SLOSL – A Modelling Language for Topologies and Routing in Overlay Networks

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

***Abstract*— Overlay networks are a fascinating field in the area of distributed systems. They combine challenges from self-organisation to extreme scalability and provide an interesting middleware layer for server-free Internet applications. The design aspects of their implementations, however, remained largely at the prototype level, which renders their integration and deployment in real applications hard.**

This paper describes an integrative, platform independent design approach for overlay networks that models topologies as data management systems. Local decisions about neighbours and message forwarding are expressed in an SQL-like language. The mapping to runnable implementations follows the Model Driven Architecture approach.

I. Introduction

Recent years have seen a large body of research in decentralised, self-maintaining overlay networks like P-Grid [1], Chord [2] or Pastry [3]. They form interesting building blocks for server-free, Internet-scale 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 [4], 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 prevents 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.

Throughout the large number of available systems, the most common approach for implementing overlay networks seems to be writing them from scratch, based on sockets or networking frameworks. Even dedicated overlay frameworks, like JXTA (`http://www.jxta.org`), have failed to reduce the design effort. They make programmers write systems with tens of thousands of code lines for an implementation that remains closely tied to the framework.

A more recent approach, called Macedon [5], introduced domain specific languages to this area. It focuses on networking protocols, which are commonly specified in Event-Driven State Machines. EDSMs are finite automata that interconnect processing states by event triggered transitions. Macedon provides an EDSM language that is targeted at message exchange and message handling in overlays. It hides trivial tasks like message serialisation and triggers programmer provided code blocks on incoming messages.

There are a number of problems that result from the approach taken by Macedon. It is generally tied to the C++ language, which forces application designers into a specific language environment. More importantly, it is designed entirely as an EDSM language, which enforces code modularisation based on I/O events rather than component semantics. As most EDSM frameworks, it even hides the graph that describes the handling and chaining of events inside of hand-written source code. This significantly complicates understanding the architecture of the resulting implementations.

EDSM models are helpful for the parts of the implementation that actually deal with event handling. For the entire system, however, the protocol design approach diverts the attention of the designer from topological properties of the overlay towards low-level tasks like message handling. It becomes hard to understand and compare the actual features of overlay systems without external, explicit descriptions of the topology and the local decisions that were intended by the specific implementation.

Another problem with protocol design is the low reusability of source code. Code blocks are closely tied to specific messages and not portable between different overlay types. This prevents integration and multiplies the resource usage of a single computer participating in multiple overlays, as different protocols cannot easily share state or maintenance procedures through components.

The sum of these problems calls for new abstractions in the design of overlay systems. In 2005, two research groups have independently discovered a relation between database technology and overlay design. This lead to the development of the domain specific languages SLOSL [6] and Overlog [7]. While Overlog is mainly concerned with recursive protocol design in these systems, SLOSL integrates with a set of XML languages named OverML (Overlay Modelling Language), that form a Model Driven Architecture for the integrative design of overlay software.

This paper builds upon previous publications by the same

author [8], [6]. It models overlay topologies and the local decisions of their nodes as data management problems. Combined with models for protocols and messages, this integrates into a high-level approach to platform-independent overlay modelling.

The remainder of this paper is structured as follows. Section II presents the general requirements on overlay design approaches and frameworks. Sections III and IV then present abstractions that facilitate a higher level design of overlay topologies and decoupled components. SLOSL and its application to topology implementation and routing is presented in sections V and VI. Sections VII and VIII describe the proposed ways for generating optimised source code from the semantically rich models and the current status of the implementation.

II. Requirements on Models for Overlay Networks

The currently wide-spread practice leaves overlay designers and overlay application developers alone with the problems described in the introduction. These problems, however, allow us to extract a number of requirements for more promising overlay modelling approaches.

A. Domain-specific, high-level, platform-independent models

First of all, such an approach must provide simplified, domain-specific models for overlay development. They must allow the overlay designer to move away from reinventing the wheel in low-level implementations and to focus on the main features of the specific overlay. These are defined by the overlay topology and the algorithms for local decisions in routing and maintenance.

An important design factor is programming language independence, including the choice of a suitable execution environment. If the designer can start with abstract specifications of the overlay, it becomes possible to implement the final system in different environments without major redesigns. This is vital for distributed systems that are expected to run on diverse architectures, like large servers, standard PCs and mobile devices.

Platform independence is also vital for the development process. The later the decision about the applied environment and language can be taken, the easier it becomes to base the decision on those performance and design aspects that ultimately prove to be most relevant.

It further allows using different environments for different steps of the development process. Environments for rapid prototyping may look very different from those enabling high-performance execution. High-level, language independent models and the resulting high-level design are crucial to support this choice of environments.

B. Integration with custom software components

To integrate the high-level design with language specific code, the abstraction requires a component model that makes reactive components reusable.

1) Reusable components: Most importantly, software components should be reusable in different implementations to reduce the necessary amount of hand written source code. Code reusability requires well-defined, generic interfaces between the high-level models and specialised components. If components are written solely against the abstract models, they can become part of a middleware tool set. This provides a common base of pluggable components and further reduces the effort necessary to design new overlays.

This obviously regards also the lower networking levels. Serialising messages, sending and receiving them, is a basic feature of any networking middleware. While a middleware may support a diversity of serialisation formats (XML, XDR, IIOP, custom binary, ...) as well as different point-to-point networking protocols (such as TCP/IP, RTP or VPNs), it should hide their deployment behind simple interfaces and generic use patterns to make their usage a matter of selection and configuration rather than programming.

2) Reactive components: As in any networking software, the components of an overlay system are naturally reactive. They respond to events such as incoming or locally generated messages, time-outs and changes to the local model. The model must therefore support the management of such events and the coordination between different components.

This property is often implemented by means of Event-Driven State Machines. Their event model is simple: messages are locally received or produced and time-outs are triggered. The system then dispatches these events to states according to the available transitions. Some systems additionally support processing chains that hand generated data objects from one processing state to the next. The output of such an object from a state becomes an additional event for the system. This reduces the complexity of each state and moves more of the control logic into the event model.

The expressiveness of the event model directly impacts the complexity of its components. Domain specific, expressive events become a key factor for the decoupling of generic, reusable components. They help in lifting the implementation dependency on specific environments and frameworks.

C. Integrative management of state and node data

A large part of an overlay implementation deals with the relation of the local node to remote nodes. Overlay models must provide support for managing this relation. This comprises handling the connections to neighbours in the topology, the decision of adding them to the local view, removing them or keeping fall-back candidates for failures. Nodes often have to store further state about their neighbours, such as running time-outs, measured latencies or active subscriptions. This form of state keeping and decision making determines the main characteristics of an overlay implementation.

To support taking local decisions about other nodes, the model cannot restrict their representation to connections. It must support the selection of neighbours and communication partners based on various criteria, even if there is not currently an open connection to them. It needs an abstraction for the

local decisions that current systems implement by hand in platform specific source code. The cross-cutting ubiquity of state management throughout current implementations is an indicator for necessary support in the modelling process.

Moving the node data management into models allows us to integrate the state handling in different overlays and applications that run on a node. Supporting different topologies obviously makes sense for debugging, testing and benchmarking 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. Supporting a choice of topologies allows an application to provide optimised solutions for different cases.

Such an integration 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[1]. Integrative approaches become crucial here. They allow us to apply automated inter-overlay optimisation techniques that reduce the accumulated maintenance overhead.

III. The Topology Perspective on Overlay Software

The main problem with current overlay frameworks is their lack of support for integration and reusability. They focus on the design of protocols and reactive components, but fail to provide abstraction layers that simplify the reuse of components in different systems. They also fail to support the management of nodes and state. This leaves the developer with the sole responsibility for the design and implementation of mechanisms for overlay integration and efficient state sharing. These mechanisms become source code level tasks that do not match well with the design of an event-driven protocol architecture. The reduced readability of their non-componentized, cross-cutting implementation hinders portability and reuse of code.

The modelling approach described in this paper moves the abstraction level for overlay systems towards the design of local decisions about their topology. To achieve this, it has to deal with four major functional levels in overlay software: topology rules, maintenance, routing and adaptation.

A. Local topology rules

Local topology rules play the most important 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 form the part that actually implements this algorithm by accepting neighbour candidates or objecting to them.

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 select the others only into overlays that they support.

Where selection is the black-and-white decision of seeing a node or not, **node 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 (or equivalence classes) to distinguish different types of *equivalent* nodes. Common 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 inferring the global guarantees provided by a topology from the local rules 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.

B. Topology maintenance

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. The detection of a situation that "breaks the rules" is obviously an event that must be extracted from the topology rules. Support for this functionality is very limited among current overlay frameworks, despite its obvious importance for the intended self-maintenance in these systems.

C. Topology adaptation

Topology adaptation is the ability of a given overlay topology to adapt to specific requirements. As opposed to the error correction provided by topology maintenance, adaptation handles the freedom of choice allowed by the topology rules. The rules therefore draw the line between maintenance and adaptation. Topology adaptation usually defines some kind of metric for choosing new edges from a valid set of candidates.

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,

[1]According to [9], PlanetLab applications (`http://planet-lab.org/`), as an extreme example, generated a total of up to 1GB of ping traffic per day in 2003.

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.

D. Topological Routing

Routing is the local decision which of the neighbours a message should be forwarded to. The aim is to deliver it to the final destination (single-hop) or to bring it at least one step closer (multi-hop). The routing algorithm exploits topology rules and adaptation to determine the best next recipients. The complete process of taking a forwarding decision is typically executed in multiple inter-dependent steps. A part of this is the decision if the local node is the destination of an incoming message. The message is then either locally delivered to a message receiver component or further treated to determine the responsible neighbour.

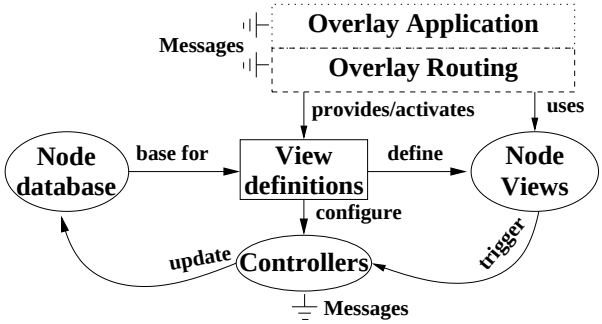
IV. Node Views, the System Model

We propose to model overlay software as data management systems by applying the well-known Model-View-Controller pattern [10]. 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. The SLOSL view definition language allows their specification in a platform-independent way.

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 local node. 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.

1: Components of the System Model



Another very important part of the architecture is an expressive *event system* for view events and messages. A notification about changes in views is triggered 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.

Message handling components are still part of the overlay specific implementation, but they now respond to expressive events and use node views for their decisions. The OverML model defines messages as hierarchical data structures and expresses subscriptions in a subset of the XPath language that is evaluated at compile time. It allows the developer to subscribe components to specific headers and data fields instead of monolithic messages. This reduces their dependency on specific overlay protocols and helps in writing generic components.

Database and views decouple the message handlers from the maintenance components and simplify the design of both considerably. Even more so, as a Node Views based architecture can provide powerful operations like selecting topologies and adaptation strategies with a single view selection command. The abstract view definition becomes the central point of control for the characteristics of the overlay.

V. SLOSL, the View Definition Language

OverML [8] is an integrated set of domain specific XML specification languages for node attribute schemas, messages, view definitions and event graphs. The view definitions for topology rules and adaptation are expressed in SLOSL, the SQL-Like Overlay Specification Language. The following is a simple example, an advanced implementation of the Chord graph [2].

```
CREATE VIEW chord_fingertable
AS SELECT node.id, node.ring_dist, bdist=node.ring_dist−2^i
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)
RANKED lowest(backups+i, node.msec_latency / node.ring_dist)
```

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. This is left entirely to the controllers. SLOSL 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* is a global constant for the life-time

of an overlay, while *backups* allows adding neighbour 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. In the example, a Chord node sorts remote nodes into ring intervals of increasing size and distance that designate equivalent neighbours: itself $\rightarrow 2^1 \rightarrow 2^2 \rightarrow 2^3 \ldots$
  The example also 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 identifier space. Other overlays may require more complex expressions or user defined functions in the ranking expression. The original Chord implementation selects exactly one node based only on its ring distance.

Other topology specifications are similarly simple. For example, Scribe [11] is a multicast scheme that was initially implemented for the Pastry overlay [3]. It is, however, applicable to many other so-called key-based routing overlays, including Chord. Scribe essentially exploits the key mapping provided by the overlay network to determine a rendezvous node for each multicast group. Subscriptions are sent towards the rendezvous that serves as the root of a multicast tree. They build up forwarding state along the way and publications follow them backwards. Subscriptions and publications simply follow the Chord routing policy towards the rendezvous node. Once a match was found however, the publications must be forwarded according to rules that are specific to Scribe.

SLOSL can model subscription state as a set of group identifiers that it keeps for each neighbour. It then builds up one view for each group topology that the local node participates in. The view selects only those Chord fingers that are subscribed for messages of this group. Publications are simply broadcasted to all nodes in the view to forward them along the multicast tree.

```
CREATE VIEW scribe_subscribed_fingers
AS SELECT node.id
FROM chord_fingertable
WITH group
WHERE group in node.subscribed_groups
```

VI. SLOSL ROUTING

The SLOSL view definitions allow for a simple extension towards a complete routing strategy. Most overlay systems base their routing decisions on more than one view. They often use different views for semantically close and far neighbours, such as the successor table and the finger table in Chord.

When a router looks for the next hop towards a given destination, it simply tests views in a sensible order to see if it knows *equivalent* nodes. This process exploits both the node categorisation and ranking features of SLOSL as follows.

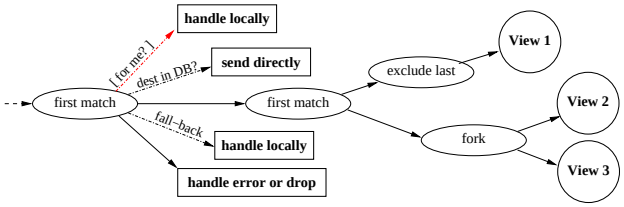
1) The SLOSL clauses HAVING–FOREACH are evaluated against the (partially) known attributes of the destination node to find the corresponding bucket. If that fails, the complete process fails for this view.
2) On success, the RANKED clause determines the best target for the bucket.

Note that the WHERE clause is not required to be evaluated. In fact, there may not even be enough local information about the destination node to do node selection. The node may only be known from the destination field of the currently forwarded message and thus may not appear in the view or even in the database. To find the next hop that corresponds to the destination, however, it is sufficient to find the category that it *would* be in if it was known. The category determines the local bucket of equivalent nodes.

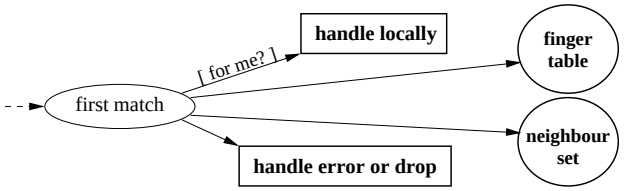
Furthermore, there is nothing that prevents the router from first querying the local database for a node that matches the destination exactly. If it is found, its physical address can be used to send the message directly. This can reduce the hop-count and therefore the latency towards certain nodes that are not regular neighbours of the local node in the overlay topology.

So far, we only regarded unicast, i.e. forwarding the message to exactly one neighbour. Some protocols will require broadcast or multicast. In SLOSL overlays, the unicast, multicast and broadcast schemes turn out to be identical, as SLOSL already selects a set of nodes. Multicasting to a subset of the neighbours is the same as broadcasting to a view that selects them. Broadcasting to a view that selects a single neighbour from the only corresponding bucket is the same as unicasting to that neighbour. SLOSL routing therefore unifies all three communication patterns into a general broadcast to all targets that result from the evaluation of a view. Note that the abstraction level does not prevent frameworks from using network level broadcast or multicast as implementations.

Figure 2 shows a general decision graph for SLOSL routing. A directed acyclic graph of this kind is all that is required to



2: General graph for SLOSL routing decisions



3: Chord routing, modelled using SLOSL views

fully describe routing components. Within the graph, messages arrive from the left and are routed as follows.

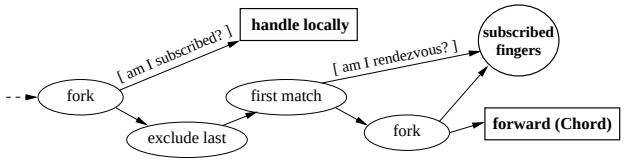
The *first match* edges simply traverse their ordered children depth-first. If a target is found, the routing process terminates and the message is forwarded to the target. This corresponds to an ordered XOR evaluation.

The first child at the top-left has a *predicate* associated with it that tests if the message is to be accepted locally. If this predicate succeeds, the previous first match edge succeeds on this transition, independent of the further evaluation down the tree. Predicates are external to the graph and must be provided by the overlay implementation. Depending on the topology, however, the same decision may be available as a generic fall-back if the attempts to route through the views fail. This is shown at the bottom of the figure.

The *fork* edge splits up the routing process and continues it in all of its child branches independently, thus yielding an OR evaluation. In the graph shown in the figure this means that the message is broadcasted to both the views 2 and 3 if no matching target is found during the evaluation of view 1. If neither of the three views yields a target, the routing procedure backtracks.

The *exclude last* edge is used to tag a sub-tree. It prevents the node that last forwarded the message from appearing in any of the target sets that are found further down the decision tree. This is commonly used to avoid duplicates in broadcast or multicast forwarding, since the last hop has already received the message. The feature obviously requires the previous hop to be identifiable, which is the case in most physical networks. If not available from the physical layer, identifiers can always be explicitly provided within higher-level message headers. When this edge is used in combination with a first match edge, the last hop is always discarded before testing the target set for success.

Typically, specific routing strategies have simpler graphs, as they do not exploit all possibilities. The Chord routing algorithm, for example, is specified as in figure 3. More



4: Multicast routing over Chord, modelled using SLOSL views

complex routing strategies, like the routing of multicast messages over a Chord graph (similar to the approach taken in Scribe [11]), become easily understandable when expressed using SLOSL routing decision graphs. Figure 4 shows the complete implementation for the forwarding of publications. The rendezvous predicate is the same as the local delivery decision in Chord. Mapping a routing graph to efficient source code is a straight forward transformation.

VII. SOURCE CODE GENERATION FROM SLOSL

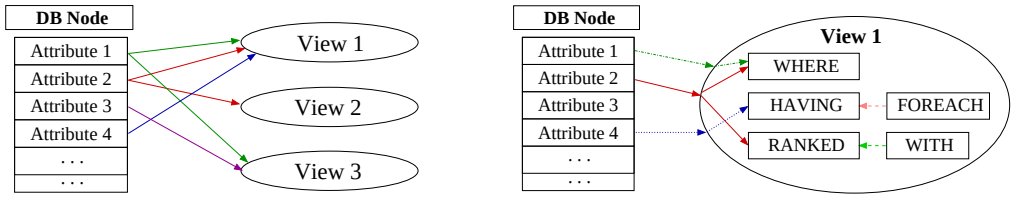
The execution of SLOSL statements is naturally event-driven. When controllers update attributes and add or discard nodes, the architecture must determine which views are affected and reevaluate them. The naive way of independently evaluating all of them can quickly become exhaustive, as it depends on the number of locally known nodes, the deployed views and their bucket structure.

SLOSL, however, is a declarative language that provides high-level semantics. Its evaluation can be implemented in different ways, depending on the requirements. The redundancies and particularities between the deployed view specifications can be exploited to further integrate their evaluation and to minimise the impact of changes in the database on event-triggered components.

SLOSL provides two main features that help in integrating views. First of all, it provides separate clauses for semantically different parts of a specification. This allows to extract overlapping expressions that follow the same semantics and to pre-calculate them for use in different views. Secondly, it makes it easy to determine the dependencies between node attributes and single clauses of views. They follow from the attributes and bucket variables used in the expressions of SLOSL statements and allow to select a minimum set of views (and clauses) for evaluation when attributes change. Note also that the schema part of OverML can declare these attributes as identifiers or otherwise static data, which allows to drop their update dependencies from the implementation.

As a result, the run-time performance of the locally running software in a SLOSL implemented system is directly linked to the preparation of an efficient multi-statement query execution plan at compile time. A SLOSL optimiser will therefore start by building a dependency graph like in figure 5a. It states which attributes each view depends on. Only the dependent views have to be re-evaluated on updates.

The next step is to open up the view declarations and to split them into their different clauses. We can then build an extended dependency graph for the clauses of each view, as in figure 5b. As in the previous examples, different clauses



(a) Attribute dependencies

(b) Dependencies between clauses

5: Example dependency graphs for attributes and clauses

do not necessarily depend on the same attributes. If an attribute changes, it can therefore not affect independent clauses. However, the clauses may also provide their own inter-dependencies. While the WHERE clause can only depend on the parents of the view, the RANKED expression may depend on bucket variables, i.e. the FOREACH and HAVING clauses. The dependency graph is therefore needed to determine an efficient execution plan.

For example, a change to attribute 1 would enforce a reevaluation of the WHERE clause for the respective node. If that succeeds, evaluating the HAVING–FOREACH clauses will tell us into which buckets the node belongs so that we can re-run the ranking only for these. Storing the information if a node is already part of a view or caching the previous result of the boolean WHERE expression can even prevent the further view evaluation entirely if the result of the WHERE clause stays unchanged. Similarly, a change to the second attribute enforces the evaluation of the RANKED expression if (and only if) the WHERE clause succeeds.

Independent expressions (even partial expressions) become candidates for pre-evaluation. Their results can be stored in the database as so-called *dependent attributes*. For example, the presented Chord statement allows for pre-evaluation of the ranking expression and the WHERE clause.

In SLOSL implemented overlay software, the detailedness of the dependency graph and the application of possible optimisations are the two main factors for the efficient execution of local decisions. The best strategies can be determined at compile time or deployment time, so that the overhead of finding them does not effect the run time performance. SLOSL optimisers can generate optimal evaluation plans for each specific update event in the model and source code generators can build efficient execution paths from each of them.

Different frameworks can take their place within the range of optimisations between additional storage for pre-calculated results, hashing, indexing and linear scans. They can trade the compile time overhead against the complexity and performance of the resulting implementation. It is an important achievement of the SLOSL language to move these trade-offs into configurations and parametrisations of frameworks and translators, away from source code implementations of overlay systems. Profiling a specific overlay implementation can yield new platform independent optimisations that all other SLOSL implemented overlays can immediately benefit from.

VIII. Implementation, current and future work

A major part of previous work went into the implementation of a graphical overlay design tool, named the SLOSL Overlay Workbench. The screen-shots in figure 6 show the interfaces for the different languages of OverML [8]. They support the semantically integrated specification of node attributes, messages, SLOSL statements and event flow graphs. The workbench exports platform independent XML descriptions of the designed overlay, which can be passed into translators and source code generators. The development of the workbench is now continued as an Open Source project at the Berlios developer site[2].

Current work is mainly guided towards the generation of efficient platform specific models and source code from OverML specifications, as well as run-time configurable component systems. This is joint work with the Reflective Middleware Group at the University of Lancaster.

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.

IX. Conclusion

This paper presented a platform-independent modelling technique for local decisions about topologies and routing in overlay networks. The specification of topology rules and adaptation strategies is expressed in a domain specific language called SLOSL. A simple extension through decision graphs allow it to support complex overlay routing decisions.

The Node Views model enforces a modularisation of state within the implementation. It decouples maintenance components from routers and shares the state between different components and overlays. It further replaces the monolithic message handlers in current implementations by control components that respond to expressive, well defined events based on structured messages and changes in the local view. This reduces their dependency on specific overlays and allows for more generic, pluggable components in frameworks that can be shared between different overlay implementations.

The general approach of modelling local decisions in distributed systems as high-level data management problems

[2] http://developer.berlios.de/projects/slow/

opens up a very interesting field of research. The proposed Model Driven Architecture combines challenges from the areas of distributed systems, databases, modelling and software engineering.

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, CA, USA (2001)

[3] Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale Peer-to-Peer systems. In: Proc. of the Int. Middleware Conference (Middleware2001). (2001)

[4] Dabek, F., Zhao, B., Druschel, P., Stoica, I.: Towards a common API for structured peer-to-peer overlays. In: Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03), Berkeley, CA, USA (2003)

[5] 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)

[6] ANONYMISED: Overlay Networks – Implementation by Specification. In: Proc. of the Int. Middleware Conference (Middleware2005), Grenoble, France (2005)

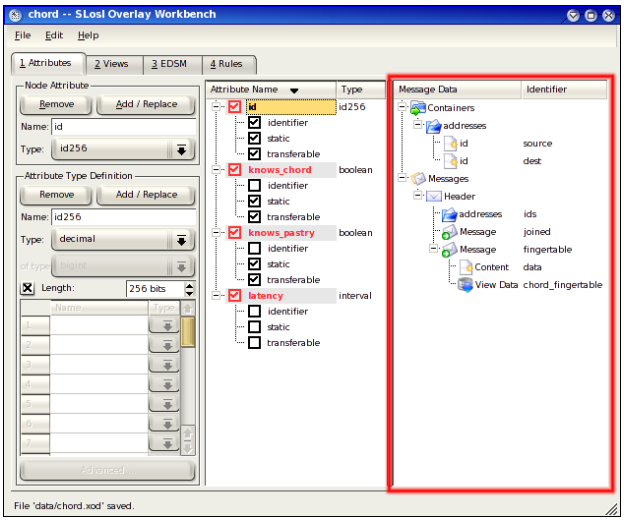
[7] Loo, B.T., Hellerstein, J.M., Stoica, I., Ramakrishnan, R.: Declarative routing: Extensible routing with declarative queries. In: Proc. of the 2005 ACM SIGCOMM Conference, Philadelphia, PA, USA (2005)

[8] ANONYMISED: 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)

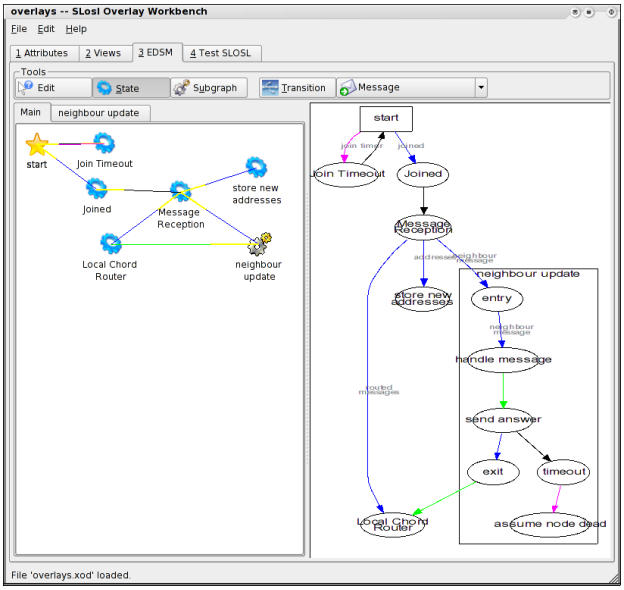
[9] Nakao, A., Peterson, L., Bavier, A.: A routing underlay for overlay networks. In: Proc. of the 2003 ACM SIGCOMM Conference, Karlsruhe, Germany (2003)

[10] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons (1996)

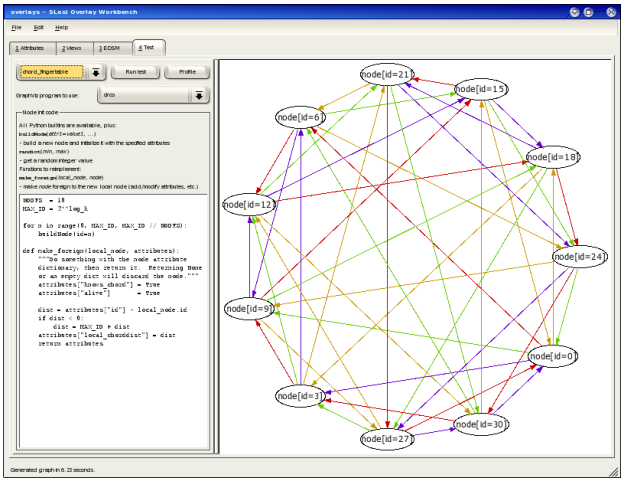
[11] Rowstron, A., Kermarrec, A.M., Castro, M., Druschel, P.: SCRIBE: The design of a large-scale event notification infrastructure. In Crowcroft, J., Hofmann, M., eds.: Proc. of the 3rd Int. Workshop on Networked Group Communications (NGC'01), London, UK (2001)



Attributes and Messages



Event flows



SLosl visualisation

6: The SLosl Overlay Workbench