Overlay Networks – Implementation by Specification ⋆

Stefan Behnel, Alejandro Buchmann

Databases and Distributed Systems Group,
Darmstadt University of Technology (TUD),
D-64289 Darmstadt, 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 such as 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 integration in frameworks. The major part of the overlay implementation is separated out into an SQL-like specification language for the topology layer. It lets us implement common overlay topologies like Chord [22] or de-Bruijn graphs [14] in a few lines of code.

1 Introduction

Recent years have seen a large body of research in decentralised, self-maintaining overlay networks like P-Grid [1], Pastry [20], ODRI [14] or Gia [3]. They are commonly regarded as building blocks for Internet-scale distributed applications.

Contrary to this expectation, current overlay implementations are built with incompatible frameworks on top of low level networking abstractions. This complicates their design and hinders the comparison and exchangeability of different topologies. As a partial solution, a recently proposed standard API for the specific case of structured overlay networks [5] has been adopted in a number of implementations. However, 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

⋆ This work was funded by the German National Science Foundation (DFG) as part of the Graduate College "System Integration for Ubiquitous Computing".

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, some of which are exemplarily outlined in section 5. The SQL-like language that we designed for implementing the different functionality levels in overlay topologies is described in section 4. We describe the status of our implementation in section 6.

2 Functionality of overlay software

We understand overlay networks as a layer for organisation and communication in distributed applications. This perspective allows us to extract different levels of functionality in overlay software. They are 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.

2.1 Overlay software from the networking perspective

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

A number of overlays, such as Bamboo [18], are implemented on top of generic event-driven state machines for Internet servers that facilitate **local message processing**. While these do not offer support for overlay specific tasks, they form a reasonable abstraction level for networking software. However, two recent frameworks, Macedon [19] and iOverlay [13], visibly raise the abstraction level and show the advantages of approaches that are specifically designed for overlay implementation.

Fig. 1. Levels of functionality in overlay software

Topology Selection	**Node views**	
Topology Adaptation		RaDP2P
Topology Maintenance		(Macedon)
Topology Rules		
Message Processing	iOverlay, Macedon	Flow-Graphs, EDSMs, ...
Message Passing		Serialisation, RPC, CORBA, ...
Network I/O		Sockets, TCP/UDP, ...

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. It is situated at the message processing layer, but uses the topology rules as explained in the next section. Macedon and iOverlay support routing protocols in different ways.

2.2 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.

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 task. Where 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 and nodes that store data of type B. 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.

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. In current structured overlay networks, topology rules are stated apart from the implementation as a local invariant whose global properties are either proven by hand or found in experiments.

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 [25] 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" subgroups of nodes in hierarchical topologies (or embedded groups as in [11]) 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 runtime 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.

2.3 Frameworks and middleware for overlay software

There have been a number of recent proposals for overlay frameworks and middleware. Macedon [19], iOverlay [13] and RaDP2P [8] are under development and evaluation in the corresponding projects. Other frameworks, like SEDA [24] or JXTA (http://www.jxta.org), have also been used for overlay implementations.

The *Staged Event-Driven Architecture* (SEDA) is a framework for scalable Internet servers. It provides an event-driven state machine for request processing

that is used in OceanStore [12] and Bamboo [18]. Besides network I/O and message processing, SEDA does not provide any specific support for overlays.

JXTA was designed as a framework for peer-to-peer applications. Its main focus are unstructured broadcast networks and it provides means for grouping and contacting nodes. There is no actual support for other topologies.

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.

RaDP2P is described as a policy framework for adapting structured overlays. User defined policies derive node IDs from application semantics and reassign them dynamically to nodes. Although this is an interesting idea, there is currently neither a working implementation nor any publicly available information describing what the policies look like or how they are implemented. Also, RaDP2P does not provide support for implementing the overlays themselves.

Macedon then forms the most interesting approach so far. It is essentially a state machine compiler for overlay protocol design. Event-driven state machines (EDSMs) have been used over decades for protocol design and specification. Macedon extends this approach to an overlay specific language from which it generates source code for overlay maintenance and routing.

2.4 What is missing?

Overlays are expected to 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 make 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 and establishes a switch that routes messages between them. 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.

Macedon provides an event-driven state machine abstraction. In a number of different proof-of-concept overlay implementations, this was shown to be very useful and efficient for implementing and testing algorithms for routing and maintenance. However, as iOverlay, it 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.

The following section presents the Node Views abstraction, complementing the networking based approaches. It provides an integrative middleware layer supporting local decisions for maintaining, adapting and selecting overlay topologies.

3 Node Views

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 (but not limited to) its neighbours in the topology.

3.1 Data about nodes

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), if a node is considered alive or if a ping was recently issued to check if it still is. The information that this was necessary may be very valuable to, say, a routing component that can take the decision to temporarily route around that queried node in order to keep the hop-by-hop loss rate low. Locally available data about remote nodes is crucial to all components of the overlay software.

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 often is more than one way of finding equivalent data. Section 5.3 describes the example of latencies being measured (ping) or estimated. As another example, a node A knows that a node B is alive if A recently succeeded in pinging B, if A received a message from B, if other nodes told A about B (gossip), etc. Note that both overhead and certainty decrease in this order. 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.

We propose to instantiate the local view of a node as a local database that keeps all locally known data about remote nodes. We can then start to apply common approaches from active database management systems in order to model and implement the data management part that is necessary to maintain and adapt topologies. The next section presents the proposed architectural model.

3.2 System Architecture

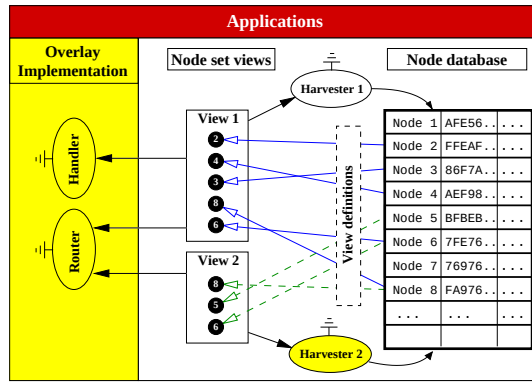


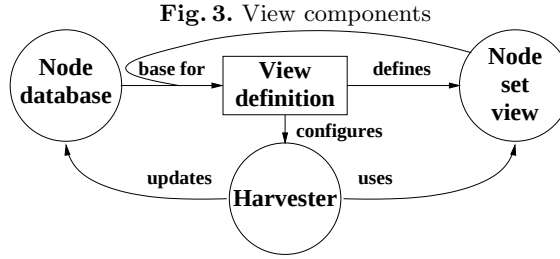
Fig. 2. The overall system architecture

A **node** is the local representation of a remote member of the topology and the "atomic unit" for overlay networks. It has a number of attributes locally assigned to it. Nodes and their attributes become available to the local view through physical links, messages or via hearsay of other members.

The active **node database** represents the complete local view of a node. It stores the attributes of all locally known nodes which makes it a locally consistent storage point for all software components. Note that the database is independent of specific topologies and overlay implementations and that it abstracts from the way how nodes and their attributes are found.

Node set views represent the views that are specific to a topology[1]. They are active views of the node database and form the most important abstraction for overlay implementations. They can also be seen as filters for the database. The software components that use a view are restricted to see only a relevant subset of nodes and their attributes. This subset is determined by a **view definition**. As generally known from database views, this is a function that constrains the node data taken from the database or recursively from other views. The interaction between database, views and their definition is illustrated in figure 3. The view definition language we propose is presented in section 4.

Fig. 3. View components



[1] Note that a single topology can have multiple views, such as the leaf-set and routing table found in Pastry [20], or different hierarchical levels as in super-node networks.

View events are changes to a view. A notification about them is fired whenever nodes enter or leave a view, or when visible node attributes change. The node database can support this with a technique that is long known in the active database area, Event-Condition-Action rules [7]. They trigger reactive components of the software when data changes occur. Views filter notifications and software components receive only those from the views they see.

Harvesters are a major part of the overlay maintenance implementation. They aim to update the database according to the view definitions (known as view maintenance in database research). This includes updating single attributes of nodes as well as searching new nodes that match the current view definitions. Note that harvesters do not aim to provide a global view. They continuously update and repair the restricted and possibly globally inconsistent local view. The node database effectively decouples them from other parts of the overlay software which makes them generic components for a framework. Harvesters are further discussed in section 5.

Other overlay components like message handlers or routers use node set views for their decisions. Again, database and views decouple them from maintenance components and simplify their design. Frameworks like Macedon then provide efficient means to implement them.

3.3 Discussion

The architecture uses the well-known Model-View-Controller pattern (database – node set views – harvesters) to simplify and decouple large parts of the overlay implementation. Since 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.

As known from database views for server applications, node set views provide simplified, decoupled layers and a common interface for overlay software components. This makes components reusable and allows their generic implementation to become part of frameworks. Even if a specialised router has to be written from scratch, it can usually be implemented in the order of tens of code lines when connected to consistent, active node set views.

Note that the node database also helps in other situations. When an overlay application terminates and is restarted, it has to reconnect to the overlay. If the local node database is persistent, it can directly start ranking nodes to find good candidates for joining. This simplifies the join procedure and provides better support for bootstrapping than a simple, semantic-free list of network addresses as currently used in most overlay implementations.

The most important feature of this abstraction, however, is the inherent support for topology rules and adaptation. The view definition becomes the central point of control for the characteristics of the overlay implementation. The language that we use for this purpose is described in the following section. Its high expressiveness for ranking, selecting and categorising nodes allows for a broad range of overlay design decisions while at the same time separating and hiding them from the implementation of overlay components.

4 The View Specification Language

For the view definitions that implement topology rules and adaptation, we developed SLOSL, the SQL-Like Overlay Specification Language. The following subsections describe the grammar in detail, first the view definition part (4.1) and subsequently the harvester configuration (4.3). However, let us start with a simple example, an implementation of the Chord graph [22].

```
1  CREATE VIEW chord_fingertable
2  AS SELECT node.id
3  RANKED highest(1, ring_dist(local.id, node.id))
4  FROM node_db
5  WITH local, log_k = log(ℵ)
6  WHERE node.supports_chord = true AND node.alive = true
7  HAVING ring_dist(local.id, node.id) in (2^i : 2^(i+1))
8  FOREACH i IN (0:log_k)
9  ;
```

The code lines do the following:

1. Define a node set view to be known under the name *chord_fingertable*.
2. The interface of this view presents the *id* attribute of its nodes to the components that use it (and no other attributes).
3. The nodes in the created view are chosen by the ranking function *highest* as the 1 top node(s) that provide the highest value for the term *ring_dist(local.id, node.id)*. The ranking function allows us to do topology adaptation. For readability, the name *ring_dist* refers to a user-provided function[2]. However, the distance is locally a constant which could just as well be a node attribute.
4. The next line states the super-views of the newly created one. In this case, it is a direct sub-view of the local node database.
5. The WITH clause defines variables or options of this view that can be set at instantiation time and changed at run-time. *local* is a very common option (and actually a keyword) that refers to the local node. The variable *log_k* is given a default value, the logarithm of the size of the key space.
6. The WHERE clause constrains nodes that are considered valid candidates for this view (node selection). Here we have defined an attribute *supports_chord* that is true for all nodes that know about the Chord protocol. The second attribute, *alive*, is true for nodes that the responsible harvester deemed alive.

7-8. The HAVING ... FOREACH clause aggregates valid candidates into buckets (node categorisation). In the example, the HAVING part states that the result of the distance function 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) of values. A discussion follows.

9. Every SLOSL statement terminates with a colon. SQL does it, we do it.

[2] In SQL: CASE WHEN $(l - n) < 0$ THEN $max\_id + (l - n)$ ELSE $(l - n)$ END

4.1 Grammar

This section describes the different clauses of SLOSL. We define the grammar in a simple, EBNF-like fashion. A few conventions are used:

- *somename-commalist* denotes a comma-separated list of *somename* elements.
- *expression* denotes more or less arbitrary arithmetic expressions, *bool-expression* is an expression that returns a boolean value (true/false), based on comparisons and the common operators *AND*, *OR*, *NOT*.
- *somename-identifier* denotes an identifier and gives it the name *somename*. This name is not used in the grammar but indicates its meaning. In any case, identifiers are lower-case alpha-numeric names (including underscores) that start with a letter.

As can be seen from the example, the complete statement of a SLOSL definition consists of seven clauses, followed by an optional set of rules (see 4.3):

```
statement :== create attr-select [ranking] parent-select \
              [options] [where] [bucket-select] [rules] ';'
```

Comments up to the end of the line are allowed everywhere (outside tokens) and begin with a double hyphen (– –).

CREATE VIEW

```
create :== 'CREATE VIEW' view-identifier [ 'AS' ]
```

This clause defines the name of the view.

SELECT

```
attr-select   :== 'SELECT' attribute-def-commalist
attribute-def :== 'node.' attribute-identifier [ '=' expression ]
```

This clause defines the node interface. Only the attributes mentioned here can be seen by components that use this view. However, all attributes found in the parent views can be used for the view definition.

It is quite a capable feature that attributes can be assigned their value by arbitrary expressions that may or may not depend on the attributes with the same name in the parent views. This can be used for renaming attributes, e.g. to trick a black-box component into using other attributes, to convert between different attribute representations (time, value ranges, ...) or to prepare data for transmission in message protocols. If no assignment is supplied, the attribute is presented unchanged as found in the parent views.

RANKED

```
ranking  :== 'RANKED' function '(' expression-commalist ')'
function :== 'lowest' | 'highest' | 'closest' | 'furthest'
```

This clause describes a filter (the ranking function) that selects a list of nodes from a list of candidate nodes. As motivated in section 2, ranking nodes is a major task in overlay adaptation. SLOSL moves it from the implementation into the view definition and makes it configurable and adaptable through view options and bucket variables.

An important feature of our approach is that an application can use multiple specifications and/or rankings to instantiate on-demand the views that currently fit best. In the Chord example, nodes are ranked by highest ID, but one could as well use the lowest latency, highest uptime, age of last contact, or some reliability or reputation measure. It is obvious that switching between different ranking mechanisms (SELECT) or node selections (WHERE) does not have any impact on the implementation of routers, harvesters or similarly connected overlay software components, as long as the bucket structure of the view (FOREACH) stays unchanged. It does, however, have a major impact on the global topology and its performance characteristics.

FROM

```
parent-select :== 'FROM' parent-identifier-commalist
```

Each view takes its nodes from its parent view(s). The set of candidate nodes is the union of all nodes in its parents. All parent views must provide at least those attributes for their nodes that are referenced in the SLOSL statement.

WITH

```
options           :== 'WITH' option-assignment-commalist
option-assignment :== option-identifier [ '=' expression ]
```

The WITH clause names configurable options of the defined view. They can be referenced in the clauses SELECT, RANKED, WHERE and HAVING–FOREACH. Options can be assigned a default value in the definition, set at view instantiation time and changed at run-time.

Special care must be taken if they are used in the FOREACH list (as shown in the Chord example). Changing their value at run-time can mutate the structure of the view buckets. This may break components connected to the view if they expect a static bucket structure.

WHERE

```
where :== 'WHERE' bool-expression
```

The boolean expression puts constraints on the attributes of nodes that are considered candidates for this view. This clause implements the node selection part of the topology rules.

HAVING ... FOREACH

```
bucket-select :== 'FOREACH' 'BUCKET'
bucket-select :== [ 'HAVING' bool-expression ] ( bucket-def )+
bucket-def    :== 'FOREACH' bucket-variable-identifier \
                  'IN' ( range | constants-commalist ) )
range         :== integer ':' integer
```

This clause aggregates nodes into buckets. It implements the node categorisation of the topology rules. In the simplest case, the FOREACH BUCKET clause copies the buckets from the parent views. View definitions can thus ignore the bucket structure of parents and still provide it as their own interface. This allows template-like view definitions that change only the interface of nodes (SELECT) or that select a subset of nodes (WHERE) from the buckets.

In the second and more common form, the HAVING ... FOREACH clause 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–IN clauses, where each clause adds a dimension. Nodes are then selected into these buckets by the HAVING expression (which is optional and defaults to true).

A node can appear in multiple buckets of the same view if the HAVING expression allows it. Note that the bucket variable can be used in the view attribute definition (SELECT). This allows the same node to carry different attributes in different buckets of the created view (see the Pastry example in 4.2).

The bucket abstraction is enough to implement graphs like Chord, Pastry, Kademlia or de-Bruijn and should be just as useful for a large number of other cases. It is not limited to numbers and ranges, buckets can be defined on any list. Numbers are commonly used in structured overlays (which are based on numeric identifiers), while strings can for example be used for topic-clustering in unstructured networks. Since SLOSL is working on a database, the values in the FOREACH list can naturally be taken from a database table. However, the warning about variable usage (WITH) applies in this case, too.

4.2 More Examples

De-Bruijn graphs [14] combine a 1-dimensional view with some calculations that yield at most one node per bucket. Note that due to the structure of these graphs, a functional overlay implementation will need more than one view here to compensate for empty buckets if the ID space is not completely filled up with nodes. Such approaches are described in [6, 14].

```
1  CREATE VIEW de_bruijn
2  AS SELECT node.id
3  FROM node_db
4  WITH local, max_id=2^64, max_digits=8
5  WHERE node.alive = true
6  HAVING node.id = (local.id * max_digits) % max_id + digit
7  FOREACH digit IN (0:max_digits)
8  ;
```

Pastry [20] uses a 2-dimensional view and two user-provided functions: *digit_at_pos* for the digit at a given prefix position in the ID and *prefix_len*, the length of the longest common prefix of two IDs. As in the Chord example, the ring distance is a simple function that is left out for readability reasons.

```
1  CREATE VIEW pastry_routingtable
2  AS SELECT node.hex_id, node.id
3  RANKED highest(1, ring_dist(local.id, node.id))
4  FROM node_db
5  WITH local, max_digit=16, max_prefix=40
6  WHERE node.alive = true
7  HAVING digit_at_pos(node.hex_id, prefix) = digit
8      AND prefix_len(local.hex_id, node.hex_id) = prefix
9  FOREACH digit IN (0:max_digit)
10 FOREACH prefix IN (0:max_prefix)
11 ;
```

Pastry uses a second view for what it calls a leaf-set. It contains the direct neighbours in the ring, both on the left and the right side of the local node. There are different ways of specifying the leaf-set in Slosl, one being the use of two views for left and right neighbours. Here, we present a second (and a little more complex) possibility that uses two buckets for this purpose. Note also the use of a new node attribute *side* to memorise where the neighbour was found. The *ncount* variable makes the number of neighbours configurable at run-time.

```
1  CREATE VIEW circle_neighbours
2  AS SELECT node.id, node.side=sign
3  RANKED lowest(ncount, ring_dist(local.id, node.id))
4  FROM node_db
5  WITH local, ncount=10, max_id=2^160 - 1
6  WHERE node.alive = true
7  HAVING abs(node.id - local.id) <= max_id / 2
8         AND sign*(node.id - local.id) < 0
9      OR abs(node.id - local.id) > max_id / 2
10        AND sign*(node.id - local.id) > 0
11 FOREACH sign IN (-1,1)
12 ;
```

4.3 Harvester configuration

```
rules :== ( update | eca )+
```

The harvester configuration is a second (minor) part of Slosl. For each view, Slosl allows the specification of constraints and event-condition-action (ECA) rules on the data acquisition process. They specify how old each attribute value may be before it is forced to be updated, or how these values should be gathered. Specialised constraints are generally preferred over generic ECA rules as they

are easier to optimise. Note that providing these rules within the view definition is only one way of specifying them. They can be set and modified at run-time.

Rules can depend on bucket variables (FOREACH). Specifying rules for views and/or buckets supports a very fine-grained configuration. Evaluating and merging these rules at the database level avoids duplication of effort (pings, etc.) when using multiple overlays and views concurrently.

As an example, Bamboo and Pastry keep most of the leaf-set nodes also in the routing table. The leaf-set nodes are queried more often as they are more important for topology correctness than the routing table entries. The maintainers for both data structures must be coordinated to avoid duplicate pings to the same node. The local consistency of the node database directly supports this coordination. The combination of fine-grained configuration and centralised coordination is a major advantage of our integrative, high-level approach.

UPDATE

```
update :== 'UPDATE' attribute-identifier-commalist \
           'BY' update-method \
           [ 'WITHIN' [ min-time ':' ] max-time ]
```

Again, we start with an Example. The following statement specifies that the last update of a node's latency and its live state must not be older than two minutes. We query the respective node by direct request if it is older, but avoid doing so within 10 seconds after the last update.

UPDATE `node.latency, node.alive` **BY** `request` **WITHIN** `10s:2m`

The *update method* is currently either 'request' for sending a direct message (to the node or some other authority), 'gossip' for including the node attribute in the gossip exchange (see section 5.1) or 'listen' for adding the attribute to the message header and reading it from incoming messages. The available update methods and their exact meaning depend on the harvester implementations that are currently instantiated or registered in the system.

The *time* values are given in milliseconds or using a modifier. The harvester responsible for finding the attribute values uses them to configure its query interval, gossip cycle length, timeout for ACKs, etc. Modifiers are preferred where possible, both for readability reasons and to avoid a source of errors. Valid modifiers are 's' for seconds, 'm' for minutes, 'h' for hours and 'd' for days. Time values can be added like in '2d+2h' or '2s-100' or they can be multiplied with a constant, bucket variable or attribute value like in '2*`node.latency`+`i`*1s'. The result must not be negative. The min-time defaults to 0.

If multiple constraints are specified on the same attribute of a node (possibly in different view definitions or buckets), the most restrictive is used. This enforces the lowest time bounds and the fastest update method (for example, it prefers direct requests over listening over gossip). If a constrained attribute depends on other attributes (e.g. by attribute definition in the SELECT clause), the constraint will be applied to all attributes it depends on.

ECA rules

```
eca :== 'ON' event ( 'OR' event )* [ 'IF' condition ] 'DO' action
```

ECA rules (Event-Condition-Action rules [7]) define actions to execute when an event occurs, but only if a condition is fulfilled at that moment. If no condition is provided, it is assumed to be always true.

Events are currently limited to database changes ('INSERT', 'UPDATE', 'DELETE') of nodes and specifically named attributes. Alternative events (separated by 'OR') are allowed for convenience, but they can also be written as multiple rules. Valid conditions are attribute comparisons as well as ('NOT') 'EMPTY' buckets. 'COUNT' is another valid aggregation function for the size of a bucket, 'COUNT(*)' denotes the total number of nodes in the view.

An example: trigger a harvester when the bucket with variable value $i = 12$ contains only one node or when any of the buckets is empty.

ON `DELETE node` **OR** `UPDATE node.alive`
IF `EMPTY(i) OR COUNT(i=12) = 1`
DO `trigger_my_bucket_harvester(i)`

5 Harvester Patterns

Harvesters are major components that aim to fill the database with data relevant to the currently available view definitions. This means finding new nodes and determining their attributes. There are a number of possible schemes and each view definition can select the most appropriate one. Having a choice of harvesters available as components allows overlays to provide very diverse characteristics. In this section, we exemplarily discuss three different kinds of harvesters. Note that much of what we say here can be applied in similar ways to other harvester types and implementations.

5.1 Gossip based node harvesters

New nodes can be found via broadcast, active lookup, overhearing routed messages, central discovery services, etc. Especially gossip or epidemic communication [10, 18, 23] has a straight forward implementation in our model that exchanges views between overlay members.

The idea behind epidemic communication is to spread knowledge to all members of a system similar to an epidemic, i.e. by periodically "infecting" randomly chosen members. On each contact, a member communicates data to another member who can then start to infect others. This leads to an exponential data distribution while keeping the load at each member low.

As an example, we can look at a Pastry derivative called Bamboo [18]. It uses multiple epidemic maintainers for its local data structures. The leafset maintainer periodically exchanges this set with one of the nodes therein. Two routing table maintainers query member nodes and try to fill empty entries.

Jelasity etal. [10] introduced the "Peer Sampling Service" (PSP). It is an overlay service for choosing good candidates for epidemic contact. Their analysis is focused on communicating the availability of nodes in unstructured networks by exchanging address data. This allows them to abstract from the actual way in which data is exchanged. However, if structured networks, multiple overlays or adaptable topologies have to be maintained, the information that must be exchanged is usually more complex.

Node set views directly support epidemic communication. Obviously, the exchange of data about nodes is an exchange of node set views. Views are created using a SLOSL statement which can be seen as a query on the node database. Once a view is defined, its data is also precisely defined and can be sent to any node by a simple call to the framework. The view definition even allows to infer an implicit message format description.

In this model, an exchange of views means the symmetric evaluation of a query in two different databases. The two results are then exchanged and used to update the other database. The PSP distinguishes three cases: sending, receiving and exchanging views. These cases determine in which database(s) the query is executed. Rhea etal. [18] describe them in a similar way and show cases where the symmetric exchange is necessary to assure eventual consistency between neighbours.

Current systems implicitly code the query into the overlay software. This resembles materialised views and stored procedures in databases. They are commonly used when performance is essential and changes are rare. In some cases, however, they are premature optimisations that come at the cost of lower adaptability. Their use makes software harder to configure and adapt at run-time.

Our model makes these queries explicit and shows that it is perfectly valid to ship the query (or a parametrisation of it) as part of the view exchange. This may be used for adapting the overlay topology to the capabilities of its members. For example, a high performance node may want to augment its view by sending broader queries. This can decrease the end-to-end latency it experiences by allowing single hops towards a larger number of nodes. Similarly, a less powerful node may decide to restrict its view by sending queries with higher selectivity.

One of the problems in gossip overlays is how to handle dead nodes. Most commonly, dead nodes are simply removed from the local view and will therefore not appear in gossip messages [18]. This forces nodes to redundantly find out about their failure. In the Node Views approach, the node database can simply keep data about dead nodes without adding any overhead to the software components. Node selection prevents dead nodes from appearing in local views. Remote nodes, however, can send queries for dead nodes as well as live nodes. Messages and database can both use timestamps to constrain the relevance of such data.

SLOSL and node set views make overlays based on gossip or other ways of exchanging node data trivial to write. At the same time, they decouple the data acquisition part of the maintenance algorithm and allow other (non-gossip) harvesters to take over if different characteristics are needed.

5.2 Distributed harvesters

Darlagiannis etal. recently presented Omicron [6], a design study for structured overlay networks. The idea is to split the algorithm that each overlay member traditionally executes into a number of simpler services that have different requirements. These services are then distributed over a cluster of nodes. This has two main advantages. Cluster nodes can provide a service that matches their capabilities instead of struggling to execute all services needed by the overlay. Secondly, it allows for replication within the cluster to increase the reliability of specific services.

However, this approach also comes at the cost of additional overhead inside each cluster. Omicron identifies four basic services in a DHT overlay: maintenance, indexing, routing and caching. Members of the same service interconnect between clusters. Therefore, each of the four services is provided by a different overlay while the cluster itself represents a fifth type of overlay. The maintainers have a special role. At the inter-cluster level, they exchange data about the nodes in their clusters. Inside their own cluster, they provide the other members with interconnection candidates from other clusters.

Node set views provide support for this exchange of views between cluster maintainers as well as between members of a cluster. The maintainer only needs to know the view definitions of each cluster member and can then send specific updates for their views. Similarly, when it communicates with the maintainers of other clusters, it can exchange its local view with them. In our model, the approach taken by Omicron mainly becomes a distributed, hierarchical database.

5.3 Latency harvesters

Among the node attributes, the latency between overlay members is the most important parameter for overlay adaptation. It can be used as a criteria whenever multiple candidates for choosing neighbours or forwarding messages exist. A latency harvester can measure or estimate this latency. While measuring usually involves pings or ACKs, there are a number of recent proposals that allow ping-free, resource efficient estimates for the physical latency between nodes. Some use virtual coordinates [4, 17], others query the routing infrastructure (BGP) of the physical network [16].

Though these are very different approaches, they all follow one common goal: determine the end-to-end latency between the local node and a remote node. The node database and its views provide an intermediate layer that decouple the harvesters from other components like routers. They hide the way how the latency information is found. SLOSL then provides direct support for converting the values of different sources when showing them in views. This allows the overlay to switch between different approaches based on accuracy and load requirements without affecting components that use this information.

Virtual coordinate systems also gain from overlay integration. Only a single harvester instance that works at the database level is needed to determine the coordinates of nodes in different overlays. This broadens the base of nodes and leads to more accurate models even for smaller overlays.

6 Implementation, current and future work

We are about to finish two different proof-of-concept implementations of this architecture. A first, light-weight prototype was written in Python, while our current work builds on the PostgreSQL database. The examples in section 4.2 were successfully tested on our current code base. We tried both a static environment and a randomised gossip scheme and verified the emerging topologies.

We continue implementing further overlay topologies in SLOSL. Since the implementations become so short, we have found that they make for comfortable unit tests in our system. However, only a broad usage of SLOSL in different overlay projects will allow us to see if further extensions to the language are needed or useful for special cases.

Our current work aims to provide 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.

For the future, we hope for diverse implementations of this architecture. The high abstraction level easily allows specialised versions for simulation and analysis, testing and debugging, and different deployment scenarios – without changes to the overlay code. Deployment environments will most likely use a rather lightweight or custom database. Simulators and debuggers can build their models on top of a single database which simplifies tracing and enables fast verification and visualisation of the system state. Recent proposals for scalable simulation environments [2] 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 recursive views and for merging view definitions when sending them over the wire (like in 5.2).

Another interesting topic to investigate in future work is (partial) source code generation from SLOSL statements. This may allow customised overlay implementations for efficient deployment.

7 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, the proposed model provides two main abstractions: node set views and harvesters. Their separation facilitates the development of generic components which enables pluggable development and integration of overlay systems.

The SLOSL language lifts the abstraction level for overlay designs 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 examples in section 5 show that the Node Views approach nicely incorporates a number of recent advances in overlay design. It relates them to well known achievements in the database area and allows to benefit from both worlds.

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. SLOSL 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 harvesters. As with any other high-level language, long-term optimisations of SLOSL based 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 harvesters, 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. K. Aberer. P-Grid: A Self-Organizing access structure for P2P information systems. In *Proceedings of the Sixth International Conference on Cooperative Information Systems (CoopIS 2001), Trento, Italy*, 2001.
2. H. Birck, O. Heckmann, A. Mauthe, and R. Steinmetz. The two-step overlay network simulation approach. In *Proceedings of SoftCOM, Split, Croatia*, pages 165–169, Oct. 2004.
3. Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making gnutella-like p2p systems scalable. In SIGCOMM2003 [21].
4. F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *Proceedings of the 2004 ACM SIGCOMM Conference*, Portland, Oregon, USA, Aug. 2004.
5. F. Dabek, B. Zhao, P. Druschel, and I. Stoica. Towards a common api for structured peer-to-peer overlays. In IPTPS2003 [9].
6. V. Darlagiannis, A. Mauthe, and R. Steinmetz. Overlay design mechanisms for heterogeneous, large scale, dynamic P2P systems. *Journal of Network and Systems Management, Special Issue on Distributed Management*, 12(3):371–395, Sept. 2004.
7. U. Dayal, A. Buchmann, and D. McCarthy. Rules are objects too: a knowledge model for an active, object-oriented database system. In *Proceedings of the 2nd International Workshop on Object-Oriented Database Systems*, volume 334 of *Lecture Notes in Computer Science*, pages 129–143. Springer, 1988.

8. D. Hughes, G. Coulson, and I. Warren. A framework for developing reflective and dynamic p2p networks (RaDP2P). In *Proceedings of the 4th IEEE Intl. Conference on Peer-to-Peer Computing (P2P2004)*, Zürich, Switzerland, Aug. 2004.
9. *The 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, USA, Feb. 2003.
10. M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In Middleware 2004 [15].
11. D. R. Karger and M. Ruhl. Diminished chord: A protocol for heterogeneous subgroup formation in peer-to-peer networks. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS04)*, San Diego, CA, USA, Feb. 2004.
12. J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gumadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*, Nov. 2000.
13. B. Li, J. Guo, and M. Wan. iOverlay: A lightweight middleware infrastructure for overlay application implementations. In Middleware 2004 [15].
14. D. Loguinov, A. Kumar, V. Rai, and S. Ganesh. Graph-theoretic analysis of structured peer-to-peer systems: Routing distances and fault resilience. In SIGCOMM2003 [21].
15. *The Intl. Middleware Conference (Middleware2004)*, Toronto, Canada, Oct. 2004.
16. A. Nakao, L. Peterson, and A. Bavier. A routing underlay for overlay networks. In SIGCOMM2003 [21].
17. M. Pias, J. Crowcroft, S. Wilbur, T. Harris, and S. Bhatti. Lighthouses for scalable distributed location. In IPTPS2003 [9].
18. S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference 2004*, Boston, MA, USA, June 2004.
19. A. Rodriguez, C. Killian, S. Bhat, D. Kostić, and A. Vahdat. MACEDON: Methodology for automatically creating, evaluating, and designing overlay networks. In *Proceedings of the USENIX/ACM Symposium on Networked Systems Design and Implemetation (NSDI2004)*, San Francisco, CA, USA, Mar. 2004.
20. A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale Peer-to-Peer systems. In *Proceedings of the International Middleware Conference (Middleware2001)*, volume 2218 of *LNCS*. Springer-Verlag, Berlin, Nov. 2001.
21. *The 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, Karlsruhe, Germany, Aug. 2003.
22. I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, San Diego, California, USA, Aug. 2001.
23. W. Vogels, R. van Renesse, and K. Birman. The power of epidemics: robust communication for large-scale distributed systems. *ACM SIGCOMM Computer Communication Review*, 33(1):131–135, 2003.
24. M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM symposium on operating systems principles*, pages 230–243, Banff, Alberta, Canada, 2001.
25. R. Zhang, Y. C. Hu, and P. Druschel. Optimizing routing in structured peer-to-peer overlay networks using routing table redundancy. In *Proceedings of the 9th International Workshop on Future Trends of Distributed Computing Systems (FTDCS'03)*, San Juan, Puerto Rico, May 2003.