Overlay Networks – Implementation by Specification

Stefan Behnel, Alejandro Buchmann

Databases and Distributed Systems Group,
Darmstadt University of Technology (TUD), Germany
{behnel,buchmann}@dvs1.informatik.tu-darmstadt.de

Abstract. Implementing overlay software is non-trivial. Current projects build overlays or intermediate frameworks on top of low-level networking abstractions. This leaves implementing the topologies, their maintenance and optimisation strategies, and the routing to the developer.
We take a novel approach to overlay implementation by modelling topologies as a distributed database. This approach, named "Node Views", abstracts from low-level issues like I/O and message handling. Instead, it moves *ranking nodes* and *selecting neighbours* into the heart of the overlay software development process. It decouples maintenance components in overlay software and allows implementing them in a generic, configurable way for pluggable integration in frameworks.

1 Introduction

Recent years have seen a large body of research in decentralised, self-maintaining overlay networks like P-Grid [1], Chord [2], ODRI [3] or Gia [4]. They are commonly regarded as building blocks for Internet-scale distributed applications.

Contrary to this expectation, current overlay implementations are built with incompatible, language specific frameworks on top of low level networking abstractions. This complicates their design and hinders the comparison and integration of different topologies. Apart from a recently proposed API for the specific case of structured overlay networks [5], there is little standardisation effort in the rest of the overlay area. And a common API does by no means simplify the design of the overlay implementation itself.

Currently, programmers who want to use overlays for their applications must decide in advance, at a very early design phase, which of the distinct overlay implementations they want to use and must invest time to understand its specific usage. This effectively prohibits testing the final product with different topologies or delivering versions with specialised overlays. Therefore, the actual usefulness of overlays for application design is currently very limited.

This paper explores the design space of overlay design frameworks and the abstractions they provide. It proposes an integrative high-level approach at a data management level rather than the networking and messaging level. Similar to the way standard DBMS's have decoupled and modularised today's server applications, the presented approach allows for a separation of concerns in overlay software and for pluggable, decoupled components in overlay design frameworks.

Section 2 investigates the major functionality blocks of overlay software and matches them with the current framework support. Section 3 then presents the Node Views abstraction that facilitates a higher level design of overlay topologies and decoupled components. The SQL-like language that we designed for topology implementation is outlined in section 4. We describe the status of our implementation in section 5.

2 Functionality of overlay software

Overlay networks form a layer for organisation and communication in distributed applications. This section describes their different levels of functionality as illustrated in figure 1. While the development process of overlay software deals with all of them, only few level are well supported by design aids and frameworks.

The lowest two levels comprise the general operating system support for Internet-level **network I/O** and edge-level **message passing**. These levels are not specific to overlays and are usually hidden by higher layers.

A number of overlays, such as Bamboo [6], are implemented on top of generic event-driven state machines like SEDA [7] that model **message processing** in Internet servers. While EDSMs were not designed for overlay development, they still provide a good abstraction level for scalable event processing (see 2.2).

Overlay routing protocols then deal with local routing decisions for scalable end-to-end message forwarding. They are distributed algorithms, executed at each member node, with the purpose of forwarding messages at the overlay level from senders to receivers. Routing is left out of figure 1 for clarity reasons. While situated at the message processing layer, it actually uses the topology rules as explained in the next section.

2.1 Overlay Software from the Topology Perspective

Where current frameworks focus on message forwarding and the protocol design part of overlay software, we propose raising the abstraction level to topology design. This is motivated by four more functional levels in overlay software.

Local topology rules play a major role in overlay software which makes them a very interesting abstraction level. The global topology of an overlay is established by a distributed algorithm that each member node executes. The topology rules on each node implement this algorithm by accepting neighbour candidates or objecting to them. Overlays traditionally implement these rules implicitly as part of their routing and maintenance algorithms, which is why frameworks currently ignore this level.

Fig. 1: Framework Support for Overlay Software

Functionality	**Support in current frameworks**	
Topology Selection	**Node views**	
Topology Adaptation		
Topology Maintenance		(Macedon)
Topology Rules		
Message Processing	iOverlay, Macedon	Flow-Graphs, EDSMs, ...
Message Passing		Serialisation, RPC, CORBA, ...
Network I/O		Sockets, TCP/UDP, ...

There are two sides to topology rules. *Node selection* allows an application to show interest in certain nodes and ignore others based on their status, attributes and capabilities. Generally, applications are only interested in nodes that they know (or assume) to be alive, usually based on the information when the last message from them arrived. But not even all locally known live nodes are interesting to the application that can select nodes for communication based on quality-of-service requirements. Furthermore, if a heterogeneous application uses multiple overlays, its participants do not necessarily support all running protocols. Each node must see the others only in overlays that they support.

Node categorisation is the second part. Where node selection is the black-and-white decision of seeing a node or not, categorisation determines *how* nodes are seen. Nearly all overlay networks know different kinds of neighbours: close and far ones, fast and slow ones, parents and children, super-nodes and peers, or nodes that store data of type A, B or C. Node categorisation lets a node sort other nodes into different buckets to distinguish them. Overlay routing and other overlay tasks are then implemented on top of the node categorisation.

In current structured overlay networks [1,2,3,6], topology rules are stated apart from the implementation as a local invariant whose global properties are either proven by hand or found in experiments. It is a hard problem but also an interesting question to what extent the process of building routing protocols from local rules and inferring the guarantees they provide can be automated.

Topology maintenance is the perpetual process of repairing the topology whenever it breaks the rules. Above all, this means integrating new nodes (i.e. selecting and categorising them) and replacing failed ones. Support for this functionality is very limited among the current frameworks, despite its obvious importance for self-maintaining overlays.

Topology adaptation is the ability of a given overlay topology to adapt to specific requirements. As opposed to the error correction of topology maintenance, adaptation handles the freedom of choice allowed by the topology rules. The rules therefore draw the line between maintenance and adaptation. An example is Pastry where evaluations have shown [8] that redundant entries in the routing table can be exploited for adaptation to achieve better resilience and lower latency. Topology adaptation usually defines some kind of metric for choosing new edges out of a valid set of candidates. Building the "right" sub-groups of nodes in hierarchical topologies also fits into this scheme.

Current overlays are designed with some kind of adaptation in mind, whereas the available frameworks do not provide support for its implementation. What is needed here is a ranking mechanism for connection candidates. Overlays usually aim to provide an "efficient" topology. The term efficiency, however, is always based on a specific choice of relevant metrics, such as end-to-end hop-count or edge latency, but possibly also the node degree or the expected quality of query results. The respective metric determines the node ranking which in turn parametrises the global properties of the topology.

Topology selection is the choice of different topologies that an overlay application can build on. Supporting multiple topologies obviously makes sense for debugging and testing at design-time. However, it is just as useful at run-time if an application has to adapt to diverse quality-of-service requirements, such as different preferences regarding reliability, throughput and latency. A given topology may excel in one or the other and this specialisation allows it to provide high performance while keeping a simple design. Topology selection allows an application to provide optimised solutions for different cases.

Topology adaptation and selection play the most important role for QoS support in overlays. However, selection obviously relies on the integration of different overlay implementations to make their topologies available to a single application. This is especially necessary to avoid duplication in effort when maintaining multiple topologies and switching between them. It is not efficient, for example, to have an application maintain several overlays if each of them independently sends pings to determine the availability of nodes. Integrative approaches like Node Views (as presented in section 3) become crucial here.

2.2 Frameworks and Middleware for Overlay Implementation

There have been a number of recent proposals for overlay frameworks and middleware. Macedon [9] and iOverlay [10] are under development and evaluation in the corresponding projects. Other frameworks, like SEDA [7] or JXTA (`http://www.jxta.org`), have also been used for overlay implementations, although they do not provide any higher-level support for topologies and other overlay specific tasks.

iOverlay essentially provides a message switch abstraction for the design of the local routing algorithm. The neighbours of a node are instantiated as local I/O queues between which the user provided implementation switches messages. This generally simplifies the design of overlay algorithms by hiding the lower networking levels. However, there is no further support for topology rules, maintenance or adaptation.

Macedon is a state machine compiler for overlay protocol design and forms the most interesting approach so far. Event-driven state machines (EDSMs) have been used over decades for protocol design and specification. Macedon extends this approach to an overlay specific, C++ based language from which it generates source code for overlay maintenance and routing. In a number of different proof-of-concept overlay implementations, this was shown to be very useful for implementing and testing algorithms for routing and maintenance.

Overlays must operate autonomously. This means that they must configure themselves and automatically adapt to a changing environment. However, this is not only a matter of designing a routing protocol. Each node in an overlay needs to take local decisions. The sum of these local decisions is the distributed algorithm that maintains the overlay. What are these local decisions based on?

iOverlay bases them on the currently available connections. It does not provide means for selecting the "right" connections or categorising them, neither does it support ranking connection candidates for adaptation and fall-back mechanisms. Similarly, Macedon does not support candidate nodes or adaptability of topologies. Modelling adaptivity in state machines is even likely to be rather complex and can lead to state explosion. Consequently, in all of these incompatible and language dependent frameworks, the designer is forced to model local decisions in framework specific source code.

2.3 Local Decisions and Data about Nodes

The local decisions, that each participant in a distributed algorithm takes, rely on the local view of that node. The local view is a node's combined knowledge about the other nodes in the system, above all its neighbours in the topology.

To establish a local view, each node has to keep data about other nodes. Examples are addresses and identifiers, measured or estimated latencies and references to data stored on these nodes. Furthermore, it is generally of interest when a node was last contacted (time-stamps or history) to determine if it is alive.

Data about remote nodes is gathered from diverse sources. Some data can be determined locally (IP address, ping latency, ...), while other information is received in dedicated messages - either directly from the node it describes or indirectly via hearsay of intermediate nodes. There is often more than one way of finding equivalent data. Latencies, for example, can be measured (ping) or estimated [11,12]. A node A knows that a node B is alive if A received a ping response or other message from B, if it heard about it from other nodes (gossip), etc. Different quality-of-service levels in an overlay application can trade load against certainty by selecting different sources.

Topology rules, maintenance, adaptation and selection mainly deal with managing data about nodes. The topology rules put constraints on the data about possible neighbour nodes. Maintenance needs to keep data about fall-back candidates that may currently not be neighbours. It also deals with gathering data about nodes that joined or finding conflicts between local and remote views. Adaptation does a ranking between candidate nodes before it decides about the instantiation as neighbours or fall-backs. Topology selection then switches between different views, i.e. ranking metrics and sets of neighbours.

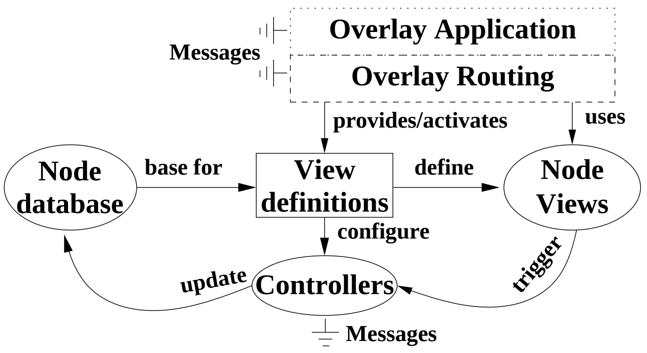
A data abstraction is obviously a good way of dealing with this diversity of sources, data characteristics and data management tasks. It allows an overlay to lift dependencies on specific algorithms and to take advantage of the different characteristics of different implementations as the need arises.

3 Node Views, the System Model

We propose to design overlay frameworks as data management systems using the well-known Model-View-Controller pattern [13]. The *model* is an active local database on each node, a central storage place for all data that a node knows about remote nodes. Once the data is stored in a single place, software components no longer have to care about any data management themselves. They benefit from a locally consistent data store and from notifications about changes.

The major characteristics of the overlay topology are then defined in *views* of the database. They represent sets of nodes that are of interest to the local node (such as its neighbours). Different views provide different ways of selecting and categorising nodes, and different ways of adapting topologies. Topology selection is then mainly a matter of selecting the right set of views.

Fig. 2: Components of the System Model



As the views form the most important overlay specific part of the implementation, they are also the most crucial part for an abstract and framework-independent specification. Their definition is the main goal of the SLOSL language that is briefly presented in the next section.

The *controllers* are tiny EDSM states that operate on the views. They are triggered by events like incoming or leaving messages, timers or changes in the views and update the database according to the view definitions. They are the actual maintenance components that perform simple tasks like updating single attributes of nodes when new data becomes available or sending out messages to search new nodes that match the current view definitions. Note that the controllers do not aim to provide a global view for the model. They continuously update and repair the restricted and possibly globally inconsistent local view. The node database decouples them from other parts of the overlay software and the node views provide them with simplified, decoupled layers and a common interface to make them generic, reusable components in frameworks.

Another very important part of the architecture is an expressive *event system* for view events and messages. A notification about changes in views is fired whenever nodes enter or leave a view, or when visible node attributes change. Views filter notifications and software components only react to events from the views that they are subscribed to.

Components like message handlers or routers are still part of the overlay specific implementation, but they can now respond to specific events and use node views for their decisions. Defining messages as hierarchical structures allows components to subscribe to data fields instead of monolithic messages. This further helps in writing generic components. Database and views decouple them from the maintenance components and simplify their design considerably. Even more so, as this architecture can provide powerful operations like topology selection and adaptation with a single view selection command. The abstract view definition becomes the central point of control for the characteristics of the overlay.

4 Slosl, the View Specification Language

For the view definitions that implement topology rules and adaptation, we developed Slosl, the SQL-Like Overlay Specification Language [14]. We present it here using a simple example, an implementation of the Chord graph [2].

```
CREATE VIEW chord_fingertable
AS SELECT node.id, node.ring_dist, bdist=node.ring_dist−2^i
RANKED lowest(backups+i, node.msec_latency / node.ring_dist)
FROM node_db
WITH log_k = log(|𝒦|), backups = 1
WHERE node.supports_chord = true AND node.alive = true
HAVING node.ring_dist in (2^i : 2^(i+1))
FOREACH i IN (0:log_k)
```

The statements CREATE VIEW, SELECT, FROM and WHERE behave as in SQL. The WHERE clause specifically implements **node selection** based on node attributes. Note that Slosl is not concerned with the source of the information that node attributes contain. It only constrains and categorises the presentation of locally available data. The remaining clauses do the following:

WITH This clause defines variables or options of this view that can be set at instantiation time and changed at run-time. Here, *log_k* will likely keep its default value, while *backups* allows adding redundancy at runtime.

HAVING–FOREACH This pair of clauses aggregates the selected nodes into buckets to implement **node categorisation**. In the example, the (constant) node attribute *ring_dist* refers to the logical distance between the local node and the remote node. The HAVING expression states that it must lie within the given half-open interval (excluding the highest value) that depends on the bucket variable i.

The FOREACH part defines the available node buckets by declaring this bucket variable over a range (or a list, database table, . . .) of values. It defines either a single bucket of nodes, or a list, matrix, cube, etc. of buckets. The structure is imposed by the occurrence of zero or more FOREACH clauses, where each clause adds a dimension. Nodes are selected into these buckets by the optional HAVING expression.

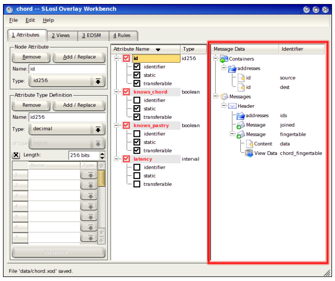
The example shows a case where the SELECT clause gives nodes a new attribute `bdist` representing their position inside the bucket. Calculating attribute values is particularly useful for HAVING expressions that allow a node to appear in multiple buckets of the same view.

RANKED To support **topology adaptation**, the nodes in the `chord_fingertable` view are chosen by the ranking function *lowest* as the $backups + i$ top node(s) of each bucket that provide the lowest value for the given expression. Rankings are often based on the network latency, but any arithmetic expression based on node attributes can be used. The expression in the example implements a simple tradeoff between the network latency and the distance travelled in the ID space. Other overlays may require more complex expressions or user defined functions in the ranking expression.

5 Implementation, current and future work

We are developing two different proof-of-concept implementations of this architecture as overlay execution environments. A first, light-weight prototype was written in Python, while our current work builds on the PostgreSQL database. It is targeted as a reference system rather than a high performance one. Once the APIs have become stable enough, we can let the architecture benefit from standard approaches used in Internet servers and application server designs.

As a major step towards simplified, abstract overlay development, we have designed a graphical editor (fig. 3) based on our system model. It allows the framework independent specification of overlay systems and outputs abstract overlay specifications in OverML [14], a new XML specification language for node attributes, SLOSL statements, messages and EDSM flow descriptions.


chord -- SLosl Overlay Workbench
File Edit Help
1 Attributes 2 Views 3 EDSM 4 Rules
Remove
Add / Replace
Advanced ...

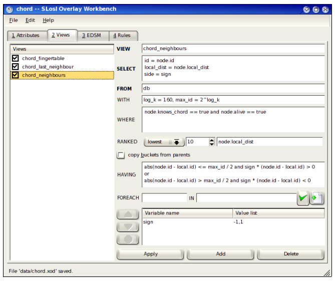
Node Attribute — Name: id — Type: id256
Attribute Type Definition — Name: id256 — Type: decimal — of type: bigint — Length: 256 bits

Attribute Name	Type
id	id256
☑ identifier	
☑ static	
☑ transferable	
knows_chord	boolean
☐ identifier	
☑ static	
☑ transferable	
knows_pastry	boolean
☐ identifier	
☑ static	
☑ transferable	
latency	interval
☐ identifier	
☐ static	
☐ transferable	

Message Data	Identifier
Containers	
addresses	
id	source
id	dest
Messages	
Header	
addresses	ids
Message	joined
Message	fingertable
Content	data
View Data	chord_fingertable

File 'data/chord.xod' saved.

Attributes/Messages


chord -- SLosl Overlay Workbench
File Edit Help
1 Attributes 2 Views 3 EDSM 4 Rules
Apply
Add
Delete

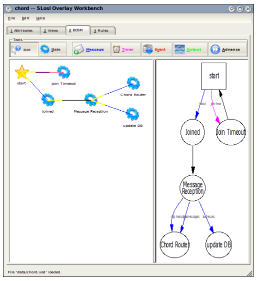
Views: ☑ chord_fingertable, ☑ chord_last_neighbour, ☑ chord_neighbours

```
VIEW      chord_neighbours
SELECT    id = node.id
          local_dist = node.local_dist
          side = sign
FROM      db
WITH      log_k = 160, max_id = 2^log_k
WHERE     node.knows_chord == true and node.alive == true
RANKED    lowest  10  node.local_dist
☐ copy buckets from parents
HAVING    abs(node.id - local.id) <= max_id / 2 and sign * (node.id - local.id) > 0
          or
          abs(node.id - local.id) > max_id / 2 and sign * (node.id - local.id) < 0
FOREACH             IN
```

Variable name	Value list
sign	-1,1

File 'data/chord.xod' saved.

SLOSL statements



Event flows

Fig. 3: The SLOSL Overlay Workbench

For the future, we hope for diverse implementations of OverML compatible frameworks as well as mappings to existing frameworks. The high abstraction level easily allows specialised environments for simulation and analysis, testing and debugging, and different deployment scenarios – without changes to the overlay specification. Deployment environments can use a rather lightweight or custom database. An interesting topic to investigate here is (partial) source code generation from SLOSL statements. This should allow customised overlay implementations for very efficient deployment.

Simulators and debuggers may prefer a single global database to enable tracing, verifying and visualising the system state. Recent proposals for scalable simulation environments [15] already take a layered approach. Simulations are carried out at a higher abstraction level and are then mapped to the network link level. We propose the database layer as a comfortable abstraction level.

Future work will also include better mechanisms for view and query optimisation. Our current PostgreSQL implementation maps SLOSL statements to rather complex, generic SQL queries. Building on the large body of literature on query modification and optimisation, we can imagine a number of ways to investigate for pre-optimising these statements. This is most interesting for views of views and for merging view definitions when sending them over the wire (like in gossip overlays [16] or hierarchical environments [17]).

6 Conclusion

This paper presented *Node Views*, a novel approach to overlay design frameworks that enables support for topology rules, maintenance, adaptation and selection at a very high level. Based on an active database, it allows for a separation of topology implementation, maintenance and message handling. This facilitates the development of generic components which enables pluggable development and integration of overlay systems.

The SLOSL language lifts the abstraction level for overlay design from messaging and routing protocols to the topology level. Its short, SQL-like statements meet the requirements for design-time specification, topology implementation and run-time adaptation of highly configurable overlay systems.

The current state of our implementation does not allow a performance comparison between the available hand-optimised overlay implementations and SLOSL based ones. In any case, the high abstraction level of Node Views will likely lead to slower systems in direct comparisons - but in a couple of hours implementation time compared to weeks for writing a traditional overlay from scratch.

Even compared to the days it takes to understand and start using one of the available overlay systems, SLOSL wins by being much easier to read and allowing overlays to gain orders of magnitude in configurability, adaptability and integration. The SLOSL Overlay Workbench makes overlay software easy and fast to write and shifts more of the development time towards testing and optimising the topology itself and choosing the right maintenance strategies. As with any other high-level language, long-term optimisations of OverML compatible platforms will improve the performance of overlays using them.

The Node Views approach encourages completely new ways of designing and testing overlays. Modifying compact SLOSL statements allows the designer to easily test and compare the impact of different selection and ranking functions on an application. Switching between different views and controllers, at design-time or run-time, enables overlay applications to adapt to the broad range from static to dynamic environments and to diverse quality-of-service requirements.

References

1. Aberer, K.: P-Grid: A Self-Organizing access structure for P2P information systems. In: Proc. of the Sixth Int. Conference on Cooperative Information Systems (CoopIS 2001), Trento, Italy. (2001)
2. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proc. of the 2001 ACM SIGCOMM Conference, San Diego, California, USA (2001)
3. Loguinov, D., Kumar, A., Rai, V., Ganesh, S.: Graph-theoretic analysis of structured peer-to-peer systems: Routing distances and fault resilience. [19]
4. Chawathe, Y., Ratnasamy, S., Breslau, L., Lanham, N., Shenker, S.: Making gnutella-like p2p systems scalable. [19]
5. Dabek, F., Zhao, B., Druschel, P., Stoica, I.: Towards a common API for structured peer-to-peer overlays. [18]

6. Rhea, S., Geels, D., Roscoe, T., Kubiatowicz, J.: Handling churn in a DHT. In: Proc. of the USENIX Annual Technical Conference, Boston, MA, USA (2004)
7. Welsh, M., Culler, D., Brewer, E.: SEDA: An architecture for well-conditioned, scalable internet services. In: Proc. of the 18th ACM symposium on operating systems principles, Banff, Alberta, Canada (2001)
8. Zhang, R., Hu, Y.C., Druschel, P.: Optimizing routing in structured peer-to-peer overlay networks using routing table redundancy. In: Proc. of the 9th Int. Workshop on Future Trends of Distributed Computing Systems (FTDCS'03), San Juan, Puerto Rico (2003)
9. Rodriguez, A., Killian, C., Bhat, S., Kostić, D., Vahdat, A.: MACEDON: Methodology for automatically creating, evaluating, and designing overlay networks. In: Proc. of the USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI2004), San Francisco, CA, USA (2004)
10. Li, B., Guo, J., Wan, M.: iOverlay: A lightweight middleware infrastructure for overlay application implementations. In: Proc. of the Int. Middleware Conference (Middleware2004), Toronto, Canada (2004)
11. Dabek, F., Cox, R., Kaashoek, F., Morris, R.: Vivaldi: A decentralized network coordinate system. In: Proc. of the 2004 ACM SIGCOMM Conference, Portland, Oregon, USA (2004)
12. Eugene Ng, T.S., Zhang, H.: Predicting internet network distance with coordinates-based approaches. In: INFOCOM 2002, New York, USA (2002)
13. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons (1996)
14. Behnel, S., Buchmann, A.: Models and languages for overlay networks. In: Proc. of the 3rd Int. VLDB Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P 2005), Trondheim, Norway (2005)
15. Birck, H., Heckmann, O., Mauthe, A., Steinmetz, R.: The two-step overlay network simulation approach. In: Proc. of SoftCOM, Split, Croatia. (2004)
16. Gupta, I., Birman, K., Linga, P., Demers, A., van Renesse, R.: Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. [18]
17. Darlagiannis, V., Mauthe, A., Steinmetz, R.: Overlay design mechanisms for heterogeneous, large scale, dynamic P2P systems. Journal of Network and Systems Management, Special Issue on Distributed Management **12** (2004)
18. The 2nd International Workshop on Peer-to-Peer Systems (IPTPS03), Berkeley, CA, USA (2003)
19. The 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM), Karlsruhe, Germany (2003)