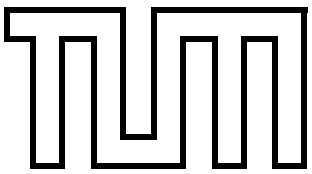


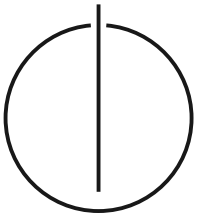
FAKULTÄT FÜR INFORMATIK
TECHNISCHE UNIVERSITÄT MÜNCHEN

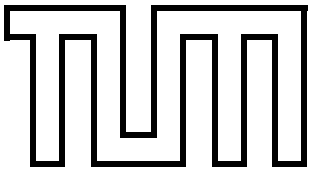
LEHRSTUHL FÜR NETZARCHITEKTUREN UND NETZDIENSTE

Master's Thesis in Informatics

LARGE SCALE DISTRIBUTED EVALUATION OF PEER-TO-PEER PROTOCOLS

Sree Harsha Totakura





FAKULTÄT FÜR INFORMATIK
Technische Universität München

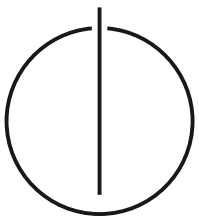
Lehrstuhl für Netzarchitekturen und Netzdienste

Master's Thesis in Informatics

LARGE SCALE DISTRIBUTED EVALUATION OF PEER-TO-PEER PROTOCOLS

VERTEILTE BEWERTUNG VON PEER-TO-PEER PROTOKOLLEN AUF HÖCHSTLEISTUNGSRECHNERN

AUTHOR: Sree Harsha Totakura
SUPERVISOR: Christian Grothoff, PhD (UCLA)
DATE: June 17, 2013



DECLARATION

I assure the single handed composition of this master's thesis only supported by declared resources.

Garching, June 17, 2013

Sree Harsha Totakura

ACKNOWLEDGEMENTS

This thesis is made possible by collective efforts of many people. Firstly, I thank my parents Yarlagadda Sri Devi and Totakura Sambasiva Rao for bestowing confidence in me and supporting my studies in a foreign country.

Many thanks go to the Free and Secure Network Systems Group (FSNSG) at TUM, Germany: Christian Grothoff, the project group leader for GNU's Framework for Secure Peer-to-Peer Networking (GNUnet), for his support during my studies at TUM and guidance throughout this thesis; Matthias Wachs for his efforts in improving the transport subsystem of GNUnet and help in debugging various bugs encountered during the development of GNUnet testbed; Maximillian Szengel for developing the profiler for GNUnet `REGEX` application and initial emulations using testbed; Bartlomeij Polot for extending the profiler and scaling up the `REGEX` emulations using testbed. I also thank Werner Koch whose recent code contributions to the libgcrypt project indirectly improved the testbed's performance significantly.

Finally, I thank all the hackers and supporters of free software who contribute to uphold an individual's right to freedom and privacy in this age of digital societies with growing censorship and surveillance. Their efforts have helped and continue to help me and many others in many ways and aspects of life.

ABSTRACT

Evaluations of Peer-to-Peer (P2P) protocols during the system's design and implementation phases are commonly done through simulation and emulation respectively. While the current state-of-the-art simulation allows evaluations with many millions of peers through the use of abstractions, emulation still lags behind as it involves executing the real implementation at some parts of the system. This difference in scales can make it hard to relate the evaluations made created with simulation and emulation during the design and implementation phases and can results in a limited evaluation of the implementation, which may cause severe problems after deployment.

In this thesis, we build upon an existing emulator for P2P applications to push the scales offered by emulation towards the limits set by simulation. Our approach distributes and co-ordinates the emulation across many hosts. Large deployments are possible by deploying hundreds or thousands of peers on each host.

To address the varying needs of an experimenter and the range of available hardware, we make our approach scalable such that it can easily be adapted to run evaluations on a single machine or a large group of hosts. Specifically, the system automatically adjusts the number of overlapping operations to the available resources efficiently using a feedback mechanism, thus relieving the experimenter from the hassles of manual tuning.

We specifically target High Performance Computing (HPC) systems like compute clusters and supercomputers and demonstrate how such systems can be used for large scale emulations by evaluating two P2P applications with deployment sizes up to 90k peers on a supercomputer.

ZUSAMMENFASSUNG

Eine Bewertung von Peer-to-Peer (P2P) Protokollen wird in der Entwurfs- und Implementierungsphase gemeinhin mittels Simulation in der Entwurfs- und Emulation in der Implementierungsphase erreicht. Während aktuelle Simulationsverfahren durch die Verwendung von Abstraktion die Bewertung von Millionen einzelner Knoteninstanzen erlauben, hinkt die Verwendung von Emulation an dieser Stelle hinterher, da bei der Verwendung von Emulation die echte Implementierung der Applikation verwendet werden muss. Diese unterschiedlichen Größenordnungen können es schwierig machen, Bewertungen, die während der Entwurfs- und Implementierungsphase entstanden sind, zu vergleichen. Weiterhin kann dies zu einer eingeschränkten Bewertung der Implementierung führen, was im Folgenden schwerwiegende Probleme bei der produktiven Verwendung des Systems haben kann.

Diese Arbeit baut auf einem existierenden Verfahren zur Emulation von P2P Applikationen auf und verbessert dieses Verfahren soweit, dass das vorgestellte Emulationsverfahren die bei der Verwendung von Simulation möglichen Größenordnungen erreicht. Der vorgestellte Ansatz verteilt und koordiniert den Emulationsvorgang über mehrere Rechnersysteme und erreicht die angestrebte Größenordnung durch die Emulation hunderter oder tausender Instanzen der Anwendung auf einem einzelnen Rechner.

Um den unterschiedlichen Anforderungen des Anwenders und der Vielzahl unterschiedlicher verfügbarer Hardware gerecht zu werden, wurde das vorgestellte Verfahren so entwickelt, dass es leicht angepasst werden kann um Bewertungen auf einem einzelnen System oder einer großen Gruppe von Systemen auszuführen. Im Besonderen passt das System die Anzahl der gleichzeitig ausgeführten Operationen auf die verfügbaren Ressourcen an, was dem Anwender die Schwierigkeiten einer notwendigen händischen Anpassung abnimmt.

Unser Augenmerk gilt im Besonderen der Verwendung auf High Performance Computing (HPC)-ähnlichen Rechenclustern und Supercomputern und wir demonstrieren, wie solche Systeme für eine Emulation von P2P Anwendungen in großem Maßstab verwendet werden können. Hierzu stellen wir eine auf einem Supercomputer erstellte Bewertung von zwei P2P Anwendungen mit jeweils bis zu 90.000 teilnehmenden Instanzen vor.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACL	Access control list. 53
API	Application Programming Interface. 8, 19, 25
API	Application Programming Interface. 7, 8, 19, 21, 24
ARM	Automatic Restart Manager. 8
AS	Autonomous System. 34, 35
CAIDA	The Cooperative Association for Internet Data Analysis. 34
CPU	Central Processing Unit. 11, 12, 15, 22, 27, 30
DFA	Deterministic Finite Automaton. 34
DHT	Distributed Hash Table. 5, 34, 35
ECC	Elliptic Curve Cryptography. 20
FIFO	First in first out. 12
FSNSG	Free and Secure Network Systems Group. vii
GNUnet	GNU's Framework for Secure Peer-to-Peer Networking. vii, 2, 7, 8, 19–21, 23, 24, 27, 28, 30, 32–34, 40
HPC	High Performance Computing. ix, xi, 2, 14, 15, 18, 27, 37, 38, 41
IO	Input/Output. 15, 23, 30
IP	Internet Protocol. 34, 35, 53
IPC	Inter-Process Communication. 7, 8, 12, 22, 24, 53
IPv4	Internet Protocol version 4. 40
LAN	Local Area Network. 5
LRZ	Leibniz Rechner Zentrum. 38
MPI	Message Passing Interface. 37
NAT	Network Address Translation. 40
NSE	Network Size Estimation. 27, 30, 32, 51

OS	Operating System. 7, 11, 12, 18
P2P	Peer-to-Peer. ix, xi, 1–7, 19, 23, 24, 28, 30, 32–34, 40, 41
SSH	Secure Shell. 37, 38
TCP	Transmission Control Protocol. 7, 13
UDP	User Datagram Protocol. 7
VPN	Virtual Private Network. 34
WLAN	Wireless Local Area Network. 5

1 INTRODUCTION

Peer-to-Peer (P2P) networks are a special type of distributed system as control and ownership are typically decentralised. This further complicates the design and implementation of P2P software: when a normal distributed system fails to scale up to expectations, the software can be revised and an updated revision be deployed quickly. With decentralised ownership and control, deploying system updates can be extremely costly and take decades. A prime example for this is the ongoing switch from IPv4 to IPv6 [Leb09], highlighting the dangers of minor design flaws (here: the IPv4 address space) in P2P networks.

Modern P2P networks are typically realised as overlay networks on top of the existing Internet infrastructure. Today, we have different applications based on such architectures offering anonymity (Tor [DMS], Freenet [Cla+01]), electronic-cash (BitCoin [Nak08]), and content distribution (Bittorrent [Coh08], Tribler [Zei+11]). The size of the resulting networks range from thousands of peers to millions; their size often being limited by the scalability of the respective architecture [Rit01], as performance and thus user-experience can degrade with the size of the network.

Thus, there exists an urgent need to evaluate the scalability of new P2P designs before deployment to avoid (or at least reduce) the need for costly updates later. Theoretical evaluations of protocols are useful to show that a protocol cannot scale; however, showing that a particular implementation does perform in practice requires experiments. Evaluating P2P protocols requires experiments that assess the networks' behaviour under conditions such as churn (peers joining and leaving), malicious participants, and possibly various application-dependent usage patterns. For P2P overlay networks, different properties of the underlying Internet infrastructure might also be of interest for such experiments.

P2P experiments can be classified into three types: *simulation*, *emulation* and *real-world instrumentation*.

In simulation, the P2P system is abstracted into an executable model which is then run and observed to obtain performance metrics. Simulations can be useful to assess the scalability and security of the protocol during the protocol's initial development. However, as the actual P2P system is abstracted, simulation cannot expose problems in a specific implementation. As in our experience, implementations tend to have more flaws than theoretical designs, this limits the utility of using simulation as a tool for quality assurance in the development of P2P systems.

On the other hand, evaluating P2P systems via real-world instrumentation fails to provide an opportunity for reducing costly re-deployments (in the case that flaws are discovered) as the real-world measurements can only happen after the system has already been deployed. Still, such experiments can deliver the most realistic data and they thus should be considered as part of an overall system life cycle management approach.

This thesis documents an approach for evaluation of P2P protocols using emulation. In emulation, the actual P2P system is executed while the underlying network infrastructure and the user behaviour are simulated. A

key challenge for emulators is to achieve scalability, as unlike simulators, the complexity of the application is not reduced by abstraction. In this thesis, we present our design and implementation of a distributed emulation method that is expected to be able to carry out large scale emulations up to millions of peers using modern High Performance Computing (HPC) systems.

Building on the recent work by Evans [EG11#1], this work represents another step towards pushing emulation to the scales offered by simulation and making the results from both approaches more comparable. Following Evans' approach, we use process-level isolation as a key lightweight method to isolate multiple peers from each other on the same host, while also enabling the system to share certain (uncritical) resources among a group of peers.

The key contributions of this thesis are: an adaptive bounded operation queuing mechanism to automatically adapt the parallelism among operations according to the load on the system, a scalable distributed command-and-control architecture for the emulation, a controlled shutdown mechanism which ensures proper experiment termination regardless of the location of the fault, and improved automated configuration generation by probing for free ports.

The presented emulation method was implemented in GNU's Framework for Secure Peer-to-Peer Networking (GNUnet), offering various usability improvements and simplified APIs over the previous testing infrastructure [EG11#1].

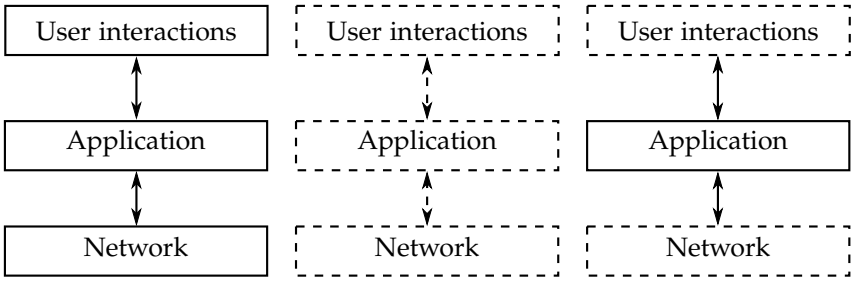
In this thesis, we demonstrate the capabilities of our method by running various P2P benchmarks from the GNUnet system on a compute cluster and the SuperMUC [Sup] supercomputer. Specifically, we conducted three sets of experiments: setting up overlay networks with a particular topology, which demonstrates the performance and scalability of our approach; decentralised network size estimation [EPG12], which uses the result aggregation and service sharing features of our system; and distributed matching of regular expressions [Sze12], which showcases emulations involving non-trivial modelled user behaviour.

The rest of the thesis is organised as follows. Work related to evaluations done through simulation, emulation and real-world instrumentation is discussed in Chapter 2. Chapter 3 describes the various design decisions that have influenced the approach, while Chapter 4 describes issues faced during implementation of the design and how they were resolved. A short description about the HPC systems which were used for conducting the experimental evaluations is presented in Chapter 6. Experimental evaluations of some of the P2P applications carried out with the implemented approach are described with their results in Chapter 5. Finally, Chapter 7 discusses some of the shortcomings faced during the evaluations, ways to resolve them and directions for future work.

2 BACKGROUND

We distinguish three methods for evaluating P2P applications, *simulation*, *emulation* and *real-world instrumentation*. The system model shown in Figure 2.1 highlights the differences between these three methods.

In this model, the network layer represents the infrastructure enabling communication among the peers. The application layer represents the behaviour of the P2P application; this is the layer which includes the new components that are to be assessed by the experiment. Examples for interactions between the application layer and the network layer include actions such as sending a message to another peer for joining the overlay network or receiving a search query from another peer. The user layer represents the user's behaviour while using the P2P application. Examples for interactions between the user layer and the application layer include actions such as starting and stopping the application, issuing search queries to the P2P network and downloading data.



(a) Instrumentation (b) Simulation (c) Emulation

Figure 2.1: P2P evaluation methods. Solid boxes indicate using real-world code and systems whereas dashed boxes indicate the use of simulators which use some abstraction over the real world. Arrows indicate interactions within the system.

Methods to evaluate P2P applications differ in how they represent the layers. Approaches using simulation abstract all three layers into some simplified representation suitable for the respective simulator and gather evaluation data from the resulting execution. In real-world instrumentation, the evaluation data is gathered by augmenting the network or application layers with additional monitoring logic. In emulation, experimental data is obtained by modelling the user and network layers and observing the interactions with the P2P application's concrete implementation. This chapter describes the state of the art for these methods.

2.1 REAL-WORLD INSTRUMENTATION & TESTBEDS

Testbeds are widely used for testing and profiling of various distributed applications and protocols. Testbeds range from a cluster of few hosts at a research organisation to an interconnection of globally located hosts such as PlanetLab [Chu+03]. Testbeds relate to the evaluation model as follows: at the application layer, concrete implementations of the protocols are run; the network layer is also realistic, for example in PlanetLab the actual Internet

is used. Depending on the experiment, the experimenter might passively observe the actions of real-world users (as done, for example, in [HG11]) or simulate user actions to trigger interesting events (as done, for example, in [Isd+10]).

Thus, depending on the type and size, testbeds can produce realistic evaluations for networking experiments; however, unless real-world users are included, the size of the experiments is typically rather limited. The problem arises when accommodating large number of peers—most of the available testbeds[Chu+03; Bol+06; Nit], comprise only of a few hundreds of nodes. Once real-world users are incorporated in the experiment, deployment (and redeployment) costs may rise dramatically and reproducibility suffers as well: global scale testbeds such as PlanetLab provide an uncontrolled environment as they use the Internet to connect the individual nodes. The outcome of an experiment performed these testbeds is thus affected by the network conditions of the Internet at the time of the experiment execution. Such testbeds thus cannot vouch for reproducibility of networking experiments.

A recent advancement to address the uncontrolled topology and traffic behaviour of in PlanetLab is to create an overlay network above the Internet to have a more controlled routing among PlanetLab nodes [Fre+12]. It is however, intentionally made, susceptible to the delays and looses experienced in the Internet. One method to address scalability issues is the integration of two or more testbeds for use in a single networking experiment [Mak+12]. Such efforts increase the number of nodes available and also extend the capabilities of a testbed with those of the others.

Real-world instrumentation is also useful to showcase the real-world trends of the Internet; however, this typically requires long-lived experiments [Spr+06]. Due to the diversified node locations, such global-scale testbeds are also sometimes used as a primary infrastructure for P2P systems [Rhe+05].

2.2 SIMULATION

Taking the definition from [Ban+98], simulation is defined as *"the imitation of the operation of a real-world process or system over time"*. The imitation is possible by constructing a model of the process or system and observing the interactions of the model within itself and with the external environment, which is also typically represented by a model.

Simulations are used in a variety of fields such as physics, chemistry, biology, warfare, meteorology, logistics and engineering. A significant property of simulations is how the interactions change the state of the model. Simulations involving continuous input from the environment (such as computational fluid dynamics) and continuous output from the model are called *continuous event simulations*. Modelling for such simulations is commonly done in the form of differential equations which act on the input events to generate continuous output. The other major type of simulations are *discrete event simulations*, where the input is in the form of discrete events (such as in logistics or computer networks), and where the model has to generate a discrete response.

The type of the simulation to use is dependent on the abstraction of the model and the questions which are to be answered from the simulations. For simulating the routing behaviour of a computer communication network and to determine the average latency of messages, it is enough to model the

sending and receiving of messages at various nodes in the network as discrete events and thus use discrete event simulation. If our aim is to observe the physical layer properties and to simulate the attenuation of signals in the network, the electrical properties of the transmission medium are typically modelled as a set of equations and continuous event simulation would be more appropriate in this case.

For evaluating P2P protocols, we are typically interested in simulating message routing among peers, per peer resource consumption, and network characteristics such as latency and loss. These requirements can be satisfied by constructing a discrete event simulation model of the underlying network where sending and receiving packets are considered as discrete events.

There has been a considerable amount of research done in the field of discrete event simulators for computer networks. Notable among the early popular network simulators is the Network Simulator 2 (NS-2) [FV07]. Simulations are given to NS-2 as Tcl scripts which specify the topology in which nodes are to be connected and the link properties, the traffic generators and sinks, various queuing mechanisms and monitoring points. NS-2 also allows simulation of custom protocols. For this a model of the protocol has to be coded in C++.

While the NS-2 simulator is a generic network simulator, it has extensions which can increase its scope to simulate a wide variety of networks — Local Area Network (LAN), Wireless Local Area Network (WLAN) and satellite links. Further extensions exist to simulate interconnection of different types of networks and radio signal propagation for wireless networks.

As P2P systems share some basic functionality, efforts were made to build simulators for P2P systems. Peersim [MJ09] is one such simulator developed using Java. The protocols to be simulated are to be modelled in Java according to the simulators exported interface.

OverSim [BHK07] and P2PSim [Li+05] improved support for unstructured and structured P2P overlay network protocols like Chord [Sto+03], Kademlia [MM02], Pastry [RD01], Kelips [Gup+03] and Tapestry [Zha+04]. The motivation behind using this type of simulators is to compare the evaluations of existing overlay protocols after modifying some parts of their functionality. OverSim is built using the OMNet++ [Var+01] discrete event simulation framework; protocol modelling is done in C++ with the help of interfaces exported from OMNet++. P2PSim is developed in C++ and requires the protocols to be modelled as C++ classes.

Apart from the generic P2P network simulators, domain or application specific P2P simulators are also available. These simulators ease the modelling required for a generic P2P network simulator to simulate a given P2P application. Examples for such simulators are ChunkSim [Kan+07] and PlanetSim [PAGL09]. Chunksim is designed to simplify simulations of P2P content distribution protocols. The authors of Chunksim claim that overlay network details are typically irrelevant while studying content distribution protocols. Using Chunksim, designers of such protocols can instead focus on content distribution and peer selection strategies. PlanetSim supports in addition to structured overlays such as Chord and Symphony [MBR+03] a few P2P application layers such as Distributed Hash Table (DHT) and churn models. Other examples for domain specific simulators are Shadow [JH11] which is designed to simulate Tor [DMS], and P2PRealm [Kot+06] which is designed to simulate P2P protocols based on neural networks. Since neural networks require a time consuming training phase, P2PRealm focuses on efficiency rather than packet level simulations.

Most of the P2P simulators discussed so far are not distributed; they are run on a single host. Even though some of them achieve large scale simulations (10^7 peers in the case of simple protocol simulations in Peersim[MJ09]), scalability is further enhanced by using distributed P2P simulators. dPeerSim [Din+08] is one such distributed P2P simulator developed by extending PeerSim. These simulators run on multiple nodes and together with some abstractions and assumptions derived from the domain knowledge of distributed systems, can achieve good simulation performance by sacrificing some accuracy [Lin+05].

A comprehensive survey of available P2P simulators is published in [Nai+07]. From the findings, it is to be noted that, no simulation model seems to be broad enough to satisfy requirements of various research groups who clearly often find it easier to create yet another simulator.

A key advantage of simulation is that the process or system in question does not have to be implemented. Thus, simulation can be used as a cost effective method to evaluate ideas before developing them. However, it suffers from the modelling inaccuracies which often cause simulation to give incorrect results or bad estimates [CFS09].

2.3 EMULATION

Emulation executes the real-world software at the heart of the experiment while using simulation for user interactions and possibly the network layer. This reduces inaccuracies when compared to simulation, as at least the component under test is not abstracted. Emulation is commonly used for networking experiments with network testbeds [Ben+06; Bol+06; Whi+02] where real protocol implementations are run on different kinds of emulated networks and devices.

The job of the emulation system is to enable the interaction between the real-world software and the simulated environment. In particular, the emulator specifies and provides an interface that the real-world application will use for its execution.

Emulators resort to virtualisation to isolate components of the system that are executed on the same hardware and to simulate realistic environmental conditions, for example by creating communication latency between two components that run on the same host. Different levels of virtualisation exist, which allows emulator designers to trade realism for efficiency. At the operating system level, processes can be used to virtualise CPU and memory resources; and process' system calls can be intercepted to provide a virtualised view of the operating system [Vah+02]. At the host level, physical hardware can be virtualised so that multiple instances of entire operating systems can run on the virtualised hardware. This gives stronger isolation, unnecessitating the sharing of file systems and networking resources, but involves higher overheads as a hypervisor has to enforce the separation between operating systems, which typically requires additional instructions or hardware support. Hypervisors like Xen [Bar+03] are widely used for employing host level virtualisation in P2P emulation experiments [Wei+11].

The overhead incurred due to host level virtualisation limits the scalability of the emulation. This is addressed by some emulators [Hib+08; LHM10; Han+12] by using lightweight forms of virtualisation through *containers* in Linux, *jails* in BSD-style systems and *zones* in Solaris. This form of virtualisation provides each guest its own file-system and virtual network interface

while sharing the same operating system kernel and thus avoiding the need for an hypervisor in addition to the operating system.

Unlike simulation, emulation typically runs in real time and is thus constrained by the physical limits of the given resources. For example, the fastest link bandwidth that can be emulated in an emulated network topology is the fastest physical link connection provided by the physical network. This limit can be overcome by emulators using Operating System (OS) level virtualisation by slowing down the time of the virtual instances [Vis+09; Gup+11; Gup+05].

Since most of the P2P applications are run as user program and often in the background, they typically do not require exclusive access to network resources. Also, since P2P protocols are designed to sustain network delays and brief network outages, they can sustain reasonable levels of contention for the shared network resources. Thus P2P application emulations do not require the high realism and isolation offered by host level or OS level virtualisation.

Taking advantage of these relaxed requirements allows P2P emulators to achieve high scalability using the lightweight isolation offered by OS processes, allowing the emulator to run multiple peers within the same operating system as processes. A key disadvantage of this method is that network properties, such as latency and loss, need to be addressed manually, either by intercepting system calls or by additional logic placed within the P2P application.

The emulation system presented in this thesis uses this approach of process-level isolation with additional logic placed within the P2P application; we are building on the basic design used by [EG11#1] to emulate large-scale P2P applications, which we will discuss in Section 2.5. While the method is not specific to GNUnet, understanding the GNUnet architecture is helpful to understand some of the nuances and specific design choices of our design and implementation, which is why we will briefly review it next.

2.4 GNUNET ARCHITECTURE

GNUnet is a framework for developing P2P applications. Peers in GNUnet comprise of several processes called as *services*. Services are responsible for providing a particular functionality and capability of a peer. For example, the `TRANSPORT` service is responsible for making connections to other peers through different transport mediums such as Transmission Control Protocol (TCP) or User Datagram Protocol (UDP).

Controlling and accessing peers' services is facilitated through respective client Application Programming Interface (API). Service and their client APIs use a client-server architecture with the service acting as a server and the respective client API as a client. Services in GNUnet can thus depend on other services' functionality by accessing respective service's client API. For example, the `CORE` service provides encrypted connections to other peers. It depends on the `TRANSPORT` service for providing the actual communication. Furthermore, the `TRANSPORT` service depends on the `PEERINFO` service to remember the addresses of known peers through peer shutdowns.

A service's client API implements the required Inter-Process Communication (IPC) routines needed for communication with the service along with some other house-keeping routines. A service can be configured to use TCP or UNIX domain sockets to listen for its client API's IPC requests. The details are specified in a configuration file which the service accesses during

start-up. In this configuration, resources such as ports numbers (for tcp) and file names (for IPC through UNIX sockets) are specified. The client-API then uses the same configuration to determine the address where the corresponding service is listening to connect to it.

Since a peer in GNUnet is made up of individual services, starting it involves starting all of its services. The `Automatic Restart Manager (ARM)` service of a peer undertakes the task of starting and stopping of the peer's services. During peer startup ARM starts the services which are marked as default services in the peer's configuration. `ARM` is also responsible for stopping services when shutting down the peer. It can also start services on-demand — services' start-ups are delayed until requests for those services arrive. Additionally, ARM monitors the services it started and restarts if any of those service crash.

2.5 DISTRIBUTED LARGE SCALE EMULATOR FOR P2P PROTOCOLS

An earlier version of the GNUnet system already used process-level isolation for large-scale experiments using emulation [EG11#1]. The approach exploits GNUnet system architecture to start multiple peers on a given host. As GNUnet is (largely) written in C, this allows the operating system to share memory for application binaries and other non-writable process resources, allowing the system to run thousands of peers on standard desktops.

Access to emulated peers is facilitated through a library which enables the profiling driver program to control peers, specify which peers are to be connected and modify the configurations of running peers. The library's Application Programming Interface (API) also supported high-level operations which could be used to control a group of peers, access services of peers and specify a network topology according to which the peers were to be connected. The architecture of the old testbed is shown in Figure 2.2.

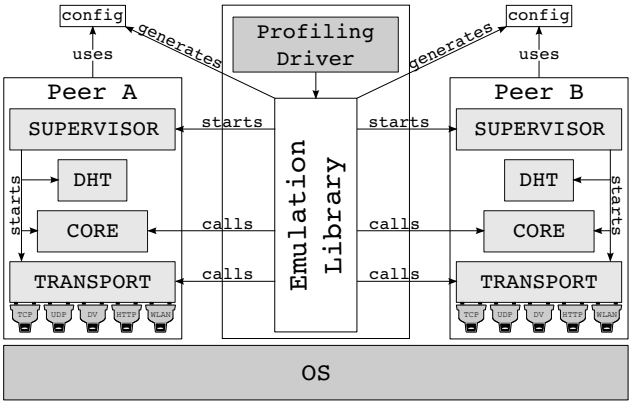


Figure 2.2: Architecture of the old GNUnet testbed (Figure from [EG11#1]).

While the profiling driver program and the emulation library are run on a single host, the approach supports distributed emulation by deploying peers on to different hosts. Connections to remote hosts are made through SSH for starting and managing peers. This has become a bottleneck while staring large number of peers on many hosts as the testing driver running on a single host has to open many SSH connections. In many cases, the operating system running the driver either runs out of source ports or hits a per-user port or file-descriptor limit. Although the limits can be relaxed, it still is a bottleneck as these resources are limited on any given host.

Another limitation of the old implementation is that the bounds on the number of parallel operations require manual tuning for each experiment. Results in [EG11#1] show that the approach is scalable: it can run on devices ranging from embedded computers to workstation class clusters. However, this is achieved at the expense of tuning the various bounds based on a host's resources to get optimal performance. The old emulation system also had serious troubles with clean experiment termination and error handling and expected exclusive access to the target systems; in particular, it did not tolerate ports being already in use for unrelated activities.

3 DESIGN

In this chapter we present our design for an infrastructure which we refer in the remainder of this thesis as *testbed* to emulate large numbers of peers. This design borrows ideas from the existing emulation method discussed in Section 2.5 using a light-weight isolation technique. Key considerations of this design are to support emulation on multiple hosts using a scalable hierarchical architecture to control the emulation process, to utilise the hosts' resources optimally using an adaptive mechanism to schedule operations, fault detection and handling in the emulation process, and the ability to modify peers' behaviour at run-time.

3.1 INSTANCE ISOLATION, CONFIGURATION AND CO-ORDINATION

In this design, we run multiple instances of an application on a single host with each instance representing a peer. This requires isolation between the instances to prevent instances from interfering with each other. Running each instance as a process is beneficial since the OS provides isolation by virtualisation of the Central Processing Unit (CPU) and memory. Further mechanisms are required for isolating peer-specific information and resources.

By generating a configuration with unique resources for each instance, multiple instances can co-exist on the same host. On the other hand, sharing of resources among a group of instances can be achieved by having a common resource in all of the generated configurations.

An *experiment driver* program directs the control of the emulation to the *experiment controller*. The driver dispatches operations to the controller to create peers, start and stop peers, access peers' services, and connect peers to form an overlay network. These operations are then executed by the controller which signals the driver about the operation's success (with an optional result) or failure. Additionally, the driver collects evaluation data from the peers.

3.2 OPERATION QUEUES

An implication from desiring a scalable architecture is to make the testbed utilise resources optimally. This is achieved by executing independent operations which can use the resources simultaneously in a parallel fashion to reduce their overall execution delay.

The number of operations started in parallel is called the *parallelisation factor* and directly impacts the load on the resources. A low parallelisation factor may result in resources being under-utilised when compared to a higher degree of parallelisation. This direct relationship, however, does not imply that a high parallelisation factor achieves better performance. This is due to that fact that with heavy parallelisation of operations, resources can get over-utilised. With over-utilisation, overheads arise from resource scheduling which degrade the overall performance. This is called *thrashing*. For example, a single core processor can be best utilised by running two processes which need CPU for only 50% of the time or by running a single CPU-intensive process which needs 100% of CPU. If two CPU-intensive pro-

cesses are run in parallel, a kernel with round-robin scheduler has to share the CPU between the processes by performing task switches which incurs overhead. This overhead can have significant impact on system performance if large number of processes are run in parallel as the scheduler has to do many task switches. The problem of thrashing can be tackled by bounding the parallelisation factor in the testbed.

Another concern with executing operations in parallel is that their resource consumption demand has to be limited to stay within bounds given by the OS. For example, on most UNIX systems a process is by default limited to have a predefined number of file descriptors open. Handles for socket communication, IPC and files all occupy a file descriptor. If a process hits this limit, then the operating system will refuse open further file descriptors. These restrictive measures by the OS can be fatal for an experiment and hence the parallelisation factor has to be bounded in a way that the parallel operations' resource requirements are below the permissible limits.

Bounding the parallelisation factor can is realised in our design by employing *operation queues* — queues which allow a fixed number of operations to be active at any time. An operation queue is a quota-based First in first out (FIFO) queue for a resource which limits the number of operations using this resource. The quota of an operation queue determines the number of units of the resource that can be used simultaneously. Operation queues have several lists in which operations are queued with their resource demand depending on their current state. The relation between the current state of an operation and its corresponding list is described in Table 3.1.

State	list	Description
Waiting	Wait list	The operation is waiting in the operation queue's wait list to be activated.
Activated	Active list	The resources required by the operation are assigned and the operation is being executed.
Finished	–	The operation has finished execution. The resources assigned to it are unassigned from it and the operation is destroyed.
Deactivated	Inactive list	The operation has finished execution, but its resources are not unassigned.

Table 3.1: States of an operation and corresponding list within the operation queue

When a new operation is queued, the operation queue checks if it has sufficient resources to meet the demand of the new operation, the operation will be activated. If the operation queue has exhausted its quota, the operation is kept in the queue's *wait* list. Here it stays until the other active jobs in the queue reduce their resource requirements sufficiently to allow the queue to satisfy the operation's demands.

Once an operation is activated, the queue allocates the resources to the operation and the available units of the resources are decreased by the operation's resource demand amount. The operation is then executed with the requested resources. Once the operation is activated, it allocates its resources and generates a result object which is returned to the experiment driver. Often, the result object relates to the requested resources and prevents them from being used by other operations. The driver is expected to explicitly mark the operation as *done* once it no longer requires the result object. The

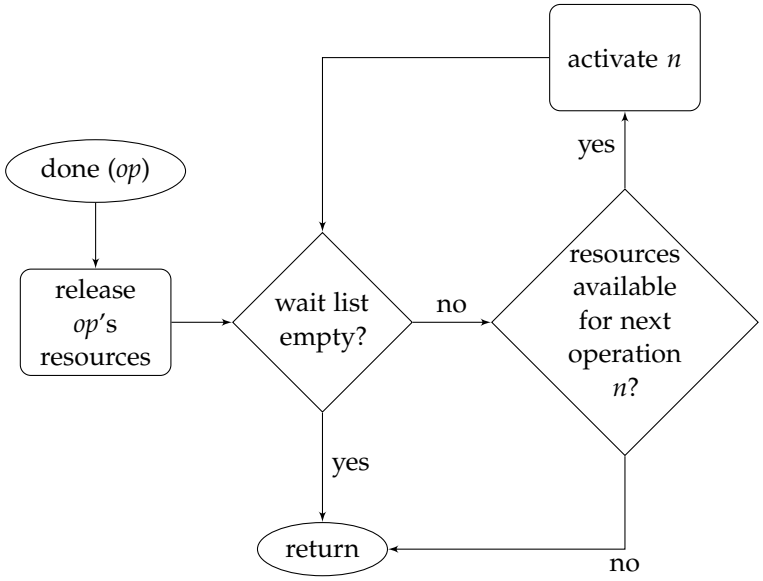


Figure 3.1: Marking an operation *op* as *done*

driver can also cancel an operation before it is activated, again by marking it as *done*.

When an operation was activated and then marked as *done* (see Figure 3.1), its resources are released and the available resource units of the operation queue it is queued into is increased by the operation's resource demand amount. The operation queue will then try to activate other operations in its wait list.

An another design decision to be considered is the re-usability of resources. Often, operations have to open handles to resources before accessing them (*e.g.* opening a socket handle for a TCP connection) and may incur some delay due to setup or overhead (delay due to TCP handshake). When an operation using these resources is marked as *done*, it relinquishes them by closing these handles. Latter operations in the same operation queue will then have to open respective handles and thus incur a delay again. These delays can be mitigated by having operations close the resource handles *on-demand*. This can be realised by deactivating operations as shown in Figure 3.2 instead of marking them as *done*.

Upon deactivating an operation, its resources are not immediately released but it is moved to the operation queue's *inactive* list. It stays there until it is activated again or marked as *done*. The idea is that, when the resource creation of a future request matches that of another recently deactivated operation (e.g., another TCP connection is needed for the same service in a situation where TCP streams can be re-used), the resource can simply be reused by activating the deactivated operation instead of queuing a new operation and performing the initialisation operations repeatedly. Naturally, if the resource is to be used in a different way (e.g., for a TCP connection to a different service) a new operation is queued instead of reusing an inactive operation. At this point, if the quota of the operation queue is exhausted, the queue then purges deactivated operations (by releasing their resources and transitioning their state to *done*) until the new operation's resource demand can be satisfied.

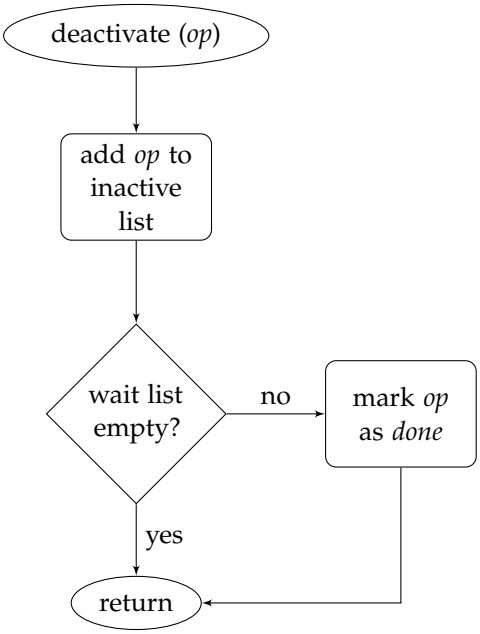


Figure 3.2: Deactivating an operation *op*

Deactivated operations require further processing while adding new operations to an operation queue since the resources of the deactivated operation. The control flow for this process is shown in the Figure 3.3.

Since an operation can require access to multiple resources, it is common for it to be queued into multiple operation queues. In this case, the operation is activated once all the operation queues it is queued into have enough available resources to meet the respective resource demands of the operation.

Another advantage of our operation queue design is that it can help avoid duplicating an operations' result objects. For example, consider the operation to fetch the configuration of a given peer. Upon its completion, the operation's result (the fetched configuration) stays valid until the operation is marked as *done*. The experiment driver can thus avoid unnecessary memory duplication by using this configuration directly as long as it is valid.

3.3 ADAPTIVE BOUNDED PARALLELISM

Having a bound on the parallelisation factor still has the risk that the configured bound may trash low-end systems where frugal resource usage is required. On the other hand, the configured bound may become too conservative for high-end systems which have more resources and can accommodate aggressive parallelisation. Thus, before starting an experiment, the given system has to be assessed to find the optimal degree of parallelisation which can then be used to bound parallelisation factor.

Such assessments are suitable for systems which are guaranteed to maintain the same number of resources between different runs of experiments. Examples for such systems include single nodes and locally networked small group of workstations. In such systems, resources are not allocated dynamically in contrast to the HPC systems (e.g, compute clusters and supercomputers) which often require resource allocation prior to starting the experiment. Assessing a bound for parallelisation factor for a group of re-

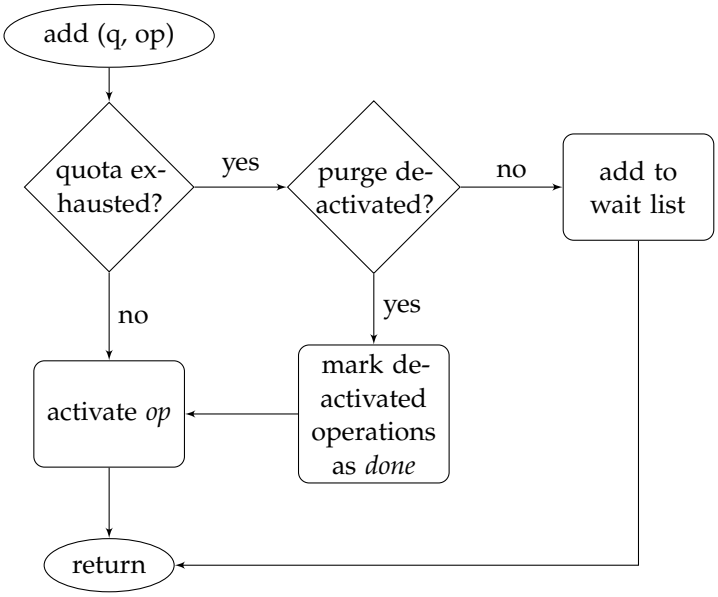


Figure 3.3: Adding an operation *op* into an operation queue *q*

sources on HPC systems becomes tiring and overwhelming for an experimenter who typically allocates varying quantities of resources based on the experiment's requirements. To address these situations, the testbed has to facilitate automatic assessment of the parallelisation factor by considering the quantity of resources available. This feature frees the experimenter from the hassles of determining the parallelisation factor manually, even for systems with fixed number of resources.

Additionally, depending on external factors (e.g., network congestion, presence of other tasks accessing CPU or Input/Output (IO)) and also on internal factors (e.g., load induced by the experiment or testbed itself), the load on the system can vary dynamically. With a varying system load, a static bound on the parallelisation factor assessed automatically or manually cannot guarantee that the resources are used optimally without thrashing. Hence, a mechanism is required to continuously adapt the parallelisation factor such that the resources are used optimally.

In our design, the bound on the parallelisation factor is determined adaptively by statistically comparing the average operation completion time (t) of the currently active operations with the set of averages of previous operations' completion times (T). The parallelisation factor (D) is adapted according to the heuristics described in the following algorithm:

With the above mechanism, testbed continuously probes the system if it can handle the varying D by having equally many active operations. If the system coped well with the given D, t is reduced which leads to an increase in D. This continues until the system is fully utilised and the overhead involved in parallelisation begins to dominate. At this point, the thrashing is realised by testbed from the increased t and D is reduced until t starts to decrease. This approach of using statistical analysis over the operation completion times has the advantage that it considers the utilisation of all resources required for the operations implicitly—any contention in any of the resources will prolong operation completion times.

Due to the computational effort involved in calculating the statistical values over of set of averages previous operations' completion times (T), T's

ALGORITHM 1: Determining the parallelisation factor D

Require:
t: the average completion time of the current set of active operations
T: the list of averages of previous operations' completion times
n: the current number of elements in T
M: the maximum number of elements that can be in T
if $t \leqslant avg(T)$ **then**
  $D \leftarrow 2D$
else if $t \leqslant (avg(T) + \sigma(T))$ **then**
  $D \leftarrow D + 1$
else if $t \leqslant (avg(T) + 2\sigma(T))$ **then**
  $D \leftarrow D - 1$
else if $t > (avg(T) + 2\sigma(T))$ **then**
  $D \leftarrow D/2$
end if
if $0 = rand(D)$ **then**
  if $M = n$ **then**
    $T \leftarrow T \setminus head(T)$
  end if
  $T \leftarrow append(T, t)$
end if
return D

size is bounded by having newer values replace older values. A new concern arises from this setting: when D is increased, the completion times of the increased number of operations will flush out the older values from T quickly as there are many of them now. This results in T having more better values quickly than before, which will make the heuristics linearly increase D when it could be doubled. Also, upon a slight increase in t, it results in quickly reducing D. Neither of these behaviours are desired as they fluctuate D frequently. For this reason, T is updated with new values with a probability of $1/D$ to limit the flushing of T when D increases.

3.4 SCALABLE ARCHITECTURE

One of the goals for the testbed is to be able to run distributed experiments on a wide variety of platforms – from desktop machines to compute clusters of different sizes and supercomputers. A characteristic difference among these platforms is the number of hosts they have. While an approach with centralised architecture [EG11#1] is well suited for distributing emulation among small number of nodes, the centralised components become a bottleneck with large number of nodes.

We address this issue by distributing the management of emulation among multiple hosts along with the emulation. This is done by starting a controller on each of these hosts which supervise the emulation locally on their respective hosts. The controllers are connected with each other in a hierarchical master-slave architecture with the experiment driver connecting to the master controller as shown in Figure 3.4. The hierarchical arrangement aids scaling as it allows for abstracting branches of the hierarchy by placing intermediate slave controllers. It also allows for the master controller to control an unlimited number of slave controllers which is not possible if the slave controllers are to be directly connected to the master controller (as

there is a limit on the number of connections a single process is allowed to have open at the same time).

The controllers manage the emulation by forwarding the operations dispatched by the driver to either all controllers or to the controller responsible for executing the operation. Upon termination of the emulation, the controllers clean up any local state associated with the emulation and signal the result.

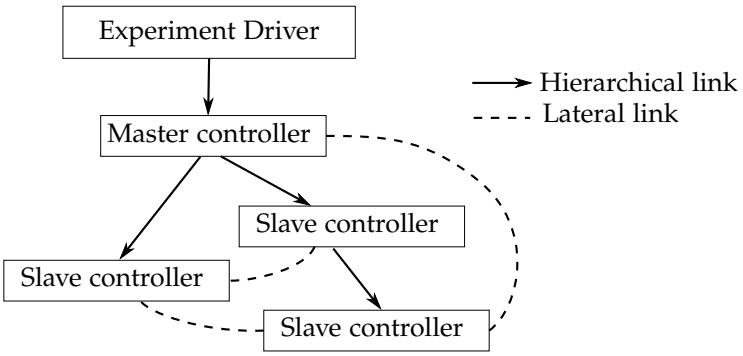


Figure 3.4: Controller hierarchical architecture

In addition to supporting hierarchical links between controllers, the design supports lateral links to be established between any two controllers. Lateral links serve as an optimisation to reduce load at the more centralised controllers situated high in the hierarchy when executing operations which require two controllers to coordinate. An example for such operations is the operation which attempts to establish a connection between two peers. If the two peers happen to be on the same host, the host's controller will solely act upon the operation. In case the peers are running on different hosts, their respective controllers co-ordinate to negotiate a connection by exchanging the peers' addresses by establishing a lateral link. These interactions are illustrated in Figure 3.5

While establishing a lateral link, it is important to establish them *on-demand* as it relives the controllers to establish them during start-up. This has the advantage that the controllers do not have to spend time at start up establishing these links and hence quickly start the experiment.

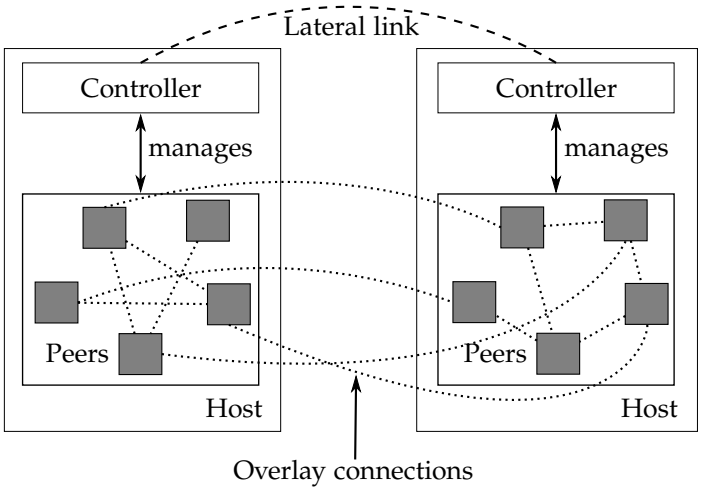


Figure 3.5: Controllers use lateral links to establish peer connections across hosts.

Since a lateral link involves a network connection it utilises a file descriptor – a critical resource since the OS has defined limits on the number of open file descriptors. For this reason, opening a lateral link has to be realised by a corresponding operation which has to be queued in the operation queue (Section 3.2) for the number of open file descriptors. This helps to keep a check on the number of file descriptors opened by a controller. Also, the corresponding operation queue logic can be used to achieve some efficiency by reusing lateral connections as described in Section 3.2.

3.5 CONTROLLED SHUTDOWN & CRASH DETECTION

To shutdown the experiment or in the case of an error while running the experiment, it is required for the testbed to provide a ability to shutdown the experiment in a controlled manner. The experiment driver initiates the shutdown by sending a shutdown signal to the controller hierarchy. Upon receiving the shutdown signal, the controllers stop the peers, destroy any state associated with them and propagate the shutdown signal to further controllers below in the hierarchy. It should be noted that a controller does not terminate while controllers further down the hierarchy (or any of their peers) are running. Instead, the controller waits until all of its child controllers are shutdown. This ensures that the shutdown call in the experiment driver waits until all controllers are shutdown successfully. Any prolonged delay during the shutdown can then be attributed to a faulty peer which refused to terminate and if such faulty situation arises such peer can be easily located and debugged by following the *shutdown-waiting* branch of the controller tree.

Controlled shutdown also proves useful while working with systems with reservation based usage schedulers. Most of the HPC systems fall under this category which require the experimenter to obtain a reservation prior to executing experiments. While some systems support extending reservations dynamically such requests can be denied. In such case, the systems signal the experimenter's processes to shutdown after the reservation has expired. A controlled shutdown ensures that the experiment driver terminates only when all peers are shutdown. This helps the driver to collect valuable data generated while the peers are shutting down. Without a controlled shutdown such data would get lost when the experiment driver is aborted.

An another desired feature is the ability to shutdown gracefully an experiment when one of the controllers in the tree crashes resulting in an invalid experiment. This ability is invaluable when working with reservations where compute time is accounted for. Leaving a failed experiment running until the reservation expires would result in wasted resources.

3.6 RE-CONFIGURING PEERS

Experiments may require peer behaviour to be changed while running the experiment. An example for such a scenario is described in [EG11#2] where benign peers had to be reconfigured to be made malicious.

To re-configure a peer, the driver generates a new configuration and passes it to the controller hierarchy. The controllers distribute the new configuration to the controller running the respective peer which updates the peer's configuration.

4 IMPLEMENTATION

We implemented the design approaches for our emulation method in the GNUnet testbed as a part of the GNUnet P2P framework. With the testbed an experimenter can evaluate P2P protocols by implementing them as peer services in GNUnet and developing an experiment driver program which interfaces with the testbed library for experiment setup and control.

In this chapter we describe the architectural overview of the testbed, how an experimenter can create and run evaluations, and some techniques to optimise emulation runs.

4.1 TESTBED ARCHITECTURE OVERVIEW

The implementation of the testbed is divided into the following parts:

- *testbed API:* is a library used by the experiment driver programs as an interface for the testbed to control the emulation. It provides the functions for managing the testbed and the peers. Testbed management functions enable setting up a testbed controller hierarchy as described in Section 3.4 by starting controllers and connecting them. Peer management functions enable creating, starting and stopping of peers, fetching a given peer's configuration and identity information, establishing overlay connections between two given peers and accessing peers' services. These functions issue the respective operations to the testbed: *create peer*, *start peer*, *stop peer*, *overlay connect*, *get info*, *service connect*. Additionally, the testbed API also provides convenience functions to connect a given set of peers in some particular topology, gather statistics from a given set of peers and to quickly setup the testbed on a given set of hosts.
- *testbed service:* is the controller which is started on every host to manage the distributed emulation. Controllers running on different hosts are connected to each other to form the controller hierarchy. The controllers in the hierarchy manage the operations generated from the testbed API by forwarding them to the corresponding controller.
- *testbed helper:* is a program to start the testbed service on remote hosts and to perform local state cleanup if the controller crashes.

4.2 EXPERIMENT START-UP

The experiment driver starts an emulation by launching the testbed helper program using the testbed API on the hosts provided to run the experiment. In addition the driver provides the helper with a configuration template used by the helper to generate unique configurations for the controllers. These unique configurations ensure that the controllers run without resource conflicts (for example, listen ports). Once a controller is successfully started on a host, further controllers can be started by this controller. This is done similarly by starting the helper programs on respective hosts to form a hierarchy as described in 3.4.

The helper programs are started on given hosts through a remote shell; `ssh` is used by default but this choice can be changed by the experimenter. The testbed API and the helper communicate through the helper program's standard input through which the configuration template is read by the helper and standard output through which the generated configuration is sent back to the experiment driver.

Peers are created by the experiment driver by issuing *create peer* operations. While creating a peer, the driver has to provide a template configuration and the host where the peer has to be created. The operation is forwarded to the controller running on that host. The controller creates the peer by generating a unique configuration based on the template configuration. The generated configuration is placed into a unique directory which is used to isolate the peer from other peers.

Since the testbed uses process-level isolation, the generated configuration for a peer must not have conflicting system resources such as already in use network sockets. This is supported by generating an unique configuration for each peer by modifying possibly conflicting variables in it with unique and non-conflicting values. For example, port numbers for peer's services in the generated configuration are only assigned if they are not used yet. Assigned ports are then marked by the controller so that they are not assigned again. The configuration variables that are used to isolate a peer's services in the current code scope of GNUnet are listed in Appendix Table B.1.

Along with the configuration, creation of a peer requires assigning a *hostkey*. Peers in GNUnet derive their identity from their respective hostkeys and need them for communicating with other peers. A hostkey of a peer consists of a cryptographic public/private key pair and is generated by the peer upon start up if it is not present. The current implementation of the GNUnet uses Elliptic Curve Cryptography (ECC) to generate and use these hostkeys.

As observed in [EG11#1], the hostkey generation is computationally expensive and incurs a significant cost if a hostkey has to be generated for all peers in the testbed. For this reason, following the approach suggested in [EG11#1], the hostkeys are pre-computed and assigned to the peers upon their creation by copying the pre-computed hostkeys into the peers' unique directories.

Once the peers are created the driver can run them by issuing the *start peer* operation which causes the peer's controller to start GNUnet's `ARM` service of the peer. The `ARM` service, described in Section 2.4, starts the peer's services. After starting a peer, the driver may shutdown it with the *stop peer* operation which causes the peer's controller to kill the `ARM` service of the peer. When terminating, the `ARM` service causes the peer's services to shut down.

4.3 RUNNING AN EXPERIMENT

After peers are started, the driver can interact with the peers to connect them in a topology, connect to services of peers and gather data from them. The following subsections describe these operations in details.

4.3.1 *Overlay connections*

Connections between given two peers are established through the *overlay connect* operation. The operation establishes a undirected overlay connection between two given peers A and B by connecting peer B to peer A.

In GNUnet, establishing overlay connections involves peers exchanging identity and address information in messages called `HELLO` messages. The work-flow of the operation using GNUnet's `CORE` service and `TRANSPORT` service is summarised below:

1. Connect to the `CORE` service of A to get its identity and to register a *peer connect* callback through which new peer connections to A are notified
2. Connect to the `TRANSPORT` service of A and acquire its `HELLO`
3. Connect to the `TRANSPORT` service of B and offer it A's `HELLO`
4. Ask the `TRANSPORT` service of B to connect to A
5. Wait until B's connection to A is notified through the *peer connect* callback

If A and B are controlled by the same controller the work-flow is handled solely by this controller. However, if peers are distributed among various controllers, the above work-flow has to be split across the controllers. In such cases, controllers forward the request to the controller of A which then connects to the controller of B by establishing a lateral link to complete the overlay connect operation. The controller of B then takes the tasks associated with B in the work-flow.

The controller forwarding an overlay connect request also takes care that controller of A is aware of B's controller. This is essential since the controller of A has to connect to the controller of B via a lateral links. The lateral links are established using operation queues as described in Section 3.4.

4.3.2 *Accessing peers' services*

In GNUnet a peer's service is accessed through the respective client API as described in Section 2.4. Accessing a service requires the configuration of the peer. Since the configuration of a peer is modified to be unique by the respective controller process, the *service connect* operation fetches the configuration from the controller which created the peer. The testbed API then provides this configuration to the experiment driver which can use the respective service's client API to connect to the peer's service. This entire process is managed using operation queues to limit the number of sockets in use and to re-use sessions (especially their respective TCP connections) if possible.

4.4 NORMAL AND ABNORMAL EXPERIMENT TERMINATION

Once the evaluations are completed, the driver may terminate the emulation by sending a termination signal to testbed. It is also possible that a component causes the experiment to terminate abnormally, for example because a host experienced hardware problems. In this case, the termination process must initiated by the neighbours of the respective faulty component in the controller hierarchy. Thus, all control channels used among the controllers (and from controllers to peers) must enable bi-directional sensing. If either side of the channel dies, the other side must then trigger a system-wide shutdown, signalling the failure up and down the hierarchy to all other components.

The components involved in managing the shutdown include: (1) the experiment driver; (2) the testbed helper, which is started via remote shell and

thus senses and signals using the respective network connection; (3) controllers which are started by the testbed helpers and use IPC signals; and (4) the `ARM` service of a peer which manages the peer's services (also via IPC signals).

For IPC signals, signalling is usually done via UNIX pipes instead of using POSIX signals as those are not available for Windows. The pipes are created when the helper and controllers are started. For the helper process, the pipe is created from the helper's `stdin` and `stdout`, effectively binding those streams to the network connection from the remote shell. This ensures that the remote shell (which is used to start the testbed helper process on remote hosts) terminates after the respective helper process has shut down. Similarly, termination from the driver is initiated by sending a termination signal via the control pipe to the testbed helper process running the master controller. The driver then waits for the master controller process to terminate.

Upon noticing the termination signal, the testbed helper process passes the termination signal to the controller it is running and waits for it to terminate. At this point, the controller receiving the termination signal will initiate shutdown by passing the termination signal to controllers further down in the hierarchy. This is done by sending a shutdown message to their respective testbed helper processes. Each controller will then wait for the slave controllers to be terminated and then terminates itself.

As part of the shutdown routine, a controller also stops all running peers which are created by it. Peers are stopped by sending termination signal to their respective `ARM` services. The `ARM` service then passes the signal to running service processes and waits until they are terminated before terminating itself.

Since a controller waits for all its peers' `ARM` processes and its slave controllers to terminate before terminating itself, the master controller terminates only after all the controllers in the hierarchy have terminated. As a result, any misbehaving peer whose processes do not terminate after getting the termination signal block the peer's controller and the whole controller chain up to the master controller from terminating. These processes can then be easily found and inspected (by attaching a debugger) by following the blocked controller chain from the master controller.

Termination resulting from a crash in the driver or hardware failure is handled by the helper process of the master controller. The helper process monitors its connection from the driver and initiates shutdown when the pipe is closed. Similarly, when a controller crashes, the connection from its helper process and those to the helper processes started by the controller on the remote hosts are broken. Broken connections are detected and the respective remote processes then immediately begin to propagate the termination up and down the hierarchy by signalling their controllers to shutdown. This minimises the chance that processes continue to run out of control regardless of the location of the failure.

4.5 OPTIMISATIONS

In this section we describe some of the techniques to optimise the experiments. They help to ease the resources requirements: memory and CPU requirements can be lowered by reducing the number of services run per peer by sharing redundant services among multiple peers; load on the file systems can be reduced by aggregating the data locally. Furthermore, we de-

scribe how peers in the testbed are passed a unique number to individualise their run-time behaviour and how churn can be efficiently emulated.

4.5.1 *Service sharing*

The GNUnet architecture as described in Section 2.4 allows peers to share services for reasons of efficiency, reduced usage of memory and other resources (such as disk storage and ports), aggregation of data, co-ordination, etc.

The shared services are not associated with any peer or a hostkey. For this reason, only services which do not depend on other services can be shared; logic which is peer-specific has to be implemented as a non-shared peer service. For example, consider the `PEERINFO` service which stores the addresses of known peers. A peer writes addresses of new peers it learns about in the network to `PEERINFO` service and use these addresses to connect to other peers. `PEERINFO` thus has no P2P-specific logic and hence can be typically shared with other peers.

Service sharing is supported by testbed through configuration variable `SHARED_SERVICES` under the `TESTBED` section in the controller's configuration. This variable is set by the experimenter in the configuration used to start testbed. The variable's value specification allows for sharing multiple services through the following format: *<service:share> [<service:share> [...]]*. Here *service* represents the name of the service to be shared and *share* represents the number of peers among which the service has to be shared. For example, the specification

```
SHARED_SERVICES = testbed-logger:1000 peerinfo:500 resolver:500
```

causes the services `PEERINFO` and `RESOLVER` to be shared among every 500 peers and the `TESTBED-LOGGER` service to be shared among every 1000 peers. Note that sharing of services is available among peers started on the same host; sharing of services among peers running on different hosts is not advisable due to the resulting network overhead and thus not supported by the current implementation.

4.5.2 *Testbed logging*

Experiment data generated by peers can be logged to files or databases for further analysis. These files or databases may exist locally on the hosts where the peers are running or on a remote host. Aggregating this measurement data involves merging the files and poses a challenge during large scale evaluations: opening too many individual files on a host may saturate the IO subsystem; if a database is used, it may not accept the required number of open connections. Also, most of the HPC systems suitable for large scale evaluations have a shared file system across all hosts and writing to it may involve network communication which will saturate the network a large number of files are opened.

For these reasons, it is advisable to group the experiment data from a group of peers and store it locally. For example, peers in groups of 10 can be configured to write to a file instead of each writing to an individual file. This approach, however, would require some co-ordination among peers to determine which shared file is to be accessed by which peer. Such co-ordination is provided by testbed for ease of use through the `TESTBED-LOGGER` service, which should be configured as a shared service to realise the performance

gains from sharing file operations with multiple peers. The service receives measurement data from the peers through IPC and writes them to a file. Peers' services send this data through the `TESTBED-LOGGER` API.

4.5.3 *Peer identities in testbed*

Often, it is required for peers in an experiment to know their identity in the experiment to differentiate their run-time behaviour. As an example, consider the `REGEX` experiments (Section 5.2.3) where each peer has to publish unique data. This data are stored in multiple files, one per peer in the experiment. Now, to answer the question "which file's data should a peer publish", the peer needs to know its identity in the testbed. While GNUnet has peer identities which are the SHA-512 hash of their respective public keys, it is useful to assign a unique, small sequentially assigned number for each peer created in the testbed. This information can be included as part of the configuration template while creating peers. However, while using the convenience functions (Section 4.6) the creation of the peers is implicitly done by convenience function and the flexibility of providing a separate configuration template for each peer is lost.

To address this requirement, testbed writes a unique peer number to the configuration of each peer in the option `PEERID` of the `TESTBED` section. The peer can then read this value through its configuration.

4.5.4 *Emulating churn*

A given P2P protocol can be affected by churn in the underlying P2P network. Thus, it is desirable to observe the functionality and measure the performance of the protocol by subjecting it to varying amounts of churn.

Emulation of churn can be done by stopping and starting some peers in the testbed. However, this approach has a drawback: the restarted peers will have to re-establish overlay connections to existing peers to be a part of the network. This is not a problem for protocols which implement a mechanism to automatically connect to known peers. However, re-establishing a particular topology that existed prior to the peer's shutdown would be somewhat inefficient, as the testbed driver would have to remember the previous state precisely and then re-initiate the key exchange protocols for the connections which were cut when the peer was stopped.

For GNUnet, the `CORE` service has a feature that enables a hack that can make experiments with churn much more efficient — and easier to code. In GNUnet, there is not a simple notion of peers being "connected". As GNUnet consists of many services, connectivity between peers is defined per service. Specifically, the `CORE` service only notifies higher layers about connections to other peers if those other peers run the same services. When a service other than the `CORE` or `TRANSPORT` services of a peer are stopped, the peer stays `CORE`-connected to its neighbouring peers. However, the `CORE` service of the peer will notify the `CORE` services of neighbouring peers that a service has been stopped. The `CORE` services of neighbouring peer will then notify each of their relevant services that a neighbour has "disconnected". Similarly, when a service on a peer is started, the `CORE` service of that peer will send notifications to its neighbour which will then notify their respective services that a new peer has "connected".

Thus, the problem of emulating churn can sometimes be addressed by stopping merely the service under test without the peer closing `CORE` and

TRANSPORT-level connections to its neighbours. Then, to emulate a peer joining the network, the service implementing the protocol of interest is simply restarted, in which case the original neighbour connections will be retained by the CORE service.

4.6 CONVENIENCE FUNCTIONS

It is common for most experiments to create peers, start them and connect them in some overlay topology before proceeding with the actual experiment. This phase of an experiment is termed as test *start-up*. Similarly, when the experiment is completed, the peers have to be stopped and destroyed. This phase is called test *tear-down*.

Since the start-up and tear-down phases are required for all experiments, the testbed API provides convenience functions to simplify the experiment driver's code to set up the experiment.

As part of the start-up phase the API provides functions `GNUNET_TESTBED_run()` and `GNUNET_TESTBED_test_run()`. As input, these functions take parameters and a configuration from where some values are read and does the following:

1. Check if the given set of hosts can start the testbed controllers. This is done by checking if the hosts are remotely accessible and the testbed controller is available in the PATH environmental variable;
2. Start a master controller on the first host;
3. Have the master controller start slave controllers on the remaining hosts;
4. Create peers by distributing them equally across all slave controllers;
5. Start peers on all slave controllers;
6. If an overlay topology is specified in the given configuration, it is passed to another convenience function `GNUNET_TESTBED_overlay_configure_topology()` to connect the peers;
7. Schedule a task to be run when the experiment driver signals to tear-down the test. This task will shutdown running peers and cleans up the unique directories used to isolate peers.

These functions signal their completion by calling a callback function they take as a parameter. The experiment driver can then start its interactions with the peers as part of the experiment. Upon completion of the experiment, the experiment driver can signal the functions to initiate tear-down by calling `GNUNET_SCHEDULER_shutdown()` which will trigger the task scheduled by the functions to execute during tear-down.

Note that if only one host is given to the above functions, all peer specific operations are executed on that host under a single master controller. No slave controllers will be started.

The function `GNUNET_TESTBED_overlay_configure_topology()` attempts to connect a given set of peers in a given overlay topology. It uses the function `GNUNET_TESTBED_overlay_connect()` to establish an overlay connection between two peers. It supports peers to be connected in the topologies given in Table B.2.

The file describing a topology in which peers are to be connected must adhere to the following format: Each line must begin with the target peer number (peers being numbered from $[0, n)$ where n is the total number of peers in the testbed). This must be followed by a colon (':') and a list of origin peer numbers separated by '|'. All white space characters (except for newline characters) is ignored. The API will then try to connect each origin peer to the target peer.

For example, a file containing:

```
1:2|3
3:4|0
0:2
```

will result in 5 overlay connections $[2 \rightarrow 1]$, $[3 \rightarrow 1]$, $[4 \rightarrow 3]$, $[0 \rightarrow 3]$, and $[2 \rightarrow 0]$.

5 PROTOCOL EVALUATIONS

In this thesis we used HPC systems to scale our emulations as they have many hosts which are connected by a high bandwidth network. We conducted the experiments described in this Chapter on an Infiniband Cluster [Inf] and the SuperMUC supercomputer [Sup]. On the cluster we ran small (1k peers) to medium (40k peers) size evaluations and a tiny partition (30 hosts) of SuperMUC was used to run large scale (90k peers) evaluations. In this chapter we describe briefly the architecture of these systems and the experiments conducted to evaluate the performance of the testbed and the protocol evaluations of the GNUnet's Network Size Estimation (NSE) approach and its REGEX library.

5.1 EVALUATION INFRASTRUCTURE

5.1.1 *Infiniband Cluster*

For the evaluations, we used 29 hosts from the cluster, each with 4 processor cores. All the hosts have 8GB of memory except for 3 hosts which have 16GB of memory. Although the hosts are connected with an Infiniband network, we used the gigabit Ethernet network of the hosts for network communications in our emulations as GNUnet does not support Infiniband communication and IPoIB [Kas06] is not available on the cluster hosts. Data is synchronised among the hosts through a shared file system. The cluster is used with exclusive access — users who want to run an experiment in the cluster submit a reservation specifying the hosts they want to use by choosing a time interval in which the hosts are free. The reservation is then displayed on the reservation system's web page which shows the time intervals in which the hosts are reserved. Other users are expected to refer this web page before submitting a reservation and only use the hosts during their reservation interval.

The number of peers we are able to emulate on this cluster is limited to 40k peers for meaningful experiments due to the limited computing power. However, during the experiments evaluating the performance of the testbed, we are able to start 60k peers as these experiments do not have CPU intensive application logic apart from establishing overlay connections among the peers.

5.1.2 *SuperMUC*

The SuperMUC supercomputer is made up of over 9400 hosts which are divided into 19 groups called islands. The islands are further divided into 18 thin-node islands and 1 fat-node island. Thin-node islands have 512 hosts each and each host has 16 processor cores and 32GB of main memory. The Fat node-island has 205 hosts each with 40 processor cores and 256GB of main memory. Nodes in an island are connected with gigabit Ethernet and high speed Infiniband network. Islands are connected in a fat-tree [Lei85] topology interconnection network. As in the case with the Infiniband clus-

ter, since GNUnet does not support communication over Infiniband, our emulation uses the Ethernet network for its communications.

Compute tasks on the SuperMUC are submitted as batch jobs with a description specifying details such as the number of hosts required for executing the job, the maximum execution duration of the job, etc. The job scheduler executes the queued job when the sufficient hosts are available.

Considering security and the primary goal to run parallel programs, `ssh` is disabled on the thin-node islands. This prevents us from starting the controllers through `ssh`. Hence we were restricted to the fat-node island from which we used a tiny partition of 30 hosts for our evaluations.

5.2 EXPERIMENTS

In all the experiments where the emulation is distributed, the testbed is setup with the following hierarchy: the experiment driver and the master controller started by the driver are placed on a single host. Further controllers are started by the master controller on the remaining hosts. The master controller acts a supervising controller and distributes the operations from the driver. Thus, the driver controls the testbed solely through the master controller.

5.2.1 *Testbed Performance*

To measure the performance of testbed in starting evaluations of various peer sizes, we conducted experiments on a workstation class machine, a server class machine and the Infiniband cluster. For experiments on workstation and the server class machines, the driver and the controller are run on the same host.

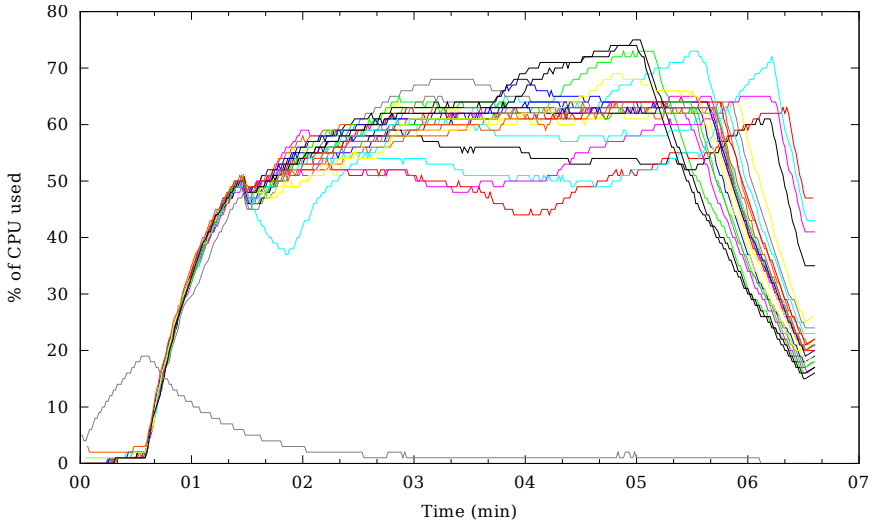
In these experiments, we measured the time taken to create peers, start them and make random overlay connections among them. We consider an overlay connection successful if the peers are connected within a timeout interval, if the timeout interval has expired the connection is considered as failed. Failed connections are retired 3 times before giving up.

Since no P2P protocol is evaluated in these experiments, we configured the peers to only run the minimal set of services required to establish connections to other peers. The measurements are shown in Tables 5.1 and 5.2.

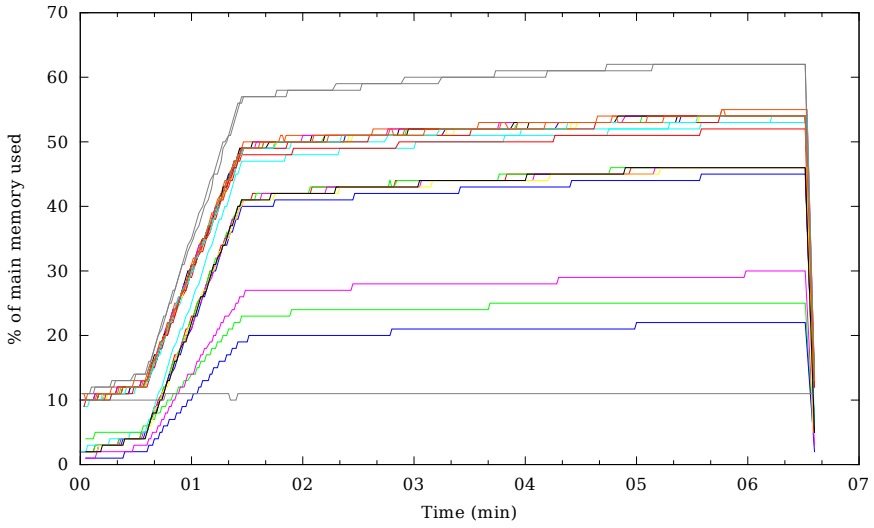
# Peers created	Time to start a peer	# Connections	# Connections per second	# Failed connections
500	18 ms	2000	15.625	0
500	18 ms	4000	17.31	0
1000	22 ms	3000	20.94	0
1000	21 ms	4000	16	0

Table 5.1: Profiling data from experiments when run on a Desktop with 2 cores and 4 GB memory

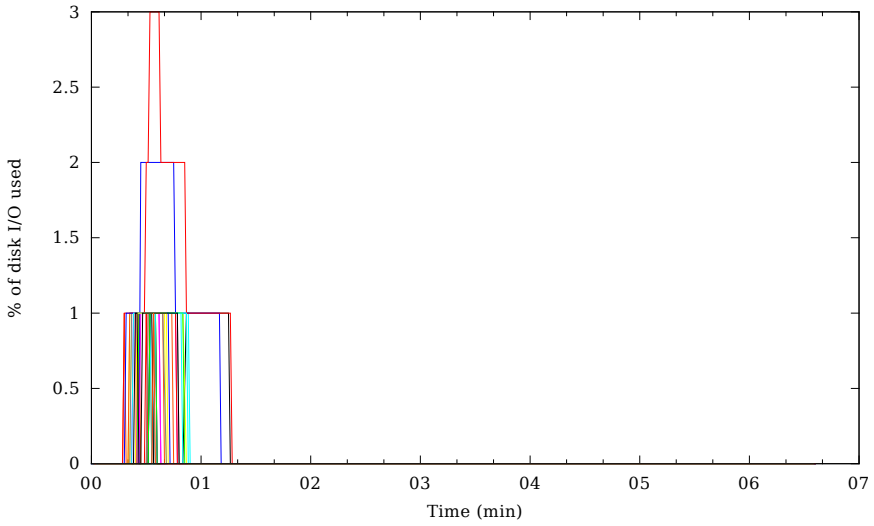
These experiments showcase our approach discussed in Section 3.3 for adapting the degree of parallelism according to the system load. We collected the monitoring data logged by the testbed controllers on the hosts of the Infiniband cluster for an experiment involving 40k peers and 100k overlay connections. Data from each host is plotted with a colour and is shown in Figures 5.1.



(a) CPU consumption



(b) Memory consumption



(c) Disk IO load

Figure 5.1: Resource consumption of testbed on 29 hosts of the Infiniband cluster while setting up an evaluation run with 40k peers and 100k random connections among them. Data recorded from each host is drawn with a unique colour.

# Peers created	Time to start a peer	# Connections	# Connections per second	# Failed connections
1000	0.063 ms	1008	77.53	0
10000	0.3462 ms	10000	93.45	1
10000	0.3458 ms	40000	341.88	1
40000	1.225 ms	40000	291.97	2
40000	1.2 ms	100000	335.57	8
60000	1.35 ms	120000	307.69	11

Table 5.2: Profiling data from experiments run on the Infiniband cluster

Note that in Figure 5.1.a, the node exhibiting the initial spike at 30 sec. is the host running the master controller and the driver. The peculiar behaviour when compared to other hosts is due to the driver compressing the configuration templates as part of the *create peer* operation. The demand for CPU on this host is remains low after creating peers as no peers are started on it. Consequently, the memory demand for this host is also less as shown in Figure 5.1.b. For the rest of the nodes, the initial spike in both CPU and memory demand is due to starting of peers. The following fluctuating pattern for CPU demand is due to adaptive parallelism; towards the end, the observed steep fall starting signifies the completion of overlay topology and the end of the experiment.

The continuous increase in memory demand while establishing overlay connections is attributed to the increase in the state information each peer maintains for each of its neighbours. The difference in the memory demand among hosts is due to hosts having different amounts of main memory and to the different amounts of free memory at the start of the experiment.

Observing the demand for disk IO shown in Figure 5.1.c, the peek in demand on all hosts at the beginning of the experiment is due to the controllers creating peer directories with their configurations and host keys. After creating the peers, no further demand is present as IO operations are disabled in the peers' services for this experiment.

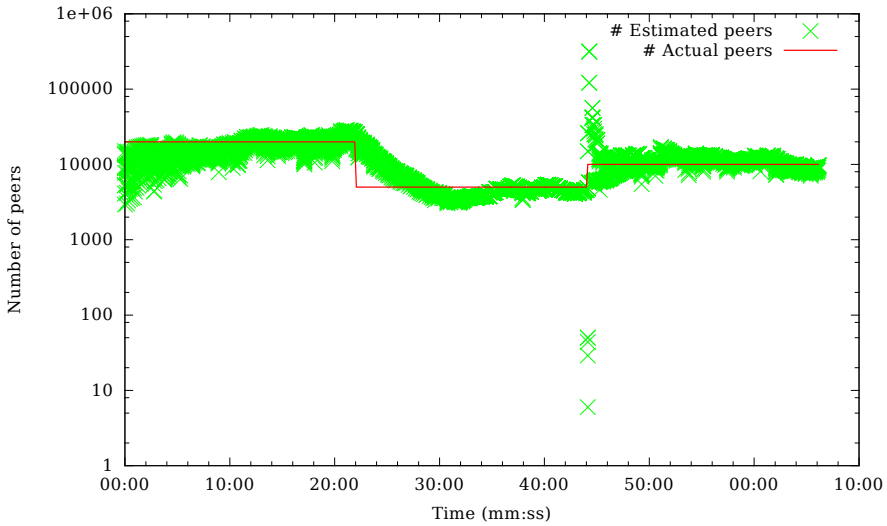
While experiments were planned to evaluate testbed performance on SuperMUC, but due to a recent security incident we were not able to run these.

Comparing with the results published in [EG11#1], our implementation lags behind in terms of number of overlay connections established per second. This can be attributed to the differences in the hardware used for evaluations and the recent code changes in GNUnet. It should also be noted that our implementation achieves the demonstrated performance without requiring any manual tuning.

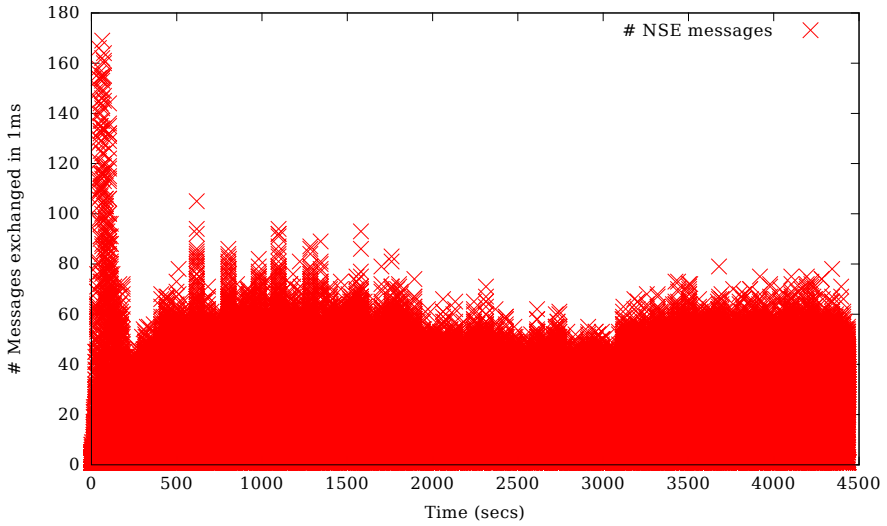
5.2.2 *Network Size Estimation*

NSE protocols are used to find the total number of peers participating in a given P2P system. Size estimations from these protocols can help applications and other protocols perform better [EG11#2] [Eug+03]. The quality of the estimation protocol is determined by the accuracy of its prediction and other factors such as keeping the number of interactions (messages transferred) and computation load to a minimum; resistance to manipulation by adversaries is also desired.

The NSE protocol demonstrated in [EPG12] is one such approach which maintains a good size estimate while keeping the total number of ex-



(a) Size estimation



(b) NSE messages exchanged

Figure 5.2: NSE evaluations of 20000 peers on the Infiniband cluster. The step sizes are 20000, 20000, 5000, 5000, 10000, 10000 with step interval of 11 minutes

changed messages low. All peers calculate the size estimate in several rounds by flooding a message from a peer whose identity is closest to a key generated from current time. This approach differs from other NSE algorithms [Mas+06] [Kos+05] by requiring all peers to participate in the estimation and hence be able to have a uniform value of estimation. To defend against Sybil attacks, the approach requires a proof-of-work to be done before participating in the network estimation rounds. As a proof-of-work the peers have to find an integer whose hash has to match a globally configured number of bits of the peer's identity.

This approach is implemented in GNUnet as the `NSE` service. A peer's `NSE` service provides size estimation to other services of the peer. This service requires the `CORE` service to provide application-level encrypted connections to the `NSE` services running on neighbouring peers. Additionally, `TRANSPORT` and `PEERINFO`, services are required to provide connections to other peers and information about known peers respectively.

In these experiments, network size estimates from the `NSE` services of various peers are evaluated by starting and stopping peers during the evaluation run. Along with these values, the distribution of the total number of NSE protocol messages communicated between the peers in a fixed time interval is also of interest. It helps to evaluate how much traffic the protocol generates in the P2P network. This data is generated in the form of a time stamp which is logged to the `TESTBED-LOGGER` by every `NSE` service whenever it sends or receives a protocol message.

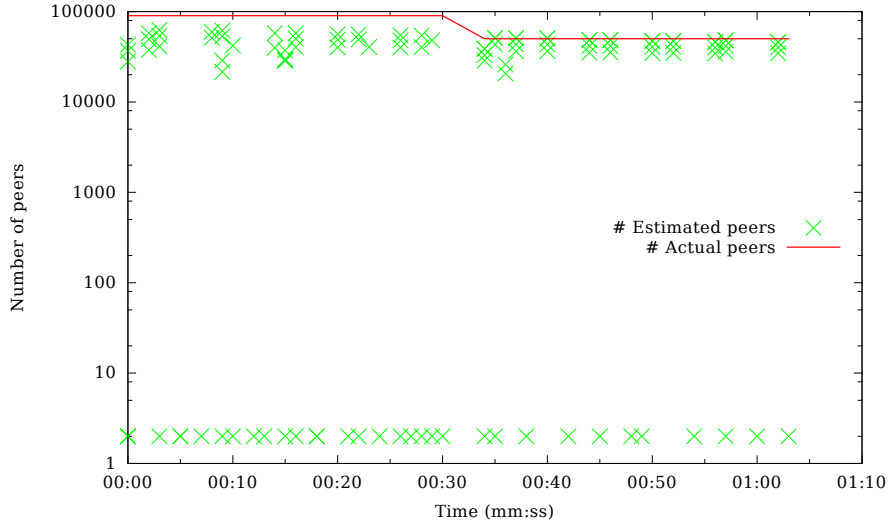
To expedite the emulation run, the round interval for the `NSE` services is reduced and the number of bits required for the proof-of-work computation are set to 0. Additionally, the services `PEERINFO` and `TESTBED-LOGGER` are configured as shared services. The corresponding configuration is given in Appendix A.1.1.

The driver for these experiments uses the convenience functions described in Section 4.6 to setup the controller hierarchy, create and start peers, connect them in an overlay topology. To emulate churn, the experiment is divided into steps with each step specifying the number of peers that should be run when the step is active. The idea is to get the network estimates during these steps and see how they change with a change in the number of running peers. Additional input options for the driver are shown in the Appendix A.1.2.

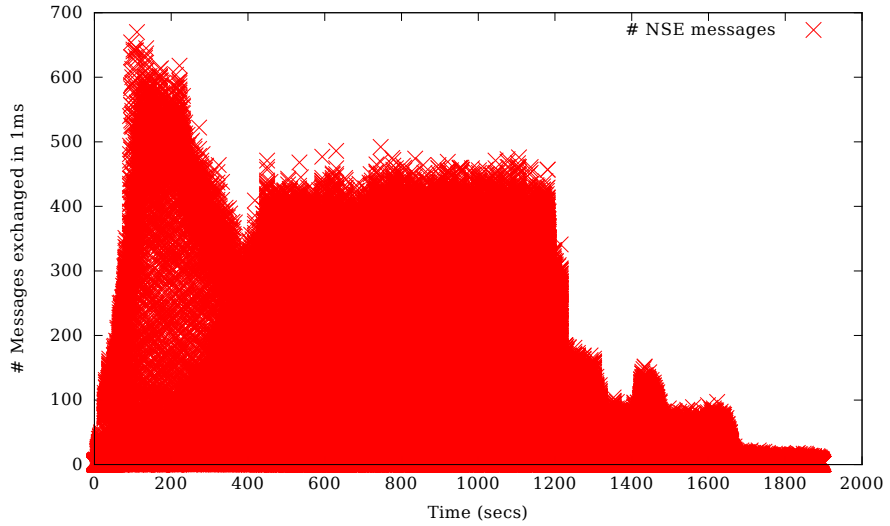
Once the testbed is ready, the driver connects to the `NSE` service of a given number of peers. These peers are chosen depending on the total number of peers running in a given step. The driver logs the size estimations from these peers' services until a given step interval has passed. It then proceeds to the next step by either starting or stopping the `NSE` services of peers to emulate churn as described in Section 4.5.4 to equal the total number peers required for that round.

We evaluated the `NSE` service on the Infiniband cluster and on 30 hosts of SuperMUC. On the cluster evaluations are conducted with up to 20k peers connected in a small world topology (Appendix B.2). The evaluation results are shown in Figure 5.2.

On the SuperMUC, with 30 nodes we were able to evaluate up to 90k peers (Figure 5.3) connected in a scale free topology (Appendix B.2). The incorrect low estimates from some peers may be due to the scale free topology; with 90k peers, some peers may have too many neighbours and hence drop connections from new peers who may remain unconnected. As a result, the overall network graph is not connected; the data suggests that the scale free



(a) Size estimation



(b) NSE messages exchanged

Figure 5.3: NSE evaluations of 90000 peers on 30 nodes of SuperMUC. The step sizes are 90000, 50000 with step interval of 30 seconds

topology created a network with at least four partitions. Thus, in the future, we might want to adapt the rules for the construction of scale-free graphs to ensure a connected graph even if peers exceed their connection limits.

5.2.3 *REGEX*

The REGEX application [Sze12] implemented in GNUnet provides distributed evaluation of regular expressions. It can be used for capability discovery among peers in the P2P network. Peers advertise their capabilities as a regular expressions using the REGEX application. Other peers searching for these capabilities will express them as a string to the REGEX application which tests if any of the published regular expressions match it. If a match is found, the peer which advertised the matching expressions is considered to have one of the capabilities expressed in the queried string.

The REGEX application uses a DHT for storing the advertised regular expressions and matching them. When peers advertise their capabilities as a regular expression, the REGEX application constructs a Deterministic Finite Automaton (DFA) from the given expressions with the accepting states containing information about the peers' identity. It then publishes each state of the DFA and its transition edges in the DHT. Consequently, searching for peers with capabilities expressed as a regular expression is accomplished by looking up the starting state corresponding to the start of the given string in the DHT and iteratively looking-up in the DHT and following the transitions to other states until an accepting state is reached.

Evaluations of the REGEX application were conducted in [Sze12] with emulation using the testbed. The motivation for these experiments is to achieve Internet routing independently in a P2P overlay: peers offering access to an Autonomous System (AS) (effectively acting as a border gateway) advertise their Internet Protocol (IP)-prefixes as regular expressions. Access to the hosts in the AS can be achieved by running the VPN-EXIT application at the gateways peers. The VPN-EXIT application allows tunnelling of Virtual Private Network (VPN) traffic in GNUnet overlay. Thus, routing to a host in a given AS can be implemented by looking up the IP address (expressed as a string) of the destination host through the REGEX application and then tunnelling the traffic to the gateway peer.

The experiments are evaluated by advertising IP-prefixes obtained from The Cooperative Association for Internet Data Analysis (CAIDA) as regular expressions and matching the strings that represent the IP address with them; tunnelling of VPN traffic is not considered. At each peer, the advertising of the IP-prefix is done by a daemon which is started as one of the peer's service. The daemon reads the AS IP-prefix data from a flat-file to determine the regular expression it has to advertise based on its unique peer number (see Section 4.5.3 for how peer identities are implemented).

After a peer has advertised its regular expression, a random peer starts searching for the advertiser by matching a string which is described by the advertised regular expression. The second peer waits for a small interval for the DHT to propagate the published regular expression's state information. The search is deemed successful by the second peer if the given regular

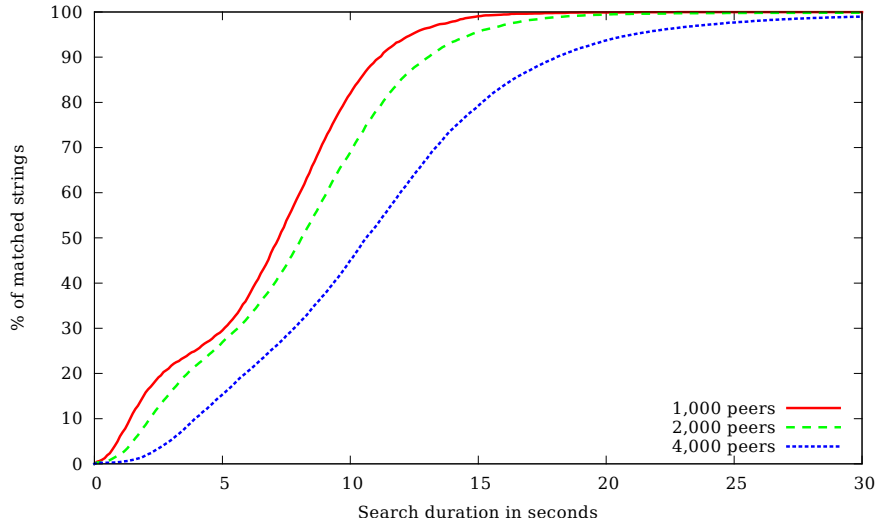


Figure 5.4: Time to match strings with regular expressions

expression can be traversed by the selected peer until its accepting state before a timeout interval.

Since advertising a regular expression induces load on the DHT, the time at which a peer advertises its AS IP-prefix data is crucial – if all peers advertise at the same time the system may get thrashed and become unresponsive. However, advertising sequentially will slow the evaluation and may cause resource under-utilisation. For this reason, the advertisements are started in parallel but with a bound on their parallelism.

Evaluations for the `REGEX` were made on the Infiniband cluster. Due to the requirement for running the `DHT` service and additional `REGEX` application logic at each peer, evaluation sizes of up to 4000 peers were possible. In these evaluations we measured the time taken to match a string after its corresponding regular expression is published. The peers in the evaluations are connected in a random graph topology with each peer having 40 random connections. The evaluation results of the `REGEX` application are shown in Figure 5.4.

6 CHALLENGES WITH HPC SYSTEMS

In the previous chapter we demonstrated our emulation method by running evaluations on HPC systems: the Infiniband cluster [Inf] and the SuperMUC [Sup] supercomputer. These systems are designed to run parallel programs developed either using OpenMP [DM98] or Message Passing Interface (MPI) [GLS99] libraries and are shared among multiple users by reserving exclusive accesses to partitions of the system. The workload from users is either managed manually by the users (through a reservation system as in the case of the Infiniband cluster) or by a scheduler as in the case of SuperMUC. The presence of this scheduler poses unique challenges for resource allocation, access and job execution since the testbed has to interface with the scheduler for setting up and starting emulation runs. We describe these challenges in detail and the resolution strategies used to address them in the following sections.

6.1 RESOURCE ALLOCATION

The simple resource allocation for the Infiniband cluster allows for an experimenter to manually specify the hosts to use for emulation when her reservation begins. However, on the SuperMUC, resource allocation is handled by a scheduler which requires an user to submit the evaluation runs as jobs. The scheduler executes the job by running the binary specified in the job's description when the required number of hosts are available in the system. This means that the specific set of hosts allocated by the scheduler for a job are determined dynamically for each job. Hence, the testbed cannot be configured to use a predefined list of hosts.

This problem also often applies implicitly for parallel programs developed with OpenMP or MPI libraries, which address it by reading the allocated hosts from a host-file. The host-file is created by the parallel environment execution wrapper executing the parallel program. Examples for such parallel environment execution wrappers are *mpiexec*[1], *mpirun*[2] and *poe*[3]. These wrappers interface with the scheduler to determine the allocated nodes where they start multiple instances of the parallel program. The instances of the parallel program then establish connections among themselves by using the host-file as a database to determine their locations.

Taking the same approach as parallel program, this problem is addressed by starting the experiment driver as a parallel program through one of the parallel environment execution wrapper available on the SuperMUC. The testbed then reads the hosts-file and starts the emulation on the allocated nodes.

6.2 RESOURCE ACCESS

Once the allocated hosts are determined, testbed needs to access them using Secure Shell (SSH) to start the controllers. However, due to security reasons

1 http://linux.die.net/man/1/mpiexec
2 http://linux.die.net/man/1/mpirun
3 http://www-03.ibm.com/systems/software/parallel/

and to avoid misuse SSH access between the allocated nodes is restricted in the SuperMUC. We also observed such restriction on other HPC systems like the Leibniz Rechner Zentrum (LRZ) compute cluster.

Currently, the problem was addressed by requesting the administration of SuperMUC to permit SSH access which has granted it to us within the hosts of the single fat-node island. This dependency on SSH should be made optional and further communication mechanisms which are widely supported in HPC systems (for example, OpenMPI messaging) ought to be added to testbed.

6.3 EMULATION START-UP

When the experiment driver is started as a parallel program through a parallel environment execution wrapper, the wrapper starts multiple instances of the driver program on the allocated nodes. Since, we only require one instance of the driver all other instances need to be terminated, else they start multiple testbeds and hence multiple evaluations which overloads the system.

This is addressed by having a helper program which is developed as a parallel program using OpenMPI to start a single instance of the driver. Since the helper program uses OpenMPI, its instances can determine a leader among themselves when multiple of them are started by the parallel environment execution wrappers. A single instance of the driver is then started by the leader, whereas the other instances terminate.

7 FUTURE WORK

During the experiments, we came up with additional features that would make emulations more efficient. We describe these in this chapter as directions for future work.

7.1 BARRIERS

In the experiments conducted in Chapter 5, the experiment driver terminates an experiment upon its completion. Determining the completion of an experiment is trivial in these experiments as the experiment's control flow is entirely implemented in the driver logic. Some experiments may, however, benefit from distributing the control flow among the peers. For example, in a planned experiment involving the GNUnet file-sharing application peers can be given some autonomy to publish and download files chosen randomly from a given set. This makes the control flow logic simpler at the experiment driver as the driver does not require to control actions at each peer. It also helps to parallelise the control flow — multiple peers may publish and download simultaneously. On the other hand, as the control flow is now distributed, determining the completion of the experiment requires extra co-ordination among the peers.

The required co-ordination is similar to the *barrier* synchronisation method used in parallel programming where threads executing in parallel are blocked when they their control flow reaches a barrier; their execution is resumed when the barrier is crossed *i.e.* when all or a predefined number of threads reach a barrier [TY86].

Implementing support for barrier co-ordination in the testbed helps experiments with control flow distributed among peers to determine experiment completion efficiently: the control flow of peers is blocked upon reaching a barrier marking its completion; once all peers' control flow reaches the same barrier, the driver can initiate the next stage of the experiment. Thus, the barrier synchronisation allows for the experiments with distributed control flow to be divided into steps where transitions between steps are guarded by barriers. This ensures that all peers' control flow is synchronised during the starting of each step.

The type of barriers described so far block the control flow until the control flows of all peers cross the barriers. Another type of barriers called fuzzy barriers [Gup89] allows applications to separate signalling that they are done with a particular phase from the blocking operation that prevents the system from executing code that requires the previous phase to be completed by all other activities. This type of barrier, which would in particular allow a peer to continue to serve requests from other peers in the current phase, would likely be the most appropriate type for an enhanced testbed API.

7.2 SIMULATION OF BANDWIDTH, LATENCY & LOSS

Since our approach simulates the underlay network without simulating network characteristics such as link bandwidth, latency and data loss, fidelity

of the evaluations remains affected as the connections between peers are not subjected to the effects observed in real networks.

The loss in fidelity can thus be mitigated by simulating the underlay network with the aforementioned network characteristics observed in real networks. However, as the isolation required for separating peers' processes in our approach does not virtualise the network interfaces, but shares them among the peers' processes, the network interfaces cannot be configured to simulate network characteristics of a underlay link between two peers without effecting the underlay links between other peers. Thus, support for simulating bandwidth, latency and loss is to be handled in the overlay network. This subjects the traffic to similar effects observed at the underlay network as the overlay links are established on top of underlay links.

Support for simulating network characteristics at the overlay links is recently provided in GNUnet by the `TRANSPORT` service. This support has to be configured by the testbed to allow the creation of overlay links in emulations with more realistic network characteristics.

Additionally, tools to gather the network characteristics from real networks should be developed. Such tools would profile real networks to generate network profiles comprising of the characteristics observed in the real world. These network profiles could then be used in emulations to simulate realistic network conditions.

7.3 SIMULATING NAT AND FIREWALLS

A consequence of exhausting Internet Protocol version 4 (IPv4) addresses spaces in the Internet is the presence of Network Address Translation (NAT) gateways. When a user runs a P2P application in networks with these gateways, connections from the peers outside the user's network to the user's peer are often restricted by the gateways although connections from the user's peer to peers outside the network may succeed. We refer to this ability of a peer to be able to be connected freely by other peers as the *underlay visibility* of that peer. In networks with NAT gateways, the underlay visibility of a peer is often restricted to the peers present in the same network.

Support for hiding a peer's underlay visibility is recently provided by the `TRANSPORT` service. This is implemented using a blacklist which determines which incoming connections a peer should accept. However, underlay visibility is not yet simulated using in testbed as the underlay network is a full mesh which allows for any two peers in the testbed to be connected. A future version of the testbed should support simulating underlay visibility using the functionality offered by the `TRANSPORT` service.

8 CONCLUSION

In this thesis we have demonstrated how emulation of P2P protocols can be scaled by taking an approach to distribute the emulation among multiple hosts and making the approach adapt the number of parallel operations automatically to fit the given experiment. We demonstrated that our method prevents the hosts from thrashing and facilitate running the evaluations at reasonable speed without manual tuning on a wide variety of systems.

From the experiments, we have shown that HPC systems are suitable candidates for large scale emulation of P2P networks. Given the distributed nature of P2P systems, the only obstacle to scaling experiments on such systems is the management of the experiment itself. By distributing the control, going as far as making each node execute an individualised profile for the external interactions of the system, we show that the experiments can scale — within the limitations of the P2P protocol and the efficiency of the implementation under test.

BIBLIOGRAPHY

[BA99] A.-L. Barabási and R. Albert. "Emergence of scaling in random networks". In: *science* 286.5439 (1999), pp. 509–512.

[Ban+98] J. Banks et al. *Handbook of simulation*. Wiley Online Library, 1998.

[Bar+03] P. Barham et al. "Xen and the art of virtualization". In: *ACM SIGOPS Operating Systems Review* 37.5 (2003), pp. 164–177.

[Ben+06] T. Benzel et al. "Experience with DETER: a testbed for security research". In: *Testbeds and Research Infrastructures for the Development of Networks and Communities, 2006. TRIDENTCOM 2006. 2nd International Conference on*. 2006, 10 pp.–388.

[BHK07] I. Baumgart, B. Heep and S. Krause. "OverSim: A Flexible Overlay Network Simulation Framework". In: *IEEE Global Internet Symposium, 2007*. 2007, pp. 79–84.

[Bol+06] R. Bolze et al. "Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed". In: *International Journal of High Performance Computing Applications* 20.4 (2006), pp. 481–494. eprint: http://hpc.sagepub.com/content/20/4/481.full.pdf+html.

[CFS09] R. Chertov, S. Fahmy and N. B. Shroff. "Fidelity of network simulation and emulation: A case study of TCP-targeted denial of service attacks". In: *ACM Trans. Model. Comput. Simul.* 19.1 (Jan. 2009), 4:1–4:29.

[Chu+03] B. Chun et al. "PlanetLab: an overlay testbed for broad-coverage services". In: *SIGCOMM Comput. Commun. Rev.* 33.3 (July 2003), pp. 3–12.

[Cla+01] I. Clarke et al. "Freenet: A Distributed Anonymous Information Storage and Retrieval System". English. In: *Designing Privacy Enhancing Technologies*. Ed. by H. Federrath. Vol. 2009. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 46–66.

[Coh08] B. Cohen. *The BitTorrent protocol specification*. 2008.

[Din+08] T. T. A. Dinh et al. "Large Scale Distributed Simulation of p2p Networks". In: *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*. 2008, pp. 499–507.

[DM98] L. Dagum and R. Menon. "OpenMP: an industry standard API for shared-memory programming". In: *Computational Science Engineering, IEEE* 5.1 (1998), pp. 46–55.

[DMS] R Dingledine, N Mathewson and P Syverson. *TOR: the onion router. Tor Project/EFF*. https://www.torproject.org/.

[EG11#1] N. S. Evans and C. Grothoff. "Beyond Simulation: Large-Scale Distributed Emulation of P2P Protocols". In: *4th Workshop on Cyber Security Experimentation and Test (CSET 2011)*. USENIX Association. San Francisco, California: USENIX Association, 2011.

[EG11#2] N. Evans and C. Grothoff. "R5n: Randomized recursive routing for restricted-route networks". In: *Network and System Security (NSS), 2011 5th International Conference on*. IEEE. 2011, pp. 316–321.

[EPG12] N. Evans, B. Polot and C. Grothoff. "Efficient and secure decentralized network size estimation". In: *NETWORKING 2012*. Springer, 2012, pp. 304–317.

[ER61] P. Erdos and A. Rényi. "On the evolution of random graphs". In: *Bull. Inst. Internat. Statist* 38.4 (1961), pp. 343–347.

[Eug+03] P. T. Eugster et al. "Lightweight probabilistic broadcast". In: *ACM Trans. Comput. Syst.* 21.4 (Nov. 2003), pp. 341–374.

[Fre+12] C. Freire et al. "Automated Deployment and Customization of Routing Overlays on Planetlab". In: *Testbeds and Research Infrastructure. Development of Networks and Communities*. Ed. by T. Korakis, M. Zink and M. Ott. Vol. 44. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer Berlin Heidelberg, 2012, pp. 240–255.

[FV07] K. Fall and K. Varadhan. "The network simulator (ns-2)". In: *URL: http://www. isi. edu/nsnam/ns* (2007).

[GLS99] W. Gropp, E. L. Lusk and A. Skjellum. *Using MPI-: Portable Parallel Programming with the Message Passing Interface*. Vol. 1. MIT press, 1999.

[Gup+03] I. Gupta et al. "Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead". In: *Peer-to-Peer Systems II*. Springer, 2003, pp. 160–169.

[Gup+05] D. Gupta et al. "To infinity and beyond: time warped network emulation". In: *Proceedings of the twentieth ACM symposium on Operating systems principles*. SOSP '05. Brighton, United Kingdom: ACM, 2005, pp. 1–2.

[Gup+11] D. Gupta et al. "DieCast: Testing Distributed Systems with an Accurate Scale Model". In: *ACM Trans. Comput. Syst.* 29.2 (May 2011), 4:1–4:48.

[Gup89] R. Gupta. "The fuzzy barrier: a mechanism for high speed synchronization of processors". In: *ACM SIGARCH Computer Architecture News*. Vol. 17. 2. ACM. 1989, pp. 54–63.

[Han+12] N. Handigol et al. "Reproducible network experiments using container based emulation". In: *Proc. CoNEXT* (2012).

[HG11] M. Herrmann and C. Grothoff. "Privacy-Implications of Performance-Based Peer Selection by Onion-Routers: A Real-World Case Study Using I2P". In: *Privacy Enhancing Technologies*. Ed. by S. Fischer-Hübner and N. Hopper. Vol. 6794. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 155–174.

[Hib+08] M. Hibler et al. "Large-scale Virtualization in the Emulab Network Testbed". In: *USENIX Annual Technical Conference*. Ed. by R. Isaacs and Y. Zhou. USENIX Association, 2008, pp. 113–128.

[Inf] *Infiniband cluster at LRR-TUM*. http://www.lrr.in.tum.de/Par/arch/infiniband/.

[Isd+10] T. Isdal et al. “Privacy-preserving P2P data sharing with OneSwarm”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 40. 4. ACM. 2010, pp. 111–122.

[JH11] R. Jansen and N. Hooper. *Shadow: Running Tor in a box for accurate and efficient experimentation*. Tech. rep. DTIC Document, 2011.

[Kan+07] J. Kangasharju et al. “ChunkSim: simulating peer-to-peer content distribution”. In: *Proceedings of the 2007 spring simulaiton multiconference - Volume 1*. SpringSim ’07. Norfolk, Virginia: Society for Computer Simulation International, 2007, pp. 25–32.

[Kas06] V. Kashyap. “IP over InfiniBand (IPoIB) Architecture”. In: (2006).

[Kos+05] D. Kostoulas et al. “Decentralized Schemes for Size Estimation in Large and Dynamic Groups”. In: *Network Computing and Applications, Fourth IEEE International Symposium on*. 2005, pp. 41–48.

[Kot+06] N. Kotilainen et al. “P2PRealm - peer-to-peer network simulator”. In: *Computer-Aided Modeling, Analysis and Design of Communication Links and Networks, 2006 11th International Workshop on*. 2006, pp. 93–99.

[Leb09] M. Leber. “Global IPv6 Deployment Progress Report”. In: *Hurricane Electric,[Online]* (2009).

[Lei85] C. Leiserson. “Fat-trees: Universal networks for hardware-efficient supercomputing”. In: *Computers, IEEE Transactions on* C-34.10 (1985), pp. 892–901.

[LHM10] B. Lantz, B. Heller and N. McKeown. “A network in a laptop: rapid prototyping for software-defined networks”. In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. Hotnets-IX. Monterey, California: ACM, 2010, 19:1–19:6.

[Li+05] J. Li et al. “A performance vs. cost framework for evaluating DHT design tradeoffs under churn”. In: *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*. Vol. 1. 2005, 225–236 vol. 1.

[Lin+05] S. Lin et al. “Simulating large-scale P2P systems with the WiDS toolkit”. In: *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005. 13th IEEE International Symposium on*. 2005, pp. 415–424.

[Mak+12] N. Makris et al. “Cross-Testbed Experimentation Using the Planetlab-NITOS Federation”. In: *Testbeds and Research Infrastructure. Development of Networks and Communities*. Ed. by T. Korakis, M. Zink and M. Ott. Vol. 44. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer Berlin Heidelberg, 2012, pp. 373–376.

[Mas+06] L. Massoulié et al. “Peer counting and sampling in overlay networks: random walk methods”. In: *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*. PODC ’06. Denver, Colorado, USA: ACM, 2006, pp. 123–132.

[MBR+03] G. S. Manku, M. Bawa, P. Raghavan, et al. “Symphony: Distributed hashing in a small world”. In: *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*. Vol. 4. 2003, pp. 10–10.

[MJ09] A. Montresor and M. Jelasity. “PeerSim: A scalable P2P simulator”. In: *Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on*. 2009, pp. 99–100.

[MM02] P. Maymounkov and D. Mazieres. “Kademlia: A peer-to-peer information system based on the xor metric”. In: *Peer-to-Peer Systems*. Springer, 2002, pp. 53–65.

[Nai+07] S. Naicken et al. “The state of peer-to-peer simulators and simulations”. In: *SIGCOMM Comput. Commun. Rev.* 37.2 (Mar. 2007), pp. 95–98.

[Nak08] S. Nakamoto. “Bitcoin: A peer-to-peer electronic cash system”. In: *Consulted* 1 (2008), p. 2012.

[Nit] *NITOS Wireless Testbed - Network Implementation Testbed Laboratory*. http://nitlab.inf.uth.gr/NITlab/index.php/testbed.

[PAGL09] J. Pujol-Ahullo and P. Garcia-Lopez. “PlanetSim: An extensible simulation tool for peer-to-peer networks and services”. In: *Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on*. 2009, pp. 85–86.

[RD01] A. Rowstron and P. Druschel. “Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems”. In: *Middleware 2001*. Springer. 2001, pp. 329–350.

[Rhe+05] S. Rhea et al. “OpenDHT: a public DHT service and its uses”. In: *SIGCOMM Comput. Commun. Rev.* 35.4 (Aug. 2005), pp. 73–84.

[Rit01] J. Ritter. *Why gnutella can't scale. no, really*. 2001.

[Spr+06] N. Spring et al. “Using PlanetLab for network research: myths, realities, and best practices”. In: *SIGOPS Oper. Syst. Rev.* 40.1 (Jan. 2006), pp. 17–24.

[Sto+03] I. Stoica et al. “Chord: a scalable peer-to-peer lookup protocol for internet applications”. In: *Networking, IEEE/ACM Transactions on* 11.1 (2003), pp. 17–32.

[Sup] *SuperMUC: Petascale System at Leibniz-Rechenzentrum*. https://www.lrz.de/services/compute/supermuc/. 2012.

[Sze12] M. Szengel. “Decentralized Evaluation of Regular Expressions for Capability Discovery in Peer-to-Peer Networks”. Masters. Garching bei Muenchen: Technische Universitaet Muenchen, 2012, p. 100.

[TY86] P. Tang and P.-C. Yew. “Processor self-scheduling for multiple-nested parallel loops”. In: *Proceedings of the 1986 international conference on parallel processing*. 1986, pp. 528–535.

[Vah+02] A. Vahdat et al. “Scalability and accuracy in a large-scale network emulator”. In: *SIGOPS Oper. Syst. Rev.* 36.SI (Dec. 2002), pp. 271–284.

[Var+01] A. Varga et al. “The OMNeT++ discrete event simulation system”. In: *Proceedings of the European Simulation Multiconference (ESM'2001)*. Vol. 9. sn. 2001.

[Vis+09] K. Vishwanath et al. "ModelNet: Towards a datacenter emulation environment". In: *Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on*. 2009, pp. 81–82.

[Wei+11] E. Weingärtner et al. "Slicetime: A platform for scalable and accurate network emulation". In: *Proceedings of the 8th USENIX conference on Networked systems design and implementation*. USENIX Association. 2011, pp. 19–19.

[Whi+02] B. White et al. "An integrated experimental environment for distributed systems and networks". In: *SIGOPS Oper. Syst. Rev.* 36.SI (Dec. 2002), pp. 255–270.

[Zei+11] N. Zeilemaker et al. "Tribler: Search and stream". In: *Peer-to-Peer Computing (P2P), 2011 IEEE International Conference on*. 2011, pp. 164–165.

[Zha+04] B. Y. Zhao et al. "Tapestry: A resilient global-scale overlay for service deployment". In: *Selected Areas in Communications, IEEE Journal on* 22.1 (2004), pp. 41–53.

APPENDIX

EXPERIMENT DRIVERS

A.1 NSE DRIVER

A.1.1 *service configuration*

```
[nse]
PORT = 12114
ACCEPT_FROM = 127.0.0.1; 10.0.0.0/8;
UNIXPATH = /tmp/test-nse-service-nse.unix
BINARY = gnunet-service-nse
AUTOSTART = NO
INTERVAL = 3 s
WORKBITS = 0
PROOFFILE = $SERVICEHOME/nse.proof
```

A.1.2 *Input options*

Input parameters for the NSE experiment driver (Section 5.2.2):

```
totakura@nautophone:~/repos/gnunet/src/nse$ ./gnunet-nse-profiler --help
nse-profiler
Measure quality and performance of the NSE service.
Arguments mandatory for long options are also mandatory for short options.
  -C, --connections=COUNT    limit to the number of connections to NSE
                               services, 0 for none
  -c, --config=FILENAME      use configuration file FILENAME
  -d, --details=FILENAME     name of the file for writing connection
                               information and statistics
  -H, --hosts=FILENAME       name of the file with the login information for
                               the testbed
  -h, --help                 print this help
  -L, --log=LOGLEVEL         configure logging to use LOGLEVEL
  -l, --logfile=LOGFILE      configure logging to write logs to LOGFILE
  -o, --output=FILENAME      name of the file for writing the main results
  -p, --peers=NETWORKSIZESPECNumber of peers to run in each round, separated
                               by commas
  -V, --verbose              be verbose (print progress information)
  -v, --version              print the version number
  -w, --wait=DELAY           delay between rounds
Report bugs to gnunet-developers@gnu.org.
GNUnet home page: http://www.gnu.org/software/gnunet/
General help using GNU software: http://www.gnu.org/gethelp/
```

B MISCELLANEOUS

B.1 UNIQUE CONFIGURATIONS

The variables of interest while generating unique configuration from configuration template are shown in Table B.1

Variable	Description
SERVICEHOME	The working directory of the peer's service. This variable is present for every service of the peer
PORT	The port number on which the peer's service has to listen. This variable is modified if the given configuration template has it specified for a service. The actual port number to use is determined at run time by querying for an unused port
UNIXPATH	The path to the UNIX socket through which a service can be accessed. This serves as an alternative to IP-based IPC for local clients trying to access the service
ACCEPT_FROM	This variable serves as an ACL to determine which hosts can access the services of the peer. The value for this variable, if present in the template configuration, will be appended with the IP address of the host running the experiment driver. If the variable is not found in template configuration then it is set to the IP address of the host running the experiment driver

Table B.1: Variables modified by testbed in generated configurations

The variable SERVICEHOME is generated by appending the local peer number to a temporary directory. For example, if /tamp/gnunet-testbed is the temporary directory, the SERVICEHOME for the first peer created will be set to /tmp/gnunet-testbed/0.

The variable PORT is set to free port number. A port is determined to be free if a IP listen socket can be opened on it. Ports assigned to a configuration are marked as used in a bitmap which is consulted later to find the next port number to check.

The variable UNIXPATH is set by appending SERVICEHOME with the name of the service. For example, the CORE service of a peer which is created after the first peer is set to have UNIXPATH as $SERVICEHOME/core.sock.

B.2 SUPPORTED TOPOLOGIES

The following are the overlay topologies supported by the convenience functions:

Topology	Description
Clique	All peers are connected to each other.
Line	Peers are connected in a line, one after another.
Ring	Line topology with an additional link between the last peer and the first peer
2D-torus	Peers are connected to form a two dimensional torus topology.
Erdos-Renyi	Peers are connected randomly [ER61]. This topology takes an argument specifying how many random connections are to be made
Small-world	Peers are connected in a 2D-torus topology with some additional random connections among peers. This topology takes an argument specifying how many random connections are to be made.
Small-world ring	Peers are connected in a ring topology with some additional random connections among peers. This topology takes an argument specifying how many random connections are to be made.
Scale-free	Peer connections are made by observing the power-law distribution and hence the generated topology resemble closely the topologies present in nature [BA99].
File	Peers are connected according to the data given through a file.

Table B.2: Supported overlay topologies

C

CORRIGENDA

The following errors were corrected in this thesis after its publication:

- The deployment size of the evaluations carried out on the SuperMUC is mentioned as 150k peers at multiple places. While we have attempted this peer deployment, due to an unknown error we were not able to gather the evaluation data. Hence, multiple references to 150k peer deployment size has been changed to 90k in the Abstract and pages 27, 32 and 33. Additionally, the Figure 5.3 also incorrectly mentions the step sizes as 150000, 90000 which do not correspond to the presented graph. The corrected step size is 90000, 50000. — *27. July 2013*