Towards a Common Interface for Overlay Network Simulators

Christian Groß[1], Max Lehn[1], Dominik Stingl[1], Aleksandra Kovacevic[1],
Alejandro Buchmann, and Ralf Steinmetz
Multimedia Communications Lab, Databases and Distributed Systems,
Technische Universität Darmstadt, Germany
Email: {gross, stingl, sandra, steinmetz}@kom.tu-darmstadt.de,
{max_lehn, buchmann}@dvs.tu-darmstadt.de

***Abstract*—Simulation has become an important evaluation method in the area of Peer-to-Peer (P2P) research due to the scalability limitations of evaluation testbeds such as PlanetLab or G-Lab. Current simulators provide various abstraction levels for different underlay models, such that applications can be evaluated at different granularity. However, existing simulators suffer from a lack of interoperability and portability making the comparison of research results extremely difficult. To overcome this problem, we present an approach for a generic application interface for discrete-event P2P overlay network simulators. It enables porting of the *same* implementation of a targeted application once and then running it on various simulators as well as in a real network environment, thereby enabling a diverse and extensive evaluation. We established the feasibility of our approach and showed negligible memory and runtime overhead.**

***Index Terms*—Simulation, Peer-to-Peer, Interface, Common API, Overlay, Testbed, Simulator-Design**

I. Introduction

Since research in Peer-to-Peer (P2P) systems became popular in the late nineties, the evaluation of such large scale and complex systems has been a challenging task. Test-bed platforms, such as PlanetLab or G-Lab, where prototypical implementations can be deployed, offer a realistic evaluation environment that is limited to a few hundred peers. Scalability of P2P systems, with several thousands and even millions of participants, can be effectively assessed only by simulation. However, more detailed simulation models (e.g., considering the underlay network) are not amenable to large-scale simulations. This is the reason why accepted network simulators such as ns-3 are seldom used in P2P research. To address this tradeoff between scalability and realism, most researchers tend to develop their own simulators that are geared to focus on specific aspects, as shown by Naiken et al. [15], [16]. This has resulted in a plethora of simulators. As simulation models of P2P systems can vary greatly in existing simulators, comparison of evaluation results is difficult if not impossible. Taking this comparability into account, having only one single simulator comprising the different models of all simulators would be an ideal solution.

In spite of their obvious shortcomings, the plethora of existing simulators has the advantage of providing various abstraction levels for different underlay models such that applications can be evaluated at different levels of granularity. However, due to incompatible simulator APIs, it is hard to exploit these simulators effectively. We therefore propose a generic application interface for P2P simulators. This allows to reuse underlay models from existing P2P simulators by porting the *same* implementation of a target application to the different simulators as shown in Figure 1, and avoids the effort of re-implementing the application on top of various simulators. Furthermore, due to the abstract nature of the interface that does not assume a specific execution environment, applications built on top of our interface can be directly tested in real network environments. Throughout this paper we use the term 'application' for the overlay (such as Chord [21], CAN [19], or Kademlia [12]), as well as for the actual application generating the workload for the particular scenario being simulated and evaluated.

To motivate the need for our generic application interface, we state some general properties regarding the utilization of the interface on top of the different simulators highlighting its benefits:

- Firstly, existing P2P overlays such as Chord [21], CAN [19], or Kademlia [12] can be implemented once based on the interface definition of our generic application interface. In doing so, these P2P overlay implementations can be re-used within all simulators implementing our generic application interface. So without the need of re-implementation new P2P

[1]Authors supported by the German Research Foundation, Research Group 733, "QuaP2P: Improvement of the Quality of Peer-to-Peer Systems".

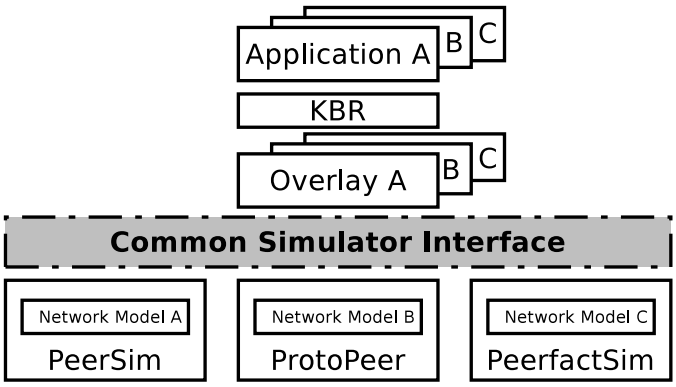

Figure 1. Schematic of the common simulator interface.

overlays, the number of available overlays for a given simulator increases, offering the possibility for researchers to evaluate their P2P mechanisms on a greater variety of P2P overlays on a single simulator, while accuracy remains unchanged due to the use of the same underlying simulator core and network model.

- Secondly, researchers may stick to their favorite simulator allowing them to work and simulate in a well-known environment and to re-use tailored tools such as data capturing and evaluation of simulated scenarios.
- Thirdly, due to the fact that each simulator offers a different set of underlay models (e.g., simple network model with static delay, GNP network model [17], wireless network model [3]) P2P mechanisms can be evaluated using the provided underlay models from the different simulators.

We implemented the interface for three current P2P simulators (PeerSim [13], ProtoPeer [6], PeerfactSim.KOM [10]) and built a prototypical P2P application performing random walks, similar to a simplified GIA [4] implementation. We thereby proved the feasibility of our approach and showed negligible memory and runtime overhead.

In Section II the background of P2P simulators is discussed and a brief overview of the architecture of existing discrete-event overlay network simulators is given. Section III deals with the requirements for a common simulator interface, followed by a discussion of design decisions in Section IV. A concrete interface proposal is introduced in Section V, and a feasibility and evaluation study of the interface is given in Section VI. Finally, we present conclusions in Section VII and present an outlook on the future work.

II. Background

P2P simulators which aim at the simulation of P2P systems in a realistic and scalable manner have been widely researched. Most simulators, such as PeerfactSim.KOM [10], PeerSim [13], ProtoPeer [6], PlanetSim [18], and OverSim [1] have a layered architecture along the lines of the ISO-OSI model. Conceptually, most P2P simulators distinguish between a network layer, an overlay layer and an application layer. The network layer is responsible for modelling delays and message loss between different peers/nodes using different approaches, such as euclidean embedding [11], [9] following the approach of Ng et al. [17], the King dataset [8], or shortest path algorithms based on generated Internet topologies [23], [24]. The overlay layer contains the corresponding overlay implementations like Chord, CAN, and Kademlia. P2P applications are built on top of these overlays. All of the three aforementioned layers use the discrete-event simulation engine for performing asynchronous wait operations.

One of these simulators following the layered architecture approach is PeerfactSim.KOM which has been introduced in [10]. The simulator is written in Java and divided into dedicated layers which are triggered by an event-based simulation engine. Beside the three identified layers described above, PeerfactSim.KOM extends this general concept with further components. These additional elements allow for modelling the user behaviour and interaction with the P2P system, while the segmentation of the underlay into a network and a transport layer facilitates the simulation of scenarios with multiple applications and different types of overlays at the same time. Furthermore, it offers an integration of different and customized churn models as well as a monitoring architecture simplifying the data collection during a simulation.

PeerSim [13] which is developed by Alberto Montresor et al., follows a slightly different architectural design. In contrast to the other simulators, PeerSim offers two different modes of operation, a cycle-based and the event-based approach. The cycle-based mode allows for large scale simulations using simplifying assumptions regarding the message transport, whereas the event-driven mode aims at simulations requiring higher granularity and more realistic results. Overlay networks in PeerSim are modelled as a set of nodes each having a set of protocols. A protocol implements application-specific behaviour, thus being the extension point for additional functionality.

ProtoPeer has been developed with the idea in mind of switching between event-driven simulations and live network deployment by exchanging the underlying network component, such that no changes to the application code should be necessary. This is achieved by sticking to a modular architecture design which defines clear interfaces between different layers. In particular, the abstract time and networking API is one of the key features of the ProtoPeer architecture ensuring that application code built on top does not need to be changed. Applications in ProtoPeer are created by using so called *Peerlets*, which define modularized, reusable and unit-testable peer functionality. The simulator offers the opportunity for running applications in a discrete event-driven mode or as a prototype on a real network.

Beside the simulators introduced above, there exist several other overlay simulators which are either not written in Java or have not been updated recently. One of the latter is PlanetSim [18]. PlanetSim is also structured into an application, overlay, and network layer. Functionality in PlanetSim is structured in two different ways. The first one defines node functionality as a protocol which determines the actions a node has to perform, e.g., in case of incoming and outgoing messages or periodic tasks such as overlay maintenance. The second one defines a *behaviour* model that allows for the implementation of differentiated aspects of node functionality in separate classes that can be composed dynamically.

A prominent C++ overlay network simulator is OverSim [1] which is based on the general-purpose discrete-event simulation engine OMNeT++ [22]. Since the simulation engine is provided by a separate project, there is a clear apportionment of discrete-event simulation and network modelling. OverSim provides a set of different network models with varying granularity. There are implementations for important overlay networks, such as Chord, Kademlia, and GIA.

Each of the simulators was developed for specific needs assuming that existing simulators do not satisfy the particular requirements. All simulators define an API for writing simulator-specific extensions that are not compatible among simulators. A comparison of the different simulators, however, shows that most simulators are conceptually similar and provide equivalent functionality such that a generic application interface for these can be derived. Each of the simulators is shipped with a set of underlay models and overlays. But the set of overlays usually consists of only a small subset of the relevant systems. For example, PeerSim offers support for the Chord, Kademlia and Pastry overlay, but lacks from overlays such as CAN, GNutella, or Gia. In applying our interface to the simulator and to the overlays, researchers using PeerSim benefit from it because a much bigger variety of overlays is available. The naive approach of increasing the availability of overlays for an given simulator by adapting existing overlays causes much more overhead as a typical overlay implementation consists of several hundreds of lines of source codes and are based on simulator specific concepts.

There is no other existing approach for defining a generic interface for discrete event-based P2P simulators that be ported to the relevant set of simulators with reasonable effort.

The FreePastry [20] library which has been developed by the Rice University defines a set of basic entities (e.g., Application, EndPoint, Message, RouteMessage, Id, and NodeHandle) called Common API (CAPI) an application should use. This API definition has proven to be applicable for a variety of prototypes using the FreePastry library ([7], [14]). Although the API has been desined to support prototypical and simulated execution, it makes FreePastry-specific assumptions (e.g., neighbor sets associated to endpoints and ID ranges). Furthermore, it does not support the full set of required functionality to run event-based simulation as identified in Section III.

Dabek et al. [5] introduced an interface definition for structured overlays (*key-based routing*, KBR) which aims for an easy interchangeability of overlay implementations. Although not developed particularly for simulators, KBR is frequently applied there. Behnel et al. [2] proposed an approach for rapid overlay implementation including a modelling framework for overlay networks. Both approaches focus on the overlay layer and its interfaces, whereas the goal of our interface is to make simulators exchangeable without the need for modifying the application or overlay code.

III. Requirements

Applications written for a specific simulator need to have access to certain functionality of the simulator. Based on our experience, the following set of basic mechanisms represents the functional requirements which need to be covered by the simulator interface in order to satisfy the typical P2P simulation needs.

- Since distributed systems consist of a collection of *nodes*, methods for starting and shutting down these nodes are required.
- To enable communication between nodes, methods for *sending and receiving messages* are necessary.

- For the evaluation of topology-aware overlays, it is desirable to have access to *network topology information*.
- An application must be able to perform temporal operations (e.g., waiting, periodic activities). Therefore, it must have direct access to the scheduler for *scheduling events*.
- A typical problem related to the start-up process of a P2P node is how to find *bootstrap nodes* to connect to. Thus the common simulator interface has to provide facilities for obtaining those in the start-up phase.
- Nodes joining the network may either start as a completely new node or as a *re-joining* node. Re-joining nodes need to read their previous session's state, requiring some means for persistence.
- An essential functionality of simulators is a *random number generator* that produces seeded pseudo random numbers, ensuring a deterministic behaviour such that experiments are reproducible.
- It might be necessary to pass certain node-specific *parameters* to the application, for instance, to let the application decide to behave differently based on its connection properties.
- *Logging* should be covered by the simulator interface for gathering information about the simulated activities. In particular, per-node logging is desirable to identify a single node's actions.
- *Statistical and analytical* functionality should be provided for obtaining detailed simulation results.

Besides the listed functional requirements, the translation layer building the generic interface on top of a particular simulator should be lightweight with respect to both number of lines of code and runtime overhead. Therefore, the interfaces should be designed with simplicity in mind and only provide a necessary minimum of functionality.

The purpose of the common simulator interface is to ensure the portability and re-usability of applications built on top of simulators. The configuration of the simulator (e.g., the layers and their parameters or the utilized network model) and of the scenario (number of nodes, churn model, bandwidth, application workload, etc.) is explicitly not part of this interface since this configuration heavily depends on the layers and components of the particular simulator's world model and thus may not be easily generalized. Taking the feasibility into account, the reason for neglecting this functionality is that it is almost impossible to define a generic configuration interface for all kinds of components that the different simulators offer. Regarding the formal requirements of the generic simulator interface, this interface is geared to ensure the portability and the re-usability of applications built on top of simulators and not to provide a uniform usability of simulators. Thus, when porting an application from one simulator to another, the scenario definition for the simulation still needs to be adapted. We see this as an orthogonal topic which may be covered separately. Therefore, the developed application on top of our simulator interface should specify its own configuration in order to be independent from the simulator specific configuration.

IV. Design Decisions

This section introduces the basic components of the interface based on the identified requirements.

A. Transport Protocol

One of the fundamental questions that must be answered is, on which transport protocol (UDP, TCP, or more abstracted messaging or streaming) the simulated communication between nodes should be based. In order to keep the communication API as simple as possible, we have chosen a UDP-like messaging model as transport protocol since it is a stateless communication protocol. Conversely to UDP, TCP is a stateful protocol which needs a more complex communication API. In addition to that, most current P2P simulators offer a transport layer API based on UDP or a more abstract messaging model, as it is sufficient for many P2P applications.

B. Addressing Scheme for Nodes

Another basic issue to be discussed is how the addressing scheme using by the simulator should look like. Most simulators use an abstract addressing scheme based on integer values to identify peers. This addressing scheme is very simple but cannot be applied within scenarios were an applications should be executed in a native execution environment with a real network layer. To enable native execution of prototypes, it is necessary to have a real address scheme based on IPv4 or IPv6 as it is required by the network device. Our common simulator interface introduces an *Address* interface which encapsulates the actual addressing scheme such that both mentioned addressing schemes can be used.

C. Node Concept for Native Execution of Prototypes

The simulator interface should provide the opportunity to easily switch applications developed for the simulator to a live network deployment without the need for

changing the application or overlay network code. This feature allows for evaluating applications even further than on different network models of various simulators. Finally, it enables a deployment of complete applications for productive use. It can be provided by implementing a scheduler running in real time and a messaging wrapper that sends and receives massages over native UDP. The main aspect enabling this live deployment is that nodes, implementing the overlay and application functionality, are started by the simulator or the translation layer. On start-up, each node gets the necessary interface objects which provide only the functionality that is available in both simulation and live network deployment. In particular, the applications built on top of our interface cannot depend on global knowledge which is only available during simulations.

D. Time Units

Another essential design decision is how time should be represented within simulated applications. Many simulators define their own classes for handling time units and offer methods for retrieving the current simulated time in different resolutions. Others directly use a long integer data type, typically with microsecond or millisecond granularity. We define an interface representing time which can be implemented depending on the needs of the underlying simulators.

E. Handling of Obsolete Events

In case of failing or leaving nodes, it happens that scheduled events become obsolete, leading to the question of how to deal with such events. There are basically two solutions for handling outdated events. The first is to delete obsolete events of failed or leaving nodes, causing additional overhead as the event queue needs to be searched for those. The second approach is to handle those events by the application itself, which means that the application has to decide whether an event fired by the simulation engine is obsolete and thus discarded or not. Depending on the approach, the interface for encapsulating the simulation engine has to provide methods for deleting events. For the sake of simplicity we have chosen the second approach and allow the application to handle obsolete events.

V. A Common Simulator Interface

Based on the requirements gathered in Section III as well as on the available design decisions identified in Section IV, we developed the following interface design covering the basic simulator functionality:

- The **`Node`** interface represents the basic entity that encapsulates the application logic. The **`NodeFactory`** interface is implemented by the application and handed over to the simulator for the creation of nodes. **`NodeInterface`** is implemented by the simulator and encapsulates the per-node API.
- The **`Scheduler`** interface represents the simulation engine and offers methods for scheduling events and requesting the current simulation time. **`Time`** encapsulates the simulator-specific time unit. **`EventHandler`** is implemented by components that contain the functionality for processing events.
- The interface of the network layer used for sending messages is **`NetworkInterface`**. The **`Address`** interface transparently encapsulates simulator-specific addresses (IPv4/v6 or just plain numbers, depending on the simulator). The basic message type used for the communication between nodes is defined by the **`Message`** interface. **`NetworkListener`** has to be implemented by an application component for receiving messages.
- **`Random`** defines methods for retrieving pseudo random numbers for reproducible simulations.
- **`Bootstrap`** allows nodes to register as a bootstrap node and retrieve adresses of online nodes available for bootstrapping.

Figure 2 shows how the above-mentioned interfaces are integrated in a typical simulator architecture. The grey shaded elements represent our common simulator interface whereas the white parts depict the most important parts of a simulator architecture. As already mentioned in Section II, most of the simulators follow a layered architecture design which mainly consists of a network layer and an overlay/application layer. To make an application or overlay portable and independent from the concepts and methods of a specific simulator, it is necessary to wrap all these concepts and methods using an interfaces. These interfaces then need to be implemented once for each simulator forming a translation layer which maps application actions onto simulator specific methods.

Among the listed interfaces, there are three core interfaces `Node`, `Scheduler`, and `NetworkInterface` which we describe in more detail in the following subsection.

A. Node Interface

The `Node` interface, as shown in Listing 1, has to be implemented by the application, and represents the basic entity in simulated P2P systems. Therefore, the

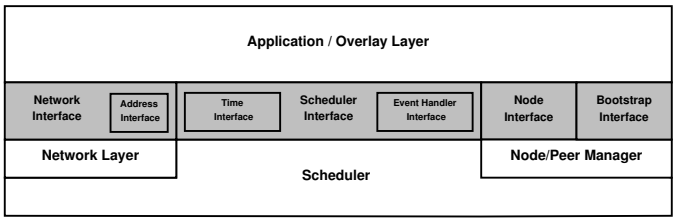

Figure 2. Integration of the common simulator interface into the simulator architecture

class implementing the `Node` interface should contain the functionality of the simulated application. The interface consists of two methods for setting up and shutting down the node, which both are invoked by the simulator. The method for starting a node passes the `NodeInterface` object which encapsulates the whole per-node API during the node's lifetime. The `shutdown()` method indicates a (graceful) shutdown or crash of the node, depending on the `crash` argument.

```
void startup(NodeInterface nodeInterface);
void shutdown(boolean crash);
```

Listing 1. Node

`NodeInterface` provides getters for the interface objects for the various categories of functionality as described above (see Section III). The current implementation contains four of them (scheduler, network interface, random source, and bootstrap address provider) as shown in Listing 2.

```
Scheduler getScheduler();
NetworkInterface getNetworkInterface();
Random getRandom();
Bootstrap getBootstrap();
```

Listing 2. NodeInterface

B. Scheduler Interface

As mentioned above, the `Scheduler` interface encapsulates the discrete-event simulation engine. It offers methods for scheduling events absolute or relative to the current simulation time. The two respective methods, as shown in Listing 3, each expect two arguments. The first one specifies the absolute or relative time at which the event should be fired. The second parameter is the event handler to be associated with the event. In addition to that, the interface contains methods for retrieving the current simulation time and for creating `Time` instances based on different time units.

```
Time getCurrentTime();
Time timeInMicroseconds(long time);
Time timeInMilliseconds(long time);
Time timeInSeconds(long time);
void scheduleIn(Time t, EventHandler h);
void scheduleAt(Time t, EventHandler h);
```

Listing 3. Scheduler

C. Network Interface

The network layer is represented by the interface `NetworkInterface` as shown in Listing 4. This interface is responsible for handling downcalls to the network and therefore provides a method for sending UDP-like messages as discussed in Section IV-A. As its first argument the method expects the message object, containing the application-defined message payload. The second argument is the receiver address of the message. Furthermore, the interface provides a method for obtaining the local address.

```
void addListener(NetworkListener l);
void removeListener(NetworkListener l);
void sendMessage(Message msg, Address to);
Address getLocalAddress();
```

Listing 4. Network Interface

For handling incoming messages, we use the Listener pattern. Components that want to receive messages have to implement the `NetworkListener` interface and register at the provided `NetworkInterface` object.

VI. Evaluation

In order to evaluate our approach, we implemented a translation layer for our generic simulator interface on top of each of three widely used discrete-event based simulators (ProtoPeer, PeerSim and PeerfactSim.KOM). The focus of the evaluation lies on the applicability of our approach and on the overhead caused with respect to time and memory consumption. Although we implemented the interface in Java, it is possible to port it to every other object-oriented programming language like C++ or C#. In order to give an impression on the implementation effort necessary for implementing our interface on top of each simulator, we have counted the physical source lines of code (SLOC) for all three interface implementations.

The results (Table I) clearly indicate that the implementation overhead for the interface to work within each simulator is low, especially when comparing it to the total number of lines of code of the simulators. The effort necessary for porting a single overlay (as an example,

Simulator	SLOC Simul.	SLOC Interf.
PeerSim	~7,300	237
ProtoPeer	~7,700	225
PeerfactSim.KOM	~88,000	225

Table I
PHYSICAL SOURCE LINES OF CODE (SLOC) FOR EACH IMPLEMENTATION OF THE GENERIC SIMULATOR INTERFACE.

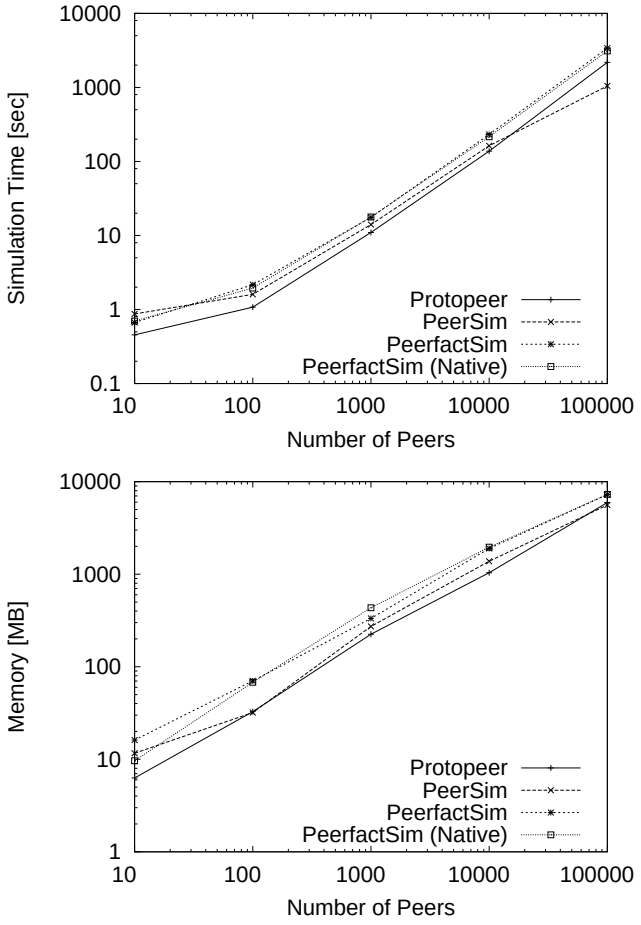

Figure 3. Simulation duration and memory consumption.

the Pastry implementation for PeerSim has 1100 SLOC) may already be higher as it builds on simulator-specific concepts.

The prototypical application that we use for the evaluation of usability of our interface and runtime performance is a simple P2P system, in which each node periodically performs a random walk. Whenever a node receives a (random walk) message from some other node, it records the initiating node in its routing table.

We measured the runtime and memory consumption for the three simulators using the translation layers with 10, 100, 1,000, 10,000, and 100,000 nodes (Figure 3). We ran each simulation five times and calculated the mean values for each simulator. We want to emphasize that these measurements are not meant as a serious simulator competition, since the performance of each of the simulators heavily depends on the particular network model. It rather gives an impression on the comparability between simulators in general.

As a reference, we implemented our prototypical P2P application natively on PeerfactSim.KOM (i.e., using the PeerfactSim.KOM API) whose measurement results are also included in the figure. With 100.000 nodes, the runtime overhead using our translation layer compared to the native implementation on PeerfactSim.KOM has a maximum of 18%, which appears justifiable. The memory overhead of less than 2% is negligible. All simulators show the same scalability behaviour with respect to runtime and memory consumption during the simulations.

VII. CONCLUSION AND FUTURE WORK

In this paper we identified the requirements for a general purpose application interface for discrete-event overlay simulation. Based on these requirements we proposed an approach towards a common interface for discrete-event overlay network simulators.

Using our interface, a lot of future implementation overhead can be avoided, as overlays and P2P applications that are implemented once can be reused with other simulators. Moreover, we also intend to improve the structure of future simulators by defining the generic simulator interface. As a result of this reuse of code, research results from existing or newly created simulators are more comparable. Evaluation shows that the overhead for using our interface on top of existing simulators is justifiable and that the integration of our interface is feasible with a reasonable implementation effort. From our point of view, agreeing on a common API for P2P overlay simulators enables a community-driven development process for each overlay. In doing so, the overlay implementations can be standardized and research results based on these overlays can be compared more easily.

For future work, we plan to implement our interface using C++ to prove the portability to other object oriented languages. Furthermore, we want to conduct a detailed performance analysis of the translation layer for each simulator that implements our common interface. Based on this analysis, we intend to gain insight on a possible impact of the translation layer on the

performance of the simulator as well as to compare the performance between discrete-event overlay network simulators. Finally, we plan to extend the interface to support logging and statistics functionality as well as to add functionality to the network interface for quering topology information of the network which forms the basis for locality-aware overlays.

VIII. Availability

The source code of our implementation is available at:

`http://www.kom.tu-darmstadt.de/~chrgross/CSI/CSI.zip`

References

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

[2] Behnel, S., Buchmann, A., Grace, P., Porter, B., and Coulson, G. A Specification-to-Deployment Architecture for Overlay Networks. In *Proceedings of the Int. Symposium on Distributed Objects and Applications* (2006).

[3] Broch, J., Maltz, D., Johnson, D., Hu, Y., and Jetcheva, J. A Performance Comparison of Multi-hop Wireless ad hoc Network Routing Protocols. In *Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking* (1998), ACM, p. 97.

[4] Chawathe, Y., Ratnasamy, S., Breslau, L., Lanham, N., and Shenker, S. Making Gnutella-Like P2P Systems Scalable. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (2003).

[5] Dabek, F., Zhao, B., Druschel, P., Kubiatowicz, J., and Stoica, I. Towards a Common API for Structured Peer-to-Peer Overlays. *Springer LNCS 2735* (2003), 33–44.

[6] Galuba, W., Aberer, K., Despotovic, Z., and Kellerer, W. ProtoPeer: Bridging the Gap Between Simulation and Live Deployment. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques* (2009).

[7] Graffi, K., Podrajanski, S., Mukherjee, P., Kovacevic, A., and Steinmetz, R. A distributed platform for multimedia communities. In *IEEE International Symposium on Multimedia (ISM '08)* (Berkley, USA, Dec 2008), IEEE, IEEE Computer Society Press, p. 6.

[8] Gummadi, K. P., Saroiu, S., and Gribble, S. D. King: Estimating latency between arbitrary internet end hosts. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment* (2002), ACM, pp. 5–18.

[9] Kaune, S., Pussep, K., Leng, C., Kovacevic, A., Tyson, G., and Steinmetz, R. Modelling the Internet Delay Space Based on Geographical Locations. In *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing* (2009).

[10] Kovacevic, A., Kaune, S., Liebau, N., Steinmetz, R., and Mukherjee, P. Benchmarking Platform for Peer-to-Peer Systems. *it - Information Technology 49*, 5 (2007), 312–319.

[11] Kunzmann, G., Nagel, R., Hossfeld, T., Binzenhöfer, A., and Eger, K. Efficient Simulation of Large-Scale P2P Networks: Modeling Network Transmission Times. In *Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing* (2007), IEEE Computer Society, pp. 475–481.

[12] Maymounkov, P., and Mazieres, D. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. *Springer LNCS 2429* (2002), 53–65.

[13] Montresor, A., and Jelasity, M. PeerSim: A Scalable P2P Simulator. In *Proceedings of the 9th P2P Conference* (2009).

[14] Mukherjee, P., Leng, C., and Schürr, A. Piki - A Peer-to-Peer based Wiki Engine. In *Proceedings of the P2P 2008* (September 2008), IEEE Computer Society Press, pp. 185–186.

[15] Naicken, S., Basu, A., Livingston, B., and Rodhetbhai, S. A survey of peer-to-peer network simulators. Citeseer.

[16] Naicken, S., Livingston, B., Basu, a., Rodhetbhai, S., Wakeman, I., and Chalmers, D. The State of Peer-to-Peer Simulators and Simulations. *ACM SIGCOMM Computer Communication Review 37*, 2 (March 2007), 95.

[17] Ng, T., and Zhang, H. Towards Global Network Positioning. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement* (2001).

[18] Pujol Ahullo, J., and Garcia Lopezs, P. PlanetSim: An Extensible Framework for Overlay Network and Services Simulations. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops* (2008).

[19] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Schenker, S. A Scalable Content-Addressable Network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (2001).

[20] Rowstron, A., and Druschel, P. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)* (2001).

[21] Stoica, I., Morris, R., Karger, D., Kaashoek, F., Chord, H. B., and A. Chord: Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (2001).

[22] Varga, A., and Others. The OMNeT++ Discrete Event Simulation System. In *Proceedings of the European Simulation Multiconference* (2001).

[23] Winick, J., and Jamin, S. Inet-3.0: Internet Topology Generator. Tech. rep., University of Michigan, 2002.

[24] Zegura, E., Calvert, K., Bhattacharjee, S., and Others. How to Model an Internetwork. In *In Proceedings of IEEE INFOCOM* (1996).