ring topology with additional, randomized long distance connections
Small World (2d-torus)	2d-torus topology with additional, randomized long distance connections
Scale-Free	Scale-free topology
InterNAT	Clique with some peers restricted to outgoing connections (similar to unpunched NAT)
File	Read topology from file

3.4.2 Executing Experiments

In order to use our emulation library for experiments, a P2P algorithm designer would first implement their application as a GNUnet service. The implementation burden is eased by the large amount of low-level P2P functionality provided by extant services and the *gnunetutil* library. Next, an application specific profiling driver which utilizes the emulation library would need to be created, again with the benefit of building on existing profiling and testing drivers. While implementers would have to learn the GNUnet framework (as they would have to learn a simulation framework), the benefits of services provided by GNUnet and a single *usable* implementation likely outweigh this overhead.

Execution of an experiment using the library via the high-level API typically proceeds in six phases. During the first phase, a hostkey is generated for each peer; at this time, the emulation driver is optionally notified of the identity (the hash of the public key) of each peer. This allows the controlling process to keep track of peer identities for later peer identification and lookup in the peer group. In the second phase, the desired network topology is computed and per-peer configuration files that specify the desired topological constraints are created. These constraints specify which connections are allowed at both the underlay and overlay level, but not which connections will actually be used. Peers are started in the third phase, and connected using the so-called *initial* network topology in the fourth phase. After all of these initial topology connections have been established, in the fifth phase, the driver performs actions specific to the experiment, during which time the network topology may evolve, configurations for individual peers can be

modified, churn can be induced, and benchmarks may be executed. In the final phase, the emulation library is used to shut down all of the peers. Peers are shut down using (fresh) `ssh` connections to the execution hosts.

3.4.3 Limitations

The emulation library exerts only very limited control over the execution of the processes in the testbed. Thus, the results are generally not timing-accurate, especially if the execution hosts run large-scale experiments. Using a virtual machine (VM) per peer with quality-of-service guarantees enabled at the VM level would eliminate this restriction; however, this would also greatly reduce the scalability of our approach.

While our design may not provide timing accuracy, we believe this is unnecessary for many important experiments. In P2P networks, individual peers often have unpredictable and highly variable amounts of resources at their disposal. A robust design should thus be able to deal with such fluctuations and not fail (for example due to timeouts) due to minor timing variation. In terms of security experiments, practical denial-of-service attacks should still show their impact on system behavior even with slight timing deviation. Many other attacks, such as those on data integrity, are also typically not impacted by changes in the timing of events. Finally, even for experiments that are inherently sensitive to timing (such as traffic analysis for deanonymization), increased fluctuations may in some cases support the robustness of the technique.

In terms of the ability to observe details of emulation execution, the emulation library provides functions to manage the experiment at the level of peers (configure, start, connect, stop) and the level of the peer group (iterate topology, gather statistics). These functions are used in concert with high-level API interactions with the service that is the subject of the experiment. However, some experiments may require more specific insights into the behavior of the network, especially at the lower layers. This can be achieved by directly connecting to the respective service APIs for the lower layers. For example, the core service provides an API for notification of every message that is sent prior to encryption or received after decryption. Similarly, encrypted traffic can be intercepted from the transport service API.

For our experiments with the DHT (Section 5.1), the DHT service API (with its purposely simple PUT/GET abstractions) did not provide the desired data collection capabilities. Here, we directly instrumented the DHT for data collection (Section 5.2). This is likely better for designers of new P2P applications; as we discuss in Section 6, it can be problematic to have the framework gather data that is not specific to the application.

4 Lessons Learned

The development process for our system was iterative: as we scaled up the network size, new bottlenecks would emerge and had to be dealt with before we were able to increase the size again. This section summarizes the most salient lessons learned about emulating large-scale P2P networks.

4.1 Cryptography

Cryptographic operations, in general, can be expensive, and given that many modern P2P networks use asymmetric key pairs for host identification and other core functionality, repeated calls to cryptographic functions (e.g. key generation) can have significant overhead.

Our experience during development was that the creation of strong private keys, even for a few dozen peers, virtually always depleted the entropy pool of the system, causing excessive delays. Our initial response was to disable entropy gathering; however, creating tens of thousands of 2048-bit RSA keys for each experiment still took a significant amount of time, even using a cluster.

We solved this problem by pre-computing a large number of public-private key pairs and reusing them between experiments. Despite the use of rather expensive cryptographic primitives, we did not have to simplify or eliminate other cryptographic operations. This also serves to make our emulations more realistic.

For emulating P2P systems that require cryptography, there are two lessons to be learned here. First, strong key-generation operations (which generally have little impact for end-users in terms of system performance) need to be simplified even for small-scale emulation experiments. Second, assuming protocols are reasonably well-designed, other typical cryptographic operations do not need to be simplified even for large-scale emulation experiments.

4.2 Execution time

One important discovery we made was that when running tests at a large scale, tasks scheduled to run at a fixed frequency are problematic, especially if their number increases with an experimental variable.

As a concrete example, when crossing the 5,000 peer threshold, our experiments were stuck at a total of 200K connections or less; any increase caused the connection process (and our test host) to grind to a halt. The reason, we discovered, was that the CPU became pegged processing latency estimation tasks. These tasks were initially set to run at a seemingly harmless frequency of once per minute per connection. With around 40

connections per peer and 5,000 peers, these tasks became backed up, effectively taking over the system. In this case, our simple solution was to decrease the frequency of the latency measurements when testing, which is harmless as our emulation setup does not model network latencies (thus, these measurements would not be accurate anyway).

The overall issue is that in a general-purpose framework, virtually any hard coded value will eventually cause problems. As such, the solution is to either make the code adaptive — for example, our system uses exponential back-offs in many places instead of fixed retry frequencies — or at least configurable by users (preferably with clearly marked default configuration values).

4.3 Latency

One important general lesson we learned in this endeavor is that while parallelism is important for avoiding idle waiting and efficient utilization of available resources, it must always be bounded and balanced in order to avoid overly negative impacts on latency. More specifically, we discovered that once we moved to distributed operations, the latency of starting peers over the network quickly became a major bottleneck, as creating an `ssh` connection and waiting for complete startup of a single peer took up to a second — even without key generation.

To solve this issue, we first reduced the startup delay by launching only the master process (Section 3), and not waiting until the peer fully initialized. Interactions with individual peers were deferred until after all peers had been launched. Still, having to interact with each execution host for each peer created unacceptable delays of many network round-trip times. Ultimately, the latency issue for starting peers was resolved by creating a helper script that starts all the necessary peers on a particular host; using this script, we only require one `ssh` interaction per host instead of one per peer.

Additionally, because the creation of an initial set of connections between peers requires the driver to communicate the desired initial connections to those peers, another general latency issue we discovered was attempting to establish too many simultaneous connections at once. In the case of initial connection setup by the driver, we cannot fully parallelize this step, as the driver would run out of file handles. Furthermore, in general, trying to establish too many connections at a single host in parallel can peg the CPU on that host while other hosts remain idle. To solve this, our driver currently imposes configurable limits on the number of concurrent connection attempts per peer and per host; this ensures that all hosts are utilized until all connections are established.

By bounding and balancing our use of parallelism and by reducing the overall number of round-trip times during network operations (in our case, done by introducing the helper script), it is possible to effectively control many of the latency issues introduced by distribution.

4.4 Sockets

Attempts to maintain network transparency also created issues. As we scaled our implementation, we repeatedly ran into socket limitations. First, we ran out of TCP ports because each peer used a TCP port for every service. After switching to UNIX domain sockets for each peer's internal interprocess communication, we were still exceeding operating system limits when creating hundreds of thousands of TCP connections between peers on the same host. We had initially expected that UDP would be a good alternative, but we quickly discovered that UDP becomes highly unreliable even over loopback once the kernel's UDP buffer becomes too small to handle all queued messages.

As a result, our large-scale experiments are typically configured to use UNIX domain sockets for all inter-peer communication on the same host and UDP or TCP for intra-peer communication. Furthermore, TCP-based control connections between the driver and peers are only established on-demand.

The lesson we learned here is that while network transparency is nice (and in fact sometimes required for interactions with the driver), using UNIX domain sockets instead of TCP/UDP wherever possible is important for scalability. Being able to choose the communication domain is important, and the most scalable end-result is typically a mixture of UNIX, UDP and TCP.[2]

4.5 Memory

When running tens of thousands of peers, every single additional private page in memory is costly; as such, a repeatedly recurring bottleneck was memory consumption. While many specific changes to data structures were made to reduce memory consumption, the single biggest improvement was obtained by changing the default size of our communication buffers from a static 64k bytes to a minimum initial size of 4 bytes, which is then re-allocated to a larger size as necessary. Most messages passed between services and peers are much smaller than 64k, so this change resulted in significant memory reductions for most services.

Per design, each of our peers is comprised of a number of services implemented as independent processes. Depending on the nature of the experiment, some of these service processes can be shared between peers. For example, sharing the DNS resolution service is typically

[2]Unless it is germane to the experiment, whereas the emulation library can be configured to only use particular transports.

unproblematic as it has no state. We typically share one instance of such services among every 100 peers. This reduces the overall memory footprint without turning these shared processes into bottlenecks.

The general conclusion here is that being able to share memory between peers is critical for large-scale emulation, and that any non-shared memory (including heap, stack and pages for global variables) must be closely inspected. We have been able to push the amount of memory shared between peers to about 80%, which represents an improvement in scalability by a factor of five.

5 Experimental Results

We have tested our emulation framework extensively on various architectures under numerous configurations. In this section we present some basic performance data for the framework as well a small selection of the results obtained from our experiments.

Table 2 shows the results of our framework in terms of scalability (measured in the total number of peers emulated) and topology setup times for various architectures and system configurations. In terms of time, the most costly part of the emulation setup is typically the peer connection phase, as it requires cryptographic key exchange and often network communication. The speed at which connections can be established is therefore an important metric.

Table 2: Relevant performance details of our framework on various architectures.

Architecture	# Hosts (Total)	Cores (Total)	Memory (Total)	# Peers	Connections per second	Time to start peer
Cortex-A8	1	1	512 MB	100	$\sim$ 1	$\sim$ 206 ms
Xeon W3505	1	2	12 GB	2,025	$\sim$ 60	$\sim$ 12 ms
Xeon W3520	1	8	12 GB	2,025	$\sim$ 188	$\sim$ 5 ms
Opteron 8222	1	16	64 GB	10,000	$\sim$ 327	$\sim$ 27 ms
Opteron 850	31	124	217 GB	80,000	$\sim$ 559	$\sim$ 1 ms

The data shows that our framework is quite scalable, running up to 100 peers on an (embedded) ARMv7 device and 80,000 on a small cluster. As one would expect, the number of peers we can emulate correlates closely with the total amount of system memory, and the connection speed relates to the processor speed. In general, peer startup time is reduced as more cores are added, except for in the case of the 16-core host. We believe this is due to I/O limitations; starting each peer requires peer-specific file accesses to the configuration and private key files.

The most important factor in terms of scalability of our framework is the memory footprint of the processes that make up each individual peer. Table 3 shows how memory is used by the various processes that each of our peers typically uses.

Table 3: Breakdown of memory usage for the relevant services extant in our framework. Each process uses an additional 84 KB for the stack. P2P connections require about 6 KB memory with the transport service, 7 KB with the core service and an additional 1 KB if the DHT is using the connection. Our code was compiled on a 64-bit Linux system using GNU GCC version 4.3.2 optimized for size ("gcc -Os").

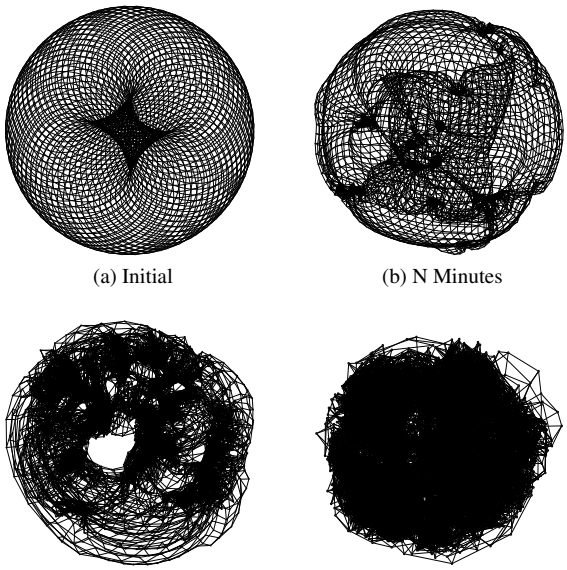
Service	Non-shared	Heap	Shared
supervisor	228 KB	32 KB	2,364 KB
transport	359 KB	99 KB	2,888 KB
core	300 KB	84 KB	2,428 KB
dht	536 KB	240 KB	3,684 KB
total	1,424 KB	456 KB	11,364 KB

5.1 DHT Testing Example

We implemented a recursive implementation of the Kademlia DHT [18] as an example for using the emulation library. DHTs are suitable for large-scale experiments since they provide efficient routing in large-scale networks with a moderate number of connections, selectively choosing peers to connect to based on their location in a global address space. In Kademlia, this peer selection is performed iteratively by repeatedly searching the network for each peer's unique identifier. To illustrate the effects of this process, we provide an example where peers are initially connected in a 2d-torus overlay topology (the underlay topology is unrestricted). We then show snapshots of the peer connections over time (Figure 2). This use of our emulation library demonstrates how access to the overlay topology can be used to gain insight into the state of the network.

We also implemented a variation of Kademlia that uses randomization to enhance resilience, for example against Sybil [7] attacks. Figure 3 shows experimental results comparing Kademlia against randomized Kademlia with an increasing number of malicious sybil peers being added to the network. These results were obtained running a total of 20,000 total peers on the 32-node cluster described previously. This experiment utilizes many facets of our emulation framework, including its topology generation and logging facilities, the ability to modify peers to run attacks, and the ability to easily distribute execution to run large-scale experiments.

As mentioned previously, we expect P2P applications written in the GNUnet framework to have diverse data logging requirements depending on the purpose of the application. For instance, in our DHT implementations we added various levels of logging. Data which can be logged in the DHT includes hop-by-hop records of mes-



(a) Initial (b) N Minutes

(c) N Minutes (d) Final

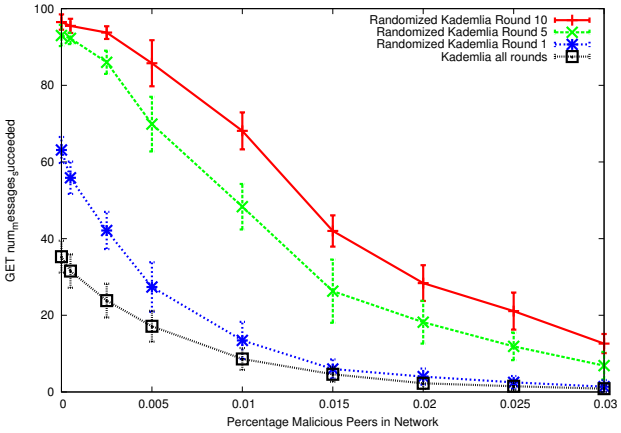
Figure 2: Topology evolution from a highly structured 2d-torus into one suitable for Kademlia. As find peer requests are sent, the torus collapses into itself.

sages, average path length for successful and failed requests, number of data replicas in the network, and myriad other data items. We chose to integrate this logging with a database; as we felt this was the best way to maintain the large amount of data. However, these metrics and data formats are quite specific to the DHT; other applications would likely prefer different data. Even somewhat common metrics such as hop counts can be interpreted differently; does "hop count" include paths through the network which failed, or only the hops from source to destination? Requiring application creators to implement specific metrics reduces the ambiguity of these metrics, and forces the experimenter to *understand* their data.

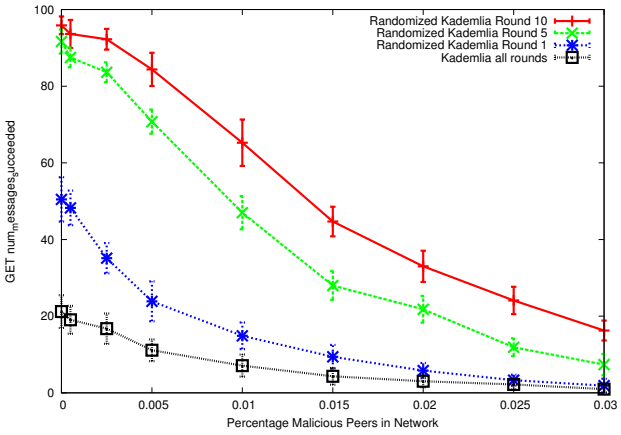
5.2 DHT-Related Performance Issues

There were two specific performance bottlenecks which caused problems for our experiments with the DHT. First, we had to throttle the initial exchange of "find peer" messages (which the DHT uses to build the overlay routing tables). Then, convergence of the DHT overlay was still a time consuming process. In the end, we used the topology capture functionality to store converged topologies to disk. Subsequent runs can then load the converged topology, eliminating this costly phase for variations of the experiment.

The second bottleneck related to logging operations. Our DHT instrumentation supports detailed logging of routing operations. Directly logging each routing operation to a central database is often too expensive. Writing the information to a peer-specific log file and importing the logs into the database in bulk after the experiment is much more scalable.



(a) Erdos-Renyi Topology



(b) Small-World Topology

Figure 3: Comparison of Kademlia vs. Randomized Kademlia in restricted underlay topologies as the number of Sybil nodes are increased in the network.

6 Related Work

Large-scale testbeds such as DETER [19], GENI [8], SecSI [3], and Emulab [12] provide realistic network conditions and operating systems for security testing. Typically this research focuses on application level tests running full VMs [12] or OSes [5], revealing high-level performance and security [4] issues. In contrast, our design focuses specifically on P2P implementations, allowing security design issues to be discovered which may only be present at large scale. However, our emulation framework can be deployed on such testbeds to incorporate the effects of differences in the underlying platforms

into results. We have used our emulation library on PlanetLab; however, the instability and limited resources of PlanetLab nodes made it more useful for us to run experiments on the cluster at our disposal. Deployment on more stable testbeds such as DETER and Emulab should work with the emulation library "as-is".

6.1 Simulation

The prevailing method for testing and verifying new P2P designs is simulation [2, 6, 10, 20, 22, 25, 27, 31].

Chunksim [14] and the Query Cycle Simulator [25] are domain-specific discrete event simulators focusing on BitTorrent and content distribution, respectively. Both were created with the assumption that real-world experiments require the ability to model at least 20K peers and the presumption that this size of network could not be emulated or deployed for experiments. Furthermore, it was thought that malicious peers could not be adequately studied in deployed networks of this size. This paper shows that such limitations no longer hold.

OverSim [2] and PlanetSim [1] are discrete event simulators for overlay protocols. Both use a layered structure to provide a common API for application development. Overlay applications are written in a domain specific dialect of C++ (OverSim) or Java (PlanetSim); this high level of abstraction makes simulation implementations quite different from their counterpart real-world implementations. Building upon OMNet++ [28], OverSim is able to simulate down to the network level, achieving high realism. PlanetSim allows the networking layer to be switched out, allowing the use of a NetworkSimulator for simulation, and a NetworkWrapper for emulation or deployment. Simulations of up to 100K peers are reported, but we are not aware of any detailed studies that were performed at this scale.

Some effort has recently been made to distribute the tasks of existing simulators [6, 17, 27]. PeerSim and dPeerSim (the distributed version of PeerSim) are currently the most scalable P2P discrete event simulators with simulations of tens of millions of peers. While this is significantly larger than what we can do with emulation, the realism and freedom of implementation provided by our framework makes it complementary to these large-scale simulators.

6.2 Emulation

Emulation frameworks [13, 23, 26, 29] are less common. While emulators are typically better at testing real-world implementations and capture more realistic data, they do so at the expense of scalability. None of the existing emulation frameworks have been used for experiments of the scale presented in this paper.

Contrary to our focus on high-level operations, ModelNet [29] is a distributed emulation environment aimed at capturing network delays, cross traffic and congestion due to underlay network configuration. In ModelNet, each emulated peer is executed with precise control over their network interactions. The largest ModelNet emulation included 10K Gnutella peers in a cluster of 100 physical nodes, though full details of this particular experiment were not given.

6.3 Combining Simulation and Emulation

Some projects try to overcome the scalability limitations of emulators and the undesirable effects of abstraction from simulators by mixing both techniques.

MACEDON [23] is a framework for the design, simulation and emulation of P2P algorithms. MACEDON requires that applications are written in a specific language; from this specification, code is generated for the simulator or emulator. MACEDON relies on ModelNet for underlay specification, and the largest reported experiment was less than 1K peers in total. We found no instances of MACEDON-generated code actually being used as a real-world implementation.

Other mixed simulation/emulation systems include Overlay Weaver [26] and RealPeer [13]. Overlay Weaver has been scaled up to 4K total peers on a 200 PC cluster. RealPeer is a framework and methodology for creating a P2P implementation in phases which include simulation and emulation. The Java-based implementation could conceivably be used as a real-world implementation. The authors successfully simulated a 20K node Gnutella network; however, no results were provided on testing the emulation or real-world implementation.

7 Conclusion

Simulation is often not suitable for security evaluations and existing emulation environments for P2P networks generally lack scalability. We believe that this is largely because they are implemented in high-level languages and often focus too much on packet-level details. We have presented a range of techniques we used to engineer our emulation framework for high performance. The GNUnet P2P framework with the emulation library exceeds previous emulators in terms of scalability and provides a comprehensive framework for large-scale experiments with P2P protocols.

Acknowledgments

This work was funded by the Deutsche Forschungsgemeinschaft (DFG) under ENP GR 3688/1-1. The authors thank Krista Grothoff for extensive editing.

References

[1] AHULL, J. P., AND LPEZ, P. G. Planetsim: An extensible simulation tool for peer-to-peer networks and services. In *Peer-to-Peer Computing* (2009), pp. 85–86.

[2] BAUMGART, I., HEEP, B., AND KRAUSE, S. Oversim: A flexible overlay network simulation framework. In *IEEE Global Internet Symposium, 2007* (May 2007), pp. 79 –84.

[3] CALVET, J., DAVIS, C. R., FERNANDEZ, J. M., GUIZANI, W., KACZMAREK, M., MARION, J.-Y., AND ST-ONGE, P.-L. Isolated virtualised clusters: testbeds for high-risk security experimentation and training. In *Proceedings of the 3rd international conference on Cyber security experimentation and test* (Berkeley, CA, USA, 2010), CSET'10, USENIX Association, pp. 1–8.

[4] CALVET, J., DAVIS, C. R., FERNANDEZ, J. M., MARION, J.-Y., ST-ONGE, P.-L., GUIZANI, W., BUREAU, P.-M., AND SOMAYAJI, A. The case for in-the-lab botnet experimentation: creating and taking down a 3000-node botnet. In *Proceedings of the 26th Annual Computer Security Applications Conference* (New York, NY, USA, 2010), ACSAC '10, ACM, pp. 141–150.

[5] CHERTOV, R., FAHMY, S., AND SHROFF, N. B. Fidelity of network simulation and emulation: A case study of tcp-targeted denial of service attacks. *ACM Transactions on Modeling and Computer Simulation 19*, 1 (2008), 4:1–4:29.

[6] DINH, T. T. A., THEODOROPOULOS, G., AND MINSON, R. Evaluating large scale distributed simulation of p2p networks. In *Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications* (2008), IEEE Computer Society, pp. 51–58.

[7] DOUCEUR, J. R. The sybil attack. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems* (London, UK, 2002), Springer-Verlag, pp. 251–260.

[8] ELLIOTT, C., AND FALK, A. An update on the geni project. *SIGCOMM Comput. Commun. Rev. 39* (June 2009), 28–34.

[9] EVANS, N. S., DINGLEDINE, R., AND GROTHOFF, C. A practical congestion attack on tor using long paths. In *18th USENIX Security Symposium* (2009), USENIX, pp. 33–50.

[10] GIULI, T. J., AND BAKER, M. Narses: A scalable flow-based network simulator. *CoRR cs.PF/0211024* (2002), 1–6.

[11] HE, Q., AMMAR, M., RILEY, G., RAJ, H., AND FUJIMOTO, R. Mapping peer behavior to packet-level details: a framework for packet-level simulation of peer-to-peer systems. In *Modeling, Analysis and Simulation of Computer Telecommunications Systems 11th IEEE/ACM International Symposium* (2003), pp. 71 – 78.

[12] HIBLER, M., RICCI, R., STOLLER, L., DUERIG, J., GURUPRASAD, S., STACK, T., WEBB, K., AND LEPREAU, J. Large-scale virtualization in the emulab network testbed. In *USENIX 2008 Annual Technical Conference* (2008), USENIX, pp. 113–128.

[13] HILDEBRANDT, D., BISCHOFS, L., AND HASSELBRING, W. Realpeer–a framework for simulation-based development of peer-to-peer systems. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on 0* (2007), 490–497.

[14] KANGASHARJU, J., SCHMIDT, U., BRADLER, D., AND SCHRÖDER-BERNHARDI, J. Chunksim: simulating peer-to-peer content distribution. In *Proceedings of the 2007 spring simulation multiconference* (2007), Society for Computer Simulation International, pp. 25–32.

[15] KARYPIS, G., AND KUMAR, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput. 20* (December 1998), 359–392.

[16] KOTILAINEN, N., VAPA, M., KELTANEN, T., AUVINEN, A., AND VUORI, J. P2prealm - peer-to-peer network simulator. In *Proc. 11th International Workshop on Computer-Aided Modeling, Analysis and Design of Communication Links and Networks* (2006), pp. 93–99.

[17] KOZLOVSZKY, M., BALASKO, A., AND VARGA, A. Enabling omnet++-based simulations on grid systems. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques* (2009), Simutools, pp. 67:1–67:7.

[18] MAYMOUNKOV, P., AND MAZIÈRES, D. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems* (2002), Springer-Verlag, pp. 53–65.

[19] MIRKOVIC, J., BENZEL, T., FABER, T., BRADEN, R., WROCLAWSKI, J., AND SCHWAB, S. The deter project: Advancing the science of cyber security experimentation and test. In *Technologies for Homeland Security (HST), 2010 IEEE International Conference on* (2010), pp. 1 –7.

[20] MONTRESOR, A., AND JELASITY, M. Peersim: A scalable p2p simulator. In *Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on* (2009), pp. 99–100.

[21] NAICKEN, S., LIVINGSTON, B., BASU, A., RODHETBHAI, S., WAKEMAN, I., AND CHALMERS, D. The state of peer-to-peer simulators and simulations. *SIGCOMM Comput. Commun. Rev. 37* (March 2007), 95–98.

[22] The Network Simulator NS-2. `http://www.isi.edu/nsnam/ns/`.

[23] RODRIGUEZ, A., KILLIAN, C., BHAT, S., KOSTIĆ, D., AND VAHDAT, A. Macedon: methodology for automatically creating, evaluating, and designing overlay networks. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation* (2004), USENIX, pp. 20–20.

[24] SALIHUNDAM, P., JAIN, S., JACOB, T., KUMAR, S., ERRAGUNTLA, V., HOSKOTE, Y., VANGAL, S., RUHL, G., AND BORKAR, N. A 2 tb/s 6 x 4 mesh network for a single-chip cloud computer with dvfs in 45 nm cmos. *Solid-State Circuits, IEEE Journal of 46*, 4 (2011), 757–766.

[25] SCHLOSSER, M., CONDIE, T., AND KAMVAR, S. Simulating a file-sharing p2p network. Technical Report 2003-28, Stanford InfoLab, 2003.

[26] SHUDO, K., TANAKA, Y., AND SEKIGUCHI, S. Overlay weaver: An overlay construction toolkit. *Computer Communications 31*, 2 (2008), 402 – 412. Special Issue: Foundation of Peer-to-Peer Computing.

[27] SIOUTAS, S., PAPALOUKOPOULOS, G., SAKKOPOULOS, E., TSICHLAS, K., AND MANOLOPOULOS, Y. A novel distributed p2p simulator architecture: D-p2p-sim. In *Proceeding of the 18th ACM conference on Information and knowledge management* (2009), ACM, pp. 2069–2070.

[28] VARGA, A. The omnet++ discrete event simulation system. *Proceedings of the European Simulation Multiconference (ESM'2001)* (June 2001).

[29] VISHWANATH, K., GUPTA, D., VAHDAT, A., AND YOCUM, K. Modelnet: Towards a datacenter emulation environment. In *Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on* (2009), pp. 81–82.

[30] VISWANATH, B., MISLOVE, A., CHA, M., AND GUMMADI, K. P. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM workshop on Online social networks* (New York, NY, USA, 2009), WOSN '09, ACM, pp. 37–42.

[31] YANG, W., AND ABU-GHAZALEH, N. GPS: a general peer-to-peer simulator and its use for modeling BitTorrent. pp. 425–432.