Exchangeable, Application-Independent Load Balancing for P2P Simulation Frameworks

Daniel Warneke
Technische Universität Berlin
Berlin, Germany
warneke@cs.tu-berlin.de

Ulf Rerrer-Brusch
Technische Universität Berlin
Berlin, Germany
urerrer@cs.tu-berlin.de

ABSTRACT

Today many peer-to-peer (P2P) simulation frameworks feature a variety of recent years' research outcomes as modular building blocks, allowing others to easily reuse these blocks in further simulations and approach more advanced issues more rapidly. However, the efforts in the field of P2P load balancing have been excluded from that development so far, as the proposed techniques often impose too many dependencies between the load balancing algorithms and the application to be put in loosely-coupled components. This paper discusses how load balancing algorithms that rely on the virtual server concept can be separated from the application and run in a modular container with a unified communication interface. We discuss design fundamentals for such a load balancing container based on a variety of existing load balancing techniques and present our implementation for the OverSim framework. With our work load balancing becomes a reusable building block for P2P applications which contributes to the process of building rich and modular simulation environments.

Keywords

Peer-To-Peer, Simulation, Load Balancing, Modularization

1. INTRODUCTION

Since the first calls for an open and extensible P2P simulation framework [13, 20] the research community has gone a long way: Starting from mainly custom-tailored solutions today many simulation frameworks [1, 18, 21] come with a variety of tools (e.g. for topology generation, controlling churn and statistics), a set of popular and ready-to-use overlay routing protocols and a highly modular design that allows other researchers to quickly adapt the simulation framework to their own needs. While more and more basic research in the field of P2P is accomplished, researchers can concentrate on more advanced issues using these frameworks.

In particular, the evaluation and testing of new applications for P2P networks can greatly benefit from these rich simulation frameworks. The demands of modern P2P applications go far beyond solving the famous "lookup" problem [6] and separating functionality into reusable, loosely-coupled building blocks can help to significantly reduce development time. However, while there has been some work [5] to promote the creation of such loosely-coupled building blocks for overlay routing protocols other components of a P2P application remain closely interwoven.

In this paper we deal with the question of how to offer load balancing for simulations of structured P2P networks in a modular and reusable manner, i.e. in a way that is open to a variety of concrete load balancing algorithms on the one hand, but independent of specific applications and routing techniques on the other hand. During the last years load balancing has been subject to wide research (e.g. [19, 7, 9, 17, 25]) and became fundamental to many applications. However, the proposed load balancing algorithms are often tied to specific routing protocols [2] or tightly coupled to certain applications [4, 10]. As a result, currently no P2P simulation environment we are aware of [1, 8, 11, 12, 14, 18, 21, 20, 23, 24] provides support for load balancing at all.

We focus on load balancing techniques that rely on the virtual server concept introduced in [22]. When using virtual servers a physical node pretends to be several distinct peers, each participating independently in the overlay network. The load of the physical node is determined by the sum of its virtual servers' loads, thus concrete load balancing algorithms can respond to heavy load by deactivating or transferring a virtual server according to specific strategies.

Compared to other approaches, load balancing based on virtual servers is especially suitable to be offered in a reusable building block, since virtual servers are typically considered to be autonomous peers that only allow little assumptions about their internals. Section 2 highlights these assumptions for a variety of popular load balancing algorithms building upon the virtual server concept. Our goal is to create an abstract load balancing container that offers a unified API to all of these algorithms so that existing P2P applications can be easily transferred into virtual servers and be managed by the algorithm within the container. In Section 3 we explain fundamental design decisions for the load balancing container and introduce the API.

As a proof of concept we implemented our load balancing container as a module for the OverSim P2P simulation framework [1]. With our implementation existing OverSim application can make use of load balancing and virtual servers without modifications to their implementation. Section 4 describes our extension and outlines how we dealt with compatibility issues. In Section 5, we evaluate our extension in terms of execution overhead and memory consumption. Finally, we conclude in Section 6.

2. RELATED WORK

The first to propose virtual servers as a concept for load balancing in P2P networks have been Stoica et al. [22]. Based on [15] they conclude that the difference in size between the largest fraction of the ID space a physical node has to manage and the smallest one can be bound by a constant factor if each physical node runs $O(log\ N)$ virtual servers (with N being the number of physical nodes in the network). Each virtual server joins the overlay with a distinct pseudo-random ID. Later Karger and Ruhl demonstrated in [16] how to achieve the same even distribution when each physical node maintains a set of $O(log\ N)$ virtual servers, but only activates one of them at a time. Both approaches only consider load imbalance as a result of unevenly sized ID space partitions. They do not use virtual servers to tackle the problem of skewed data distribution or popularity, although these two aspects are major reasons for load imbalance in many practical applications.

The distributed storage system CFS [4] uses virtual servers to deal with the different amount of storage space the participating physical nodes offer. Files are split into blocks which are equally distributed over the available virtual servers. In CFS the ID of a virtual server is determined by hashing the physical node's IP address and the virtual server's index number. Similar to [22] the authors of CFS do not address load imbalance due to varying data popularity.

A more flexible approach to load balancing with virtual servers is offered by Rao et al. [19]. The authors assume that each physical node runs several virtual servers. Load is considered as the utilization of a bottleneck resource (e.g. CPU utilization, network traffic or storage space) and the physical node's load is defined as the sum of the virtual servers' loads. In case the physical node's load exceeds the specific threshold the node is regarded as overloaded and tries to transfer one of its virtual servers to another, less-loaded node. Therefore the overloaded node shuts the respective virtual server down and advises the less-loaded node to restart a new virtual server with the same ID. In order to identify these less-loaded nodes the authors propose to maintain several directories on top of the overlay which can be used by the physical nodes to publish their current load situation. The concrete transfer strategies have later been expanded to dynamic structured P2P networks in [9].

The idea of using the overlay network to store load information has also been picked up by Zhu and Hu [25]. The authors define the physical node's load in the same way as Rao et al., though they construct a k-ary tree on top of the overlay network to maintain the information. The tree's nodes are mapped to the virtual servers in the network according to a specific scheme. The scheme guarantees that at least one leaf node of the k-ary tree is planted in a virtual server. Periodically, the physical node chooses one of its virtual servers to report its current load situation. The information is then propagated towards the tree's root node and used to coordinate rendezvous between overloaded and less-loaded physical nodes.

Contrary to the previous approaches, k-choices [17] does not start virtual servers with random or pseudo-random IDs. This load balancing technique relies on probing a constant number of candidate IDs for a new virtual server and to estimate the expected load for the physical node in advance. As a result, k-choices requires to contact the overlay network before starting a virtual server. Again, the physical node's load is defined as the sum of virtual servers' loads, however, bottleneck resource is expected to be the network bandwidth. Similar to [19] and [25] k-choices assumes that every physical node has a distinct load threshold. When the node enters the network, it starts virtual servers until its designated load threshold or the upper bound for the number of virtual server is reached.

3. DESIGN

The previous section highlighted the functionality required by a selected set of load balancing techniques which rely on virtual servers. When designing a load balancing container that is capable of running all of these techniques, two basic challenges must be solved: First, the container must provide an interface that allows the load balancing algorithms to access the requested functionality in a unified manner. Second, the container itself must implement the interface in a way that reduces the dependencies to the applications within the virtual server as much as possible.

In the following we focus on both aspects. At first we discuss the fundamental dependencies that arise between the load balancing container and the virtual servers if the entire functionality described in Section 2 should be supported. Afterwards, we propose two interfaces which unify the communication between the load balancing algorithms and the load balancing container and, optionally, between the load balancing container and the applications running inside virtual servers.

3.1 Fundamental dependencies

A virtual server can be considered to be a self-contained block which is assigned a unique ID from the overlay ID space and runs its own instance of the overlay routing protocol (Tier 0) as well as application tiers (Tier 1 - Tier m). Typically virtual servers do not share components and, hence, can be started and stopped independently. Although the load balancing algorithms outlined in the previous section deal with virtual servers only in terms of these self-contained blocks, their features implicitly create dependencies to the concrete application running inside that must be kept in mind.

The first dependency emerges when the load of a virtual server is regarded as the consumption of a bottleneck resource other than network bandwidth. CPU load or storage consumption is hard to capture in simulation environments that focus on message flows. Usually these values must be provided by the application running inside the virtual server and, consequently, the application must be modified to implement a respective interface.

However, for many P2P applications the occurring network traffic remains the major bottleneck [3]. In this case an application-independent way of collecting the virtual server's load information is possible. As depicted in Figure 1, the load balancing container can be positioned between the underlay network and the virtual servers. In this position it can act as a message dispatcher and can collect information on what virtual server accounts for what percentage of the overall network traffic.

Several load balancing algorithms [19, 17, 25] use the over-

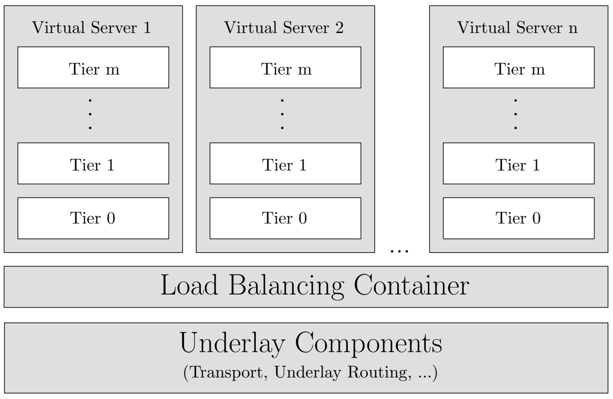

Figure 1: Load balancing container positioned to act as a message dispatcher for virtual servers.

lay network to store or probe for load information. They send special messages through the overlay network to arrange encounters between overloaded and less-loaded nodes and to coordinate the migration of virtual servers between those. To support these features the load balancing container needs access to an overlay routing tier (Tier 0) which is capable of routing a message towards an ID. Running a separate overlay routing tier inside the load balancing container would reduce the dependencies to the virtual servers the most, however, this design would have a significant drawback.

A separate overlay routing tier inside the load balancing container would be assigned a distinct ID from the overlay ID space. Depending on the concrete overlay routing protocol, the container would be put in charge of a specific range of the ID space. As a result, the P2P network would consist of two different types of peers, one running the actual P2P application, one running the load balancing algorithms. This clearly violates the basic principle of P2P network that every peer can perform the same tasks.

We favor a design that increases the dependencies to the virtual servers, but leaves the load balancing transparent to the overlay network. Thereby the load balancing container is allowed to access the overlay routing tier of one of its virtual servers. Messages from the load balancing algorithm are specially tagged by the load balancing container so that the applications inside the virtual servers can distinguish application from load balancing messages and pass the latter ones down to the load balancing container.

[17] requires to route messages through the overlay network even if no virtual server is running on the node. In this case the load balancing container could contact a bootstrap node and request it to forward the message.

3.2 The LoadBalancer interface

As the concrete load balancing algorithms that run inside the load balancing container should be exchangeable the container must provide a generic interface with well-defined semantics. We propose the `LoadBalancer` interface for this purpose. Similar to [5] we adopt a language-neutral notation to describe its methods. The data type *key* refers to an ID from the P2P network's overlay ID space, *message* is used for messages containing application data of arbitrary length. We assume that messages are attached with information on their senders, so any component that has received a message can respond to its originator. The data type *transportaddress* encapsulates a transport address, e.g. an IP address and a port. Furthermore, *T [] p* denotes an array *p* with objects of type *T*. All parameters are input parameters.

key startVirtualServer(key id, double delay) A load balancing algorithm can call this method to start a new virtual server on the physical node. The parameter *id* can be used to start the new server with a predetermined ID. This is required by a variety of load balancing algorithms [19, 17, 25]. If *id* is NULL, the ID is chosen at random from a uniform distribution. The parameter *delay* specifies the delay for the initialization of the new virtual server in seconds. This is helpful because often P2P routing protocols and applications run repair routines on a periodic basis. When multiple virtual servers are started at once, these repair routines are triggered simultaneously. According to our observations this can lead to congestions in the send queue of the physical node underneath. The return value of the method is the actual ID that was assigned to the virtual server upon startup. The value only differs from the parameter *id* when *id* has been set to NULL or the desired ID is already occupied by another virtual server on the physical node.

boolean stopVirtualServer(key id, double delay) This method is called by a load balancing algorithm to shut down a virtual server. The parameter *id* specifies the server. If the physical node runs a virtual server with such an ID it is shut down, removed from memory and the method returns TRUE. If no such ID is assigned to any of the node's virtual servers it returns FALSE. The parameter *delay* can be used to facilitate a graceful shut down. If *delay* is non-zero and the application within the virtual servers implements the `VirtualServer` interface, the method `gracefulShutdown` is invoked and the actual shutdown is delayed by *delay* seconds. If the application does not implement the `VirtualServer` interface, the shutdown is delayed anyway.

boolean routeToID(key destid, message msg, key vsid) A load balancing algorithm can use this method to route a message *msg* to a physical node responsible for the overlay ID *destid*. As the load balancing container does not participate in the overlay network directly, the parameter *vsid* can be used to specify the virtual server to forward the message towards its destination. Through this mechanism it is possible to use the overlay network to store or access load information and to contact physical nodes whose transport address is unknown. If *vsid* is NULL, the load balancing container chooses the virtual server whose ID is closest to *destid* in terms of overlay routing distance. If the load balancer has not yet started its own virtual server the method attempts to forward the message to a bootstrap node, which is then put in charge of the delivery. The method returns TRUE if *msg* could be forwarded to the specified virtual server if *vsid* is not NULL, to any server if *vsid* equals NULL, or to a bootstrap node if *vsid* is NULL and no virtual server has been started yet. Otherwise, it returns FALSE.

void routeToAddress(transportaddress dest, message msg) This method can be used to send message *msg* to the transport address *dest*. That way messages can be exchanged between different load balancing containers without taking the detour through the overlay network. E.g., an overloaded and a less-loaded node who found out about each other through the load information stored in the overlay can now call this method to communicate directly and coordinate the migration of a virtual server.

void receive(key destid, message msg) This method is invoked by the load balancing container upon reception of an incoming message *msg*. A concrete load balancing algorithm must overwrite this method to react to the message. The method is called with an additional parameter *destid*, which states the ID the message was routed to. This parameter is NULL if the message was send directly and did not traverse the overlay network.

double getLoad(key vsid) This method returns the current load of the virtual server with ID *vsid* as a floating point value. The semantics of the return value depends on whether the application within the queried virtual server implements the `VirtualServer` interface. If this is the case, the return value is an abstract application-specific figure which represents the current consumption of one (or possibly a combination) of the application's bottleneck resources. Several load balancing techniques [19, 17, 25] consider a physical node overloaded if the sum of its virtual servers' loads exceeds a specific threshold. We expect load balancing algorithms to provide a variable that can be set by the user to specify that threshold for a concrete application. If the application within the virtual servers does not implement the `VirtualServer` interface, the load balancing container assumes network bandwidth to be the bottleneck resource. In this case the method's return value represents the data rate for the virtual server's incoming messages in KBit/s as observed during message dispatching. A negative value is returned if no virtual server with the ID *vsid* is currently running.

key [] getVirtualServers() A load balancing algorithm can call this method to obtain an array containing the IDs of all virtual servers that are currently running on the physical node. The method is important to implement higher level aggregation methods.

3.3 The VirtualServer interface

While the `LoadBalancer` interface is mandatory for load balancing algorithms to communicate with the load balancing container, applications running within virtual servers can optionally implement the `VirtualServer` interface. With the interface implemented the applications can react to graceful shut downs and provide application-specific values that describe their current load situation.

double getLoad() This method is invoked when a load balancing algorithm has requested information about the virtual server's load situation. The method must be overwritten by the application developer and return a floating-point value that represents the current consumption of one (or possibly a combination) of the application's bottleneck resources. The return value must be non-negative.

void gracefulShutdown(double delay) This method is invoked when a load balancing algorithm has requested a virtual server to shut down gracefully. The application developer must overwrite this method to respond to the shutdown request. The parameter *delay* specifies the number of seconds before the virtual server is finally destroyed.

4. IMPLEMENTATION

In the following we demonstrate how the design considerations from the previous section can be transferred into a concrete implementation for the popular P2P simulation framework OverSim [1]. Based upon OMNeT++'s INET framework OverSim already puts special emphasis on a modular design with exchangeable, loosely-coupled components. Therefore it was suited best for our purposes. We designed our load balancing extension with backward compatibility in mind, i.e. existing applications that use OverSim's tier abstraction can run inside virtual servers without modifications to their implementation.

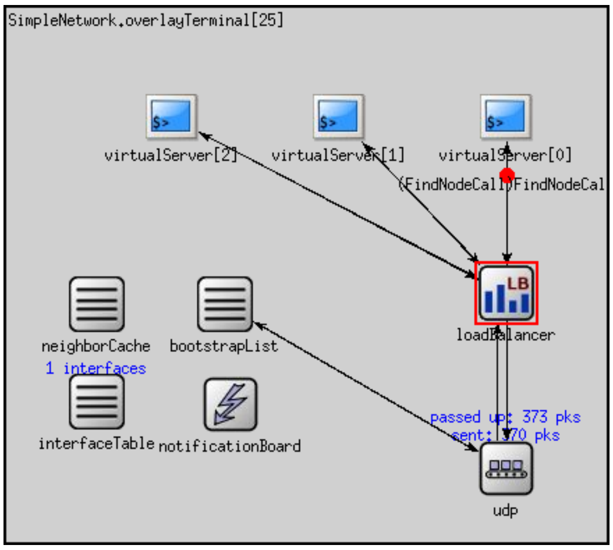

Figure 2: Components running inside the overlayTerminal module.

Figure 2 illustrates the internal structure of an OverSim overlay terminal with our load balancing extension enabled. The module `loadBalancer` represents the load balancing container that encapsulates the actual load balancing algorithm. As discussed in the previous section, it is situated between the `udp` module, the transport layer of the underlay network, and the virtual servers it controls.

A load balancing algorithm that is supposed to run inside the `loadBalancer` module must be derived from the base class `BaseLoadBalancer`. `BaseLoadBalancer` implements the `LoadBalancer` interface described in Section 3 and, hence, provides an API to start/stop virtual servers, route messages towards IDs and query for load information. The class also contains functionality to act as a message dispatcher for the virtual servers and can keep track of the traffic distribution among the servers.

Upon initialization of a new virtual server the `loadBalancer` modules checks all of its submodules for the `VirtualServer` interface. If at least one module inside the virtual server implements the interface, the return value of the `getLoad()` method is used as load information instead of the collected network traffic data. The total load of a virtual server is the sum of the load of all its submodules that implement the `VirtualServer` interface.

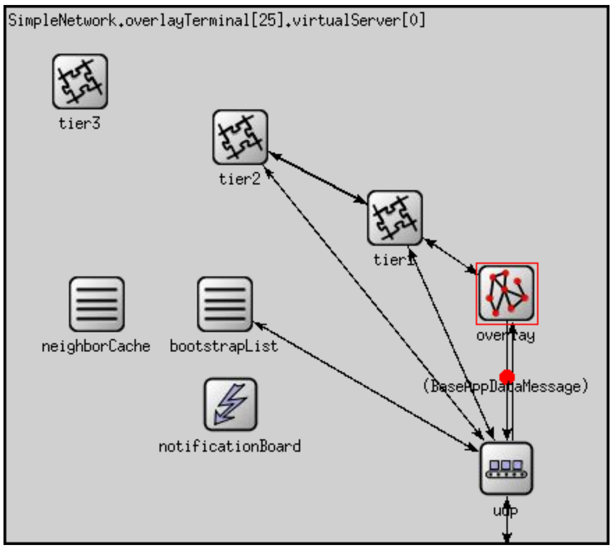

Figure 3: Components running inside the virtualServer module.

Figure 3 depicts the internal structure of a virtual server. For compatibility purposes we mostly copied the layout of a regular `overlayTerminal` module and modified the behavior of specific submodules.

The `udp` module is a replacement that simply forwards outgoing UDP messages to the load balancing container and disseminates incoming ones according to their destination port. Since overlay nodes can run multiple instances of the same application tier stack, we experienced the problem that several binds were requested on the same UDP port. We solved the problem by modifying the `bindToPort` method from the `BaseOverlay` and `BaseApp` class. When called inside a virtual server, the method now contacts the `loadBalancer` module to negotiate an unbound UDP port. The modification works without any difficulties as long as the overlay network is used for message delivery and an application does not expect to be contacted directly on a specific port. Standard OverSim applications do not make use of TCP as a transport protocol, thus we have disregarded it so far. However, the same approach seems to be applicable to TCP as well.

The behavior of the modules `bootstrapList` and `neighborCache` has been modified as well. When running inside a virtual server, these modules forward any requests from the application tiers to the respective global modules located in the `overlayTerminal` module. The modules maintain information on physical nodes for proximity aware routing and bootstrapping. As all virtual servers run on the same physical node and, thus, share the same network address, they can all benefit from gathering the information within a global module.

We left the `notificationBoard` module unchanged since it ships with OMNeT++'s INET framework and we did not want to change any code which is different from the OverSim framework. In our current implementation the load balancing module therefore subscribes to all events of the node's notification board that can be triggered by OverSim's underlay configurator and re-sends them to the notification boards in all the virtual servers.

5. EVALUATION

Finally, we want to evaluate the simulation overhead introduced by the virtual servers in terms of the memory consumption and execution speed. We conducted a series of small measurements with the OverSim framework (release 20080919) on an Intel Xeon 2.66 GHz machine (64-bit architecture) running Ubuntu Linux (Hardy Heron). As a test application the following application tier stack was used: On tier 0 we ran OverSim's `Chord` module, the `DHT` module was used on tier 1 and tier 2 was set to `DHTTestApp`.

We conducted each measurement with both virtual servers enabled and disabled and varied the number of overlay terminals in the system between the individual measurement runs. To ensure comparability the number of overlay terminals was divided by the number of virtual servers running on each terminal, so e.g. a simulation run with 500 overlay terminals and no virtual servers is compared to a simulation run with 100 overlay terminals, each running 5 virtual servers. Churn was not applied in any of our experiments. A complete listing of the parameter settings can be found in the appendix.

In order to capture the simulations' memory consumption we called the standard Unix tool `ps` right before the program finished, the duration of each measurement run was determined with the command `time`.

According to our measurements starting an empty virtual server (with no application tiers) increases the memory footprint of an overlay terminal by about 28 KBytes. However, since all virtual servers within an overlay terminal share the same modules for accessing the underlay network (e.g. UDP module, transport layer and PPP interface) the overhead can be compensated.

Figure 4 shows the memory consumption of the simulation with OverSim's *INET* model depending on the number of peers in the overlay network. The detailed simulation of the lower network tiers with this model leads to an increase of memory consumption of about 35 KBytes per node compared to the more abstract *Simple* model. Consequently, running multiple virtual servers on the same node results in substantial memory savings when the number of nodes in the system is increased. For the experiment the underlay network was configured with 10 backbone and 50 access routers.

With respect to execution speed the impact of the virtual servers is negligible. Neither simulations with the *Simple* nor with the *INET* model showed a significant change in performance with or without virtual servers.

6. CONCLUSIONS

State-of-the-Art P2P simulation frameworks feature modular building blocks to accomplish rapid research results. Unfortunately, currently no simulation framework provides support for load balancing although it is vital to many types of applications. This paper introduced how load balancing algorithms that rely on the concept of virtual servers can run in a modular container with little or even no dependen-

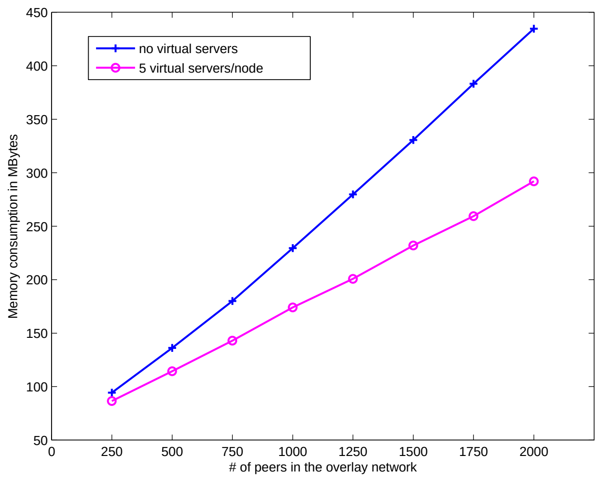

Figure 4: Memory consumption of test simulation with *INET* model.

cies to the application. We discussed design decisions for such a container and proposed two interfaces to unify the communication between the load balancing algorithms and the virtual servers. As a proof of concept, we presented an implementation of our approach for the OverSim P2P simulation framework and demonstrated how virtual servers can improve the scalability of simulations.

Currently, our implementation is in an early development phase. When it has reached a more mature state, we intend to contribute it to the OverSim project. In general, we hope our work encourages other developers to foster modularization and standardization in simulation environments. Both aspects are crucial for building more complex applications in the future and facilitating the exchange of components between different simulation tools.

7. REFERENCES

[1] I. Baumgart, B. Heep, and S. Krause. OverSim: A Flexible Overlay Network Simulation Framework. In *Proceedings of 10th IEEE Global Internet Symposium (GI '07) in conjunction with IEEE INFOCOM'07*, pages 79–84, 2007.

[2] S. Bianchi, S. Serbu, P. Felber, and P. Kropf. Adaptive Load Balancing for DHT Lookups. In *Proceedings of the 15th International Conference on Computer Communications and Networks (ICCCN'06)*, pages 411–418, 2006.

[3] C. Blake and R. Rodrigues. High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems (HOTOS'03)*, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[4] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area Cooperative Storage with CFS. *SIGOPS Operating Systems Review*, 35(5):202–215, 2001.

[5] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Berkeley, CA, February 2003.

[6] V. Darlagiannis, O. Heckmann, and R. Steinmetz. Peer-to-Peer Applications Beyond File Sharing: Overlay Network Requirements and Solutions. *Elektrotechnik und Informationstechnik*, 123(6):242–250, 2006.

[7] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 444–455. VLDB Endowment, 2004.

[8] T. Giuli and M. Baker. Narses: A Scalable, Flow-based Network Simulator. Technical Report arXiv:cs.PF/0211024, Computer Science Department, Stanford University, Stanford, CA, 2002.

[9] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load Balancing in Dynamic Structured P2P Systems. In *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'04)*, volume 4, pages 2253–2262, 2004.

[10] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher. Adaptive Replication in Peer-to-Peer Systems. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 360–369, 2004.

[11] M. Jelasity, A. Montresor, G. P. Jesi, and S. Voulgaris. The Peersim simulator. `http://peersim.sf.net`.

[12] S. Joseph. NeuroGrid: Semantically Routing Queries in Peer-to-Peer Networks. In *Proceedings of the International Workshop on Peer-to-Peer Computing*, volume 2376 of *LNCS*, pages 202–214. Springer, 2002.

[13] S. Joseph. An Extendible Open Source P2P Simulator. *P2P Journal*, 1:1–15, 2003.

[14] K. Kant and R. Iyer. Modeling and Simulation of Ad-Hoc/P2P File-sharing Networks. In *Proceedings of the 13th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, 2003.

[15] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC'97)*, pages 654–663, 1997.

[16] D. R. Karger and M. Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. In *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'04)*, pages 36–43, 2004.

[17] J. Ledlie and M. Seltzer. Distributed, Secure Load Balancing with Skew, Heterogeneity and Churn. *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'05)*, 2:1419–1430, March 2005.

[18] J. Li, J. Stribling, R. Morris, F. M. Kaashoek, and T. M. Gil. A Performance vs. Cost Framework for Evaluating DHT Design Tradeoffs Under Churn. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'05)*, volume 1, pages 225–236, 2005.

[19] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load Balancing in Structured P2P Systems. In *Proceedings of the 2nd International Workshop on Peer-To-Peer Systems (IPTPS'03)*, 2003.

[20] M. T. Schlosser, T. E. Condie, and A. D. Kamvar. Simulating a File-Sharing P2P Network. In *Proceedings of the 1st Workshop on Semantics in P2P and Grid Computing*, 2003.

[21] K. Shudo, Y. Tanaka, and S. Sekiguchi. Overlay Weaver: An Overlay Construction Toolkit. *Journal of Computer Communications (Special Issue on Foundations of Peer-to-Peer Computing)*, 31(2):402–412, Feburary 2008.

[22] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'01)*, pages 149–160, 2001.

[23] N. S. Ting and R. Deters. 3LS - A Peer-to-Peer Network Simulator. In *Proceedings of the 3rd International Conference on Peer-to-Peer Computing (P2P'03)*, pages 212–213, 2003.

[24] W. Yang and N. Abu-Ghazaleh. GPS: A General Peer-to-Peer Simulator and Its Use for Modeling BitTorrent. In *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 425–432, 2005.

[25] Y. M. Zhu and Y. Hu. Efficient, Proximity-aware Load Balancing for DHT-based P2P Systems. *IEEE Transactions on Parallel and Distributed Systems*, 16(4):349–361, April 2005.

8. APPENDIX

The simulation used to obtain the experimental results in Section 5 was conducted with the following parameter settings:

```
num-rngs = 1
rng-class = "cMersenneTwister"
*.underlayConfigurator.churnGeneratorTypes = "RandomChurn"
*.churnGenerator*.creationProbability=0.0
*.churnGenerator*.migrationProbability=0.0
*.churnGenerator*.removalProbability=0.0
*.churnGenerator*.churnChangeInterval=0
IPv4Network.backboneRouterNum=10
IPv4Network.overlayBackboneRouterNum=0
IPv4Network.accessRouterNum=50
**.measurementTime = 18000
**.transitionTime = 1000
**.overlayType = "ChordModules"
**.numTiers = 2
**.tier1Type = "DHTModules"
**.tier2Type = "DHTTestAppModules"
**.globalObserver.globalFunctionsType = "GlobalDhtTestMap"
**.globalObserver.useGlobalFunctions = 1
**.initPhaseCreationInterval = 0.1
**.debugOutput = false
**.drawOverlayTopology = false
**.tier1.dht.numReplica = 4
```

The parameter `seed-0-mt` was initialized with a random number before each measurement run. Depending on the experiment, the network type (parameter `network`) was either set to `IPv4Network` or `SimpleNetwork`. When we ran an experiment with n peers (i.e. instances participating in the overlay network), we set `**.targetOverlayTerminalNum` to n for regular overlay terminals and to n/v for our modified overlay terminals, each running v virtual servers.

All simulation parameters not covered in this section comply with the standard values as defined in the file *default.ini* of OverSim release 20080919.