Mesmerizer: A Effective Tool for a Complete Peer-to-Peer Software Development Life-cycle

Roberto Roverso
Peerialism Inc.
Stockholm, Sweden
roberto@peerialism.com

Sameh El-Ansary
Peerialism Inc.
Cairo, Egypt
sameh@peerialism.com

Alexandros Gkogkas
Peerialism Inc.
Stockholm, Sweden
alex@peerialism.com

Seif Haridi
Royal Institute of Tech. (KTH)
Stockholm, Sweden
haridi@kth.se

ABSTRACT

In this paper we present what are, in our experience, the best practices in Peer-To-Peer(P2P) application development and how we combined them in a middleware platform called Mesmerizer. We explain how simulation is an integral part of the development process and not just an assessment tool. We then present our component-based event-driven framework for P2P application development, which can be used to execute multiple instances of the same application in a strictly controlled manner over an emulated network layer for simulation/testing, or a single application in a concurrent environment for deployment purpose. We highlight modeling aspects that are of critical importance for designing and testing P2P applications, e.g. the emulation of Network Address Translation and bandwidth dynamics. We show how our simulator scales when emulating low-level bandwidth characteristics of thousands of concurrent peers while preserving a good degree of accuracy compared to a packet-level simulator.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems—*distributed applications*; C.4 [**Performance of Systems**]: Modeling techniques; D.1.3 [**Programming techniques**]: Concurrent programming—*distributed programming*; D.2.11 [**Software Engineering**]: Software Architectures—*domain-specific architectures*; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*discrete event*

General Terms

Design, Performance

Industry track 2011, March 22-February 22

DOI 10.4108/icst.simutools.2011.245501

Keywords

peer-to-peer systems design, simulation-based development, NAT emulation, bandwidth dynamics emulation

1. INTRODUCTION

Peer-to-Peer (P2P) systems have passed through a number of evolution eras. Starting from being an exotic practice in hacker communities to a rigorously researched and decently funded academic field. Nowadays, products based on P2P technologies such as Bittorrent and Skype are mainstream brands in internet technologies. Despite of that, while algorithms and research ideas about P2P systems are abundant, software engineering practices of developing P2P systems, especially in an industrial setting are less shared. Compared with the process of developing web-based applications, the amount of best practices that has been iterated, re-factored and publicly shared within the communities is huge. Examples include model-driven frameworks such as Ruby on Rails [1] or Django [2] or communication patters like AJAX [16] and COMET [3] and their variants. We argue that while the art of developing P2P applications in terms of shear algorithmic complexity is far beyond web-applications, there are very few best practices shared on how to develop P2P systems.

The point of this paper is to the share best practices that worked for Peerialism. We do that by articulating three main areas where we think we have gained maturity. The first is simulation tools. Namely, how they are an integral part of the software development and not just an assessment tool. We highlight how a clear concurrency model can significantly simplify development and then which modeling aspects are critical for a successful P2P system design and implementation process.

Simulation-Based Development cycle. P2P algorithms are in general complex due to the high amount of exchanged messages, asynchrony and the fact that failures are the norm rather than the exception. Consequently, simulation is not a luxury but a necessity when validating P2P protocol interactions. Even a very few lines of code running simultaneously on one thousand peers result in interactions that are rather challenging to debug. We started with the common practice of authoring the algorithms on our own discrete-event simulator and, when the algorithms were mature enough, we transitioned to real implementation. How-

ever, maintenance of a dual code base and irreproducibility of bugs were main concerns that led us to attempt injecting a discrete event simulator underneath our production code. The results of this effort, where mainly a simulated network, simulated time and simulated threading were provided, were published in the MyP2PWorld system [33]. The main advantage of the approach was that developers wrote exactly in the same style they were familiar with. This approach made it possible to debug complex interactions of hundreds of peers on a single development machine and also share reproducible scenarios with the development team. In a sense, the simulated mode served as an extremely comprehensive integration testing tool. That is an achievement which is hard to obtain in uncontrolled, real-world deployments.

Over time, we found that we are using component-based frameworks like Guice [18] extensively to organize and decouple various parts of our code. We realized quickly that the task of switching between real and simulated runs could be achieved in a more elegant fashion using component frameworks where for example the network is a component with one interface and two implementations, one real and one simulated. We noticed that others in the field have later independently reached the same conclusion and we see systems like Kompics [6] and Protopeer [14] which adapted the same practice. We expect more wide dissemination of systems like these in the future.

Clear Concurrency Model. In general, the classical way of dealing with concurrency is to use threads. For I/O intensive applications, the network programming community has been advocating the asynchronous/event-based concurrency model. That is more or less an established consensus. In a programming language like Java, one could observe the transition of the standard library to provide more off-the-shelf code for asynchronous I/O based on Java NIO [19]. With that model, any network related activity is done in an event-based fashion, while the rest of the concurrent modules in the application are written using threads. Our first application was developed in such a way; the co-existence of blocking (thread-based) and non-blocking (event-based) concurrency models rendered the applications much harder to design and maintain. The practice that we finally settled on was to unify the concurrency model by having a pure event-based system. We have also seen that frameworks like Twisted for Python [4] have a similar concept. It is worth stressing that, when injecting a DES underneath real application code, the non-blocking model is much more suited.

More Realistic Network Model. Other than being injected underneath the real application code, our discrete-event simulation layer is in principle very similar to every other peer-to-peer simulator out there. That said, we have two unique features. The first is the ability to simulate the behavior of NAT boxes and the second is an efficient and accurate bandwidth allocation model. For the former, we have taken all the real-life complexities we found in actual deployments and implemented that logic in a NAT box emulator integrated in our framework. Up to our knowledge, this is the first emulator of its kind. For the latter, we have surveyed how others crafted bandwidth allocation in their simulators. Packet-level simulators like NS-2 [29] are the winners in terms of accuracy but fall short on efficiency for the desired scale of thousands of peers. A more efficient solution is to use flow-based simulation where we have found that the max-min fairness approach [9] is the most-widely used model. Nevertheless, implementations of max-min fairness vary a lot, not only in accuracy but in efficiency as well. We have implemented our own max-min based bandwidth allocation model where we substantially improved on efficiency without sacrificing too much accuracy. The ability to provide a more realistic network model is important. In the absence of NAT box emulation, P2P algorithm designers would have a very naive picture of phenomena like long connection setup times, the impossibility of connectivity between different combinations of NAT boxes, and hence peers behind them. Similarly, sloppy models for bandwidth allocation result in overly optimistic estimation of data transfer delays, especially in P2P networks where a lot of transfers are taking place simultaneously.

Our P2P Development Framework. We have combined all of our best-practices in a P2P Java middleware called Mesmerizer. At the moment, all Peerialism applications are written on top of that platform. The rest of this paper is dedicated to explaining how our best practices are realized in the Mesmerizer framework. We start by describing the Mesmerizer programming model in Section 2 as well as its internals in Section 3. Then, we present the execution modes available to users in Section 4 and give more details about the network layer, both real and simulated, in Section 5. Finally, we present our conclusions and future work in Section 6.

2. THE MESMERIZER FRAMEWORK

Applications developed in Mesmerizer consist of a set of *components*. Every component has an *interface* and one or more corresponding *implementation(s)*. An interface is bound to a component implementation with an explicit *binding*. A component instance belongs to a certain *group*. Groups contain one or multiple component instances, have explicit identifiers and define the scope of communication between components. A component may communicate with other components in a message-passing fashion using *events*. When an event is triggered by a component, Mesmerizer broadcasts it to the the component group it belongs. Events may be triggered for immediate execution, e.g. messaging, or future execution, e.g. timeouts. Inter-group communication is allowed by using explicit addressing of groups. Static tunnels between groups can be defined to automatically forward one or many types of events from one group to the other in order to avoid the need of explicit addressing. Every handler processes a single type of event. Handlers belonging to a certain component may be executed *sequentially*, i.e. a component instance processes a single handler at a time. This allows for simple concurrency semantics and isolation of components' state. However, event handlers can be defined as *concurrency-safe*. In that case, no synchronization control is applied, i.e. many "safe" handlers may be executed, while only a single "unsafe" handler is running at the same time. Explicit *filters* may be defined as a protection in order to avoid execution of handlers based on the runtime characteristics of events.

3. IMPLEMENTATION

Mesmerizer is implemented in Java as an extension of Google Guice. Guice is a lightweight injection framework. It was principally developed to alleviate the use of the explicit factory pattern in Java. It also provides a tight degree of

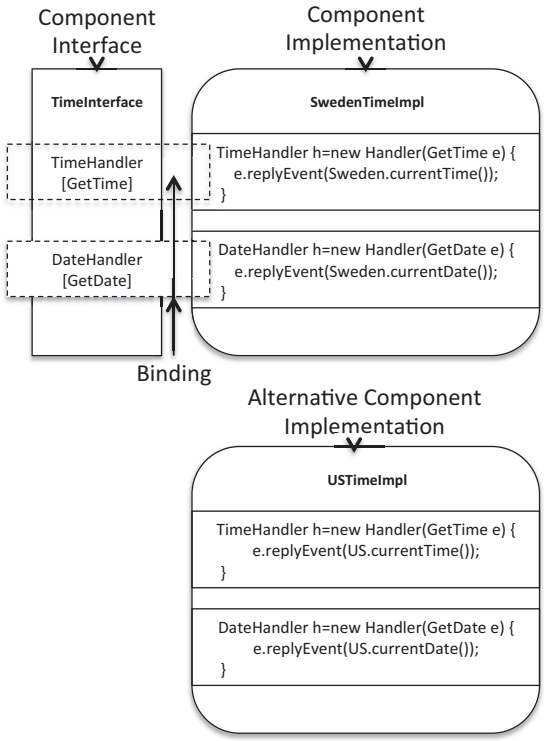

Figure 1: Example Component Interface and implementations

```
.....
// Binding
bind(TimeInterface).to(SwedenTimeImpl).as(
    Scopes.GroupSingleton)
.....

// Interface
trait TimeInterface extends ComponentInt {
 @Handler def timeHandler():Handler[GetTime]
 @Handler def dateHandler():Handler[GetDate]
}

// Implementation
class SwedenTimeImpl extends TimeInterface {
 def timeHandler():Handler[GetTime] = {
  return timeHandler;
 }
 def dateHandler():Handler[GetDate] = {
  return dateHandler;
 }
 val timeHandler = new Handler[GetTime]() {
   def handle(e: GetTime):Unit = {
   e.reply(new RespTime(currentTime(SWEDEN)))
 }}

 val dateHandler = new Handler[GetDate]() {
  def handle(e: GetDate):Unit =  {
     e.reply(new RespDate(currentDate(SWEDEN)))
 }}
}
```

Listing 1: Example Component Interface and corresponding Component Implementations in the Scala language

control on how, which and when implementations of a certain interface should be instantiated. In Guice, the instantiation mechanism can be configured using explicit static *bindings*, i.e. interface to implementation mappings, and *scopes*. By default, the library makes use of a completely stateless scope, instances of a specific interface implementation are allocated every time the application requests a new instance. The library provides two stateful scopes: singleton, which causes Guice to return always the same instance of a certain interface, and a request scope, used mainly for Servlet applications, which allocates instances on a request-by-request basis. In general, Guice is principally used for unit-testing. It is common practice to swap dependencies of a Guice component with mock implementations in order to test its functionalities independently.

Mesmerizer uses Guice bindings to map component interfaces to corresponding implementations. Component interfaces are simple Java interfaces which define a number of handlers. Handlers are objects which inherit from a specific abstract Handler class and are statically typed by the event class which they take as an argument. Figure 1 illustrates an example of component interface and two corresponding component implementations. In this case, a binding has been created between $TimeInterface$ and $SwedenTimeImpl$.

Component instances live in the context of groups. Upon the creation of a group, a set of the aforementioned Guice bindings must be provided together with the corresponding allocation policy. The latter defines if a component instance should be considered as a singleton or not in the context of the group or of the application. Component instantiation calls are made on group instances. Internally, Mesmerizer groups make use of Guice's scope mechanism to create and keep track of instances in their context. As a consequence of this design, two different groups may be able to bind the same interface to different implementations. When a component implementation gets instantiated, Mesmerizer uses Guice to parse both the component interface and the bound implementation. After the parsing, the framework registers each handler as a possible destination for the event it is typed with. This information is then used upon event triggering to retrieve the handlers to execute. As a design choice, we let many different components implement a handler for the same event type, whereas we allow only a single handler for a specific event type on the same component.

In Figure 2 we show the structure of Mesmerizer. The most important part of the platform is the *Core*, which contains the implementation of the model's semantics and wraps around the Guice library. It provides mainly the group and component instantiation mechanisms, in addition to the registration and resolution mechanisms used for event routing. The actual handling of events, that is the scheduling and the execution, is carried out by the *Scheduler*, while the *Timer* is entitled with the task of handling timeouts and recurring operations. A glue *Interface and Management layer* is added to handle setup of one or multiple application instances and provide logging through the *Logger* embedded in the system. As we can see from the figure, one or many component groups can be created by an application, in this case G_1 and G_2 and filled with component instances C_1, C_2 and C_3, which are different instances of the same component

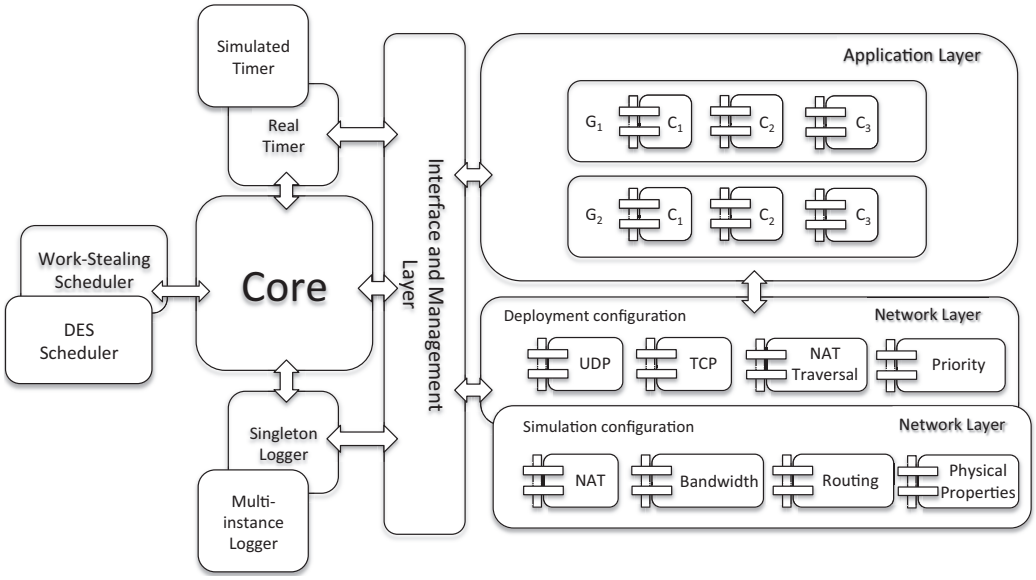

Figure 2: The structure of the Mesmerizer Framework

implementations residing in the two different groups.

3.1 Scala Interface

We implemented an interface layer over Mesmerizer to be able to develop components in Scala [30]. Scala is a programming language designed to express common programming patterns in an easier and faster way. It is an object-oriented language which also supports functional programming. Our experience is that Scala allows for faster prototyping than Java. Algorithms and other complex routines can be written in much shorter time than in Java and expressed in a clearer and more compact way making it easier for the programmer/researcher to implement, understand and improve both its logic and code. An example of the Scala code corresponding to Figure 1's depiction of component interface, implementation and binding is shown in Listing 1.

We are currently working on a layer to interface the Python language with Mesmerizer by using the Jython library [22]. In principle, any programming language that compiles to Java bytecode can be interfaced with Mesmerizer. The Mesmerizer framework provides two different execution modes: *simulation* and *deployment*. The former allows for a strictly controlled execution of multiple instances of an application. It enables both large-scale reproducible experiments and smaller-scale testing/debugging of applications over an emulated network layer.

Deployment allows for parallel processing of events. In this mode, one or multiple application instances are executed using a concurrent environment while network services are provided by a library which enables TCP and UDP communication between hosts.

4. EXECUTION MODES

The design of Mesmerizer allows an application to be run either in simulation or in emulation mode by simply changing some of the Mesmerizer's system bindings, namely the Scheduler, Timer, Logger and Network layer components as shown in Figure 2.

4.1 Simulation Mode

In simulation mode, multiple instances of the same application are spawned automatically by Mesmerizer according to a specified configuration provided to the Management Layer. Event execution in this case is controlled by a Scheduler based on a single-threaded Discrete Event Simulator (DES). During execution, triggered events are placed into a FIFO queue and the corresponding handlers executed in a sequential manner. Our previous experience in using a multi-threaded DES based on pessimistic lock-stepping [24], where events of the current time step are executed in a concurrent fashion, has shown that the amount of overhead required for this technique to work is larger than the actual benefits. We found the event pattern to be very sparse for the applications we developed using Mesmerizer: a live streaming platform and a Bittorrent client. We noticed that the synchronization burden required to achieve barrier synchronization between consecutive time intervals is far greater than the speed-up obtained by the actual processing of the few events present in the "concurrency-safe" intervals.

Trying to scale simulation, we mostly concentrated our efforts in improving the performance of what we experienced to be the biggest bottleneck in our P2P simulations: emulating bandwidth allocation dynamics of the network. We will show in Section 5.2.2 how the Mesmerizer simulation environment performs in terms of scalability when emulating such phenomena and its level of accuracy with respect to other more costly solutions.

Simulation Setup. We provide a set of APIs which enable users to fully configure the execution of multiple application instances and carefully control the behavior of the simulated network layer. Typically in simulation mode, Mesmerizer isolates an application instance in its own component group containing all of its components. Application instances communicate using the same network inter-

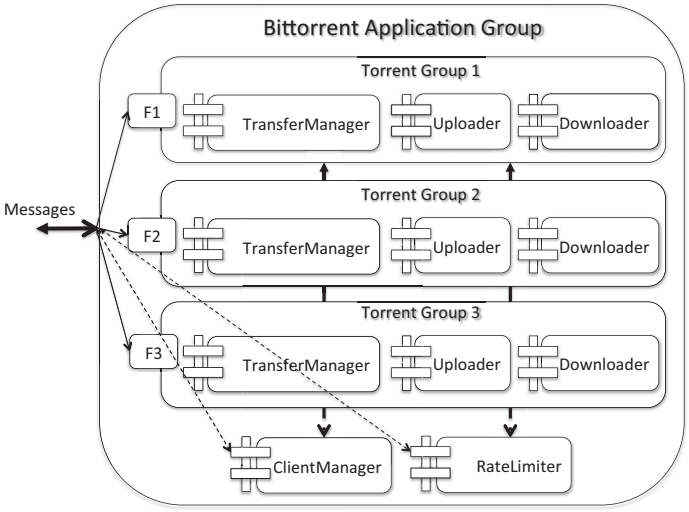

Figure 3: The structure of our Bittorrent Application

face provided in deployment mode. Messages are however routed to an emulated network layer which models a number of characteristics found in real networks.

For large-scale experiments, we implemented a number of churn generators based on probabilistic and fixed time behaviors which can be configured either programmatically or through an XML scenario file. The emulated underlying network can also be configured in the same way. The Mesmerizer simulator allows for the creation of scenarios containing a predefined number of peers interconnected using routers and NAT boxes [36] on simple end-to-end topologies or more complicated consumer LAN infrastructures and/or complex multi-layered corporate networks. The simulation APIs make possible to define bandwidth capacities for each peer/router in the network, dictate the behavior of NAT boxes and configure in detail network characteristics such as delay patterns, packet loss and link failures.

4.2 Deployment Mode

Deployment allows for the execution of application instances in a concurrent environment. In this mode, the handlers of the application's components are processed on multiple threads concurrently. From a component's instance point of view, a number of its *safe* handlers can run together at once, however, its *unsafe* handlers will be executed sequentially. On the other hand, many unsafe handlers of different components may be active at the same point in time. In this mode, execution is controlled by a concurrent Scheduler which implements the work-stealing paradigm introduced by Blumofe et al [10] on a number of Java threads, similarly to Kompics [6]. Work-stealing is an efficient scheduling scheme for multi-core machines which is based on an queuing mechanism with low cost of synchronization. We make use of the work-stealing paradigm in the following way: when an event gets scheduled, Mesmerizer finds the component instances which subscribed to that particular event type and hands them to the Scheduler. The Scheduler then checks the availability of a number of free threads corresponding to that of the passed handlers. If enough free threads are found, it schedules the handlers for immediate execution. If no sufficient number of threads is available, the scheduler distributes randomly the remaining handlers to the waiting queues of the currently occupied threads. When a thread completes the execution of a handler, it tries either to take an event from its queue or, if its queue is empty, it steals handlers from another thread's queues. In our experience, this type of scheduling guarantees a good level of fairness and avoids starvation of handlers.

In Figure 3 we show the structure of a Bittorrent client that we developed using Mesmerizer, which is currently deployed in our test network. The application is made by two basic components which reside into the main *Bittorrent Application* component group: the *Client Manager* and the *Rate Limiter*. The former is entitled with the task of setting up and remove Bittorrent transfers, while the latter controls load balancing and priority levels between transfers. When a new torrent file is provided to the application, the $ClientManager$ creates a component group for the new transfer, for instance $TorrentGroup1$. The group contains all transfer's components, which are instances of the interfaces $Downloader$, $Uploader$ and $TransferManager$. Which implementation should be used for the aforementioned interfaces is defined in a Guice binding when adding the torrent. We designed a number of different implementations of the transfer components which provide various transfer strategies, such as partial in-order or random. This kind of design allow for the use of multiple transfer policies in the same client by simply providing the right binding when adding new transfers. During transfer setup, the $ClientManager$ also proceeds to create automatic tunneling of message events from the network layer to $TorrentGroup1$ using filters based on message characteristics. We use the mechanisms of tunneling and filtering for automatic inter-group routing and multiplexing of incoming messages respectively. Routing of events is also carried out internally to the application between $Uploader/Downloader$ component instances and both the $ClientManager$ and the $RateLimiter$.

The latter in particular has the important task of keeping the view of all transfer rates and dynamically adjust, by issuing the correct events, the uploading/downloading rates of the *Uploader* and *Donwloader* components, when they excess the priority level or max speed.

5. NETWORK LAYER

We provide two different sets of components to be used as network layer: one for deployment and another for simulation mode. In deployment mode, components need to transfer messages, i.e. remote events, to other remote hosts using the TCP or UDP protocol. In simulation mode instead, the network layer is entitled with the task of modeling those same transfers between a number of application instances which are running in the same context, i.e. the same instance of the Mesmerizer framework. We detail the composition of the network layer in both modes in the following sections.

5.1 Deployment Configuration

Our network layer deployment includes a number of components that offer TCP and reliable UDP communication to the overlying applications through a simple event-based interface. Our TCP and UDP components deliver out-of-the-box support for transparent peer authentication and encrypted data exchange. We have implemented two components to achieve NAT traversal and improve peer-to-peer connectivity; support for explicit NAT traversal protocols such as UPnP and NAT-PMP, and state-of-the-art techniques for UDP hole-punching, based on our previous work [34]. If direct connectivity between two peers cannot be achieved, the network layer automatically relays the communication over a third host. However, we use this feature only for signaling purpose since relaying traffic is a very expensive operation, in particular for data intensive applications such as video streaming or content delivery platforms where the amount of data to be transfered is significant.

On top of NAT traversal and reliable communication, the deployment network layer provides three strategies for traffic prioritization based on different UDP congestion control methods. For low priority traffic, we implemented the LEDBAT delay-based congestion control [35], which yields with respect to other concurrent TCP streams. For what we call fair level of priority, or medium, we adopted a variation of the TCPReno [27] protocol which enables equal sharing of bandwidth among flows generated by our library and other applications using TCP. Finally, when the delivery of data is of critical importance, the library provides high priority through an adaptation of the Mul-TCP [12] protocol. The level of priority can be dynamically chosen on a flow-to-flow basis at runtime.

5.2 Simulation Configuration

In Simulation mode, we make use of a number of components which emulate different aspects of the network. For instance, on the IP layer, we provide routing and NAT emulation through the corresponding components. For modeling lower layer network characteristics, we implemented components that model bandwidth allocation dynamics, non-deterministic delays and packet loss. As mentioned in Section 1, some of these emulated characteristics are found in almost all P2P simulators. We detail the NAT and bandwidth emulation components; the first being notable for its novelty and the second for its high level of efficiency/accuracy trade-off.

5.2.1 NAT Emulation

Network Address Translators constitute a barrier for peer-to-peer applications. NAT boxes prevent direct communication between peers behind different NAT boxes [36]. Even though an attempt has been made to standardize Network Address Translation [7], in particular to improve support for Peer-To-Peer applications, not all vendors have complied to the standard. This is either because most of the routers ship with legacy NAT implementations or because manufacturers claim the standard to be too permissive. In particular, in the context of corporate NAT implementations, the translation behavior may vary drastically between different vendors or even different models of the same manufacturer.

In our previous work, we have tried to classify most of the behaviors of current NAT implementations using a real deployment of our test software. The outcome of this effort is a model that encompasses 27 types of NATs as a combination of *three* behavioral policies: filtering, allocation and mapping. NAT traversal is a pairwise connection establishment process: in order to establish a communication channel between machines behind two different NAT boxes, it is necessary to carry out a connection setup process dictated by a NAT traversal strategy, which should vary according to the type of the two considered NAT boxes. Given a set of *seven* known traversal strategies, the NAT problem translates to understanding which strategy should be used in each one of the 378 possible combinations of the 27 NAT types. On top of that, each traversal strategy has its own configuration parameters. These parameters could be as detailed as deciding which source port should be used locally by the two source hosts to initiate the setup process in order to maximize the connection establishment success probability.

At first, we tried to understand the mapping between NAT type combinations and traversal strategies by formal reasoning. However, this task turned out to be too complex due to the size of the possibility space. As a consequence of that, we developed a configurable NAT box implementation which could emulate all the aforementioned NAT types. On top of it, we implemented a small simulator which would perform the process of traversal, according to the *seven* strategies, on all of the emulated 240 NAT type combinations. By using this small framework, we discovered many counter-intuitive aspects of NAT traversal and mistakes in the strategy's decision process which we would not have discovered by simple reasoning. The outcome of our effort of mapping strategies to combinations is described in [34].

For the purpose of testing the correct and non-trivial implementation of the Traversal strategies in our application, we included NAT emulation in the network layer configuration of Mesmerizer. In other words, we built an emulated network where the connection establishment process by means of NAT traversal is modeled in a very detailed manner. Thus, we were able to test not only the correctness of our implementation of the traversal strategies, but also to measure the impact of the connection establishment delays on the overlying application. Delays are very important factors to be considered when designing audio and video streaming systems, due to the time-constrained nature of the application.

The model which we designed for the emulation of NAT

boxes encompasses most of the behaviors which are found in current routers and corporate firewalls. However, after the deployment of our live streaming application on a real network, we noticed a number of exceptional characteristics, e.g. non-deterministic NAT mapping timeouts, inconsistent port allocation behaviors and the presence of multiple filtering policies on the same router. We thus tried to formalize those exceptions and integrate them into our simulator. We further improved our simulator by modeling connection establishment failures based on real-world measurements. We emulate the observed probability of success between NAT types combinations according to what we observed during our real-world tests. Currently, we still experience cases that are not covered by our emulation model and we keep improving our model based on those observations. The source code of our NAT box emulator is publicly available as an open source project [32].

The detailed emulation of NAT boxes has provided us with better insight on how peer-to-peer applications should be designed and implemented in order to avoid connectivity issues.

5.2.2 Bandwidth Modeling

It is common for P2P networks to create complex interactions between thousands of participant peers, where each peer typically has very high inbound and outbound connection degree [5] [39] [38]. Connections are used either for signaling or for content propagation. In the latter case, each peer implements intricate multiplexing strategies to speed up transmission of large chunks of data [11], thus creating complex effects on the underlying physical network which translate into varying transmission delays and packet loss at the receiving side.

Most of the existing P2P simulators abstract away network interactions by modeling only the structural dynamics of the overlay network [8] [21] [31] [26] [37] and thus totally ignoring the impact of the actual network on application performance. On the other end, accurate packet-level simulators like SSFNet [41] and NS-2 [29] can usually scale up only to a limited number of simulated peers. This limitation makes it infeasible to capture the behavior and issues of a larger P2P real-word deployment, such as the effect of network congestion on segments of the overlay network. In order to study the complex interactions between large scale overlays and the physical network, a proper network simulation model is required. The level on which the model abstracts the network transfers directly affects both its scalability and accuracy.

Flow-level network simulation focuses on a transfer as a whole rather than individual packets, introducing a viable trade-off between accuracy and scalability. A flow abstracts away the small time scale rate variation of a packet sequence with a constant rate allocated at the sender/receiver's bandwidth. The rate remains allocated for an amount of time which corresponds to the duration of the flow, i.e. the simulated packet sequence transmission time. This approach reduces drastically the number of events to be simulated. The driving force behind the event creation in flow-level simulation is the interaction between the flows since an upload/-download link might have many flows happening at the same time. A new or completed flow might cause a rate change on other flows competing for that same link's capacity. A flow rate change may also propagate further in the simulated network graph. This phenomenon is known as *"the ripple effect"* and has been observed in a number of studies [23] [15]. The impact of the ripple effect on the scalability of the model is directly dependent on the efficiency of the bandwidth allocation algorithm which is used to mimic the bandwidth dynamics.

Bertsekas and Gallager [9] introduce the concept of *max-min fairness* for modeling Additive-Increase Multiplicative-Decrease congestion control protocols like TCP. Max-min fairness tries to maximize the bandwidth allocated to the flows within a minimum share thus guaranteeing that no flow can increase its rate at the cost of a flow with a lower rate. In every network exists a unique max-min fair bandwidth allocation and can be calculated using the progressive filling algorithm [9]. The basic idea behind this algorithm is to start from a situation where all flow rates are zero and then progressively increment each rate equally until reaching the link's capacity, i.e. the sum of all flow rates of a link equals its capacity. In this algorithm, the network, including its internal structure, e.g. routers and backbone links, is modeled as an undirected graph. A recent accuracy study [13] showed that this approach offers a good approximation of the actual network behavior. Nevertheless, having to simulate the flow interactions that take place on the internal network links magnifies the impact of the ripple effect on the algorithm's scalability by making the simulation significantly slower.

In order to gain more scalability, the *GPS* P2P simulator [40] uses a technique called *minimum-share allocation*, defined in [17], which avoids the propagation of rate changes through the network. Instead, only the flow rates of the directly affected nodes are updated, i.e. only the flow rates of the uploading and downloading nodes of the flow triggering the reallocation. Not considering the cross-traffic effects of the flows obviously has a positive impact on the simulation time but also makes the model highly inaccurate. *Narses* [17] uses the same technique as GPS but it further promotes scalability by ignoring the internal network topology and considers only the bandwidth capacity of the access links of the participating peers. The result is what we call an *end-to-end* network overlay where the backbone network is completely abstracted away from the modeling and rate changes happen between pairs of peers. This is a reasonable abstraction if we consider that the bottlenecks on a P2P network usually appear in the "last mile" rather than the internet backbone. In doing so, the number of events simulated is further reduced, however in this case the inaccuracy remains since only the end-to-end effects are taken into account while the cascading effect on other nodes, as modeled by max-min fair allocation, is completely overlooked.

There exists two bandwidth allocation algorithms in the state of the art which apply the progressive filling idea on end-to-end network models, thus keeping the advantages of simulating only access links but still considering the effects and propagation of rate changes throughout the peer interconnections. The first algorithm proposed by F. Lo Piccolo et Al. [25] models the end-to-end network as an undirected graph. In each iteration, the algorithm finds the *bottleneck* nodes in the network, the nodes with the minimum fair bandwidth share available to their flows. Then it proceeds to allocate the calculated minimum fair share to their flows. The algorithm iterates until all nodes are found saturated or a rate is assigned to all their flows. The main disadvantage of this node-based max-min fair bandwidth allocation algo-

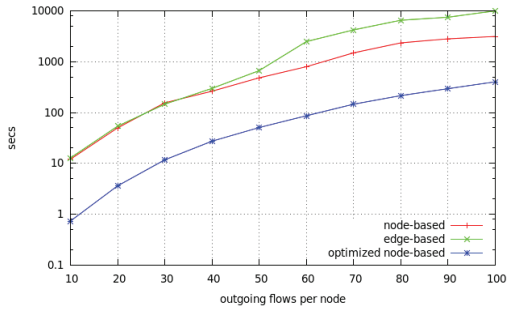

Figure 4: Performance comparison for structured network overlays with 1000 nodes and different number of outgoing flows per node.

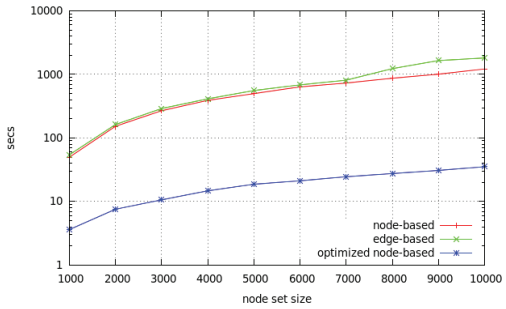

Figure 5: Performance comparison for structured network overlays with varying size and 20 outgoing flows per node.

rithm lies in the modeling of the network as an undirected graph. In order to simulate a network with separate upload and download capacities, two node instances are required per actual network peer. The memory footprint is therefore larger than the one needed to model a direct network graph.

An alternative edge-based max-min bandwidth allocation algorithm is given by Anh Tuan Nguyen et al. [28]. It is an edge-based algorithm which uses a directed network model, differently from the approaches we introduced until now. In one iteration, the algorithm calculates the minimum fair share of the two ends of every unassigned flow. Then, on the same iteration and based on the previously calculated shares, the algorithm finds the bottleneck nodes, derives the flows' rates and applies them. The algorithm iterates until all flows have a rate assigned. It is important to underline that during the second phase of each iteration, the algorithm might find one or multiple bottleneck nodes, thus assigning rates to the flows of multiple nodes at the same iteration. This edge-based max-min fair bandwidth allocation algorithm addresses the shortcoming of the undirected network modeling, that is the memory footprint. However, the algorithm performance's dependence on the edge-set size constitutes a major drawback. On top of that, a further iteration of the node set is required in order to find the saturated nodes.

It is common in large simulated networks for a new or finished flow to only affect the rates of a subset of the existing network flows, as the propagation of a rate change does not reach all nodes in the network but rather few of them. Based on this observation, F. Lo Piccolo et Al. [25] partially outline an affected subgraph discovery algorithm that can be applied on an undirected network graph.

Using this optimization algorithm before applying an undirected node-based max-min fair bandwidth allocation algorithm leads to a large performance gain. Unfortunately, F. Lo Piccolo et Al. apply this only on an undirected network model. Moreover, the authors provide only a sketch of the affected subgraph idea rather than a state-complete algorithm. In our simulator we leverage the benefits of the affected subgraph optimization on a directed network model. The result is an algorithm whose computational complexity is independent of the edge set size. Our node-based max-min fair bandwidth allocation algorithm iterates until all flows have a rate assigned. Each iteration has two phases. In the first, we find the node(s) which provide the minimum fair share by calculating the fair share of the upload and the download capacity of each node. The lower of these rates is set as the minimum fair share and the corresponding node sides (uploading or downloading) are considered saturated i.e. they constitute the bottleneck of the network in this iteration. In the second phase, we allocate this minimum fair share to the flows of each saturated node, downloading or uploading depending on their saturated side. In order to improve scalability, we adapt the affected subgraph discovery algorithm for use with directed end-to-end network models. Given a flow that triggers a bandwidth reallocation, we initiate two graph traversals that each one has as root one of the flow's end nodes. In each hop of a traversal we find the affected flows of the last reached nodes and continue the traverse to their other ends. This procedure continues until no newly affected nodes are discovered by any of the two traversals.

Evaluation.

For our scalability evaluation, we consider structured overlay scenarios with two main parameters: the size of the node set and the number of outgoing flows per node. The nodes enter the system in groups at defined time intervals and the starting times their flows are distributed uniformly in that same time interval. The destination of the flows is chosen following a specific structure, i.e. a DHT-based one. The bandwidth capacities of the nodes are chosen randomly from the set: {100Mbps/100Mbps,24Mbps /10Mbps,10Mbps/10Mbps,4Mbps/2Mbps,2Mbps/500Kbps} with corresponding probabilities of {20%,40%,10%,10%}. Our experiments show, Figures 4-5, that our node-based max-min fair allocation algorithm constantly outperforms the edge-based algorithm proposed by Anh Tuan Nguyen et al. for large-scale and structured network overlays. An important conclusion drawn from these results is that the number of connections per node has a bigger impact in the performance of the simulation models rather than the node set size. For example, the simulation of a random overlay of 1000 nodes with 70 outgoing flows requires a similar simulation time to a 10000 nodes random overlay having 20 outgoing flows per node. We also would like to point out that the performance gain when using flow-level simulation instead of packet-level is paramount. In order to simulate a random scenario of 1000 nodes with 10 flows each, a time of three orders of magnitude longer is required. Running

c^u/c^d	flows	std. deviation	avg. deviation
20/10	10	3.8±0.4%	3±0.4%
	20	3.9±0.2%	3.1±0.1%
	30	4.1±0.3%	3.3±0.2%
	40	3.4±0.2%	2.8±0.2%
	50	3±0.1%	2.5±0.2%
20/10,10/10	10	6.4±0.4%	5±0.4%
	20	6±0.4%	4.9±0.3%
	30	4.8±0.4%	3.9±0.3%
	40	3.4±0.9%	3.2±0.3%
	50	3.5±0.2%	2.8±0.2%

Table 1: Deviation of simulated transfer times.

the same scenarios using the optimization algorithm significantly reduces the simulation time. In Figure 4 we can see that the required time is one order of magnitude lower when simulating network overlays with the same size but different number of outgoing flows per node. The performance improvement is much higher, three orders of magnitude, when we increase the network size and keep the number of flows per node fixed, as shown in Figure 5.

Finally, in our accuracy study we compare the simulated transfer times of our proposed solution with the ones obtained with NS-2 for the same scenarios. In NS-2, we use a simple star topology, similar to the one used in [13] [20]. Each node has a single access link which we configure with corresponding upload and download capacities. All flows pass from the source access link to the destination access link through a central node with infinite bandwidth capacity. Unfortunately, the size of our experiments is limited by the low scalability of NS-2. We run scenarios of 100 nodes with a number of flows per node that varies between 10 and 50. The size of each flow is 4MB and the bandwidth capacities of a node are either asymmetric, 20Mbps/10Mbps, or mixed, 20Mbps/10Mbps and 10Mbps/10Mbps. The results of our experiments are shown in Table 1. We can see that our flow-level max-min fair bandwidth allocation follows the trends of the actual packet-level simulated bandwidth dynamics by a nearly constant factor throughout the experiments. We can see that the presence of the symmetric capacities affects the transfer time deviation negatively. The negative impact is more visible when fewer outgoing flows per node are used. When the links are less congested, the slower convergence of the flow rates of the nodes with smaller symmetric capacities is more apparent.

6. CONCLUSION & FUTURE WORK

In this paper we presented what we found to be the three most important practices in P2P software development: a simulation-based development cycle, a clear concurrency model, and a realistic network model when running in simulation mode. We then presented the Mesmerizer framework which we built to encompass all the aforementioned. We detailed its design and implementation and how it can be used both for controlled evaluation/testing/debugging and deployment of production code. Regarding simulation-based development, we stress how NAT box and bandwidth dynamics emulation are of vital importance when testing a large number of a P2P application instances. From the point of view of the scalability, we demonstrate that our simulation framework is able to emulate the bandwidth characteristics of thousands of peers while preserving a good level of accuracy compared to the NS2 packet-level simulator.

Our ongoing work includes the improvement of the NAT and bandwidth emulation model's accuracy and the release of all our utilities as open source software.

7. REFERENCES

[1] Ruby on rails. http://rubyonrails.org/.

[2] django. http://www.djangoproject.com/.

[3] Comet. http://cometdaily.com/.

[4] Twisted Metal for Python. http://http://twistedmatrix.com/trac/.

[5] A. Al Hamra, A. Legout, and C. Barakat. Understanding the properties of the bittorrent overlay. Technical report, INRIA, 2007.

[6] C. Arad, J. Dowling, and S. Haridi. Building and evaluating p2p systems using the kompics component framework. In *Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on*, pages 93 –94, sept. 2009.

[7] F. Audet and C. Jennings. Network Address Translation (NAT) Behavioral Requirements for Unicast UDP. RFC 4787 (Best Current Practice), Jan. 2007.

[8] I. Baumgart, B. Heep, and S. Krause. Oversim: A flexible overlay network simulation framework. In *GI '07: Proceedings of 10th IEEE Global Internet Symposium*, pages 79–84, Anchorage, AL, USA, May 2007.

[9] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, second edition, 1992.

[10] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 356 –368, Nov. 1994.

[11] B. Cohen. Incentives Build Robustness in BitTorrent. In *Econ '04: Proceedings of the Workshop on Economics of Peer-to-Peer Systems*, Berkley, CA, USA, June 2003.

[12] J. Crowcroft and P. Oechslin. Differentiated end-to-end Internet services using a weighted proportional fair sharing TCP. *ACM SIGCOMM Computer*, 1998.

[13] A. Dandoush and A. Jean-Marie. Flow-level modeling of parallel download in distributed systems. In *CTRQ '10: Third International Conference on Communication Theory, Reliability, and Quality of Service*, pages 92 –97, june 2010.

[14] G. W. et al. Protopeer: A p2p toolkit bridging the gap between simulation and live deployement. *2nd International ICST Conference on Simulation Tools and Techniques*, May 2009.

[15] D. R. Figueiredo, B. Liu, Y. Guo, J. F. Kurose, and D. F. Towsley. On the efficiency of fluid simulation of networks. *Computer Networks*, 50(12):1974–1994, 2006.

[16] J. J. Garrett. Ajax: A new approach to web applications. http://adaptivepath.com/ideas/essays/archives/000385.php, February 2005. [Online; Stand 18.03.2008].

[17] T. J. Giuli and M. Baker. Narses: A scalable flow-based network simulator. *Computing Research Repository*, cs.PF/0211024, 2002.

[18] Google guice. http://code.google.com/p/google-guice.

[19] R. Hitchens. *Java Nio*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[20] T. Hossfeld, A. Binzenhofer, D. Schlosser, K. Eger, J. Oberender, I. Dedinski, and G. Kunzmann. Towards efficient simulation of large scale p2p networks. Technical Report 371, University of Wurzburg, Institute of Computer Science, Am Hubland, 97074 Wurzburg, Germany, October 2005.

[21] S. Joseph. An extendible open source p2p simulator. *P2P Journal*, 0:1–15, 2003.

[22] The jython project. http://www.jython.org/.

[23] G. Kesidis, A. Singh, D. Cheung, and W. Kwok. Feasibility of fluid event-driven simulation for atm networks. In *GLOBECOM '96: Proceedings of the Global Communications Conference*, volume 3, pages 2013 –2017, November 1996.

[24] S. Lin, A. Pan, R. Guo, and Z. Zhang. 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*, pages 415 – 424, sept. 2005.

[25] F. Lo Piccolo, G. Bianchi, and S. Cassella. Efficient simulation of bandwidth allocation dynamics in p2p networks. In *GLOBECOM '06: Proceedings of the 49th Global Telecommunications Conference*, pages 1–6, San Franscisco, California, November 2006.

[26] A. M. Mark Jelasity and O. Babaoglu. A modular paradigm for building self-organizing peer-to-peer applications. In *Engineering Self-Organising Systems*, pages 265–282, 2003.

[27] J. Mo, R. La, V. Anantharam, and J. Walrand. Analysis and comparison of TCP Reno and Vegas. *IEEE INFOCOM '99*, 1999.

[28] A. T. Nguyen and F. Eliassen. An efficient solution for max-min fair rate allocation in p2p simulation. In *ICUMT '09: Proceedings of the International Conference on Ultra Modern Telecommunications Workshops*, pages 1 –5, St. Petersburg, Russia, October 2009.

[29] The ns-2 network simulator. http://www.isi.edu/nsnam/ns/, October 2010.

[30] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.

[31] J. Pujol-Ahullo, P. Garcia-Lopez, M. Sanchez-Artigas, and M. Arrufat-Arias. An extensible simulation tool for overlay networks and services. In *SAC '09: Proceedings of the 24th ACM Symposium on Applied Computing*, pages 2072–2076, New York, NY, USA, March 2009. ACM.

[32] R. Roverso. NATCracker Box Emulator Software. http://code.google.com/p/natcracker/.

[33] R. Roverso, M. Al-Aggan, A. Naiem, A. Dahlstrom, S. El-Ansary, M. El-Beltagy, and S. Haridi. Myp2pworld: Highly reproducible application-level emulation of p2p systems. In *Decentralized Self Management for Grid, P2P, User Communities workshop, SASO 2008*, 2008.

[34] R. Roverso, S. El-Ansary, and S. Haridi. Natcracker: Nat combinations matter. In *Proceedings of the 2009 Proceedings of 18th International Conference on Computer Communications and Networks*, ICCCN '09, pages 1–7, Washington, DC, USA, 2009. IEEE Computer Society.

[35] S. Shalunov. Low extra delay background transport (ledbat) [online]. http://tools.ietf.org/html/draft-ietf-ledbat-congestion-02.

[36] P. Srisuresh, B. Ford, and D. Kegel. State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs). RFC 5128 (Informational), Mar. 2008.

[37] N. S. Ting and R. Deters. 3ls - a peer-to-peer network simulator. In *P2P '03: Proceedings of the 3rd International Conference on Peer-to-Peer Computing*, page 212. IEEE Computer Society, August 2003.

[38] G. Urvoy-Keller and P. Michiardi. Impact of inner parameters and overlay structure on the performance of bittorrent. In *INFOCOM '06: Proceedings of the 25th Conference on Computer Communications*, 2006.

[39] C. Wu, B. Li, and S. Zhao. Magellan: Charting large-scale peer-to-peer live streaming topologies. In *ICDCS '07: Proceedings of the 27th International Conference on Distributed Computing Systems*, page 62, Washington, DC, USA, 2007. IEEE Computer Society.

[40] W. Yang and N. Abu-Ghazaleh. Gps: a general peer-to-peer simulator and its use for modeling bittorrent. In *MASCOTS '05: Proceedings of 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 425–432, Atlanta, Georgia, USA, September 2005.

[41] S. Yoon and Y. B. Kim. A design of network simulation environment using ssfnet. pages 73–78, 2009.