A Model Driven Architecture for Adaptable Overlay Networks

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

vorgelegt von

Stefan Behnel
aus Helmstedt

Referenten:
Prof. Alejandro P. Buchmann, Ph. D.
Prof. Geoff Coulson, Ph. D.

Datum der Einreichung: 12. Dezember 2006
Datum der mündlichen Prüfung: 5. Februar 2007

Darmstadt 2007, D17

Für Gabriele -
Danke. Für Alles.

Preface

Acknowlegements

Credit is due to many people who inspired, encouraged and supported me during my research and in the process of writing this thesis. The first to mention at this point is my advisor Professor Alejandro P. Buchmann. He opened the door for me into an interesting and inspiring research environment and supported my work with comments, advice, patience and a remarkable degree of freedom. This work would not have started nor would it have been possible to complete without him.

The very next person to thank is Ludger Fiege for numerous discussions and his priceless support as a friend, colleague, critic and (co-)author of papers.

An important part of this thesis originated from a collaboration with Geoff Coulson and Paul Grace of the Distributed Systems Group at Lancaster University. I thank both of them for a warm welcome, for fruitful discussions and interesting insights that helped in improving and rounding up the work presented here. Their involvement opened interesting new paths for future research.

At a very personal level, I want to thank Wesley Terpstra and Dimka Karastojanova for joining me on the long way since we started. Their friendship and collaboration was invaluable to me.

My work was generously supported by a national grant of the Deutsche Forschungsgesellschaft (DFG) as part of the graduate college 749 "System Integration for Ubiquitous Computing". The graduate college and its participants provided a friendly, inspiring environment for my research.

Finally, in the long tradition of naming the most valuable people last, I thank Gabriele, my parents and my sisters for their enduring support and encouragement during the creation of this thesis and ever before.

Scientific Curriculum Vitae

1997 - 1999	Computer Science studies at the Braunschweig University of Technology, Germany
1999 - 2000	Academic year of Computer Science in Limerick, Ireland
2000 - 2001	Specialised studies on Networks and Distributed Systems in Lille, France. Final degree: *Diplôme d'Études Supérieures Spécialisées* in Distributed Systems.
2001 - 2002	Professional software development for the leading european online fright exchange service Téléroute in Seclin, France
2002 - 2007	Doctorate as part of a DFG financed Graduate College in the *Databases and Distributed Systems Group* at the Darmstadt University of Technology, Germany. Final degree: *Dr. Ing.* in Computer Science.
since 2006	Professional software development und architecture in the financial sector.

Wissenschaftlicher Lebenslauf

1997 - 1999	Studium der Informatik an der Technischen Universität Carolo-Wilhelmina in Braunschweig, Deutschland
1999 - 2000	Studium der Informatik in Limerick, Irland
2000 - 2001	Spezialisierungsstudium Netzwerke und Verteilte Systeme in Lille, Frankreich. Abschluss als Jahrgangszweiter mit dem *Diplôme d'Études Supérieures Spécialisées* im Bereich Verteilte Systeme.
2001 - 2002	Professionelle Software-Entwicklung für die führende europäische Online-Frachtbörse Téléroute in Seclin, Frankreich
2002 - 2007	Vorbereitung der Promotion in der Fachgruppe *Datenbanken und Verteilte Systeme* an der Technischen Universität Darmstadt, Deutschland, mit Bestenförderung der Deutschen Forschungsgesellschaft im Rahmen eines Graduiertenkollegs. Abschluss mit dem Titel *Dr. Ing.* im Bereich Informatik.
seit 2006	Professionelle Software-Entwicklung und Architektur im Finanzsektor

Publications

[BB05a] Stefan Behnel and Alejandro Buchmann. 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, August 2005.

[BB05b] Stefan Behnel and Alejandro Buchmann. Overlay Networks – Implementation by Specification. In *Proc. of the Int. Middleware Conference (Middleware2005)*, Grenoble, France, November 2005.

[BBG+06] Stefan Behnel, Alejandro Buchmann, Paul Grace, Barry Porter, and Geoff Coulson. A specification-to-deployment architecture for overlay networks. In *Proc. of the Int. Symposium on Distributed Objects and Applications (DOA)*, Montpellier, France, October 2006.

[Beh05a] Stefan Behnel. MathDOM – A Content MathML Implementation for the Python Programming Language. `http://mathdom.sourceforge.net/`, 2005.

[Beh05b] Stefan Behnel. The SLOSL Overlay Workbench for Visual Overlay Design. `http://developer.berlios.de/projects/slow/`, 2005.

[Beh05c] Stefan Behnel. Tin Topologies – Designing Overlay Networks in Databases. In *Proc. of the Int. Middleware Conference (Middleware2005)*, Grenoble, France, November 2005. Demonstration and Poster Presentation.

[Beh05d] Stefan Behnel. Topologien aus der Dose – Ein Datenbank-Ansatz zum Overlay-Design. In Paul Müller, Reinhard Gotzhein, and Jens B. Schmitt, editors, *KiVS Kurzbeiträge und Workshop*, volume 61, Kaiserslautern, Germany, March 2005. Poster.

[Beh07] Stefan Behnel. SLOSL - a modelling language for topologies and routing in overlay networks. In *Proc. of the 1st Int. Workshop on Modeling, Simulation and Optimization of Peer-to-peer environments (MSOP2P)*, Naples, Italy, February 2007.

[BFM06] Stefan Behnel, Ludger Fiege, and Gero Mühl. On quality-of-service and publish-subscribe. In *Proc. of the 5th Int. Workshop on Distributed Event-based Systems (DEBS'06)*, Lisbon, Portugal, July 2006.

[TB04] Wesley W. Terpstra and Stefan Behnel. Bit Zipper Rendezvous - Optimal data placement for general P2P queries. In *International Dagstuhl Seminar 04111 on Peer-to-Peer-Systems and Applications*, Schloss Dagstuhl, Germany, March 2004.

[TBF$^+$03] Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Andreas Zeidler, and Alejandro Buchmann. A Peer-to-Peer Approach to Content-Based Publish/Subscribe. In *Proc. of the 2nd Int. Workshop on Distributed Event-based Systems (DEBS'03)*, San Diego, CA, USA, June 2003.

[TBF$^+$04] Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Jussi Kangasharju, and Alejandro Buchmann. Bit Zipper Rendezvous - Optimal Data Placement for General P2P Queries. In *Proc. of the 1st Int. Workshop on Peer-to-peer Computing and Databases*, Heraklion, Crete, March 2004.

Abstract

Recent years have witnessed a remarkable spread of interest in decentralised infrastructures for Internet services. Peer-to-Peer systems and overlay networks have given rise to a major paradigm shift and to novel challenges in distributed systems research. Numerous projects from several research communities and commercial organisations have started building and deploying their own systems. Adoption, however, has been restricted to sparsely selected areas, dominated by few applications.

Among the technical reasons for the limited availability of deployable systems is the complexity of their design and the incompatibility of systems and frameworks. Applications are tightly coupled to a specific overlay implementation and the framework it uses. This leaves developers with two choices: implementing their application based on the fixed combination of overlay, networking framework and programming language, or reimplementing the overlay based on the desired application framework. Both approaches have their obvious draw-backs.

Implementing an overlay, even as a reimplementation, is a task that exhibits considerable challenges. Protocols have to be adapted, completed and partially reverse engineered from the original system to implement them correctly. Message serialisations and interfaces have to be rewritten for the new framework. State maintenance and event handling are programmed in very different ways in different environments, which typically requires their redesign. Reimplementing an overlay for a new environment is therefore not necessarily less work than designing a new one.

As for the alternative, being tied to a specific environment prevents the application designer from freely choosing the best suited framework for the specific application. Deploying different overlays in one application is only possible if they were written for the same framework, and even then, running multiple non-integrated overlays at the same time can become prohibitively resource extensive. Testing with different overlay topologies and adapting to different deployment environments is similarly hard in this scenario. The current techniques used for overlay implementation turn out to become limiting factors in the design of overlay applications.

The approach taken by this thesis tackles these issues at design time, at a point long before integration problems arise. It presents a modelling framework that allows to express overlay specific semantics in a platform-independent, domain specific language called the Overlay Modelling Language, OverML. A Model Driven Architecture maps these abstract overlay specifications to concrete, framework specific implementations.

Applications based on OverML benefit from state sharing between different overlays and from short topology implementations in the SQL-Like Overlay Specification Language SLOSL. Their conciseness allows an easy adaptation of

SLOSL statements to specific quality-of-service requirements. The architecture provides a clean separation between the generated implementation and hand-written components through generic, event-driven interfaces.

To evaluate the approach, a series of topologies is exemplarily specified in OverML and/or SLOSL. A complete walk-through from the specification to the deployment of an OverML implemented overlay is additionally presented.

The major contribution of this thesis is the first complete design methodology for the integrative, high-level development of portable, adaptable overlay networks.

Zusammenfassung

Seit einigen Jahren zeigt sich ein stark zunehmendes Interesse an dezentralen Infrastrukturen für internetweite Dienste. Ausgelöst durch den massiven Erfolg von Peer-to-Peer Systemen und Overlay-Netzwerken hat vor allem in der Forschung ein Umdenken eingesetzt. Wie sehr diese Netzwerke als ernsthafte Alternative zu Client-Server Systemen wahrgenommen werden, belegt die große Menge an Systemen, die Forschungsgruppen und Firmen weltweit entworfen haben. Bemerkenswert ist jedoch auch, wie wenige dieser Systeme den Weg in reale Anwendungen gefunden haben. Abgesehen von Programmen zur Distribution großer Datenmengen (E-Donkey oder BitTorrent) gibt es nur wenige Systeme mit nennenswerter Verbreitung.

Für diesen Mangel an Anwendungen gibt es durchaus auch technische Gründe. Zu nennen sind hier vor allem die Komplexität dieser verteilten Systeme und die Inkompatibilität der existierenden Implementierungen. Anwendungen sind dabei sehr stark an die zu Grunde liegende Overlay-Software gekoppelt. Ihren Entwicklern bieten sich heute nur zwei Möglichkeiten. Sie können ihre Anwendungen für existierende Overlay-Implementierungen maßschneidern und sich auf die damit einhergehende Kombination aus Software-Umgebung und Programmiersprache festlegen. Oder sie wählen eine Sprache und Umgebung, die zu ihrer Anwendung passt und sind dann gezwungen, das benötigte Overlay in dieser Umgebung neu zu implementieren. Aus Entwicklersicht kann keiner dieser Ansätze zufriedenstellend sein.

Ein bestehendes Overlay zu portieren ist nur selten weniger Aufwand als es neu zu schreiben. Das liegt einerseits an der zumeist unzureichenden Spezifikation, die eine Anpassung der Protokolle nach sich zieht. Teilweise muss dabei sogar auf Reverse-Engineering der ursprünglichen Implementierung zurückgegriffen werden. Zudem ist die Implementierung von Nachrichtenverwaltung und -serialisierung, Komponentenschnittstellen und Zustandsverwaltung sehr stark von der verwendeten Software-Umgebung abhängig, so dass diese Teile neu entwickelt werden müssen. Der hierfür benötigte Aufwand steht in keinem Verhältnis zu der Erleichterung, die der Einsatz von Overlays für Anwendungsentwickler bringen soll.

Die Alternative ist die Übertragung der vom Overlay genutzten Software-Umgebung auf die Anwendung. Dies macht jedoch nur dann Sinn, wenn Umgebung und Programmiersprache auch angemessene Unterstützung für die Entwicklung der spezifischen Anwendung bieten. Selbst dann besteht jedoch weiterhin das Problem der Inkompatibilität zwischen Overlay-Implementierungen, so dass ein späterer Wechsel oder auch nur Tests mit anderen Overlays mit großem Aufwand verbunden sind. Eine gemeinsame Nutzung mehrerer Overlays zur Laufzeit verbietet sich schon durch die fehlende Integration der Systeme, da sich ihr Resourcenverbrauch zumeist summiert.

Die vorliegende Arbeit verfolgt einen umfassenderen Ansatz, der die beschriebenen Integrationsprobleme von vornherein umgeht und sowohl die Entwicklung als auch den Umgang mit Overlay-Software vereinfacht. Er erlaubt erstmals die plattformunabhängige, abstrakte Modellierung von Overlay-Implementierungen. Die Beschreibung erfolgt in einer eigens zu diesem Zweck entwickelten domänenspezifischen Sprache namens OverML, der Overlay Modelling Language. Entsprechend dem Model-Driven-Architecture Ansatz erfolgt anschließend eine maschinelle Übersetzung der abstrakten Modelle in konkrete Implementierungen für spezifische Software-Umgebungen.

Auch für Anwendungen bietet OverML Vorteile. So werden die Topologieeigenschaften von Overlay-Implementierungen durch die Verwendung der kompakten, SQL-ähnlichen Sprache SLOSL übersichtlich und konfigurierbar und lassen sich sehr leicht an spezielle Anforderungen anpassen. Zur Integration mehrerer Overlays wird ein gemeinsamer Zustandsspeicher verwendet, der eine Duplizierung des Verwaltungsaufwandes zu vermeiden hilft. Die Komponentenarchitektur wird durch generische, datengesteuerte Schnittstellen entkoppelt, was zu einer sauberen Trennung von generiertem und handgeschriebenem Code führt.

Zur Evaluierung werden verschiedene Overlay-Topologien spezifiziert. Zudem erfolgt eine Beschreibung des vollständigen Entwicklungsablaufes von der Spezifikation einer Overlay-Implementierung bis hin zu ihrem Einsatz in wechselnden Anwendungsumgebungen.

Somit stellt diese Arbeit eine neuartige Methodik zur Verfügung, die erstmals die integrative, abstrakte Entwicklung plattformunabhängiger, leicht adaptierbarer Overlay-Implementierungen erlaubt.

Contents

List of Figures

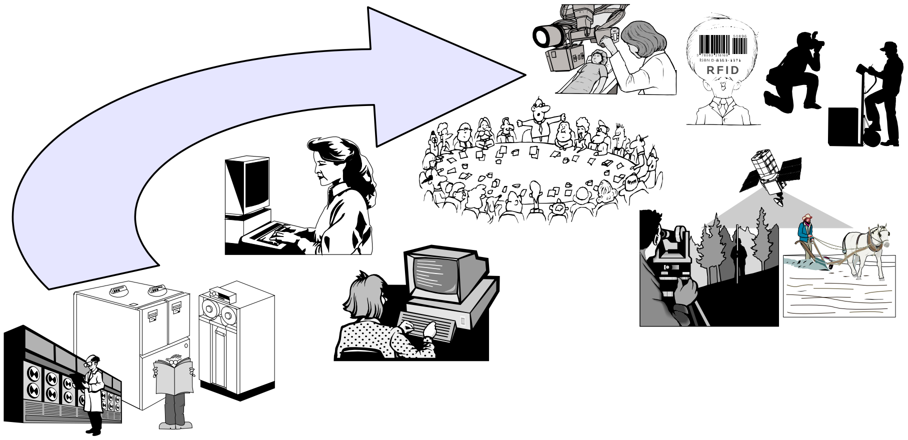
List of Listings

Chapter 1

Introduction

Contents

Since the early 1990's, we can witness an exponential growth of participation in the Internet. Only a few years ago, the main drive came from the incremental inclusion of stationary PCs, either from individuals connecting over Internet service providers or from entire enterprise networks. Today, however, we can see that Internet connected devices have become smaller and smaller and thus entered our daily mobile life.



Following the history of networked computing, we can see a major trend starting from the *one computer, many users* pattern in the days of expensive and large mainframes. It shifts towards a *one computer, one user* paradigm in the home computer and PC era during the 1980's and 1990's and finally becomes *one user, many computers* in today's mobile environments. Powerful cell phones and pocket computers (like PDAs) have found their wide-spread use. Wireless Internet access points are steadily increasing their density. Embedded devices like environmental sensors, surveillance cameras or stationary public phones are increasingly connected through to Internet. RFID tags are becoming more and more ubiquitous and, besides security and privacy concerns, raise questions about a suitable global infrastructure.

On the other side of the *digital gap*[1], poverty, socioeconomic background, disabilities and general knowledge deficiencies prevent major parts of the world-wide population from access to learning resources [DOT70]. This was found to be a problem in post-industrial societies, but even more so in developing countries. Especially for the latter, there is a substantial dispute whether the growing ubiquity of Internet access and the freely available resources in the World Wide Web can help in bridging the knowledge gap or not.

At the time of this writing, a number of projects world-wide are trying to provide people from rural areas with hardware and Internet connections. A common saying in this context is "One child, One laptop", which underlines the hope for a major educational impact. Some of these projects were thus initiated by governmental interest in education (such as the Simputer[2] in India), others come from private initiatives (like the *Children's Machine*[3] initiated by Nicholas Negroponte). While none of them is free of commercial interest, they all aim to give poor, under-educated people access to globally available knowledge and to enable their active participation in broader communication through cheap, portable computers. Wide spread acceptance and increasing quantities are expected to further decrease the costs of these systems.

The before mentioned technology centred initiatives have yet to see their break-through. However, certain results have already become visible and have shown to be rather promising. This further feeds the growing interest in connecting more and more people to knowledge resources. Global events like the "World Summit on the Information Society", held by the United Nations in 2003, encourage world-wide discussion on this topic and may well lead to increasing support for sensible projects.

We currently see that a large part of the world population is still excluded from the usage of Internet resources. This simple fact exhibits a non-negligible chance that today's Internet still has its major scalability issues lying ahead.

[1]`http://en.wikipedia.org/w/index.php?title=Digital_divide&oldid=72467327`

[2]`http://en.wikipedia.org/w/index.php?title=Simputer&oldid=68302591`

[3]`http://en.wikipedia.org/w/index.php?title=The_Children%27s_Machine&oldid=72726137`

1.1 New Services in Globalising Environments

These emerging trends towards a globalising ubiquity of connectivity are and will continue to be a driving force in the increasing requirements on Internet services. The most widely used services today are e-mail (including mailing lists, news-groups and SPAM) and the world wide web. Although mail and web servers have shown a high scalability in every-day situations, their scalability as centralised systems is still limited by hardware performance and financial considerations.

Especially free and non-profit services, which continue to fill a major portion of the Internet, show these insufficiencies. They simply cannot afford to provision resources over demand to satisfy extraordinary request peaks. The well-known results are slow and unpredictable responsiveness of mass mail servers and news-groups (SourceForge mailing lists[4], Yahoo-Groups[5], ...). Web-servers have the same problems with flash-crowds ("Slashdotting", reaction to Software announcements, catastrophic news of mass interest, ...). This reveals the need for low-budget scalability of mass communication services.

Despite the fact that e-mail and world wide web are still the most widely used services, the major part of the internationally generated network traffic (60% and more[6]) is currently produced by file swapping networks such as Gnutella[7], KaZaA[8], EDonkey/Overnet[9] or BitTorrent[10]. Their whole purpose is to make large files (from megabytes to several gigabytes) available for efficient multi-source download. Since they came into use around the year 2000, these systems have already proven their scalability to more than a million concurrent users [SGG02] which is achieved mainly by the decentralisation of their download service.

There is currently a major focus in research to see whether an equivalent decentralisation makes sense for other services in these systems, such as the search for files (as exemplified by Gnutella and Overnet), and how this can be achieved in a similarly scalable way. First attempts to decentralise e-mail [KRT03, MPR$^+$03] and news dissemination [SMPD05] show the potential of server-free, decentralised services.

Another major part of the current network traffic comes from server based file distribution, like software updates, CD/DVD image downloads, etc. While most of the files are still copied by FTP or HTTP to the clients or via RSync between mirrors, systems like BitTorrent show that a partially decentralised

[4]`http://www.sourceforge.net/`

[5]`http://groups.yahoo.com/`

[6]`http://www.cachelogic.com/home/pages/understanding/identifying.php` (Aug. 30, 2006)

[7]`http://en.wikipedia.org/w/index.php?title=Gnutella&oldid=72696718`

[8]`http://www.kazaa.com/`

[9]`http://edonkey2000.com/`

[10]`http://www.bittorrent.com/`

approach can achieve much higher scalability [QS04]. It uses a centralised server (or tracker) only for delegating requests for file slices to clients that already downloaded them. The dominating work of copying these slices is done by the participating clients that interact directly. This relieves the need for expensive server replication and over-demand bandwidth provisioning on the side of the distributor. Newer versions of BitTorrent even remove the remaining bottleneck at the tracker by decentralising the client delegation. They use an overlay network similar to the one in the Overnet file sharing network to store the slice availability.

1.2 Requirements on Overlay Networks

Due to such promising achievements, interest in these decentralised overlay networks has exploded over the last years, both in research and in practically deployed applications. They are generally perceived as a comfortable building block for large-scale distributed systems. The proven mass scalability of Overnet, KaZaA or Gnutella, but also the extended BitTorrent design show their appealing potential. They can be deployed where applications require decentralised communication over highly scalable, self-maintaining infrastructures.

Especially self-maintenance is a crucial feature when it comes to large-scale systems. Overlay networks are commonly constructed based on distributed, self-organising algorithms that aim to achieve a high resilience against arbitrary failures. The ultimate goal is to build administration free virtual networks that provide simple communication abstractions and allow the deployment of diverse applications on top of them.

The recently proposed systems are rather diverse in their major characteristics. An extensive overview is provided in the related work sections 9.1 and 9.2 of this thesis. One of the examples above was the Gnutella network, which deploys a power-law topology. Despite the simplicity of its algorithms, it achieves impressively good resilience properties and a low diameter of the overall topology. Other overlays, like P-Grid [Abe01] or Chord [SMK$^+$01], are organised as distributed trees or rings, which allows them to execute efficient lookup operations directly on top of their topology.

Having different characteristics also means that these networks solve different problems with varying effectiveness. It is rather difficult to find or build overlays that can fulfil the entire range of requirements for a given, non-trivial application. This holds especially in large systems which have to respond to the diverse interests of thousands to millions of users. Hybrid search systems like [LHSH04] are an example. They combine the characteristics of different types of overlays to achieve a similar quality-of-service for finding rare items as for the broader search of well replicated data.

The following (non-exhaustive) list tries to give an idea about the diversity of requirements that applications can pose on overlay networks. They may benefit from

- (average) single-hop delivery for reduced latency
- (provably) good resilience for high reliability and availability
- rapid reconciliation when nodes join or fail
- practical deployability (in a specific scenario)
- high scalability to millions of participants in the Internet
- optimal performance in small installments and ad-hoc networks
- low maintenance overhead
- ping/ack at low rates (or none at all)
- low per-node degree and state
- high degree of freedom in optimal neighbour selection
- optimisability for high throughput or low latency
- good search properties for both needles and hay

Some of these properties are even mutually exclusive. Obviously, a combination of a low node degree and single-hop delivery will not result in a scalable topology. Therefore, each specific application that runs on top of an overlay will require a specific subset of these characteristics at the cost of others being ignorable or even impossible. This subset determines the choice of the right overlay.

As an example, overlay-multicast applications [RKCD01, ZH03] may prefer a low or a high node degree. A low degree generally leads to deeper graphs and higher end-to-end latency but allows for higher throughput at each node. A higher degree reduces the tree depth and therefore the latency and the probability of node failures along each path. It also tends to increase the resilience against failing neighbours. Varying the degree may yield the desired throughput versus latency and reliability tradeoff.

Distributed content based publish-subscribe systems (see chapter 2) are another example. They base their routing decisions on filter matching. Filter updates, especially to new nodes, can be costly depending on the complexity of filters. In this setting, a small number of neighbours may turn out to be more desirable to keep the number of updates low. As with multicast, however, a higher degree may yield better performance for content forwarding, so the right tradeoff is again application specific.

For relatively small overlays, single-hop delivery may be optimal and very desirable. However, the overlay may have to switch to multi-hop delivery when the number of nodes increases dynamically or if the dynamism (or "churn rate") of the network exceeds a certain threshold [RB04].

In many use cases, the members of an overlay network form subgroups at run-time, e.g. for better efficiency, resilience or security. This may include multicast groups, semantic clusters or functional clusters (like in Omicron [DMS04]). These subgroups will most likely have different requirements than the "global" overlay (after all, that is why they are formed in the first place).

A low maintenance overhead is always desirable, while in small or well-connected intra-organisational installations other properties, like rapid reconciliation, may take precedence.

A quality-of-service aware overlay must provide very different service types to applications, as different users usually have very different ideas about the QoS they need.

So far, these requirements stayed at a rather abstract level. The second chapter of this thesis aims to substantiate the diversity of the requirements posed by applications and the characteristics provided by overlay networks. It presents an extensive case study in the concrete field of publish-subscribe applications. By building up a meaningful set of quality-of-service metrics for this area, it provides a solid ground for the evaluation of overlay networks against concrete requirements.

Section 2.3 uses the presented metrics for an exemplary evaluation of overlay based publish-subscribe systems. It tries to show the capabilities, weaknesses and limits of different systems. From this analysis, it becomes obvious that no single overlay will ever satisfy the needs of all possible applications, not even in a single application domain such as publish-subscribe. Even within a specific application there may still be situations where a choice between different topologies makes sense. Having to make this decision at implementation time (or even design time) is obviously premature, since it prevents tests with different topologies as well as an adaptation at run-time.

As an approach towards the integration of similar overlay types, a common API for structured overlays was proposed by Dabek and others [DZDS03]. This allows an application to abstract from the specific details of different structured overlays and use the one at hand.

The main problems, however, arise at a different level, which cannot be approached with an API. The available structured overlays continue to provide distinct implementations in different frameworks and different programming languages. Applications that want to deploy overlays can hardly support more than one framework in one language. This complicates comparisons and effectively prevents a hybrid combination of overlay systems. Even if it was possible to use them combined, the potential gain would be limited by the fact that each overlay requires its own maintenance algorithm and introduces its own overhead.

Due to the current complexity of overlay implementations, porting an overlay between frameworks basically means reimplementing it, which is a huge obstacle for application developers. In this context, the building block idea degenerates into an additional block that has to be built rather than a helpful support for the design of an application.

There were a few attempts to aid in the implementation of overlay networks, mainly JXTA [JXT, JXT03], iOverlay [LGW04] and Macedon [RKB$^+$04]. Section 3.2 presents them in detail. However, the analysis following their description shows that they fail to make integrative, framework independent models available to overlay designers and application integrators.

1.3 Integrative Design of Overlay Networks

As we will see further on in section 3.4, implementing overlay networks is far from trivial. Implementations typically require between 10,000 and 30,000 lines of source code. As distributed networking systems, overlay implementations are often written in event-driven I/O patterns. This leads to counterintuitive code modularisation, cross-cutting interdependencies between modules and limited support for code reuse. These obstacles render the understanding of the actual system and its topological features difficult.

As noted in 3.1, a simplification in overlay design requires high-level models that capture the important characteristics of overlay networks. While the componentisation of overlay software must respect the event-driven nature of the running system, it is more important to encourage a clean, high-level design of the implementation. The goal is to make it understandable for other developers and application integrators without requiring additional, external documentation.

The need to integrate different overlays into adaptable hybrid systems poses further demands on the design process and its tool support. Different overlays must be able to share their system state. Otherwise, the independent maintenance strategies cannot become aware of each other, which requires them to duplicate their effort. Shared state also reduces code redundancy between overlay implementations. Every participant has to store state about neighbours and maintain its connections to them. A common ground for handling neighbour state allows to move this functionality into middleware components.

The main characteristics of overlays result from their topology. As we will see in chapter 3, however, the currently available models and abstractions for these systems do not capture this part of their design. If the topology is meant to become a readable (and thus understandable) part of overlay software, new models are needed to support its design.

The *topology perspective* on overlay software, as described in section 4.1, is an approach to make the five main functionalities in overlay networks visible in their implementation: topology rules, overlay routing, topology maintenance, topology adaptation and topology selection.

Rules define the topology from the local point of view of each participating node. Routing describes how to use the rules to forward messages. Adaptation honours the fact that the rules may leave a certain degree of freedom in neighbour selection and forwarding decisions and tries to optimise the topology according to specific policies. Maintenance deals with the actions that must be taken when the local view breaks the rules. Selection finally deals with the integration of different topologies (i.e. rules, forwarding/maintenance schemes and adaptation policies) to let an application select the right topology for its current requirements.

Each overlay implementation must deal with these functionalities. This motivates the design of a domain specific language called Slosl (described in chapter 5) for the platform independent implementation of rules and adaptation policies. It is easily extended to support routing decisions defined in the Edgar language (6.3). To capture the entire design process of overlay implementations, and to provide high-level support for topology maintenance and selection, this thesis proposes a Model Driven Architecture based on Slosl, the XML Overlay Modelling Language OverML (chapter 6) and the Node Views system architecture (4.3).

OverML is a set of five domain specific languages for the area of overlay design. The combination of these languages allows for far-reaching, semantically rich models of overlay systems.

Nala The Node Attribute Language specifies data schemas for the attributes of overlay nodes. It serves as a common model for local state keeping.

Slosl The SQL-Like Overlay Specification Language defines local views for each node in the overlay. It describes the topology as the union of all local views and expresses topology rules and adaptation strategies.

Edgar The language defines Extensible Decision Graphs for Adaptive Routing. Based on the rules defined by Slosl, it specifies the local routing decisions taken by each node to forward messages through the topology.

Himdel The Hierarchical Message Description Language models layered messages based on Nala data types and Slosl view data.

Edsl The Event-Driven State-machine Language defines overlay protocols and connects implementation specific maintenance components with events defined by the architecture, especially by the languages Slosl and Himdel.

Chapter 8, as well as the various examples in other chapters, show the broad applicability of these languages to the design of different overlays and their topologies.

1.4 Overview – how to read on

To give an idea of the diversity of requirements on overlay networks, the following chapter presents an extensive case study of a specific class of overlay applications: distributed publish-subscribe systems. First, it derives quality-of-service metrics that allow to compare systems from this area. They are used in section 2.3 to show the differences between a number of well-known implementations that deploy decentralised overlay networks to support publish-subscribe at a global scale. Their different levels of quality-of-service make a case for middleware architectures that integrate different systems. Only a resource efficient integration allows applications to benefit from the diversity of features that different systems provide.

A new methodology for the integrative, platform-independent, high-level design of data-driven overlay networks is the major contribution of this thesis. Chapter 3 overviews different abstraction levels that are currently used in overlay software. It shows the deficiencies of recent frameworks when requiring a substantial simplification of the integrative, portable design of overlay networks. The main reason is that current frameworks have largely focused on networking and protocols. Section 4.1 replaces this focus by a new perspective that targets the main characteristics of overlay networks, which are determined by their topology.

Taking the topology perspective on overlays, chapter 4 derives a new architectural model for overlay design, the Node Views architecture. It underlies the domain specific OverML languages that are presented in chapter 6, with a special emphasis on SLOSL, a specification language for local decisions in topologies (chapter 5). The OverML languages form the foundation of the Model Driven Architecture proposed in this thesis. Chapter 7 finally sketches ways of implementing common communication patterns using the new architecture.

Chapter 8 presents the current implementation of an Integrated Development Environment (IDE) for overlay design, based on the OverML language, as well as a mapping of the abstract design models to executable implementations. It finishes by presenting a complete walk-through from the specification of an overlay to its deployment.

Related work in different areas as well as an overview of available overlay networks is presented in chapter 9. The work is finally concluded by chapter 10.

Chapter 2

A Case Study: Publish-Subscribe Overlay Applications

Contents

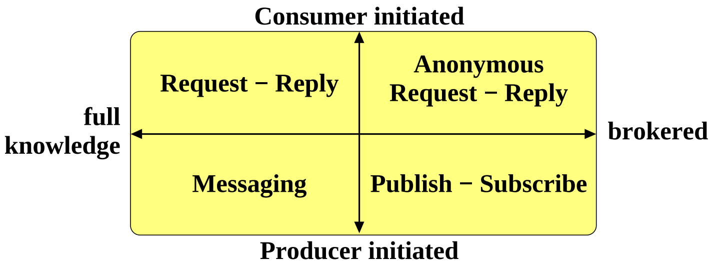


Figure 2.1: Distributed communication paradigms

Figure 2.1 shows the known paradigms for distributed communication. The two on the left are in widespread use today. Following the examples from the introduction, the e-mail service represents a messaging scheme, although the internal server-to-server interaction is based on request-reply. Web and file servers are built on centralised request-reply interaction. In both cases, the producer of data (i.e. the server) knows all receivers that initiated a request and addresses them explicitly and directly for its reply, which is a major problem for server scalability.

Commercial web server clusters already take a step towards higher scalability through a variant of the anonymous request-reply scheme. They exploit the indirection provided by DNS and load balancers to decouple service names from servers and spread the load over a larger number of servers. Large sites like the Amazon web shop[1] or the Google search engine[2] are prominent examples that make tens of thousands of servers available under the same name. They can even be globally distributed over different locations, as Google shows. Addressing works in a randomised way based on implicit request semantics such as the host domain or preferred language of the client.

The original BitTorrent design uses request-reply for all services (finding sources and downloading), but the tracker randomises the delegation to download providers. Again, this shows ideas of anonymous (or delegated) request-reply, which is even more visible in the newer overlay design of the system. Current file swapping networks commonly use a delegated or anonymous form of request-reply for search and direct request-reply for download.

As the examples show, the flavours of request-reply are the most widely used paradigms. Simple variations of anonymous request-reply have been adopted fairly often in the form of DNS delegation or centralised load-balancing. Scalability is commonly achieved by delegating resource intensive tasks to a large number of nodes, while keeping a single entry point for clients. Overall, the technical trend is clearly guided towards reduced knowledge at the initiator.

An interesting and rather old example are mailing lists and news groups, which deploy the fourth paradigm, publish-subscribe. Despite its appealing simplicity, this scheme is so powerful that it can provide advantages to many other Internet services, especially where scalability is an issue. The two brokered paradigms on the right virtualise and decouple the connection between consumers and producers of data. This inherently makes them very interesting for decentralised communication infrastructures.

2.1 The Publish-Subscribe Paradigm

The system model of the publish-subscribe communication paradigm is surprisingly simple. It provides three roles: publishers, subscribers and brokers. Publishers (aka producers) provide information, advertise it and publish notifications about it. Subscribers (aka consumers) specify their interest and receive

[1] `http://www.amazon.com`

[2] `http://www.google.com`

relevant information when it appears. Brokers mediate between the two by selecting the right subscribers for each published notification. In the following, the term "client" will additionally be used for publishers and subscribers to distinguish their roles from the broker infrastructure. A more thorough discussion of the publish-subscribe model is provided in [Fie05].

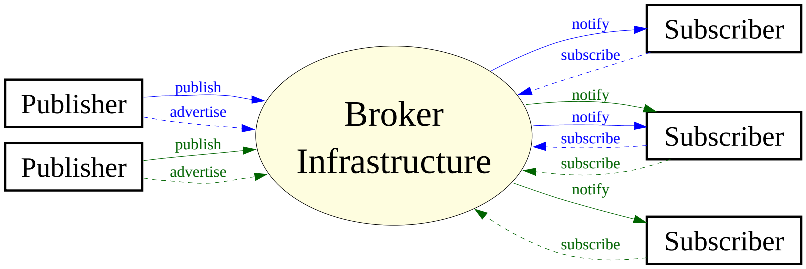


Figure 2.2: The publish-subscribe model

This scheme decouples publishers and subscribers and thus simplifies their roles and interfaces considerably. The drawback is the infrastructure in the form of brokers that is necessary to provide the publish-subscribe service. In the mailing list example, the mailing list servers are brokers that mediate between authors and readers of e-mail, i.e. publishers and subscribers. Similarly, in IP-Multicast [Dee91], the IP routers take the role of brokers.

However, publish-subscribe is not limited to mailing lists, where the broker is a server that has a simple list of receivers available for each message type. This case is commonly known as centralised *subject based* publish-subscribe where consumers subscribe to a subject that producers explicitly assign to their notifications. In *content based* publish-subscribe, subscriptions express more general filters on the actual content of notifications. This scheme is both more powerful and more complex than fixed lists of recipients.

An important property of the publish-subscribe model is the level of abstraction at which publishers and subscribers communicate. They are not aware of the organisation of the infrastructure or of the size of the system. There can be a single centralised broker, a cluster of them or a distributed network of brokers. All that participants see is their specific brokers through which communication partners are self-selecting by interest. This makes this model appealing for highly scalable systems that need to hide varying complexity and adaptable infrastructures from the participants.

However, finding a suitable infrastructure organisation is not possible without further knowledge about the specific needs of the participants. If the infrastructure is supposed to provide a sufficient level of quality-of-service (QoS) to all participants independent of the current scale or infrastructural organisation, explicit QoS distinctions become vital to the system.

2.2 Quality-of-Service in Publish-Subscribe

The publish-subscribe model suggests a number of different quality-of-service (QoS) metrics, as described by the author in [BFM06]. Some are related to subscriptions and single notifications while others describe end-to-end properties of flows of notifications. The following sections describe the specific meaning of these metrics when requested by publishers or subscribers, in subscriptions, notifications or advertisements. Their context in publish-subscribe systems is briefly summarised at the beginning of each section (☞).

Their implementation in a distributed publish-subscribe system then has mainly two dimensions: The topological organisation of the brokers and the local decisions of each broker regarding filtering, scheduling, etc. Their interaction with and impact on the QoS metrics are briefly summarised in figure 2.3.

2.2.1 QoS at the Global Infrastructure Level

End-to-end latency, bandwidth and delivery guarantees form low-level properties of the broker infrastructure. In a centralised infrastructure in which the client connections also implement QoS, there are ways to impose hard limits on them. In any less predictable environment, especially distributed multi-hop infrastructures, they should not be understood as real-time guarantees. Here they become probabilistic options or even hints about preferences of clients. Note that even hints can be helpful to the infrastructure if it has to determine which notifications to drop from overfull queues or which broken inter-broker connection to repair first.

Latency

☞ *Subscriptions* - Subscribers request a publisher with a maximum delay.

The end-to-end latency between producers and consumers depends on the number of broker hops between them, the travel time from hop to hop and the time it takes each broker to forward a notification.

In a centralised system, the travel time between broker and clients gives a hard lower bound and the additional forwarding time depends on the broker load. Even in a distributed infrastructure, measured lower bounds can give hints if a requested QoS level is achievable at all. In general, however, they do not allow to give absolute guarantees in distributed systems.

A broker infrastructure can deal with latency requirements by pre-allocating fixed paths between senders and receivers. This avoids the overhead of having to establish connections on request.

A further speed-up can be achieved by tagging notifications and merging them into channels [Fie05]. This approach effectively maps content-based filtering to subject-based filtering and thus simplifies the routing on each broker

QoS Metric	***Topology Impact***	***Local Broker Decisions***
Latency	end-to-end network latency and hop count, reorganisation delays, global load distribution, adaptation to physical topology, degree of freedom in routing	time complexity of filter evaluation, local routing choices
Bandwidth	end-to-end bandwidth, global load distribution, path independence, adaptation to physical topology, degree of freedom in routing	local routing choices
Message priorities	topology shortcuts	local scheduling
Delivery guarantees	reliability, reorganisation, path redundancy, trusted or reliable subgraphs	persistence and caching, routing decisions based on reliability, trust or reputation
Selectivity of subscriptions	adaptation to subscription language (e.g. subject grouping)	decidability, local load
Periodic or sporadic delivery	-	conversion
Notification order	single/multi-path delivery, consistency during reorganisation	reordering
Validity interval	deterministic delivery paths for follow-up messages	message drops after delay or on arrival of follow-ups
Source redundancy	path convergence	identity decision
Confidentiality	trusted subgraphs	encrypted filtering
Authentication and integrity	-	client authentication, access control

Figure 2.3: Relation of topologies and local broker decisions to QoS metrics: Impacts and implementation choices

along the path. Note, however, that such a mapping is not always possible as it can lead to state explosion [Müh02]. The reduced expressiveness of channel identifiers compared to content filters can also introduce excessive forwarding of unwanted information. The most efficient form is obviously a mapping to the network level through native IP-Multicast [Dee91, Hui99].

A combination of both measures may allow to reduce the path length by skipping those brokers that only forward a stream to one or very few connections. Still, the observed latency may vary considerably over time, as Internet paths commonly serve multiple concurrent streams. The optimal case is a physical network with native quality-of-service provisioning and IP-Multicast.

Bandwidth

☞ *Advertisements* - Producers specify the minimum and/or maximum bandwidth necessary for the stream they produce.

☞ *Subscriptions* - Subscribers restrict the maximum amount and size of notifications they want to receive per time unit.

The overall bandwidth used by the system depends on the throughput per broker and the size of each notification. Today's Internet-level connections tend to be sufficiently dimensioned to allow many concurrent high-traffic streams, but they usually do not provide physical quality-of-service and bandwidth provisioning. This holds especially when streams cross the borders of autonomous systems.

Therefore, bandwidth requirements should rather be regarded at a per-broker level. If each broker knows the bandwidth that it can make locally available to the infrastructure, this gives an upper bound for the throughput of a path. Although not necessarily accurate, such an upper bound allows to route notifications based on the highest free bandwidth on the neighbouring brokers. It can be used to avoid high-traffic paths and to do local traffic optimisation.

If channel merging is applied (as described for latency), the channels can be tested for their capability of providing the required bandwidth. However, the observed bandwidth in general purpose networks may show high variations over time that cannot be foreseen.

Message priorities

☞ *Notifications* - Producers specify the relative priority between notifications that they produce themselves or their absolute priority compared to other (foreign) notifications.

☞ *Subscriptions* - Subscribers specify the relative priorities between their subscriptions or the absolute importance of a subscription.

As with latency and bandwidth, message priorities have both a per-broker side and an end-to-end side to them. They are, however, easier to establish in a distributed broker network than latency bounds, as their conception does not aim to provide absolute or real-time guarantees.

Priorities between notifications can be used to control the local queues of each broker which will eventually lead to their end-to-end application along a path. Again, channels can be used to shorten the path for high-priority notifications, the extreme case being to send them directly from the system entry point to the recipients. More commonly, however, high-priority notifications will be allowed to overtake those with a lower priority during the forwarding and filtering process at each broker, subject to a weighted scheduling policy. This increases the importance of their per-broker part.

Absolute subscription priorities can be merged on their delivery paths. Only the maximum priority is required at the publisher end. Brokers further down the path can store the more exact priorities for their forward scheduling to local subscribers and neighbouring brokers.

Delivery guarantees

☞ *Subscriptions* - Subscribers specify which notifications they must receive, which are less important, and where duplication matters.

☞ *Notifications, Advertisements* - Producers can specify if receivers should be guaranteed to receive their notifications.

These guarantees can be as simple as a tag for notifications stating if they *may be dropped* on the delivery path or not. It is even straight forward to merge this tag from multiple subscriptions along the delivery path with a simple boolean “and”. This approach is especially applicable in combination with message priorities, where the least important message is the first to be dropped.

More demanding guarantees regard the completeness and duplication of the delivery. Notifications can be delivered *at least once*, *at most once* or *exactly once* to a subscriber, the latter being the combination of the first two. If the infrastructure is not reliable itself, the first requirement can be achieved by meshing which increases the delivery probability at the cost of generating duplicates and thus increasing the message overhead. A request for at most one delivery encourages either single path delivery or duplicate filtering before the arrival at the subscribers.

In infrastructures with unreliable brokers, reputation systems or availability histories may allow to route notifications through more reliable brokers first. This lowers the chance of brokers failing during delivery or maliciously suppressing notifications. However, this should be taken as a hint and by no means as a guarantee since a history does not allow a reliable prediction of the future availability or behaviour of network and brokers.

Another problem with delivery guarantees regards temporarily disconnected subscribers [CFH+03], e.g. in a mobile environment. The broker infrastructure must store notifications that were guaranteed to be delivered at least once until the subscribers become available again. Note that this may be after an arbitrarily long time or never, so in practice, the infrastructure may choose to store notifications only for a reasonable time interval.

Finally, it is possible to define a *quorum*, i.e. to specify the minimum or maximum number of senders that must receive a notification or the minimum or exact number of receivers that a subscriber wants to subscribe to. In a decoupled, brokered environment like publish-subscribe systems, where publishers and subscribers are not supposed to know anything about each other, this criterion should only be available at an administrative level, e.g. within scopes [FMG03].

Note that the requests of subscribers override the advertisements of publishers. If a publisher requests guaranteed delivery and all subscribers agree that its notifications may be dropped, the infrastructure may ignore the request of the publisher. Subscriber requests are generally more important than publisher requests, which become not much more than a hint or default to the infrastructure.

2.2.2 QoS at the Notification and Subscription Level

A number of QoS properties touch the semantics of subscriptions and notifications. They are periodic or sporadic delivery, the order in which notifications arrive, priorities between them, the duration of their validity and redundancy of producers. Security issues like authentication or confidentiality also fall into this scheme. A very important factor is the selectivity of subscriptions.

Expressiveness and Selectivity of Subscriptions

☞ *Subscriptions* - Subscribers define their subscriptions in a specific language.

Subscriptions can be expressed in different classes of languages, thus allowing different levels of expressiveness. Simple subscriptions can contain a subject for identity matches, whereas more complex ones can be augmented by further attribute restrictions. The most complex subscriptions possible are filters implemented as Turing complete computer programs.

All of these languages have a specific level of complexity with respect to filter identity tests and merging, distributed matching, global message overhead and false positives on delivery. The filter language therefore represents a tradeoff between the local overhead at brokers or subscribers and the distributed overhead inside the broker infrastructure. In general, more complex languages can support a higher expressiveness and therefore a higher selectivity of subscriptions. This allows them to reduce the number of false positives (or uninteresting notifications), but tends to make filter optimisations and distributed matching more difficult.

The language is not necessarily the same within the entire infrastructure. Scoped subgraphs [FMG03] may decide to provide languages internally that have a different expressiveness than the outside world, and then convert between the two at the interfaces. Similarly, a broker may decide to accept complex subscriptions from its local subscribers and forward simplified versions that remote brokers understand or that are more suitable for the topology. If it then filters incoming notifications based on the more expressive subscriptions, it can increase the satisfaction level of the local subscribers.

Periodic or sporadic delivery

☞ *Advertisements* - Producers advertise their way of publishing.

☞ *Subscriptions* - Subscribers specify which notifications they want to receive sporadically and which they need periodically.

This captures the difference between an interest in changes of information and the information itself. Periodically published notifications become a data flow that represents the status of requested information at the moment of each publication. Sporadically published notifications occur only when this information changes. One way of looking at them is as prefiltered events that pass when the data change exceeds a certain threshold. This is an interesting option for reducing the amount of data sent by accepting a certain inaccuracy.

The infrastructure may either match subscriptions to corresponding publishers or try to emulate the requested delivery mode by itself. Periodic notifications may be emulated by storing and repeating the latest sporadic notification. Sporadic delivery can be based on a configurable threshold that blocks the delivery of periodically published static values [MUHW04]. Note that in a content-based system with arbitrary filters, the necessary comparison of notifications may be restricted to an identity test or may not be possible at all.

Notification order

☞ *Advertisements* - Producers advertise for which of their notifications the ordering matters.

☞ *Subscriptions* - Subscribers specify in their subscriptions which notifications they want to receive in order.

The order in which notifications arrive may or may not be relevant. In many cases, ordering is easy to achieve by either using centralised ordering, ordered transports (ATM, TCP) or by letting the producer (or its broker) impose an explicit order and sorting the notifications on delivery.

Special care must be taken if the broker topology is allowed to change during the delivery process or if priorities are used, as both may impact the order in which notifications arrive and may even result in message loss. Since ordered delivery does not imply guaranteed delivery, a very efficient solution is to deliberately drop notifications if a successor has already been delivered. If this is not acceptable, notifications must be reordered before delivery, which may introduce arbitrarily long delays.

The distributed ordering of events coming from different sources is another problem. For content-based subscriptions, it is even hard to define a meaningful ordering in this case. It is therefore largely dependent on the subscription language and the application if such an order is applicable. A generic (and very simple) approach is the deployment of a dedicated, central broker to enforce a global ordering, which can in turn limit the scalability of the overall infrastructure. Note, however, that this would only impact notifications for which ordering was requested.

Validity interval

☞ *Notifications, Advertisements* - Producers advertise or specify a timeout for their notifications, or a successor message that renders them irrelevant.

It is important for the infrastructure to know how long a notification stays valid, either specified in terms of time or inferred by the arrival of other (follow-up) messages. A validity based on a hop count (as used in many low-level transport protocols) would be meaningless in the decoupled publish-subscribe model. If only the most recent event is of interest, the validity specification by follow-up messages is a particularly efficient approach. It allows the infrastructure to reorder and shorten its queues in high traffic situations.

Source redundancy

☞ *Subscriptions* - Subscribers request redundant sources for the same event.

An example for redundant sources is a set of temperature sensors inside a room that publish more or less the same value. They generate semantically similar notifications which allows subscribers to double check events. In some cases, the increased fault tolerance may be sufficient to replace at-least-once guaranteed delivery.

Note that a request for redundancy requires the notifications or advertisements of publishers to be comparable based on a subscription. Rice's Theorem[3] implies that complex subscription languages may hinder or prevent this. In this case, explicit hints such as subjects or type identifiers are required to assure the comparability of notifications.

Confidentiality

☞ *Subscriptions* - Subscribers encrypt their subscriptions or send them only to trusted brokers.

☞ *Notifications* - Publishers connect only to trusted brokers or send partially or completely encrypted notifications.

Confidentiality can obviously be achieved in (sub-)networks that contain only trusted brokers [FZB$^+$04]. Similarly, any message can be hidden from untrusted brokers on the delivery path by using standard encryption mechanisms between trusted brokers. Untrusted brokers that cannot read and evaluate the content are then forced to broadcast to all of their neighbours. Apart from higher load at those brokers, this also introduces potentially large numbers of duplicates in the system.

Confidentiality becomes a difficult problem if untrusted brokers must be allowed to participate in the matching process. This is a minor problem in subject-based publish-subscribe, where the low expressiveness of subscriptions makes the actual content of notifications opaque to the matching process. It can just as well be encrypted, which may be exploited within untrusted broker networks. A reduction of content filters to channels can achieve the same effect. In both cases, the information gain of untrusted brokers depends on the amount of information revealed by the subject or the channel ID. The selectivity of subjects can therefore be seen as a tradeoff between message overhead and revealed information. Untrusted brokers can thus be prevented from overhearing the information, but malicious brokers can still suppress messages in this scenario.

Content-based matching requires much higher insight into the content of notifications. A number of recent publications from the database area show ways for matching encrypted data against certain types of queries [AKSX04, DAK00, AKD03] without revealing any of the two. Publish-subscribe systems could apply similar schemes to allow blind matching on untrusted brokers. However, solving this problem for arbitrary queries and data without revealing any information about them is likely impossible.

Authentication and Integrity

☞ *Subscriptions* - Subscribers authenticate their subscriptions.

☞ *Notifications* - Publishers authenticate their notifications.

[3] `http://en.wikipedia.org/w/index.php?title=Rice%27s_theorem&oldid=64121863`

If publishers want to enforce access control mechanisms [BEP+03], it becomes necessary for subscribers to authenticate their subscriptions using techniques like digital signatures. On the other side, publishers can sign or watermark their notifications to assure data integrity. The evaluation can then either be done end-to-end or within the broker infrastructure.

2.2.3 Discussion

The analysis presented in this chapter provides extensive insights into the diversity of requirements in publish-subscribe systems with respect to the broker implementation and the topological organisation of distributed broker networks. It must be noted that very few quality-of-service levels can be enforced locally on each single broker, in which case they are mainly related to the filtering process, to scheduling concerns or to local delivery.

On the other hand, almost all metrics are impacted by the topology of the broker infrastructure. One of the major limiting factors is the overall message overhead. Message multiplication in multi-hop infrastructures reduces the bandwidth that is available for each unique message. At the same time, it increases the per-broker load and therefore the end-to-end latency. It can even impact delivery guarantees if brokers become overloaded and are forced to drop messages.

Two major factors impact the global message overhead of a publish-subscribe architecture: The expressiveness of the subscription language and, again, the topology of the broker network. While subscription languages are an interesting topic for further investigation, they are also out of scope for this thesis. We will therefore continue with a strong focus on those QoS constraints for which the topological infrastructure is the dominating factor.

The topology of the broker infrastructure has been a major playground for research in recent years. Overlay networks were widely adopted as the basic building block for publish-subscribe systems. The current state of the art provides a system designer with a considerable body of choices amongst the variety of different overlay implementations.

The next section overviews how overlay networks have been used to implement scalable publish-subscribe systems and evaluates the QoS capabilities they provide.

2.3 Publish-Subscribe on Overlay Networks

An important development in recent years was the introduction of overlay networks as building blocks for the distributed broker infrastructure of publish-subscribe systems [PB02, TBF+04]. This allows new approaches to distributed filtering as well as a design simplification of distributed publish-subscribe systems.

The first attempts were targeted at replacing application-level multicast, while later approaches aimed to support content-based publish-subscribe filtering. The following sections describe different systems and mention the specific quality-of-service properties that they provide for broker infrastructures.

2.3.1 Multicast Approaches

While many autonomous systems in the Internet have IP-level multicast enabled, a successful implementation across borders is still not available. It is even unpredictable if the switch to IPv6 will bring substantial advances for the support of native multicast at the physical layer, as its availability to customers will continue to depend on commercial incentives, network management and security concerns of Internet service providers. This is why many distributed applications have long used application level multicast for their needs.

Overlay networks are expected to simplify the design of these applications by providing generic infrastructures as building blocks. While rather limited from a publish-subscribe point of view (low expressiveness of subscriptions), multicast has the advantage of being easily and efficiently implemented on any key-based routing (KBR) network (see 9.2).

✔/✘ *Selectivity* — The selectivity of subjects (or multicast groups) is relatively low. This usually means that subscribers are left to do their own filtering. On the other hand, if expensive subscription languages render matching and optimisations difficult, it may be worth considering a combination with subjects to provide a fallback through subject matches. This may prevent broadcasting in many cases, which can considerably reduce the overhead of messages and filtering.

Multicast was one of the major applications proposed for various KBR networks. There are two different ways of implementing multicast on top of them: overlay internal and overlay external.

Overlay external multicast

These implementations use a global overlay to index groups and then simply create a separate broadcast overlay for each multicast group. This was exemplified by the CAN overlay [RHKS01], but is obviously possible with any other overlay that supports broadcasting. A second example is CAM-Chord [ZCLC05] that uses Chord for capacity adapted multicast.

The major advantage of external multicast is that each group overlay only has to scale with the size of a group. A disadvantage is the increased overhead of maintaining multiple independent overlays simultaneously. If the overlay is as resource consuming as CAN, external multicast may become very expensive [CJK+03].

✔ *Delivery guarantees* — All participants in the group overlay are known to be interested in all messages. This diminishes the cost of deliberate duplications which can be exploited to increase the probability of delivery, even under topological reorganisation. The redundancy that is already available in a resilient overlay can directly be exploited to assure the broadcast delivery to all live parties as long as the network stays connected. On the other hand, most KBR broadcast schemes can efficiently avoid message duplication as long as there are no reorganisations during the delivery process.
If a subset of the group members caches the received notifications, a node that missed notifications during a temporary failure can try to ask any of the other members for passed notifications, possibly using a random walk or an attenuated broadcast scheme. Similar ideas were presented in [CFH+03].

✔ *Latency* — The group overlay has the same size as the group. This minimises the end-to-end hop count, which most likely becomes smaller than in the global overlay. Obviously, this reduces the number of required routing decisions and reduces the end-to-end latency for the given topology. Note that the group overlays are independent of the global overlay and may even use a different topology to optimise for their specific group size. Overlay broadcast commonly features very good load balancing.

✔ *Confidentiality* — This scheme effectively prevents non-subscribed brokers from seeing the notifications. If some form of access control is used in the subscription process, untrusted brokers can be prevented from joining the multicast network. This enables distribution networks of trusted brokers.

✔/✘ *Selectivity* — Apart from the matching process executed for subscriptions, the distribution scheme is agnostic to the content of notifications. This allows the deployment of arbitrary delivery algorithms and additional subscription languages within the group. Systems like Choreca (2.3.5), that are built on top of broadcast delivery, can run on top of the external multicast scheme to extend its capabilities to content-based filtering. However, this requires a multi-step subscription process: a subject-based subscription to find the multicast group and a content-based subscription within the group. Depending on the application, this can be an improvement if used to reduce the number of brokers that must touch a content-based subscription. In other content-based filtering scenarios that do not benefit from subjects, the doubled subscription overhead may become a bottleneck for the overall system performance.

Overlay internal multicast

Overlay internal multicast uses features of the respective overlay to emulate or establish groups within the overlay itself. Since DHTs provide efficient intrinsics for lookup operations, the obvious approach is to have notifications and subscriptions meet at the node that a lookup yields for the multicast address, the rendez-vous node. Examples are Scribe [RKCD01] or Hermes [PB02] on Pastry [RD01], Bayeux [ZZJ$^+$01] on Tapestry [ZKJ01] and a number of proposals for Chord [SMK$^+$01], like [KR04].

These systems use two different schemes for building the multicast tree: Scribe exemplifies reverse-path forwarding (RPF) and Bayeux uses forward-path forwarding (FPF). Scribe's RPF uses the reverse path from the subscriber to the rendez-vous node to send notifications back, while FPF sends a routing message from the rendez-vous node back to the subscriber to find a new forward path. The latter tends to be more efficient in the beginning of the path as overlays commonly optimise routes in this direction. However, due to the topological organisation of the underlying DHTs, it is less efficient than RPF at the end of the path. Both schemes are described and compared in [ZH03], where a hybrid scheme is proposed to overcome their respective disadvantages.

All of these systems use distribution trees (commonly one per group) that are an integral part of the overlay topology. This usually leads to a tree depth of $O(\log N)$ where N is the number of participants in the system. The common disadvantage of these systems is that intermediate nodes must forward messages that they are not interested in. Above all, this regards the root node of the respective tree. It even represents a single point of failure that must see all notifications and subscriptions for the group to assure correctness and completeness of delivery.

- ✘ *Latency* — The global multicast network is potentially much larger than each of its groups. This generally increases the end-to-end latency experienced in each group. Also, close to the rendez-vous node, the total number of subscriptions of a node's neighbours is likely to be higher than elsewhere in the network. This further increases the load of those nodes and therefore the overall latency.

- ✘ *Delivery guarantees* — Again, due to higher load close to the rendez-vous node, the delivery guarantees may be impacted. Message duplication is more likely than in the external scheme, since the probability of topological reorganisation increases with the number of participants. On the other hand, a higher number of participants provides higher redundancy in the overall network that can be exploited to re-enhance the probability of delivery. However, achieving this is at least as hard as in the dedicated broadcast networks of external multicast.

- ✔ *Selectivity* — Due to the use of explicitly allocated delivery paths, this multicast scheme is easily extended with more advanced filtering techniques. Hermes (2.3.2) is a good example that matches on subjects and attribute-value pairs, thus providing different grades of content-based filtering with little overhead.

Discussion

It is not surprising that external multicast appears to outrun the internal scheme in terms of quality-of-service support. It simply uses a partial replica of the global overlay, so it immediately gains the same general properties as the internal multicast without requiring additional overhead at uninterested nodes.

Still, the evaluation in [CJK+03] suggests that the internal scheme has the advantage of avoiding additional maintenance overhead compared to multiple overlays in the external scheme. It also notes that this is payed for by longer end-to-end paths. Intermediate schemes as proposed for Scribe [RKCD01] can reduce the path length by skipping intermediate nodes and merging their children. Since this deviates from the original overlay topology, additional maintenance is necessary to handle these optimisations. Consequently, this approach can also be seen as the construction of a new overlay where the bootstrap overhead is reduced by reusing known nodes. We can therefore suspect the transition between the two methods to be rather smooth.

When deciding the right tradeoffs, it must also be noted that current evaluations only consider a single overlay topology for the global and all group overlays in external multicast. It would be interesting to see comparisons where the overlay topology is chosen specifically for the size, selectivity and other quality-of-service requirements of the respective multicast group. Especially the support for arbitrary subscription languages inside of multicast groups could turn the balance in some applications.

The Node Views approach, as presented in this thesis, makes the implementation and integration of different overlay topologies simple and thus enables this kind of comparisons. As CAM-Chord [ZCLC05] shows, an inherent advantage of overlay external multicast is the simpler design of overlay topologies and their implementation. Their specialised simplicity helps when introducing new features.

2.3.2 Hermes

Hermes [PB02] was the first system to exceed the limitations of overlay multicast by implementing a form of content-based publish-subscribe (named *type and attribute based filtering*) on top of overlay networks. It uses the same multicast scheme as in Scribe, but installs content-based filters along the delivery path that filter on additional attributes of the notifications. Filter merging is possible along the path to reduce the status overhead per node. Apart from these features, Hermes has the same characteristics as Scribe and other overlay internal multicast systems.

- ✔ *Selectivity* — Filtering on arbitrary attributes along the multicast paths substantially increases the selectivity of subscriptions and therefore reduces the number of false positives. The down side is an increased message overhead for filter updates compared to multicast systems.

- ✔ *Confidentiality* — Compared to multicast, the brokers in content-based publish-subscribe systems require higher insight into the notifications. Hermes uses the overlay internal multicast scheme. This means that it can only make the tradeoff between confidentiality and message overhead by letting untrusted brokers either broadcast completely encrypted messages or forward them exclusively based on their subject without revealing further content.

2.3.3 IndiQoS

IndiQoS [CAR05] reuses the design of Hermes and augments it with quality-of-service awareness, as first presented in [AR02]. The authors distinguish between a *content profile* (such as the precision of a sensor) and a *QoS profile*, allowing to request periodic delivery or latency bounds. The IndiQoS description also refers to bandwidth as an additional metric. QoS requirements are expressed as additional attributes of subscriptions and advertisements that are evaluated during forwarding.

As further improvement, IndiQoS replicates the rendez-vous node and lets the broker infrastructure select an appropriate one based on the requirements of subscribers. This is obviously payed for by a multiplication of the message overhead per notification.

- ✔ *Selectivity* — IndiQoS inherits the selectivity properties of Hermes.

- ✔ *Latency* — Subscribers can explicitly specify a maximum latency. Brokers forward subscriptions according to latencies known from prior advertisements. The rendez-vous replication then enables a lower average latency.

2.3.4 Rebeca

The Rebeca system [MFB02, Reb] is somewhat of an outsider in this list since it was initially developed as a static broker network, where joining and leaving was limited to the leaves of the delivery tree. It supports content-based publish-subscribe and was intended as a proof-of-concept system for filter merging and administration in publish-subscribe systems.

- ✔ *Bandwidth, Latency* — The major disadvantages result from the static distribution tree that currently lacks support for self-maintenance. If applicable, however, such a static environment allows a higher predictability of available resources than a continuously changing and adapting overlay. Where a dynamic overlay must reallocate resources when delivery paths change, a static installation does this only when new resources are requested or old ones deallocated. If nodes are reliable and resource allocation or rerouting is expensive, this can be an interesting alternative to dynamic overlays.
- ✔ *Selectivity* — Rebeca supports arbitrary filters as subscriptions, which can provide a high selectivity and therefore low rate of false positives. On the downside, highly expressive filters may prove opaque to filter merging.

2.3.5 Choreca

Choreca [TBF$^+$03] is an attempt to map the ideas of REBECA on a dynamic overlay. It routes content-based subscriptions and notifications over a Chord broadcast tree [EAABH03] of depth $O(\log N)$. The main feature is the usage of redundant trees, one rooted in each publisher. The overlap of these trees allows to exploit filter similarities to reduce the filter forwarding overhead. Compared to the multicast based systems, Choreca avoids any single point of failure.

- ✔ *Delivery guarantees* — The path redundancy in Choreca's Chord graph lies within $O(\log N)$. To assure the eventual delivery of a notification in the case of a lower number of link failures, it is sufficient to reroute it through a different neighbour than the failed next hop. This allows Choreca to provide a very high probability for delivery.
- ✔ *Selectivity* — Choreca inherits the filter characteristics of Rebeca.

2.3.6 BitZipper

The BitZipper Rendezvous [TBF$^+$04] is a generic approach to the decentralised evaluation of data and queries. It can be implemented over prefix-routed overlay networks, including most key-based routing networks, like Chord or Pastry (see 9.2). Its overhead is within $O(\sqrt{N})$ messages, but it is independent of any specific feature of data or queries as opposed to the restrictions posed by subject-based multicast or the NP-completeness of filter merging [Müh02]. General content-based publish-subscribe filtering can be implemented with the same message overhead when using the BitZipper.

- ✔ *Delivery guarantees* — As a general-purpose solution, the BitZipper Rendezvous protocol has a high resource profile. However, the implementation on top of structured overlays provides it with global load distribution and an intrinsic forwarding redundancy that is also within $O(\sqrt{N})$. Exploiting this redundancy for the matching process can dramatically increase the reliability in the face of arbitrary node failures.

- ✔ *Selectivity* — The BitZipper Rendezvous is agnostic to the subscription language. Even Turing complete subscriptions can be supported without impacting the message overhead or the number of brokers required for filtering.

2.3.7 Conclusion

The diverse characteristics of the presented systems leave the deployer with a large grey area of tradeoffs. Each has its own advantages and disadvantages in specific situations. In the case of multicast, the best possible solution is IP-Multicast – but only if it is available in a scalable fashion. Especially a global mapping of reasonably expressive subjects to IP addresses exposes considerable administrative barriers and may quickly lead to address exhaustion. In this case, overlay multicast leverages an increased overhead to provide a higher-level approach that circumvents these problems.

Overlay internal multicast provides average $O(\log N)$ overhead and tree depth, but the single rendez-vous node may become a problem in high-traffic scenarios. Replication helps in balancing the load, but at the expense of even higher overhead in common scenarios. More general approaches, like Choreca or the BitZipper, remove the single point of failure. They combine high selectivity with global load balancing at the expense of generally increasing the overall overhead. On the other hand, the Hermes system provides similar characteristics and problems as overlay multicast systems, but increases the per-broker state to push the expressiveness of subscriptions towards the more general solutions.

Obviously, different applications have different requirements. Where one application may find a single rendezvous point acceptable as it cares more about latency and message overhead, a second one may benefit well enough from the high redundancy of Choreca to accept the additional overhead. Designing overlays that work optimally in all situations is nearly impossible and will most likely lead to extremely complex systems that handle many special cases. Understanding and evaluating the behaviour of such a system will be at least as hard as their initial design.

A more promising (and more common) approach is the development of systems that work well in well defined cases. Their characteristics can be proven or evaluated. By relying on explicitly known quality-of-service capabilities of these systems, applications can select the best candidate system for a specific problem in a specific environment. Only the integration of specialised overlays, each with its own simple design, will lead to highly configurable and adaptable systems that keep the high predictability of their components.

Chapter 3

Middleware Abstractions for Overlay Software

Contents

As we have seen so far, the support for different levels of quality-of-service in large-scale overlay applications relies on an integration of different overlay networks. This requires support in frameworks and design tools to make overlay implementations available in different run-time environments. Platform independence and high-level design support are crucial for the implementation of overlay applications that are expected to adapt to diverse requirements in highly-scalable systems.

This chapter makes a case for new abstractions in the design of overlay software based on the deficiencies of current approaches. Based on the examples in the introductory chapters, the first section presents general requirements on overlay middleware systems. Section 3.2 then reviews current frameworks and matches them with the requirements.

Section 3.3 generalises the current approaches into two main perspectives on overlay software: the *networking perspective* and the *overlay protocol perspective*. Both approaches support developers in the implementation of overlay networks. As section 3.4 shows, however, neither of them provides an abstraction level that substantially reduces the design effort by capturing the specific characteristics of overlay networks. This outcome motivates the development of higher-level abstractions, as presented in the next chapter.

3.1 Requirements on an Overlay Middleware

To provide helpful design abstractions, overlay middleware must handle, simplify and pre-implement common tasks that help in maintaining overlays in general. It can then leave the implementation of overlay specific components to the developers. This section describes general requirements on overlay middleware systems. An overview of their incarnations in different frameworks is provided in 3.2.

3.1.1 Generic, High-Level Models

First of all, overlay middleware must provide simplified, domain specific models for overlay development. This allows 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 models must support the implementation of overlay topologies and algorithms for routing and maintenance.

An important 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 different architectures, like large servers, standard PCs and mobile devices.

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

Platform independence 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.

The Node Views architecture presented in this thesis aims to provide such platform-independent models. It combines them with a Model Driven Architecture for overlay implementation.

3.1.2 Software Components

To connect the high-level design with executable code, a middleware requires a component model that enables the integration of language specific implementation parts.

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 interfaces between high-level models and specialised components. If components are written solely against the generic models provided by the middleware, they can become part of the middleware itself. This provides a common base of pluggable components and further reduces the effort necessary to design new overlays.

It is obvious that this also regards 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 to make their use a matter of selection rather than programming.

One framework that supports this layering of components is Gridkit. It provides a dynamic component architecture and layered networking interfaces for overlay networks. Section 8.4 describes an implementation of the Node Views architecture on top of Gridkit.

Reactive components

As in any networking software, the components of an overlay middleware are naturally reactive. They respond to events such as incoming or locally generated messages, time-outs and changes to the local model. The middleware must therefore manage these events and the coordination between different components.

This feature is often implemented by means of Event-driven State Machines, finite automata that interconnect processing states by event triggered transitions. 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, like SEDA [WCB01] (see 3.2.1), support processing chains that forward 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 approach reduces the complexity of each state and moves more of the control logic into the middleware architecture.

More expressive event models can push this even further. By moving more of the complexity into the model that underlies the middleware, components can further reduce their implementation dependency on specific environments and middleware implementations. EDSL (see 6.5) is an abstract EDSM language that supports modelling the event processing of overlay implementations in a framework independent way.

3.1.3 Management of State and Node Data

An important part of overlay software deals with the relation of the local node to remote nodes. Overlay middleware must obviously provide support for managing this relation. This comprises handling the connections to neighbours in the topology, adding them to the local view and removing them, but also keeping fall-back candidates for the maintenance case. Nodes often have to store further state about their neighbours, such as running time-outs, measured latencies or active subscriptions. The ubiquity of state management throughout the system calls for support in middleware. The database provided by the Node Views architecture (see 4.3) aims to become such a central, integrative point of state management.

A further step towards a high-level middleware is to manage nodes not only on a connection level but to provide support for selecting neighbours and communication partners based on various criteria. This provides an abstract base for the respective local decisions that are currently implemented by hand, outside of frameworks. SLOSL, as presented in chapter 5, is a domain specific language that provides such a middleware abstraction level.

3.1.4 Integration and Resource Sharing

The last major feature of a common middleware is the integration of different overlays. In an integrative middleware, multiple applications can run in the same environment and each application can deploy multiple overlays. This requires the middleware to provide ways for reducing their aggregated resource usage. This encourages connection sharing between different topologies and applications as well as ways to minimise the general maintenance overhead that is generated, for example, by pings[1]. The Node Views approach, that this thesis proposes in chapter 4, aims to provide an integrative data layer for local decisions based on locally consistent views.

3.2 Frameworks and Middleware for Overlays

There have been a number of recent proposals for overlay frameworks and middleware. Macedon, iOverlay and RaDP2P are under development and evaluation in the corresponding projects. Other frameworks, like SEDA or JXTA, have also been used for overlay implementations.

Figure 3.1 compares the size[2] of some of these frameworks (as far as they are publicly available). As we will see in section 3.4, the code size of a framework

[1]PlanetLab applications (`http://planet-lab.org/`), as an extreme example, were found to generate a total of up to 1GB of ping traffic per day in 2003 [NPB03]

[2]Note that these numbers depend on which modules are counted as major (i.e. relevant) parts of the frameworks.

Framework	Type	Size
JXTA	Protocol framework	1.5 MB jar (+XML+...)
SEDA	EDSM framework	15-30,000 lines of Java
Macedon	EDSM/overlay framework	15,000 lines of C++

Figure 3.1: Approximate code size of overlay frameworks

is in no relation to the size of overlay implementations written on top of them. It is clearly the abstraction they provide that makes the difference.

3.2.1 SEDA

The *Staged Event-Driven Architecture* (SEDA [WCB01]) is a framework for scalable Internet servers. It provides an event-driven state machine (EDSM) for request processing that is used in OceanStore [KBC$^+$00] and (in a modified form) in Bamboo [RGRK04]. While designed as a generic architecture, the only available implementation is written in Java.

SEDA handles network I/O and builds an abstraction layer for message processing in reactive state components. It does not provide any higher level overlay models, language independence or support for node management. State transitions are specified by class types and dispatched in source code, which makes state implementations hard to reuse.

3.2.2 JXTA

The *JXTA* project[3] was started by Sun Microsystems in the year 2000. It builds a protocol framework for peer-to-peer applications. The main focus are unstructured broadcast networks and it provides means for connecting and grouping nodes, discovering services and dispatching messages.

There is no support for node management or for topologies and their design. The ancestry of a broadcast model makes writing structured overlays like HyperCUP [SSDN02] tedious work (see comparison in figure 3.4, page 40).

3.2.3 iOverlay

iOverlay [LGW04] 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. The overlay implementation then forwards incoming messages between them. This approach provides an abstraction level that simplifies the design of overlay algorithms by hiding the lower networking

[3] `http://www.jxta.org`

levels. It also has the advantage of being an integrative approach that allows resource sharing.

On the down side, the connection based model provides only basic support for node management and no higher-level model for the implementation of topologies. Furthermore, the router is a monolithic component in the middleware model, which leaves the concern of code reusability entirely to the developer.

3.2.4 RaDP2P

RaDP2P [HCW04] is described as a policy framework for adapting structured overlays to applications. User defined policy components 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. This makes it hard to estimate the support for the before mentioned list of middleware requirements.

The idea behind RaDP2P provides an extremely high-level model that abstracts from specific overlays. However, as it stands, RaDP2P does not provide support for implementing the overlays themselves. It should therefore rather be considered an application level framework.

3.2.5 Macedon

Macedon [RKB$^+$04] constitutes 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 based on C++ from which it generates source code for overlay maintenance and routing.

The Macedon EDSM provides a high-level, domain specific model for reactive overlay implementations. On the down side, Macedon's support for node management is as limited as in iOverlay. It does not tackle the problem of overlay integration and, as the frameworks above, it does not provide any further models for topology implementation. Its language dependence on C++ is certainly a further drawback. Still, the general approach could be adapted to generate source code for different environments, e.g. for debuggers and simulators.

3.2.6 What is missing in current frameworks?

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, as the models in current frameworks seem to suggest. Each node in an overlay needs to take local decisions. The sum of these local decisions is the distributed algorithm that maintains the overlay. What are these local decisions based on?

iOverlay bases them on the currently available connections and establishes a switch that routes messages between them. It does not provide means to select the "right" connections for specific requirements or to distinguish between different types of connections. It does not support any node management that could simplify these decisions.

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 any node management for selecting connections. Topologies only arise "accidentally" from the source code level implementation of message handling decisions. There are no higher-level models to support the decisions that are necessary to establish them.

This overview shows that the implementation of message protocols is the dominating model in current frameworks. Event-driven state machines form the state-of-the-art in the design of overlay software. They support the implementation of reactive components that handle messages. They do neither support the implementation or integration of different topologies nor the node management that is necessary for the local decisions to become meaningful.

3.3 Current Views on Overlay Software

Overlay software has been implemented on different abstraction levels, ranging from low-level sockets to high-level protocol models. Taking a look at the currently available systems, we can find two basic abstractions in their design: the low-level networking layer and the overlay routing layer. Let us now look at each of them to see what overlay developers have to face when building their software on top of these abstractions.

3.3.1 The Networking Perspective

At the lowest level, we find the most ubiquitous abstraction for networking software: **sockets**. They provide ways to send raw blocks or streams of bytes between application end-points on different machines. There can hardly be an operating system that does not provide them as a programming primitive. Their use has been largely standardised through Unix-flavoured operating systems and POSIX.1g. While they provide the most portable abstraction, their

↑ *Overlay Application* ↑	
↑ *Overlay Implementation* ↑	
Message Processing	Flow-Graphs, EDSMs, ...
Message Passing	Serialisation, CORBA, ONC-RPC, SOAP, ...
Network I/O	Sockets, TCP/UDP, ...

Figure 3.2: Overlay software from the networking perspective

help for overlay development is extremely limited. Still, there are overlay systems that were implemented directly on top of sockets, like the original Chord implementation [SMK+01].

The **message passing** abstraction allows software to send well-defined data items between end-points in a physical network. A large number of distributed applications build on top of this abstraction. Consequently, there is a remarkable set of tools, ranging from libraries for remote procedure calls (RPC, such as Sun/ONC-RPC or SOAP) over language support for serialisation as in Java or Python, up to messaging frameworks as in CORBA or JMS. Most of these have been ported to a large number of platforms and therefore form an acceptable programming level that hides data issues and platform heterogeneity. For the specific requirements of overlay network development, they still provide a very low level.

A number of overlay networks, such as Bamboo [RGRK04], are implemented on top of generic event-driven state machines for Internet servers. They facilitate **local message processing** by connecting components to request processing chains and triggering their execution. Examples are SEDA [WCB01] or the Twisted-Framework[4]. While these frameworks do not offer support for overlay specific tasks, they form a reasonable abstraction level for message-driven networking software in general.

3.3.2 The Overlay Protocol Perspective

A more interesting abstraction level for overlay development is based on the actual requirements that overlays have. They must route messages between nodes, maintain their topological structure and adapt to application needs.

During the year 2004, two interesting frameworks appeared that follow this approach, namely Macedon [RKB+04] and iOverlay [LGW04] (see 3.2). They visibly raised the abstraction level that was available at the time and thus show the advantage of designing frameworks specifically for overlay implementation.

As the major functionality in this abstraction level, **Overlay routing protocols** implement the local routing decisions for scalable end-to-end message

[4]`http://www.twistedmatrix.com`

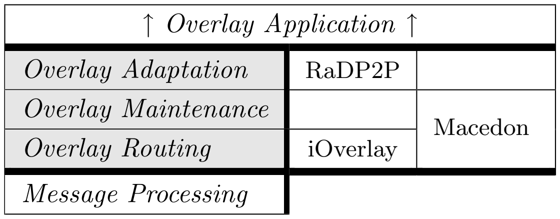


Figure 3.3: Overlay software from the overlay protocol perspective

forwarding. They are distributed algorithms, executed at each member node, with the purpose of forwarding messages at the overlay level from senders to receivers.

Both Macedon and iOverlay take useful approaches here. iOverlay provides a simple message switch abstraction that largely reduces the programming overhead when writing a router. Macedon takes a more sophisticated approach. Its EDSM is controlled by an overlay specific language that supports the complete protocol implementation.

Apart from the forwarding feature that is obviously crucial for overlay networks, routing protocols also have a number of implicit semantics. As a requirement for their algorithm, they usually define the valid neighbours for each node that assure the correctness of the routing. This leads us to **Overlay Maintenance**, the perpetual process of finding these neighbours and replacing failed ones.

Usually, maintenance and routing are tightly integrated through their interdependency. The same holds for the third functionality, **Overlay Adaptation**. As we have seen in 3.2, the approach of RaDP2P is targeted rather at applications than at overlay design. This means that there is currently no actual framework support for the implementation of adaptable overlays, although most of the existing overlay systems provide some form of adaptation. It is usually implemented as an optimisation of the overlay topology based on a specific metric, most commonly the hop-by-hop latency. Whenever the choice of neighbours leaves a certain degree of freedom (which is common case in large-scale systems), a specific adaptation algorithm is executed to find the best candidate.

3.4 Code Complexity – why is this not enough?

Figure 3.4 compares a number of recent overlay implementations with respect to their code size. All of these were written by hand in a general purpose programming language. It shows that the expected code size that results from this approach goes beyond 20,000 lines of code. It also shows that the use of inappropriate frameworks can result in considerably more code being written. Especially JXTA appears to fail entirely in its goal of simplifying the design of overlay applications.

Overlay	***Environment***		***Lines of code***	
	Framework	*Language*	*by hand*	*Macedon*
Chord	sockets	C++	8,000	500+
Pastry	sockets	Java	16,000	1300+
Bamboo	SEDA	Java	20,000	-
Tapestry	SEDA	Java	23,000	-
HyperCUP	JXTA	Java	30,000	-

Figure 3.4: Code size of well-known overlay implementations

As suggested in section 3.3.1, all of these implementations are very similar in their main components, so one can expect positive synergy effects when they are implemented on top of a common framework. This is one of the reasons why Macedon's implementations are considerably shorter. Note, however, that they are not equivalent in terms of features and that the small number of comparable overlays does not yet support any extrapolation. It would therefore be wrong to commonly expect an order of magnitude here.

Macedon's main advantage is the higher abstraction level as compared to the networking layer. On the down side, however, it replaces the challenge of writing low-level networking code by the new challenge of writing an entire overlay implementation as an event-driven state machine. This has several drawbacks.

While non-blocking state-machine programs are generally considered more scalable than their major competitors, threaded programs, they are also much harder to write and understand. State-machines require a fundamentally different and very specific approach to code modularisation. They split code into processing stages based on intermediate I/O instead of semantic vicinity. Macedon supports neither visual development nor any higher-level encapsulation of semantically close code. So the gain of a factor 10 in code size is payed for with a considerable increase in complexity and a reduced readability of this code.

Another problem with Macedon is its tight coupling to the C++ language. Porting a Macedon implemented overlay to a new programming language basically involves reimplementing both the overlay and Macedon itself.

A more general problem is the event model of current event-driven state machines. It is commonly based on messages as monolithic blocks of data that event handlers must handle in their entirety. There is no support for more fine-grained subscriptions that could allow more generic, reusable event handlers.

It should also be stressed that support for overlay software functionality is not currently complete. Writing overlay software in an adaptable way must still be done by hand. Even Macedon does not provide any help here, despite the fact that adaptation is a very important feature for overlay applications, especially if they need to be quality-of-service aware.

As we have already seen, the overlay protocol approach enforces the plain source code implementation of a number of implicit semantics that largely impact the entire overlay. This regards routing protocols, maintenance algorithms and adaptation. It therefore becomes hard to understand this kind of overlay software without further documentation – external to the implementation.

This is especially true for the topology, the most important characteristic of the overlay. It is not explicitly described in any single software component, but scattered throughout the implementation. EDSM implementations are the extreme variant, as they structure their code entirely based on overlay specific I/O. This makes it hard to reuse parts of the overlay software for different overlays. It also hinders the integration of different overlays and their combined usage in a single application. This currently makes overlay implementations a rather monolithic piece of software.

Chapter 4

A Model Driven Architecture for Overlay Implementation

Contents

The *Model Driven Architecture* [MM03] is a software design approach proposed by the Object Management Group (OMG[1]). It aims at enforcing the power of abstract models over platform specific implementations. MDA applications are designed in platform independent models (PIM) that are then transformed into platform specific models (PSM) and finally augmented with platform specific code to provide a complete implementation of a system. It is therefore a much more high-level and comprehensive technique than a middleware design approach. The OMG's MDA is based on the Unified Modelling Language (UML [Obj01, Fow04]) and thus largely targeted at business logic and enterprise-level software design.

This chapter will derive and present models that enable a Model Driven Architecture approach in software design for overlay networks. The next section tries to straighten the perspective on overlay software by introducing the topology itself as a design target. This leads to a novel approach, named Node Views, that is presented in section 4.3. Its major advantages are the decou-

[1]`http://www.omg.org/`

pling of generic maintenance components and the inherent support for explicit topology specifications as the central part of the implementation.

4.1 The Topology Perspective

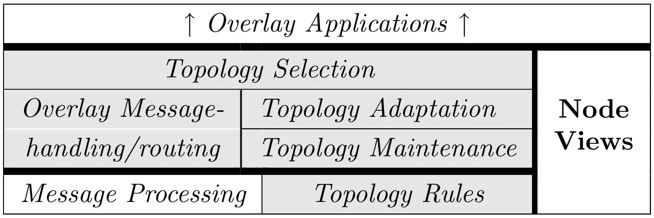


Figure 4.1: Overlay software from the topology perspective

While current frameworks focus on message forwarding and the protocol design part of overlay software, it should have become clear that these abstractions stay at a relatively low level. Even the EDSM approach forces the developer to design code in a very specific way that is tightly coupled to a framework and leaves source code too monolithic for reuse. We will now raise the perspective to the design of topologies that establish the global characteristics of overlay networks. From this perspective, we can establish five major functional levels in overlay software, as shown in figure 4.1.

4.1.1 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 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 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. The Node Views architecture proposed in this thesis is the first to move these rules into machine readable overlay models. This makes them available for automated model transformation and analysis.

4.1.2 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 amongst current overlay frameworks, despite its obvious importance for the required self-maintenance in these systems.

4.1.3 Topological Routing

Routing captures the local decision to which of the neighbours a message should be forwarded. The goal is to deliver it to the final destination or to bring it at least one step closer. The routing algorithm exploits the topology rules to determine the best next recipient. 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.

4.1.4 Topology Adaptation

Topology adaptation is the ability of a given overlay topology to adapt to specific requirements. As opposed to the error correction done by 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 that redundant entries in the routing table can be exploited for adaptation to achieve better resilience and lower latency [ZHD03]. Topology adaptation usually defines some kind of metric for choosing new edges out of a valid set of candidates. Building the "right" sub-groups of nodes in hierarchical topologies (or embedded groups as in [KR04]) also fits into this scheme.

Most 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 some expected quality of query results. The respective metric determines the node ranking which in turn parametrises the global properties of the topology.

4.1.5 Topology Selection

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

4.1.6 Discussion

While overlay networking stays an important part of the implementation, the topology perspective allows developers to take a broader and more abstract view on the design. It focuses on the major characteristic of overlay networks: their topology. This provides a cleaner view on the intentions of overlay implementations.

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 to have an application maintain several overlays if each of them independently sends pings to determine the availability of nodes. An integrative approach is needed to avoid this kind of overhead.

The following section motivates the background for Node Views, an integrative, high-level approach to overlay software design, that is based on the topology perspective.

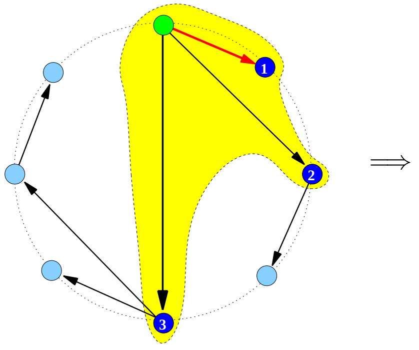
4.2 Background: The Local View of a Node

In scalable, distributed overlays, the global topology and all properties of the overlay result from the sum of local actions of its participants. All actions must be based on local decisions that each single node takes depending on its local view. 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.

4.2.1 The Chord Example

We take Chord [SMK+01] as an example, a structured overlay network based on a ring topology (see 9.2). The top node of the ring in figure 4.2 sees the nodes within the shaded area.

The local view of this node is 1) the successor link, i.e. the direct clockwise neighbour, plus 2) the finger table, a list of nodes further away along the ring that are chosen according to an overlay specific rule. For better resilience, Chord actually keeps a list of successors.



successor	Node 1
finger table	Node 2
	Node 3
	:

Figure 4.2: The Chord example - from global to local view

Two other structured overlays, Pastry [RD01] and Tapestry [ZKJ01], assemble their local view in a similar fashion. They keep a list of direct neighbours and a routing table with members that are further away. The rule for choosing these nodes is similar in both systems, but differs from the rule used in Chord.

Unstructured networks also keep a set of neighbours that are chosen based on attributes known about them. These may include network latency, uptime, node degree, available resources, data references or categories about content stored on them. Super-Peer networks [YG03] form a variant that keeps two distinct sets of peers and super-peers.

In all of these overlay networks, each participant sees a number of other members through specific data structures that represent these neighbour lists, successor lists, routing tables, leafsets, multicast groups, etc.

Abstracting from specific implementations, we can now see that each of these data structures provides a node with a partial view on the network. What we previously called the local view of a node is therefore the union of all its partial views on remote nodes.

4.2.2 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 questionable 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 7.2 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 inconsistencies 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. The Node Views approach provides such an abstraction.

4.3 The System Model: Node Views

The Node Views approach instantiates the local view of a node as a local database that keeps all locally known data about remote nodes. This allows 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 rest of the architecture, that is shown in figure 4.3, results from following the well-known Model-View-Controller pattern [BMR$^+$96]. It aims to decouple software components by separating the roles for state and data storage (model), data presentation (views) and data manipulation (controllers, named harvesters in the Node Views context).

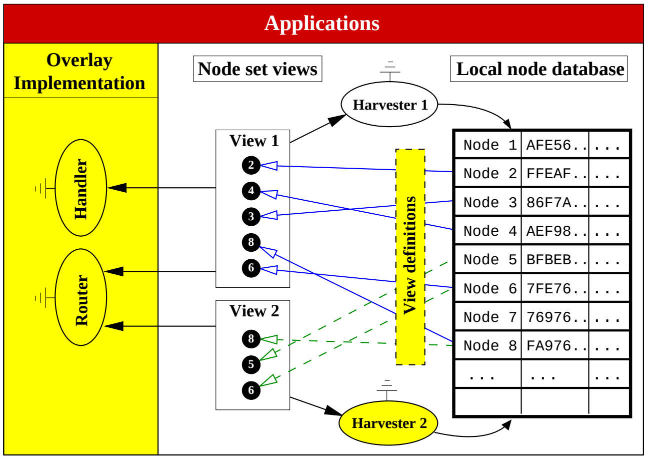


Figure 4.3: The overall system architecture of Node Views

4.3.1 Nodes

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. There are physical and logical attributes.

Logical attributes include the node identifier used in structured networks. Similarly, related or precomputed data like distances between identifiers can be stored in attributes. Besides that, an application may add semantic information about data stored on that node or references to content filters associated with it.

The most important *physical* attribute is the network address of the node, but others, like measured or estimated latency and throughput, may also be of interest. Note that this model abstracts from the source of such attributes and thus unifies estimates and measurements. This allows to exploit both in view definitions and to make them exchangeable as needed, without any impact on the implementation of components that use this information.

When using multiple (logical and physical) identifiers at runtime, they can potentially become inconsistent. This may happen if two different nodes have taken the same logical identifier by accident or if a node failed and a new one took over its physical address with a different logical identifier. These problems have a number of possible solutions, which are normally implemented as part of the overlay protocol. The Chord overlay, for example, tests for duplicated logical identifiers in its join procedure and repeats the join until an unused identifier was found. In general, the right solution is very overlay specific. Therefore, this kind of inconsistency cannot be prevented or resolved automatically by an overlay-agnostic middleware layer and sometimes not even locally between neighbours. The runtime system should therefore simply trigger a database event if any inconsistency is detected and otherwise leave the resolution to the overlay implementation.

Further physical attributes regard the communication status of a node. As described before, it is generally of interest if a node is currently considered alive and when it was last contacted (time-stamps or history). A component that pings nodes if they do not respond to messages within a certain time interval can set a boolean node attribute when starting this action. A routing component can then take the decision to temporarily route around that questionable node in order to proactively keep the hop-by-hop loss rate low. Note that this kind of information is independent of any specific overlay and can therefore be shared between different running overlays to minimise the amount of outgoing pings.

4.3.2 Node Database

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. All components in the system can easily contribute and benefit from it. 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.

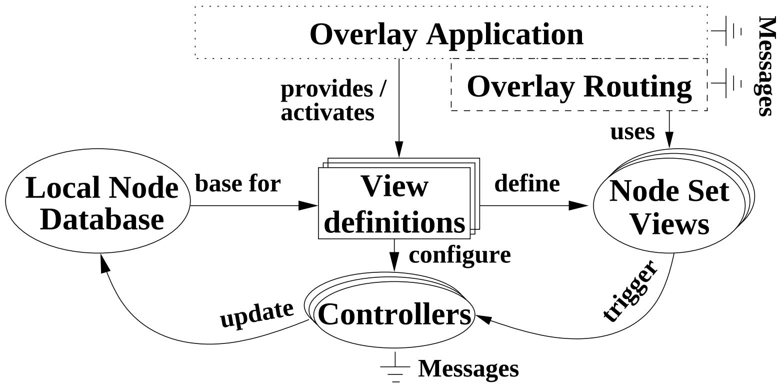


Figure 4.4: View components in the Node Views architecture

When an overlay application terminates and is restarted, it has to reconnect to the overlay. If the local node database is made persistent, it can directly be used to find good candidates for joining. A (partially) persistent node database provides better support for this than a simple, semantic-free list of network addresses as currently used in most overlay implementations.

The high value of a consistent local view for making local decisions makes nodes and the node database central components of an overlay middleware. They store information that is relevant to all other components and act as a knowledge base for the local decisions of the overlay implementation and the running applications.

4.3.3 Node Set Views

Node set views represent the views that are specific to a topology[2]. 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.

We can extract node set views from current overlays in many ways. Unstructured overlays usually do not characterise their neighbours and keep only a single set of nodes based on a pre-defined ranking. In a similar way, Super-Peer networks [YG03] use two distinct sets of peer nodes and super-peer nodes.

Most structured overlays are based on two or more distinguished node sets. As shown for the Chord example in 4.2.1, the topologies of Pastry [RD01] and Tapestry [ZKJ01] also use two node sets: leafset and routing table. Structured

[2]Note that a single topology can have multiple views, such as the leaf-set and routing table found in Pastry [RD01], or different hierarchical levels as in super-node networks.

overlays based on de-Bruijn graphs (9.2.5) have only one node set. It contains the direct neighbours in the graph.

4.3.4 View Definitions

The subset of nodes that are available through a view 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 4.4. The view definition language SLOSL, that was developed for this architecture, is presented in chapter 5.

4.3.5 View Events

As in any networked system, overlay components are naturally reactive and respond to changes in the local view. View events represent these changes. 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 [DBM88, Pat99]. They trigger reactive components of the software when data changes occur. Views generate and filter notifications and software components receive only those from the views they see. Event-triggered components are a common idea in software design, where it is best known as the observer pattern [BMR$^+$96]. It is also related to publish-subscribe systems (see 2.1). The view events supported by the SLOSL view definition language are described in section 5.5.

4.3.6 Harvesters and Controllers

Harvesters are a major part of the overlay maintenance implementation. They aim to update the database according to the view definitions. 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 enables their implementations as generic components in frameworks. Harvesters are further discussed in chapter 7.

From an implementation point of view, harvesters are mainly a design concept. MVC terminology uses the term controller (as in figure 4.4), which is any component that modifies the system state. A harvester in the Node Views architecture is a special controller or a complex composition of controllers that targets a specific functionality in an overlay network. The term harvester will be used wherever complex actions such as replacing nodes on failure or repairing views is involved.

4.3.7 Routers and Application Message Handlers

Other overlay components like message handlers or routers depend on node set views. They need node data for their decisions when accepting, routing, or otherwise handling messages. Structured overlays base this decision mainly on the ID assigned to a node. However, there is a substantial amount of research dealing with better decisions for more efficient overlays. These decisions are based on latency measurements and estimates, path redundancy, node histories, node capabilities, etc.

Database and views support these decisions by providing locally consistent views on the nodes and pluggable, generic harvesters that maintain them. They decouple message handlers from maintenance components and reduce their implementation to very specialised components that become pluggable themselves.

4.3.8 Discussion

The architecture uses the Model-View-Controller pattern (database – node set views – harvesters) to split the previously monolithic implementations of overlay software. The resulting System Model simplifies and decouples 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 component has to be written from scratch, it benefits from consistent, active node set views and the data-driven decoupling from other components.

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 chapter. 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.

Chapter 5

The SLOSL View Definition Language for Overlay Topologies

Contents

This chapter presents SLOSL, the SQL-Like Overlay Specification Language. It was specifically designed for topology view definitions and extends the SQL database query language to support the special semantics of topology rules and topology adaptation. The additional semantics also form the basis of overlay routing, as section 5.4 explains. The clear design goals of SLOSL were a high abstraction level and short, data-driven expressions.

Section 5.1 exemplifies how the different clauses of SLOSL interact for creating views and selecting nodes, based on the Node Views architecture described in the previous chapter. The subsequent section 5.2 describes the grammar in detail. Section 5.3 provides some more examples for SLOSL statements that implement different topologies. The next section (5.4) then outlines how SLOSL views are used to automate routing decisions. The final section 5.5 explains the different event types that the active SLOSL views provide, and that the Node Views architecture exploits to trigger controllers.

5.1 SLOSL Step-by-Step

This section describes the different clauses of SLOSL by evaluating an example step-by-step. It aims at providing a clear idea of the general execution semantics of SLOSL statements.

It is important to note beforehand that this scheme is only one way of evaluating these statements. As a declarative language, SLOSL does not provide any guarantees about the order or time of evaluation of its clauses or expressions in execution environments. See section 8.3 for an introduction to possible implementations and optimisations.

Here, our running example is a SLOSL specification of the Chord graph as presented in the original publication [SMK+01]. See 4.2.1 for a brief description of the Chord topology or 9.2 for the general concepts behind this kind of overlay.

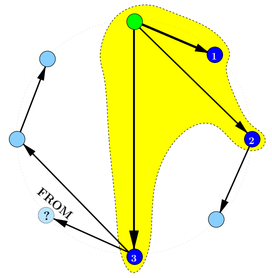
```
1  CREATE VIEW chord_fingertable
2  AS SELECT node.id
3  FROM node_db
4  WITH log_k = log(𝒦)
5  WHERE node.supports_chord = true AND node.alive = true
6  HAVING ring_dist(local.id, node.id) in [2^i, 2^{i+1})
7  FOREACH i IN [0, log_k)
8  RANKED highest(1, ring_dist(local.id, node.id))
```

As known from SQL, the first 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. No other attributes are presented, although others may be used in the view specification itself.
3. This line states the super-views of the newly created one. In this case, it is a direct sub-view of the local node database.

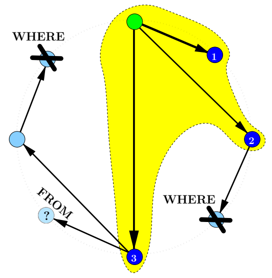
The next thing to note is the WITH clause in line 4. It defines variables or options of this view. They can be set at instantiation time and changed at runtime. In the example, the variable *log_k* is given a default value, the logarithm of the size of the key space. It is used by Chord to determine the length of node identifiers. For the Chord topology, this is a global constant that must be fixed at deployment time.

The following figure presents the local view on the topology that we are constructing in this view. It results from the partial evaluation of the SLOSL clauses up to this point.



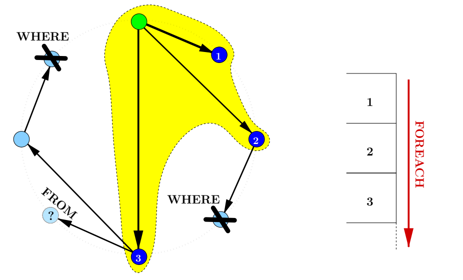
CREATE VIEW – **AS SELECT** – **FROM**

As can be seen in the bottom-left corner of the graph, there is currently only one node missing in the local database. This may mean that the local node has not yet heard of it or has already removed it from the database due to space constraints. In large overlay networks, the database will usually only contain a small subset of all nodes, most notably those that are visible in the locally active views. Previously known nodes that have gone out of sight may be removed based on an implementation specific garbage collection or caching policy (see 7.3).



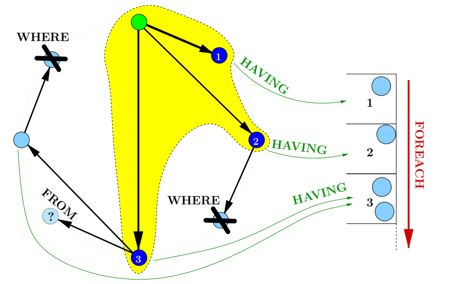
WHERE node.supports_chord AND node.alive

The next step is to evaluate the WHERE clause. It constrains nodes that are considered valid candidates for this view (node selection). Here we exploit an attribute named *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 harvesters deemed alive. As the figure shows, there are two nodes that are rejected by the boolean expression because the local node assumes them to be dead.



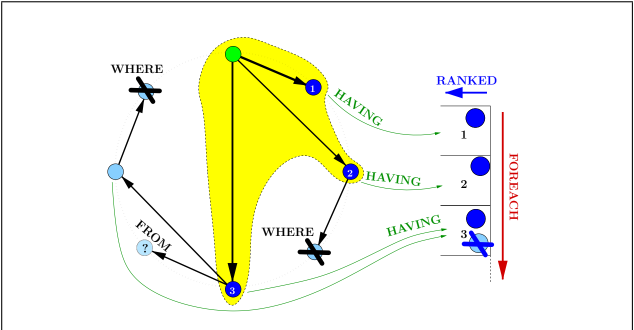
FOREACH i **IN** [0, log_k)

We can now evaluate the FOREACH clause. Its purpose is to describe a set of buckets, the *node categories* or node equivalence classes. They are enumerated by the values of the run variable i in the example. If more than one FOREACH clause is given, each combination of values references one of the buckets.



HAVING ring_dist(local.id, node.id) in $[2^i, 2^{i+1})$

Having determined the available buckets, we can start evaluating the HAVING expression for each node and each bucket. If it evaluates to true, we store the node in the bucket. In the example, the HAVING clause states that the result of the Chord distance function must lie within the given half-open interval (excluding the highest value) that depends on the bucket variable i. Here, the name *ring_dist* refers to a user-provided function. However, the distance is locally a constant which could just as well be stored in a node attribute.



RANKED highest(1, ring_dist(local.id, node.id))

Finally, the nodes are chosen from each bucket 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. It selects nodes from each category that match specific criteria best.

5.2 Grammar

This section describes the different clauses of SLOSL in more detail. It defines 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, while *bool-expression* refers to an expression that returns a boolean value (true or 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 parts:

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

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

5.2.1 CREATE VIEW

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

This clause creates and identifies a view.

5.2.2 SELECT

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

This clause defines the interface of nodes in this view. Only the attributes listed here can be seen by software components that use the view. However, all attributes found in the parent views can be used for the view definition. The expression in the last line is a common short form for $attrib = node.attrib$, where the new attribute name is identical with the original one.

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.

5.2.3 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.

5.2.4 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.

5.2.5 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.

5.2.6 HAVING . . . FOREACH

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

This clause aggregates nodes into buckets. It implements the node categorisation of the topology rules. In its simplest incarnation, 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 absence of any FOREACH clause, the view will only have a single flat bucket containing all visible nodes.

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 optional HAVING expression (which defaults to true).

A node can appear in multiple buckets of the same view if the HAVING expression allows it. If bucket variables are used in the view attribute definition (SELECT), the same node can carry different attributes in different buckets of the created view (see the Pastry example in 5.3).

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 sequence of values. 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.

5.2.7 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 chapter 3, 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 this 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 also 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 (RANKED) 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.

One interesting property of the Chord example is that the ranking function is independent of the bucket where it is evaluated. It only depends on the attributes of the node itself and can be calculated in advance. While this is not the case in all topologies, it is an obvious optimisation where available. See section 8.3.2 for a discussion of possible optimisations.

5.3 More Examples

This section briefly presents a number of topology implementations in SLOSL. The diversity of the examples motivates the broad applicability of SLOSL to the design of various different types of overlays.

5.3.1 De-Bruijn Graphs

De-Bruijn graphs [dB46] 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 [LKRG03, DMS04].

```
CREATE VIEW de_bruijn
 AS SELECT node.id
FROM node_db
WITH max_id=2^64 ,  max_digits=8
```

```
WHERE node.alive = true
HAVING node.id = (local.id * max_digits) % max_id + digit
FOREACH digit IN (0:max_digits)
```

5.3.2 Pastry

Pastry [RD01] 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.

```
CREATE VIEW pastry_routingtable
AS SELECT node.hex_id, node.id
FROM node_db
WITH max_digit=16, max_prefix=40
WHERE node.alive = true
HAVING digit_at_pos(node.hex_id, prefix) = digit
   AND prefix_len(local.hex_id, node.hex_id) = prefix
FOREACH digit IN (0:max_digit)
FOREACH prefix IN (0:max_prefix)
RANKED highest(1, ring_dist(local.id, node.id))
```

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* option makes the number of neighbours configurable at run-time.

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

5.3.3 Scribe on Chord

Scribe [RKCD01], as presented in section 2.3.1, is a multicast scheme that was initially implemented for the Pastry overlay. It is, however, generally applicable to key-based routing overlays (see 9.2), which allows its implementation also on top of Chord. Scribe essentially exploits the key mapping provided by the overlay network to determine a rendezvous node for each multicast group. It then sends group messages towards the rendezvous, which serves as the root of a multicast tree. Subscriptions 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 models 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
```

5.4 SLOSL Routing

The SLOSL view definitions allow for a simple extension towards a complete routing strategy. As the Pastry specification in 5.3 exemplifies, most overlay systems base their routing decisions on more than one view. They commonly use different views for semantically close and far neighbours, such as a neighbour table and a routing table.

When a router looks for the next hop for a given destination, all it has to do is test the relevant views in a sensible order to see if such a node exists. 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. If at least one bucket is found, its RANKED clause is executed to determine the best target.

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 normally be in. The category determines the local bucket of equivalent nodes.

Furthermore, there is nothing that prevents the router from first querying the 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 be used to reduce the latency towards certain nodes that the local node is not normally connected to 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 incorporates all three types of message forwarding in a general broadcast to all targets that result from the evaluation of a view.

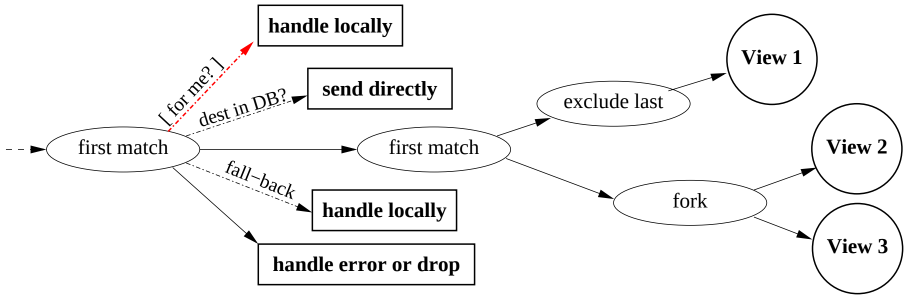


Figure 5.1: General EDGAR graph for SLOSL routing

Figure 5.1 shows a general decision graph for SLOSL routing. In general, a graph of this kind is all that is required to fully describe routing components. The language used to express this graph is called EDGAR (see 6.3), Extensible Decision Graphs for Adaptive Routing. Within the graph, messages arrive from the left and are routed as follows.

The *first match* edges simply traverse their children in top-down order. 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. Such a predicate is 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 in the lower part 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 case 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, routing continues with the two fall-backs at the bottom.

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 last hop to be identifiable, which is the case in most physical networks. If it is not available from the physical layer, this information can always be explicitly provided within higher-level message headers. When used in combination with a first match edge, the last hop is always discarded before testing the target set for success.

5.4.1 Examples

Specific routing strategies typically have simpler graphs than the one above, as they do not exploit all possibilities. The Chord routing algorithm, for example, is specified as in figure 5.2.

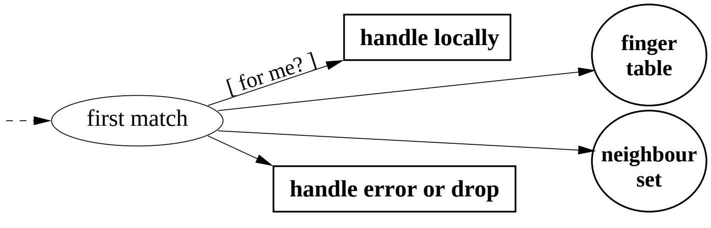


Figure 5.2: Chord routing, implemented using SLOSL views

However, even more complex routing strategies, like the routing of multicast messages over a Chord graph (similar to the Scribe approach), become easily understandable when expressed using SLOSL routing graphs. Figure 5.3 shows the complete implementation for the forwarding of publications. The example of implementing Scribe on top of the Chord overlay based on the Node Views architecture is discussed in more detail in section 8.4.

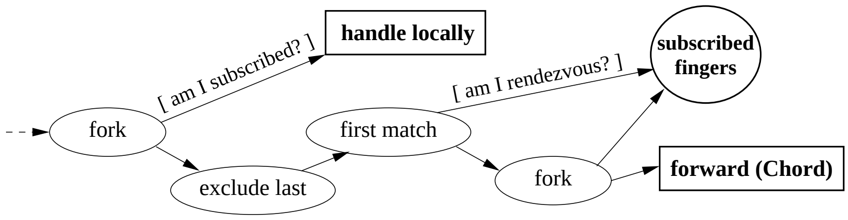


Figure 5.3: Scribe routing over Chord, implemented using SLOSL views

5.4.2 Iterative and Recursive Routing

As stated before, implementing EDGAR in a recursive routing scheme is straight forward. Each node receives the message, takes its decisions and forwards it to the next hop. However, it is also possible to implement EDGAR in iterative routing schemes. Here, the source node repeatedly asks the next hop for its successor, until it finds the destination node and sends the message directly. Both schemes are displayed in figure 5.4.

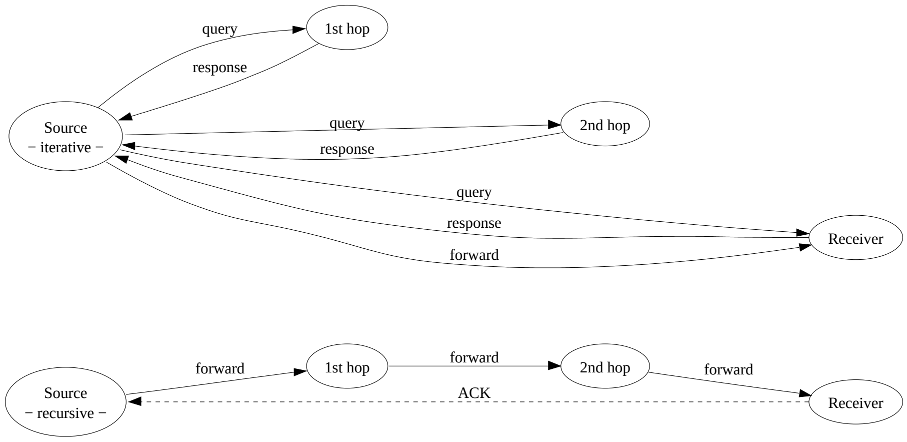


Figure 5.4: Iterative and Recursive Multi-Hop Routing

The two routing schemes have different quality-of-service characteristics. In the iterative scheme, the source knows all hops and can deploy arbitrary policies to decide if it trusts them and what it sends them. As opposed to recursive routing, it has immediate feedback about the routing delay and about node failures along the path. Obviously, the message overhead of the sender is substantially higher.

In recursive routing, the message overhead is the same for each intermediate node (unless recursive ACKs are used) while the sender is relieved from the main work. If intermediate node failures occur, the delivery can only be guaranteed if additional ACKs are in place. This also implies that multi-hop forwarding can introduce arbitrary forwarding delays that are not controllable by sender or receiver. An advantage for overlay networks is that the forwarding path follows the topology, which can be optimised for that purpose (topology adaptation).

Given the different characteristics of both schemes, it can be interesting to configure the forwarding scheme as part of the implementation. This provides an additional level of adaptation to application requirements.

The evaluation of Edgar results in a set of nodes. However, the node that takes the routing decision is not required to have the complete message available. It only needs to have the node data about the destination that is required by the graph. The sender can create an incomplete node representation and send it to a remote node to request its Edgar evaluation. The remote node responds with the resulting set of nodes and thus allows the sender to continue its traversal of the topology.

5.5 Event Types of Slosl Views

The Node Views architecture is largely event-driven. The events of incoming and outgoing messages are handled by harvesters, controllers and routers. For maintenance, however, the most important event source are Slosl views. They generate events for nodes appearing in or disappearing from views, or for views or buckets that turn empty. This section overviews the different types of Slosl events that emanate from nodes, buckets and views.

Nodes. The simplest form of event is a node event. Nodes can *enter* or *leave* views, and their attributes can be *updated*. The latter regards either the attributes stored in the database or those calculated in the SELECT clause of Slosl statements. Note that the update of a node attribute can trigger enter or leave events in dependent views after their re-evaluation.

Buckets. At the next level, buckets can turn *empty* when their last node leaves or can become *non empty* when a node enters a previously empty bucket. For views with a RANKED clause, buckets can also produce *full* and *not full* events respectively, when the number of nodes reaches the size of the bucket or falls below it.

Views. Being a set of buckets, views support the same event types. They can become *empty* when the last node from the last non empty bucket leaves or *non empty* when a node enters a previously empty view. Additionally, views become *full* if a node enters the last empty bucket, or *not full* when all buckets of the view were non empty and one of them turns empty. Note that a full view does not require all of its buckets to be full. The full event therefore means that the view no longer contains any empty buckets. The case where all buckets of a view get filled up triggers the *buckets full* event.

The descriptions above make it clear that there are dependencies between the event types. They are displayed in figure 5.5. Note that the graph shows different ways of generating *view (not) empty* events, either from bucket events or node events (dashed lines). Implementations can decide which one is more efficient for them. They may let this depend on the events used in an implementation. If no bucket events are used, it is likely more efficient to generate these events from node events directly.

There is an edge case where generating these events is not obvious from the graph. As buckets can have a bounded size through the RANKED clause, a node update can replace a node in a bucket. This will generate enter and leave events for the impacted nodes. However, if the size is unchanged after the update propagation, this operation must not generate bucket events or view events in this case. From the point of view of event generation, each view update should therefore be considered an atomic operation. Although this is rarely hard to achieve in practice, implementors must be aware of it.

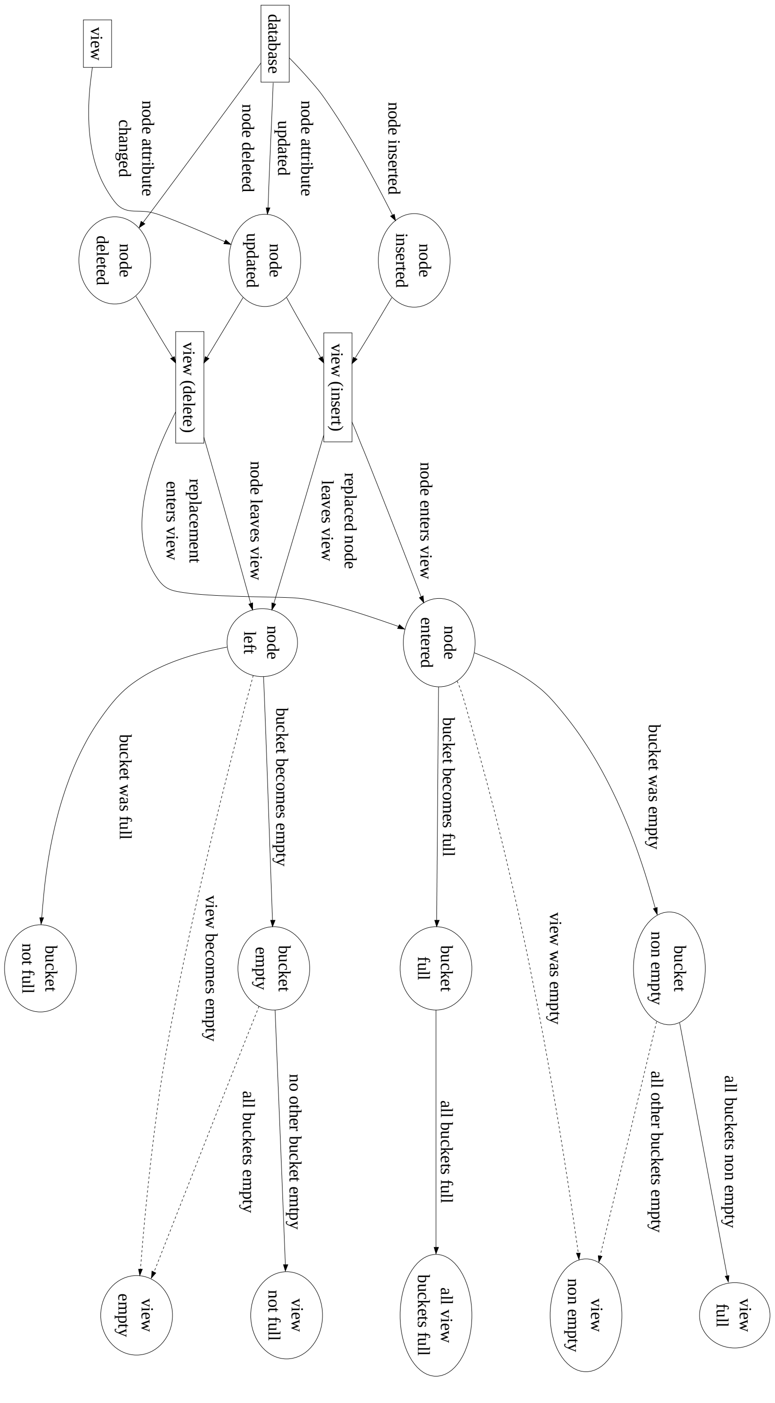


Figure 5.5: Dependencies of SLOSL view events

Chapter 6

OverML

Contents

The Overlay Modelling Language OverML is a set of XML languages for overlay specifications. It comprises the five necessary parts as described in the previous chapters: node attributes, messages, view definitions, routers and EDSM graphs. The following sections describe the syntax and semantics of each language. A formal RelaxNG schema [CM01] is provided in appendix A.1.

Figure 6.1 illustrates the interaction between the five languages Nala (the Node Attribute Language), Slosl (the SQL-Like Overlay Specification Language), Edgar (Extensible Decision Graphs for Adaptive Routing), Himdel (the Hierarchical Message Description Language) and Edsl (the Event-Driven State-machine Language).

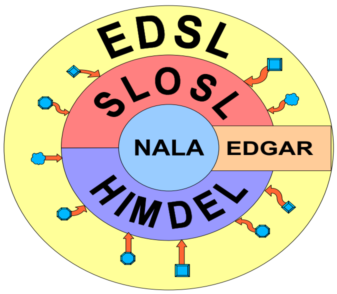
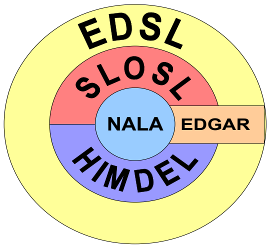


Figure 6.1: The OverML Languages

The main advantage of having an integrated set of languages is their closure, i.e. their self-containing interaction as the figure shows. SLOSL requires NALA for a definition of the underlying database schema. HIMDEL relies on NALA and SLOSL to define the semantics of message fields and view containers. EDGAR uses SLOSL to make routing decisions about HIMDEL defined messages. EDSL uses HIMDEL and SLOSL to infer the available events for messages and views.

6.1 NALA - The Node Attribute Language



NALA serves two purposes. First, it provides data types for OverML that can be restricted or composed into more complex types. Its main purpose, however, is the definition of node attributes that SLOSL works on.

Attributes can currently use a subset of the data types defined for SQL [SQL92], but the XML Schema data types [XSD04] are an alternative from the XML world. A mapping between the two is provided as part 14 of the SQL 2003 standard [SQL03b]. In both cases, NALA allows the definition of custom data types based on the available standard types.

6.1.1 Type Definition

The `nala:types` section in OverML defines the data types available to the application. A possible list containing all standard types follows.

```
<nala:types
     xmlns:nala="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/nala"
     xmlns:sql= "http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/sql">
  <sql:bigint       name="bigint" />       64 bit integer
  <sql:boolean      name="boolean" />      true/false
  <sql:bytea        name="bytea" />        binary byte array
  <sql:char         name="char" />         fixed length unicode string
  <sql:date         name="date" />         calendar date
  <sql:decimal      name="decimal" />      arbitrary length integer
  <sql:double       name="double" />       64 bit float
  <sql:inet         name="inet" />         IPv4/IPv6 address
  <sql:integer      name="integer" />      32 bit integer
  <sql:interval     name="interval" />     relative time span
  <sql:macaddr      name="macaddr" />      network MAC address
  <sql:money        name="money" />        fixed point currency
  <sql:real         name="real" />         32 bit float
  <sql:smallint     name="smallint" />     16 bit integer
  <sql:text         name="text" />         variable length unicode string
  <sql:time         name="time" />         time of day
  <sql:timetz       name="timetz" />       time plus timezone
  <sql:timestamp    name="timestamp" />    date and time
  <sql:timestamptz name="timestamptz" /> timestamp plus timezone

  <sql:composite   name="tcpaddress" />  composite data type
    <sql:inet      name="address" />
    <sql:shortint  name="port" />
  </sql:composite>

  <sql:decimal name="id128" bits="128"/> restricted decimals
  <sql:decimal name="id256" bits="256"/>

  <sql:array       name="integers" />    typed sequence
    <sql:int />
  </sql:array>

  <sql:set         name="IPs" />         typed set
    <sql:inet />
  </sql:set>
</nala:types>
```

Note how **sql:decimal** appears multiple times. The first (named "decimal") is the normal SQL data type while the others are custom types. They were given different names and restricted to a fixed bit size. As the names suggest, they can be used for node IDs.

As this example also shows, the **sql:composite** data type allows composing multiple simple types into a single structured type. Similarly, the **array** and **set** types define sequences and sets that contain items of a specific type. Besides their wide-spread support in current programming languages, most relational databases also provide an array type, whereas they would represent the set type as a table.

6.1.2 Attributes

Any of the defined types can be used to specify node attributes. Each attribute has a unique name. To further specify the semantics of the attribute, the following flags can be used.

identifier If set, the attribute can be assumed to uniquely identify the node. If a node carries multiple identifiers, each one is treated independently as a unique identifier. This allows different levels of identification, most notably physical and logical addresses. Note that multiple types (like IP address and port) can be composed into a single new type, which can then be used as a single identifier.

static If set to true, the attribute is assumed to be static once it is known about a node. All identifiers are implicitly static, but not all static attributes fulfil the uniqueness requirement of an identifier. Again, inconsistencies must trigger events.

transferable specifies whether it makes sense to include the attribute in messages. Some attributes (like the network latency to a neighbour or other locally calculated distances) only make sense locally, so they should be marked non-transferable.

selected activates the attribute for use in the database schema. Unselected attributes can be defined in the specification without actually being part of the data schema during execution. This is a pure convenience option for developers.

There is one special type of attribute: a *dependent attribute*. Dependent attributes are calculated based on other attributes. They must be updated whenever the underlying attribute is modified. An example is the constant ring distance between the identifier of a remote node and the local one in a Chord overlay. It only depends on the identifier attributes and can be calculated once and then stored in the database. Other examples are aggregated values, like the average latency over a sliding window. The lower change rate of averaged attribute values is helpful when trading optimal topology adaptation against the cost of reconfiguration, such as opening connections and exchanging state with new neighbours.

Simple dependent attributes can be expressed using MathML expressions (type "math"), while more complex ones reference an external function (type "external") that must be provided at compile time or deployment time. An interesting future extension to OverML could support portable implementations of more elaborate functions. However, it is not easy to provide such implementations in a programming language independent way, while at the same time coming close enough to the expressiveness of a general programming language. It is currently left to further research to find a suitable tradeoff.

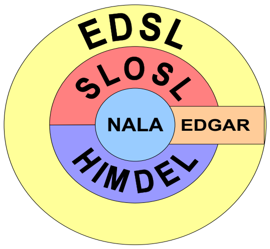
There is another important application of dependent attributes. When multiple overlay specifications from different authors are integrated into a single application, they may not obey the same naming scheme for attributes or use different data type representations for the same node attributes. Developers can deal with this by modifying the OverML specifications to use the same data schema, or they can introduce dependent attributes that match the schema of other specifications. If the overlap of different schemas is high, this can be a fast and easy approach for an integration, as the overall number of attributes in a Nala data schema tends to be small.

Nala makes the database aware of these dependent attributes. Threaded implementations can therefore update them in a synchronised or transactional manner to prevent premature update events. After calculating all dependent attributes, a collective event can be triggered that avoids unnecessary view updates or inconsistent attributes. Another possibility is to use lazy updating and only mark dependent attributes as outdated. They can then be re-calculated the next time they are accessed.

An XML example specification of different attributes follows. Note that (except for `selected`) the flags are represented as elements to simplify future extensions.

```
<nala:attributes
    xmlns:nala="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/nala">
  <nala:attribute name="id" type_name="id256"
                  selected="true">
    <nala:static/>
    <nala:transferable/>
    <nala:identifier/>
  </nala:attribute>
  <nala:attribute name="knows_pastry" type_name="boolean"
                  selected="true">
    <nala:static/>
    <nala:transferable/>
  </nala:attribute>
  <nala:attribute name="knows_chord" type_name="boolean"
                  selected="true">
    <nala:static/>
    <nala:transferable/>
  </nala:attribute>
  <nala:attribute name="chord_distance" type_name="id256"
                  selected="true">
    <nala:static/>
    <nala:depends type="external">
      <nala:attribute-ref name="id" />
      <nala:call name="calculate_chord_distance" />
    </nala:depends>
  </nala:attribute>
  <nala:attribute name="latency" type_name="interval"
                  selected="true" />
</nala:attributes>
```

6.2 Slosl - The SQL-Like Overlay Specification Language



The Slosl language for view definitions is presented in chapter 5. It describes the topology characteristics of the resulting implementation. It is based on node attributes as defined by Nala and provides view definitions that can be referenced by the event subscriptions of Edsl. Similarly, Himdel references Slosl views to include their content in messages, and Edgar uses them for routing decisions.

As Slosl has its own chapter in this thesis (chapter 5), this section only describes the XML representation of Slosl as part of the OverML languages. The format is presented for the following example, taken from 5.3.

```
CREATE VIEW chord_neighbours
 AS SELECT id=node.id, local_dist=node.local_dist, side=sign
RANKED lowest(10, node.local_dist)
FROM db
WITH log_k=160, max_id=2^log_k - 1
WHERE node.knows_chord == true and node.alive == true
HAVING abs(node.id - local.id) <= max_id / 2
         AND sign*(node.id - local.id) > 0
      OR abs(node.id - local.id) >  max_id / 2
         AND sign*(node.id - local.id) < 0
FOREACH sign IN (-1,1)
```

The XML representation follows. Note that mathematical expressions are written in Content MathML[1] [Mat03], which is a rather verbose language. It is, however, also the best choice for representing the semantics of mathematical expressions within XML languages.

```
<slosl:statements
      xmlns:slosl="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/slosl"
      xmlns:mathml="http://www.w3.org/1998/Math/MathML">
  <slosl:statement name="chord_neighbours" selected="true">
    <slosl:select name="id"> node.id </slosl:select>
    <slosl:select name="local_dist">
            node.local_dist </slosl:select>
    <slosl:select name="side" type_name="smallint">
            sign </slosl:select>

    <slosl:parent>db</slosl:parent>

    <slosl:with name="log_k"> 160 </slosl:with>
    <slosl:with name="max_id">
      <m:math>
        <m:apply>
          <m:minus/>
          <m:apply>
            <m:power/>
            <m:cn type="integer"> 2 </m:cn>
            <m:ci> log_k </m:ci>
          </m:apply>
          <m:cn type="integer"> 1 </m:cn>
        </m:apply>
      </m:math>
    </slosl:with>

    <slosl:where>
      <m:math>
        <m:apply>
          <m:and/>
          <m:apply>
```

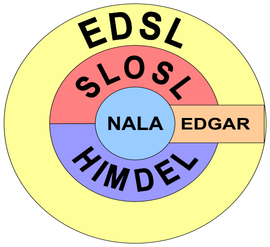
[1]http://www.w3.org/TR/MathML2/chapter4.html (Aug. 30, 2006)

```
              <m:eq/>
              <m:ci> node.knows_chord </m:ci>
              <m:true/>
            </m:apply>
            <m:apply>
              <m:eq/>
              <m:ci> node.alive </m:ci>
              <m:true/>
            </m:apply>
          </m:apply>
        </m:math>
      </slosl:where>

      <slosl:having>
        <m:math>
          <m:apply>
            <m:or/>
            <m:apply>
              <m:and/>
              <m:apply>
                <m:leq/>
                <m:apply>
                  <m:abs/>
                  <m:apply>
                    <m:minus/>
                    <m:ci> node.id </m:ci>
                    <m:ci> local.id </m:ci>
                  </m:apply>
                </m:apply>
                <m:apply>
                  <m:divide/>
                  <m:ci> max_id </m:ci>
                  <m:cn type="integer"> 2 </m:cn>
                </m:apply>
              </m:apply>
              <m:apply>
                <m:gt/>
                <m:apply>
                  <m:times/>
                  <m:ci> sign </m:ci>
                  <m:apply>
                    <m:minus/>
                    <m:ci> node.id </m:ci>
                    <m:ci> local.id </m:ci>
                  </m:apply>
                </m:apply>
                <m:cn type="integer"> 0 </m:cn>
              </m:apply>
            </m:apply>
            <m:apply>
              <m:and/>
              <m:apply>
                <m:gt/>
                <m:apply>
                <m:abs/>
                <m:apply>
                  <m:minus/>
                  <m:ci> node.id </m:ci>
                  <m:ci> local.id </m:ci>
                </m:apply>
              </m:apply>
              <m:apply>
                <m:divide/>
                <m:ci> max_id </m:ci>
                <m:cn type="integer"> 2 </m:cn>
              </m:apply>
            </m:apply>
            <m:apply>
              <m:lt/>
              <m:apply>
                <m:times/>
                <m:ci> sign </m:ci>
                <m:apply>
                  <m:minus/>
                  <m:ci> node.id </m:ci>
                  <m:ci> local.id </m:ci>
                </m:apply>
              </m:apply>
              <m:cn type="integer"> 0 </m:cn>
            </m:apply>
          </m:apply>
        </m:apply>
      </m:math>
    </slosl:having>

    <slosl:buckets>
      <slosl:foreach name="sign">
        <m:math>
          <m:list>
            <m:cn type="integer"> -1 </m:cn>
            <m:cn type="integer"> 1 </m:cn>
          </m:list>
        </m:math>
      </slosl:foreach>
    </slosl:buckets>

    <slosl:ranked function="lowest">
      <slosl:parameter> 10 </slosl:parameter>
      <slosl:parameter> node.local_dist </slosl:parameter>
    </slosl:ranked>
  </slosl:statement>
</slosl:statements>
```

The XML representation is rather straight forward, although verbose due to the lengthy MathML expressions. The main advantage of a structured language like Content MathML over a string representation is the clearer, well-defined semantics. It becomes trivial, for example, to determine the node attribute dependencies of an expression via a simple XPath expression like `.//math:ci`. These dependencies are helpful for optimisations during execution or model transformation and source code generation (see 8.3).

Note the `type_name` attribute in line 8 which refers to a NALA type definition. It allows specifying a type for a calculated view node attribute. This overrides the type that would otherwise have been inferred from the expression.

6.3 EDGAR - Extensible Decision Graphs for Adaptive Routing



The simplicity of the routing decision graphs as described in 5.4 directly translates into a simple XML structure for EDGAR. It is a straight forward hierarchical representation of the directed graph. The example below is the XML representation of the general EDGAR graph in figure 5.1.

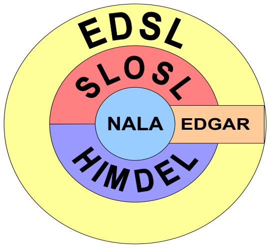
All nodes in the EDGAR graphs are represented as XML elements. The obvious execution plan is a depth-first tree traversal. Predicates also become elements that surround their target. A predicate element blocks traversal if the predicate fails. The *exclude last* element is represented as a more general *tag* element that sets a tag on the tree traversal process. Tags are evaluated before termination.

There are three ways to terminate the routing process. The *exit* elements always terminate and jump to a specific target. The *tryview* elements evaluate their target view for possible candidates. If this succeeds, the process terminates. The third way is to terminate the traversal without having hit a matching target. In this case, the behaviour is platform-specific. Platforms may raise exceptions or call global error handlers to handle this. The XML code below prevents this case by terminating on an explicit error handler.

```xml
<routers
      xmlns="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/edgar">
  <router name="examplary_router">
    <firstmatch>
      <predicate name="for_me">
        <exit target="local_handler" />
      </predicate>
      <predicate name="db_lookup">
        <exit target="forward" />
      </predicate>
      <firstmatch>
        <tag type="exclude_last">
          <tryview target="View1" />
        </tag>
        <fork>
```

```
          <tryview target="View2" />
          <tryview target="View3" />
        </fork>
      </firstmatch>
      <exit target="error_handler" />
    </firstmatch>
  </router>
</routers>
```

6.4 Himdel - The Hierarchical Message Description Language



Messages combine attributes and other data into well defined data units for transmission. They have two sides: a software interface for reading and writing data field and a serialisation format for the wire. Himdel is a hierarchical message specification language that defines a generic software interface to messages and allows for mappings to different serialisation formats as binary data or XML.

As usual, messages are encapsulated in headers which are in turn encapsulated in network protocols. This makes them conceptually hierarchical. In Himdel, message definitions consist of three top-level parts: protocols, top-level headers and pre-defined containers. The rest of the specification follows a hierarchy rooted in a top-level header, followed by encapsulated headers and finally a sequence of data fields as content. Being an XML language, Himdel presents this hierarchy in a natural way.

```
<msg:message_hierarchy
     xmlns:msg="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/himdel"
     xmlns:sql="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/sql">
  <msg:container type_name="ids">
    <msg:attribute access_name="source" type_name="id" />
    <msg:attribute access_name="dest"   type_name="id" />
  </msg:container>
  <msg:header access_name="main_header">
    <msg:container-ref access_name="addresses"
                       type_name="ids" />
    <msg:message type_name="join_request" />  <!-- 1st message -->
    <msg:message type_name="view_message">    <!-- 2nd message -->
      <msg:viewdata structured="true"
                    access_name="fingertable"
                    type_name="chord_fingertable" />
    </msg:message>
    <msg:header>
      <msg:content access_name="type"
                   type_name="sql:smallint" />
```

```
        <msg:message type_name="typed_message"> <!-- 3rd message -->
          <msg:content access_name="data"
                       type_name="sql:text" />
        </msg:message>
      </msg:header>
    </msg:header>
    <msg:protocol access_name="tcp" type_name="tcp">
      <msg:message-ref type_name="view_message" />
      <msg:message-ref type_name="typed_message" />
    </msg:protocol>
    <msg:protocol access_name="udp" type_name="udp">
      <msg:message-ref type_name="join_request" />
    </msg:protocol>
  </msg:message_hierarchy>
```

In this representation, messages become a path through the hierarchy that describes the ordered data fields (i.e. content and attribute elements) that are ultimately sent through the wire. Message data is encapsulated in the header hierarchy that precedes it on the path. Headers and their messages are finally encapsulated in a network protocol, apart from their specification. This makes it possible to send the same message through multiple protocol channels and to decide the best protocol at runtime.

As the example shows, multiple messages can be defined within the same header, which makes them independent messages using this header. When following a message path, other messages and headers are completely ignored. For example, the 'join_request' message in the example is not part of the 'typed_message', although it precedes it on the path.

To assure the uniqueness of these paths, all children of a parent element must have distinct names. Additionally, fields and containers must be uniquely named along each path, meaning that no field or container has the same name as any of its ancestors. However, only the names of top-level containers and headers, and of all messages and protocols must be globally unique in the specification.

The tag order on the message path is also important. It describes the field order when serialising data, but it also defines the data fields that are actually contained in a message. If a header is extended by content or container elements after the definition of a message, the preceding messages will *not* contain the successor fields, which are not on its path. In the example, the 'view_message' will not contain the content field named 'type'. This field is, however, available in the 'typed_message' and all messages that are defined later under the same header tag.

As shown in the example, container elements can also be used at the highest level inside the message_hierarchy tag. However, their definition is not part of the message hierarchy itself. They only predefine container modules for replicated use in headers and messages where they are referenced by the type_name attribute of container-ref elements.

6.4.1 Programmatic Interface to Messages

The programmatic access to messages and their fields is defined using the access_name tag. Note that this follows the hierarchy only for containers and message content. Headers are accessed directly, as is protocol data. Accessing the fields of a message from an object oriented language should look like the following Python snippet.

```
def receive_view_message(view_message):
    net_address  = (view_message.tcp.ip, view_message.tcp.port)
    main_header  = view_message.main_header
    source_id    = main_header.addresses.source
    finger_nodes = view_message.fingertable
```

The rules for building the access path are as follows. They allow for a relatively concise, but nevertheless structured and well defined access path to each element. The reference implementation provides this algorithm as an XSL transformation [XSL99] of Himdel (see appendix A.6).

1. As the basic unit of network traffic, a message is always the top-level element.
2. Every child of a message is kept as a second level element, referenced by its access name.
3. Entries within containers are referenced recursively, namespaced by the access name of their parent.
4. Following the path from the message back to the root header, all headers and also the protocol become additional second-level elements, referenced by their access name. Their child fields and container elements are referenced recursively as before. Children of nameless headers become elements of the parent header.

Software components can then subscribe to message names or header names in a hierarchical way and thus define the part of the message that is actually of interest to them. A simple subset of the XPath language [XPa99] naturally lends itself for defining these subscriptions. Note that even expensive abbreviations like '//' can be resolved at compile time or deployment time based on the message specifications.

The programmatic interface additionally defines two message properties *last_hop* (a node if the last hop of a message is known) and *next_hop* (a node if the next hop was already determined by a router).

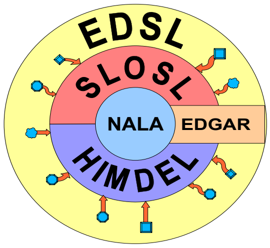
6.4.2 Network Serialisation of Messages

There are a number of possible network representations for messages. Their choice depends on frameworks and languages and is therefore outside the scope of the platform independent model provided by OverML. The following describes two possible mappings that model transformations can follow.

The SQL/XML standard [SQL03b], as published in 2003, provides a well defined mapping of SQL data types to XML Schema. This should be considered the preferred method for message serialisation to XML. In this case, the hierarchical structure of the message specifications can directly be mapped to an XML serialisation format.

Another, currently more common serialisation is the XDR data model [Sun87, Sri95], originally developed by Sun Microsystems[2] for their ONC-RPC [Sun88]. The mapping from the message specification to a flat serialisation is straight forward when laying out the data top-down along the message paths.

6.5 EDSL - The Event-Driven State-machine Language



EDSL is a graph language for describing event-driven state machines. It is based on states and transitions, like any state machine description. States represent stages of code execution, while transitions connect processing chains and describe events that trigger new states. Being a graph language, EDSL has a straight forward transformation to the well-known DOT language of graphviz. An implementation is provided in appendix A.2. It allows for easy visualisation of EDSL graphs. Figure 6.2 illustrates an example of such a graph.

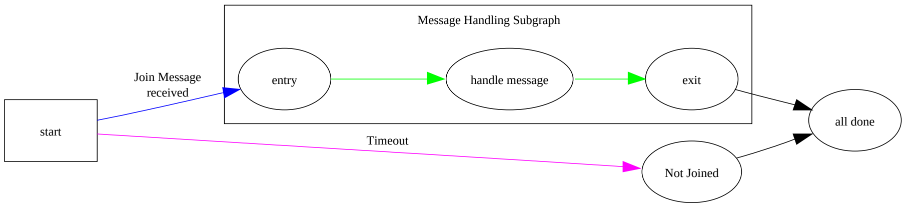


Figure 6.2: Example of an EDSL graph

The main advantage of EDSL over other languages for state machines[3] and graphs (like the DOT language[4]) is its awareness of SLOSL and HIMDEL. EDSL allows transition triggers to be expressed based on data fields that occur in

[2]http://www.sun.com

[3]http://www.elude.ca/ragel/, http://fsmlang.sourceforge.net/, http://research.microsoft.com/fse/asml/default.aspx, etc. (Aug. 30, 2006)

[4]http://graphviz.org/doc/info/lang.html (Aug. 30, 2006)

incoming messages as defined by Himdel, or based on events that occur in views defined by Slosl. It therefore becomes possible to raise the level of framework independence by specifying semantically rich transitions based on OverML instead of hand-written code.

The envisioned execution model behind Edsl is a state machine that supports multiple active states at the same time. In literature, such concurrent states are commonly referred to as "and" states, as in state charts [Har87, HP98]. They are common case in event-driven state machines, which keep multiple pending states and select between them based on I/O events.

The execution model also allows for long-running states. These are states that stay active even if one of their transitions has fired. This is helpful for generic message dispatcher states that keep receiving messages and forward them through their transitions. It also allows for concurrency in output chains where a state outputs several data items, one after the other, that are then handled by the successor states concurrently. Exiting such a state and controlling its runtime behaviour has to be done programmatically, though, as it is not part of the graph description that Edsl provides.

The following listing shows the Edsl description of the graph in figure 6.2:

```
<edsl:edsm
     xmlns:edsl="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/edsl">
  <edsl:states>
    <edsl:state name="start" id="1137068428">
      <edsl:readablename>start</edsl:readablename>
    </edsl:state>
    <edsl:state name="state000001" id="1137069916">
      <edsl:readablename>Not Joined</edsl:readablename>
    </edsl:state>
    <edsl:subgraph name="subgraph000001" id="1137119516"
          entry_state="1137119324" exit_state="1137119612">
      <edsl:readablename>Message Handling
                         Subgraph</edsl:readablename>
      <edsl:states>
        <edsl:state name="entry" id="1137119324">
          <edsl:readablename>entry</edsl:readablename>
        </edsl:state>
        <edsl:state name="exit" id="1137119612">
          <edsl:readablename>exit</edsl:readablename>
        </edsl:state>
        <edsl:state name="state000002" id="1137121100">
          <edsl:readablename>handle message</edsl:readablename>
        </edsl:state>
      </edsl:states>
      <edsl:transitions>
        <edsl:transition type="outputchain">
          <edsl:from_state ref="1137119324"/>
          <edsl:to_state ref="1137121100"/>
        </edsl:transition>
        <edsl:transition type="outputchain">
          <edsl:from_state ref="1137121100"/>
```

```
            <edsl:to_state ref="1137119612"/>
          </edsl:transition>
        </edsl:transitions>
      </edsl:subgraph>
      <edsl:state name="state000003" id="1137122012">
        <edsl:readablename>all done</edsl:readablename>
      </edsl:state>
    </edsl:states>
    <edsl:transitions>
      <edsl:transition type="message">
        <edsl:from_state ref="1137068428"/>
        <edsl:to_state ref="1137119324"/>
        <edsl:readablename>Join Message received</edsl:readablename>
        <edsl:messagetype>/join_message</edsl:messagetype>
      </edsl:transition>
      <edsl:transition type="timer">
        <edsl:from_state ref="1137068428"/>
        <edsl:to_state ref="1137069916"/>
        <edsl:timerdelay>120000</edsl:timerdelay>
        <edsl:readablename>Timeout</edsl:readablename>
      </edsl:transition>
      <edsl:transition type="transition">
        <edsl:from_state ref="1137069916"/>
        <edsl:to_state ref="1137122012"/>
      </edsl:transition>
      <edsl:transition type="transition">
        <edsl:from_state ref="1137119612"/>
        <edsl:to_state ref="1137122012"/>
      </edsl:transition>
    </edsl:transitions>
  </edsl:edsm>
```

Comparing the verbosity of state machine languages like EDSL with the clean and readable graph in figure 6.2 makes a convincing case for the visual design of event-driven state machines. This argument also holds against the implicit implementation of these graphs in source code, as required by most current EDSM frameworks. It is therefore a substantial improvement for developers to design protocols visually (as with the SLOSL Overlay Workbench, see 8.1.4) and to simply generate machine readable EDSL specifications and source code implementations from these graphs.

6.5.1 States

States in EDSL are points of execution. There is one special state called *start* at the top level that is activated at startup. All other states are activated through subsequent transitions. As for terminology, the *transitions of a state* designate the transitions that lead away from that state towards other states.

States are components of the EDSM that follow a simple, but very generic model of component interaction. It was exemplified by the Axon framework[5], which was in turn inspired by Unix pipes. The Axon model provides components with named pipes for input and output. Components react to objects being passed in through any of their input pipelines and respond by writing objects to any of their output pipelines. The model provides a very generic view on component interaction, similar to subject based publish-subscribe. EDSL inherits the simplicity of this model. It extends it to a complete graph description that makes transitions explicit and augments the semantics of events.

States can currently have two additional boolean attributes that are described further down: `long_running` and `inherit_context`. Both default to false. While concrete frameworks may encourage diverging ways of implementing states, the abstract behaviour of a state is as follows.

1. On activation, the state is instantiated and receives the status *waiting*.
2. Depending on the framework specific scheduler, the state eventually becomes *running* and is executed within the EDSM.
3. It can then produce output or request a modification of its status.
4. The execution of a state is atomic with respect to its transitions. This means that none of the transitions can fire away from a state during the execution of that state, which is the common case in EDSM frameworks. Note that atomicity only addresses the transitions of the state itself. Frameworks can execute as many states in parallel as they support.
5. Short running states (having `long_running=false`) receive the status *terminated* when they
   - produce output
   - request to be terminated
   - have finished execution and one of their transitions has fired. Since execution is atomic, there is always a first event that fires after execution, so the transition that terminates the state is well defined.
6. Long running states are only terminated when they explicitly request to be terminated. This implies that any number of transitions can fire on them after execution has finished. Note that it is up to concrete frameworks to provide ways to let long running states determine whether their execution should be restarted after having finished or whether they should stay idle but active (sometimes called a *zombie* status).
7. Depending on the framework, active states (long running or not) may be able to temporarily interrupt their execution by actively yielding control back to the framework. The framework can then decide when to continue the execution. This is commonly used in single threaded EDSM

[5]`http://kamaelia.sourceforge.net/Docs/Axon.html` (Aug. 30, 2006)

frameworks that cannot afford to miss I/O events during long calculations. This feature does not impact the atomicity constraint and leaves the state *running*.

The `inherit_context` flag of a state is meant to simplify the programming of individual states. Very often, processing chains in EDSMs are interrupted by I/O activity like sending messages and having to wait for an answer. The exact meaning of this flag is framework specific. However, setting it on a state will ask the framework to propagate an execution context from the previous state to this one during the transition. It therefore provides a simple way of propagating variables, configuration or other forms of processing status within a chain of execution.

6.5.2 Transitions

Transitions connect states and define the flow of execution. They are triggered by events which EDSL defines based on the following types:

Messages The arrival of a message. Subscriptions to messages are expressed in XPath based on the message content paths as described in section 6.4.1. In figure 6.2, the message transition is subscribed to the reception of any `join_message`, without further restrictions.

View events Events that occur in views defined by SLOSL (see 5.5).

Timers Delay triggered events. The delay is expressed in milliseconds.

Output chains All output of the source state is handed on to the target without modification. Frameworks may force the target state to become long running if the source state is.

Direct advances Immediately activate the target state when entering the source state, thus essentially merging the two states. This special transition does not require any events or output of the state to be fired. Since it is only fired on state activation, even long running states can deploy it. A possible use case is the design decision to split a single I/O state into semantically different parts.

6.5.3 Subgraphs

EDSL supports subgraphs within a state machine. These are complete state machines themselves that may have further subgraphs besides their states and transitions. Subgraphs have two predefined states, *entry* and *exit*, that connect with the parent graph by reproducing their incoming events.

The main use cases for subgraphs are harvesters and other complex control components. They can be plugged into the graph and are globally triggered and configured as part of the EDSM. Their integration and interconnection through the EDSL component model simplifies their reuse in different overlays.

Subgraphs are only used at design time to reduce the complexity of EDSM designs by semantic modularisation. States are always referenced by internal identifiers that are unique to the complete graph. It is therefore a straight forward transformation to flatten subgraphs by merging them into their parent graph. Flat graphs can then be mapped to EDSM frameworks more easily.

Chapter 7

Harvesters

Contents

As companions of database and views, harvesters are a very important part of the middleware. Their main purpose is to maintain the local view of a node by updating the node database with data relevant to the currently available view definitions. This means finding new nodes, as well as determining and maintaining their attributes, but also deleting nodes from the database.

There are a number of possible schemes for these tasks out of which the designer of an overlay system can select the most appropriate ones. Having a choice of harvester components available allows overlays to provide very diverse characteristics.

Harvesters work on views, nodes and node attributes. They can be simple controllers, but are more commonly complex EDSL compositions of controllers, i.e. EDSL subgraphs. Where controllers are more of an atomic implementation detail of a protocol, harvesters are functional components that implement a specific subtask in the maintenance of an overlay. As seen in Gridkit (8.4), they are often independent of specific overlays and can be provided as middleware components.

For example, the harvester task of finding new nodes can be implemented in various different ways based on broadcast, active lookup, overhearing routed messages, central discovery services, etc. Gossip (or epidemic communication [VvRB03, JGKvS04, RGRK04]) has a straight forward implementation in the Node Views architecture that exchanges views between overlay members. The attributes of nodes are either measured locally or requested from the nodes. It is up to the specific harvester implementation how this is done and up to its configuration how often this happens. Also, when nodes have locally disappeared from all views, the application must decide if (and how long) they should remain in the database.

The following sections exemplarily discuss four different kinds of harvesters that implement these kinds of functionality. Note that much of what is said here can be applied in similar ways to other harvester types and implementations.

7.1 Gossip-based Node Harvesters

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 and the communication cost configurable.

As an example, we can look at a Pastry derivative called Bamboo [RGRK04]. 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.

Note that in Bamboo and Pastry, most of the nodes in the leafset will also be in the routing table. If an attribute of a local node representation states that a node has not been queried by another component or otherwise responded for a while, it may be a good candidate for a ping, while a node that is already frequently queried by the leafset maintainer should not be additionally queried by the routing table maintainer. The local consistency provided by the node database naturally supports the coordination between these maintenance components and simplifies their implementation.

Jelasity etal. [JGKvS04] 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 well-selected 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. describe them in a similar way for the Bamboo system [RGRK04] 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.

The Node Views 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 [RGRK04]. 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 it trivial to write overlays based on gossip or other ways of exchanging node data. 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.

7.2 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 criterion 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 [PCW$^+$03, DCKM04], others query the routing infrastructure (BGP) of the physical network [NPB03].

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.

7.3 Sieving Harvesters

So far, we have only regarded the process of adding new nodes and updating their attributes. However, when harvesters keep adding nodes, there must also be a mechanism that removes nodes that are no longer in scope of any view. If nodes are not visible in views, their attributes will tend to become outdated more easily than those of active communication partners. On the other hand, keeping track of too many nodes can lead to an exhaustive message load. Collecting the data of too many nodes in the local database can also reduce the run-time performance of the system. Overlay nodes must therefore trade their resource consumption against the value of locally available data.

In the Node Views architecture, the problem of keeping nodes available in the database is reduced to a caching problem. Nodes that are visible in views *must* be kept in the database. Nodes that are not visible in views *may* be kept in the database. Which invisible nodes should be removed?

In many overlay implementations today, only the topology neighbours are kept in the local view. This is equivalent to removing each node from the database that is discarded from a view and no longer visible in any other. A harvester connected to the enter and leave events can easily achieve this.

However, some systems have found additional knowledge to be a good thing, e.g. for neighbour redundancy [ZHD03] or fallback candidates [MCKS03]. Systems based on the Node Views architecture can trivially benefit from additional locally available knowledge, as they already select subsets of nodes from a larger database. More knowledge allows them to take more decisions locally, without necessarily having to send messages. These systems should prefer a garbage collection scheme where nodes are removed from the database based on common caching policies.

Due to the generic interfaces of views and the specific events they generate, generic harvesters can implement basically any kind of caching scheme. It is an important property of this approach that the chosen policies are independent of overlay implementations and may be freely changed at run-time. Usually, this can even be done locally on a per node basis without remote interaction, since it only impacts the local representation of nodes that are not currently chosen as communication partners. This allows each node to determine and follow its locally optimal strategy.

7.4 Distributed Harvesters

Darlagiannis and others recently presented Omicron [DMS04], 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 straight forward 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 this model, the approach taken by Omicron mainly becomes a distributed, hierarchical database.

There is a possibility of extending this scheme into the architectural design. A distributed implementation of the Node Views architecture could reduce the overhead of the lower participants even further. Such a system would implement the complete architecture at the maintainers and replicate only the materialisation of views at the lower cluster participants. Their updates and view events would then be generated remotely by the cluster maintainer. Such a design allows routers to use static routing tables and to update them very specifically without the overhead of event generation from a local database.

The tradeoff between fast, static tables at the lower participants and the availability of the more capable general architecture for taking their own local decisions is a design choice for the implementation. It mainly depends on the expected capabilities of the cluster participants. The Model Driven Architecture approach makes it possible to configure these kinds of implementations details at the model transformation and code generation level.

Chapter 8

Implementation

Contents

The concepts developed in this thesis were validated through a proof-of-concept implementation. It currently comprises three parts:

1. The *SLOSL Overlay Workbench*[1] [Beh05b], a graphical overlay design front-end for OverML specifications. It is written in the Python language and uses the Qt toolkit for its graphical user interface.

2. A preliminary, interpreted OverML run-time environment. It is written in the Python language and uses the Twisted EDSM framework[2].

3. Exemplary model transformers for SLOSL and OverML that show how to build source code generators for different programming languages.

The Workbench allows the development of language neutral and platform independent overlay models (PIM) in OverML that can be transformed to language specific EDSM environments via the Model Driven Architecture approach.

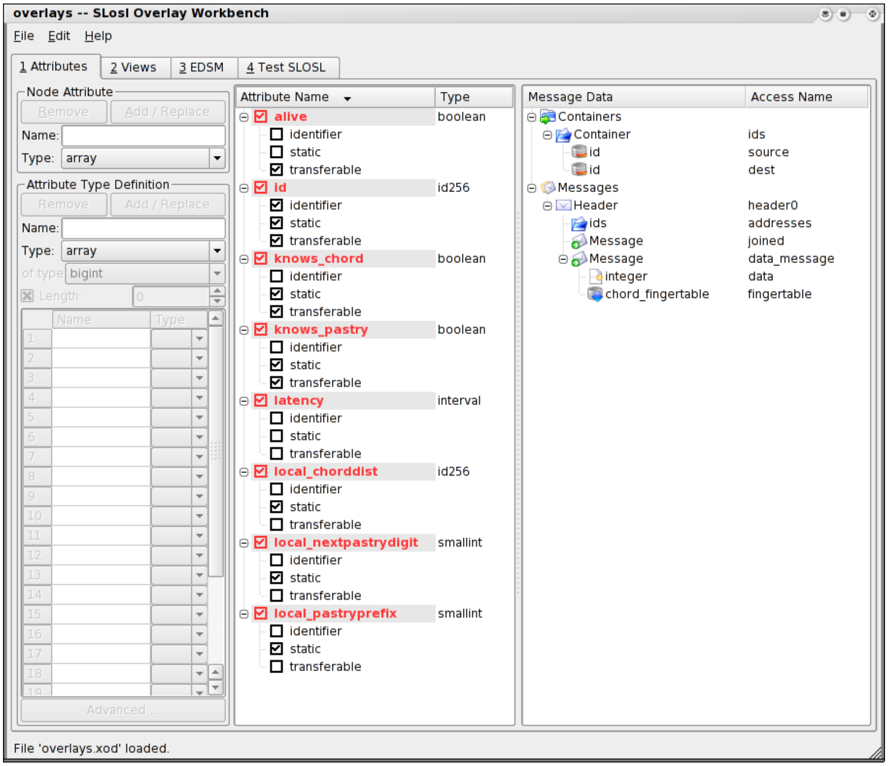
[1]`http://developer.berlios.de/projects/slow/` (Aug. 30, 2006)
[2]`http://www.twistedmatrix.com`

8.1 The Slosl Overlay Workbench

In the Model Driven Architecture of Node Views, the abstract specification of overlay software is developed in five steps by defining node attributes, messages, Slosl views, routing decision graphs and an EDSM graph to connect the controller states. Being an OverML front-end, the workbench directly operates on an XML model. This enables common XML technologies like XPath, XSLT, RelaxNG and XInclude for its implementation, that allow for model modularisation, validation and transformation.

8.1.1 Specifying Node Attributes and Messages

The first step in the design of an overlay network is the specification of node attributes. The user interface for this is shown on the left side of figure 8.1.



Name:
Type: array

Name:
Type: array
of type bigint
Length: 0

Attribute Name	Type
☑ alive	boolean
☐ identifier	
☐ static	
☑ transferable	
☑ id	id256
☑ identifier	
☑ static	
☑ transferable	
☑ knows_chord	boolean
☐ identifier	
☑ static	
☑ transferable	
☑ knows_pastry	boolean
☐ identifier	
☑ static	
☑ transferable	
☑ latency	interval
☐ identifier	
☐ static	
☐ transferable	
☑ local_chorddist	id256
☐ identifier	
☑ static	
☐ transferable	
☑ local_nextpastrydigit	smallint
☐ identifier	
☑ static	
☐ transferable	
☑ local_pastryprefix	smallint
☐ identifier	
☑ static	
☐ transferable	

Message Data	Access Name
Containers	
Container	ids
id	source
id	dest
Messages	
Header	header0
ids	addresses
Message	joined
Message	data_message
integer	data
chord_fingertable	fingertable

File 'overlays.xod' loaded.

Figure 8.1: Defining node attributes and messages in the Overlay Workbench

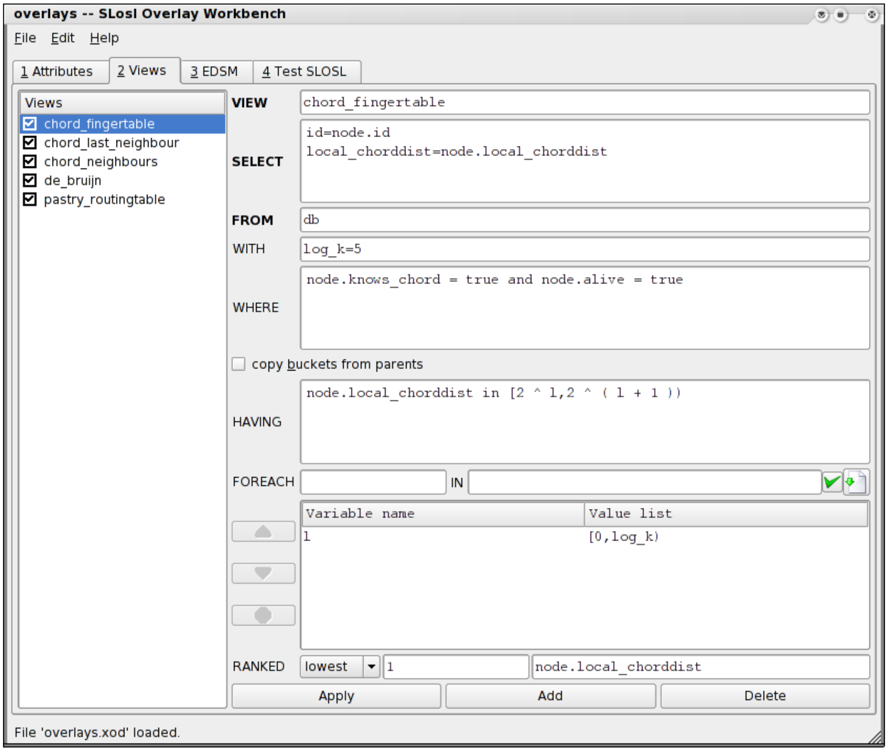
In the bottom left corner, the developer specifies and restricts data types, as supported by Nala. Based on these types, node attribute specifications are added to the list in the centre of the window. The attribute flags *identifier*, *static* and *transferable* are set directly below the names of the attributes.

The right most part in the figure allows to integrate the attributes into HIMDEL messages. Messages are data structures that build on a basic network protocol. They contain embedded header and content parts in a hierarchical fashion. This hierarchy is directly reflected in the tree structure on the right side of the GUI. The messages are defined below the entry *Messages*.

Above the actual messages, the workbench allows the definition of predefined containers. They are modules that can be reused in different parts of the message hierarchy. Container references are added to messages via drag-and-drop.

8.1.2 Writing SLOSL Statements

The next step in overlay design is the definition of the local views. This is done with SLOSL, as shown in figure 8.2.



VIEW chord_fingertable

SELECT
```
id=node.id
local_chorddist=node.local_chorddist
```

FROM db

WITH log_k=5

WHERE
```
node.knows_chord = true and node.alive = true
```

copy buckets from parents

HAVING
```
node.local_chorddist in [2 ^ 1,2 ^ ( 1 + 1 ))
```

FOREACH IN

Variable name	Value list
1	[0,log_k)

RANKED lowest 1 node.local_chorddist

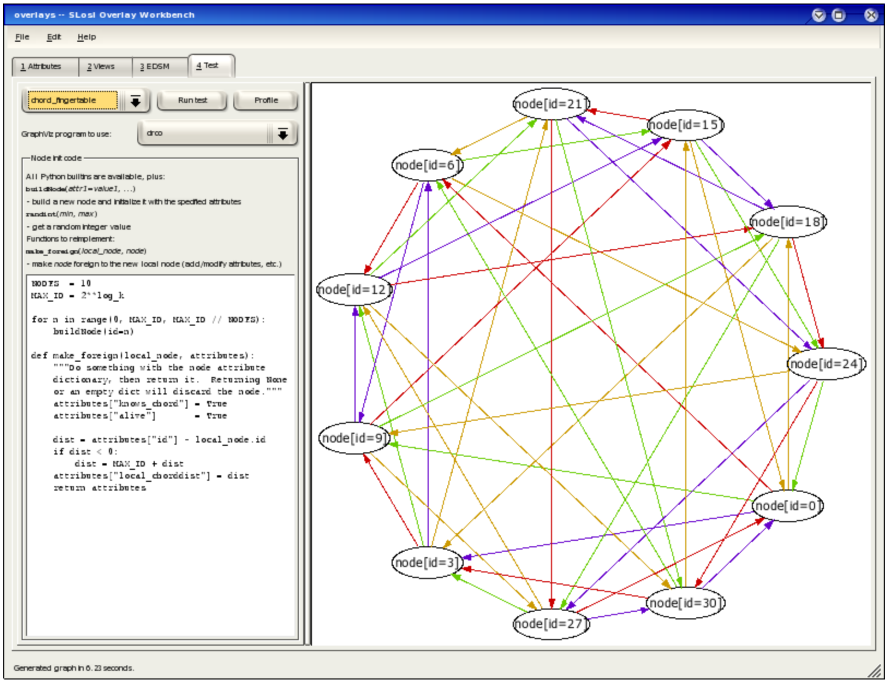
File 'overlays.xod' loaded.

Figure 8.2: SLOSL statements in the Overlay Workbench

The currently defined SLOSL statements are listed on the left and can be selected by mouse click. The right part of the window shows the editor for defining the different SLOSL clauses.

8.1.3 Testing and Debugging SLOSL Statements

Once the SLOSL statements are defined, the topology simulator can be used to test and simulate their effect by executing them in different scenarios. The SLOSL language makes the simulator a powerful tool for this purpose. The developer implements the scenarios in source code, but can completely abstract from the networking nature of the simulated topology. This makes the scenario implementation both simple and short.



All Python builtins are available, plus:
buildNode(*attr1=value1, ...*)
- build a new node and initialize it with the specified attributes
randint(*min, max*)
- get a random integer value
Functions to reimplement:
make_foreign(*local_node, node*)
- make *node* foreign to the new local node (add/modify attributes, etc.)

```
NODES  = 10
MAX_ID = 2**log_k

for n in range(0, MAX_ID, MAX_ID // NODES):
    buildNode(id=n)

def make_foreign(local_node, attributes):
    """Do something with the node attribute
    dictionary, then return it.  Returning None
    or an empty dict will discard the node."""
    attributes["knows_chord"] = True
    attributes["alive"]       = True

    dist = attributes["id"] - local_node.id
    if dist < 0:
        dist = MAX_ID + dist
    attributes["local_chorddist"] = dist
    return attributes
```

Figure 8.3: Visualising and testing SLOSL implemented topologies

The idea follows directly from the Node Views architecture. The setting deploys a number of nodes in the topology that have a certain knowledge about the attributes of the other nodes. Their local view is then used to establish the SLOSL views that form the topology graph. The example in figure 8.3 shows the following source code for the *chord_fingertable* view (see 5.1). Note that the code has global access to the options defined in the SLOSL WITH clause.

```
 1  NODES  = 10
 2  MAX_ID = max_id      # use the option value from the SLOSL statement
 3
 4  for n in range(0, MAX_ID, MAX_ID // NODES):
 5      buildNode(id=n)
 6
 7  def make_foreign(local_node, attributes):
 8      """Do something with the node attribute dictionary,
 9         then return it.  Returning None or an empty dict
10         will discard the node."""
11      attributes["knows_chord"] = True          # set trivial attributes
12      attributes["alive"]       = True
13
14      dist = attributes["id"] - local_node.id   # calculate ring distance
15      if dist < 0:
16          dist = MAX_ID + dist
17      attributes["local_chorddist"] = dist
18      return attributes                         # return node attributes
```

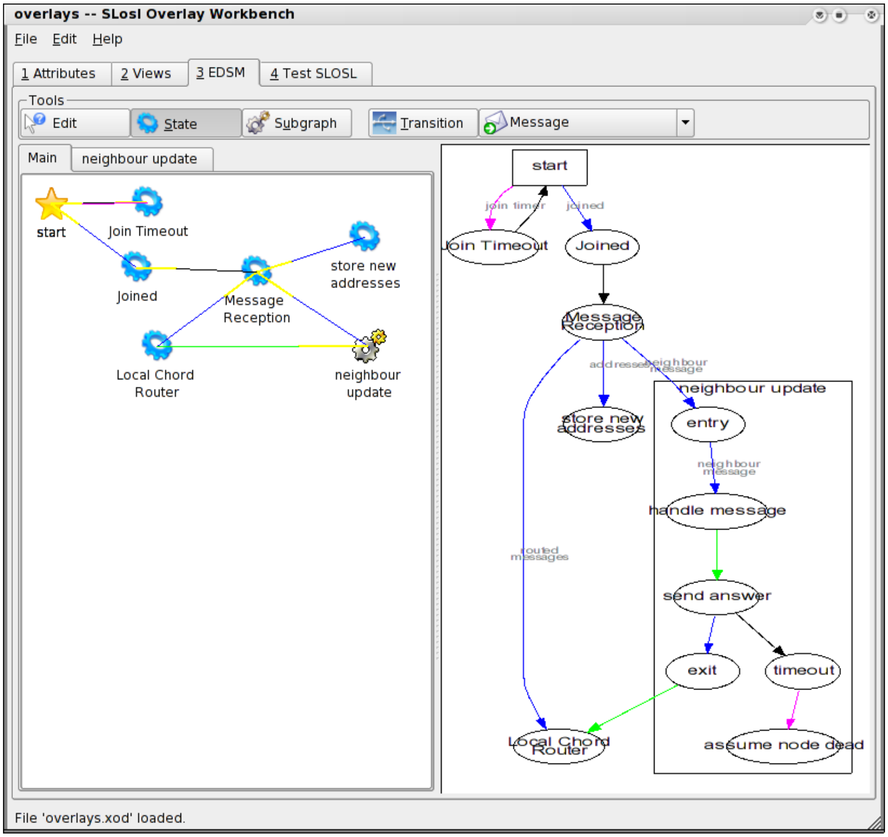
In line 4-5, the script sets up the nodes that participate in the simulation. The nodes are created and given a specific value for their node attribute *id*. This basically defines the global view of the system. The function *make_foreign* is the place where the local view of each node is constructed. At each node, it is called once for each node that is made known locally. It can then modify the attributes of that foreign node before it is added to the local database. It can even decide to ignore the node completely.

The example above only calculates the exact mapping from the global view to the global view of each node, which provides each node with a complete local view. However, by simply varying the node attributes within this function, the developer can implement arbitrary scenarios of nodes knowing each other or not, nodes having failed but still being considered alive by others, or different update propagation states of node attributes. The resulting topology will then be calculated and visualised based on SLOSL evaluation against the globally inconsistent local database of each node.

8.1.4 Designing Protocols and Harvesters

As the final design step, the developer can now build the general structure of the overlay protocol by defining an EDSM graph that interconnects states of execution by events. The user interface for this design phase is shown in figure 8.4. While simple harvesters can be represented as a single state, more complex ones can be expressed as subgraphs, which makes the overall design modular.

The graph at the right of figure 8.4 visualises the entire EDSM graph. Subgraphs are merged into the graph and displayed within a rectangular border. The left part of the window shows the design area. Here, subgraphs are repre-



File 'overlays.xod' loaded.

Figure 8.4: EDSM definition in the Overlay Workbench

sented as icons, just like states. Their innards are accessible through the tab bar above the design area.

8.1.5 OverML Output

At any time, the current OverML specification can be written to a file that the user can feed into an execution environment or a source code generator. The result obeys the schema in appendix A.1.

The workbench also supports a transformed output format that translates the HIMDEL hierarchy into separate messages and flattens the EDSL graph. This simplifies the output and makes it more suitable for subsequent transformations into executable code.

8.2 Implementing Slosl

Slosl is the most important language within OverML. The other four languages have either straight forward implementations (Nala and Edgar) or can deploy well-known EDSM and messaging frameworks for their implementation (Edsl and Himdel). This section therefore describes a simple Slosl interpreter that was written as a proof-of-concept implementation, as well as a transformation from SLOSL to the Structured Query Language (SQL).

8.2.1 Transforming Slosl to SQL

As an SQL-like language, it is possible to map Slosl to nested SQL statements, although in a rather expensive way. Some possible optimisations and tradeoffs are mentioned in this section, but were not further investigated in the course of this work. Due to the expensive evaluation of the resulting SQL queries, it is unlikely that frameworks will deploy such a mapping for real-time execution in deployment environments. Therefore, this SQL implementation is only provided as a proof of concept and for general interest.

Mapping the clauses that were borrowed from SQL is trivial. The general idea is to create the node database as a table with node attributes. This supports a direct mapping of these Slosl clauses to the equivalent SQL clauses. The WHERE clause is evaluated directly on this table or on the union of the parent views, depending on the FROM clause. For simplicity, the result will be referred to as the *node table* from now on. A mapping of the remaining clauses WITH, HAVING–FOREACH and RANKED, that are specific to Slosl, is worth further explanation.

The WITH clause can trivially be mapped to tables holding the values of the declared names. This implies joining them with the node table on query evaluation. Another possibility is to replace options by their constant value within the statement. This would require views to be re-instantiated when options are modified. Obviously, this is the most efficient solution for design-time and deployment-time options that are never (or rarely) changed at run-time.

The next step is to instantiate a table for the values of each bucket variable declared by a FOREACH statement. These tables are joined with the node table based on the HAVING expression. The result is a table which relates node attributes to matching variable values. Note that the node attribute rows of a node may end up in multiple rows of the result.

The next step is the evaluation of the RANKED clause. This means evaluating the ranking expression for each row. Obviously, if the ranking expression does not depend on the variables, it may be more efficient to evaluate it only against the node table before running the joins. In some cases, it may even make sense to store its precomputed result directly in the database to avoid further evaluations. This is the usual tradeoff between space and time, which parametrises the mapping between SLOSL and SQL.

The second part of the RANKED evaluation can only be done after the HAVING joins. The ranking and elimination step implies grouping the rows by distinct sets of variable values, sorting the rows in each group by their ranking, counting the rows and eliminate the super-numerous ones.

The following listing presents a possible template for generating the corresponding SQL statement.

```
CREATE VIEW [view_name] AS
SELECT _noderank, _db_node_id,
       [variables], [select_attribute_calculation]
FROM (
  SELECT node._db_node_id, [variables], [select_attributes], (
    SELECT COUNT(*)
    FROM (SELECT *
      FROM [parents] AS node,
      WHERE ([where_expression]) AND ([having_expression])
     ) AS _node
    WHERE ([_node_rank_expr]) [rank_cmp_op] ([rank_expr])
   ) AS _noderank
  FROM [local_node_table] AS local,
       [variable_source],
       (SELECT node.*
        FROM [parents] AS node,
             [local_node_table] AS local
        WHERE ([where_expression])
       ) AS node
  WHERE ([having_expression])
 ) AS node
WHERE (_noderank <= [rank_count])
ORDER BY [variables], _noderank
```

The HAVING joins and the RANKED sorts show where the costs of this mapping come from. Their expense depends on the number of values per FOREACH variable and on the total number of nodes available in the node table. The declarative nature of the separate SLOSL clauses helps in making tradeoffs. The visible dependencies between attributes and clauses allow to parametrise and optimise the evaluation process, especially if multiple SLOSL views are in use. Common Subexpression Elimination [Muc97], a well-known technique in compiler optimisation, can be deployed to extract and precompute common node attributes, expressions or sub-views of different views.

Also, note that the above template is unnecessarily redundant due to the intent of providing a single statement. One redundant subexpression inside the statement itself is the evaluation of the WHERE and HAVING expressions against the source tables, which occurs a second time during RANKED evaluation. Future work could investigate various optimisation strategies in this context.

8.2.2 Interpreting SLOSL

Where the SQL mapping leads to a rather heavyweight implementation, a more promising approach for a SLOSL implementation is a dedicated interpreter, which was also developed as part of this thesis. It is currently used in the SLOSL Overlay Workbench to visualise topologies (see 8.1.3).

The interpreter is written in the Python language. It uses a generic sequential stream evaluation engine based on chained Python generators. The evaluation process follows the step-by-step scheme presented in section 5.1. For simplicity, the database is represented as a simple set of hash tables (or *dictionaries* in Python terms) that map node identifiers to node objects. The entire generic SLOSL infrastructure of database, views and node implementations is implemented in about 700 lines of code, of which the evaluation engine itself occupies less than 200 lines.

The MathDOM XML library [Beh05a] is used to transform arithmetic expressions and boolean expressions of the different SLOSL clauses from platform independent MathML into Python expressions. Their evaluation then deploys the Python interpreter directly. MathDOM also originated from the work on this thesis. It already supports a variety of target languages.

On a 1.6 GHz AMD64 machine, the simulator builds a static Chord network of 128 nodes in a key space of 2^8 IDs in about 3.7 seconds. This includes the database setup and involves 1,024 ($8 * 128$) node evaluations in 128 views, i.e. 131,072 in total. According to profiling data, a single view evaluation takes less than 2 milliseconds in these tests, a single bucket can therefore be evaluated in about 0.25 milliseconds.

These numbers are already in an acceptable range for a deployed system such as a current peer-to-peer overlay. Still, the interpreter uses a generic evaluation engine without any further query optimisation techniques. Future implementations of OverML will generate efficient overlay implementations from specifications through a model transformation process. This will allow the integration of SLOSL optimisers that tune the resulting code for the specific SLOSL statements used.

8.3 OverML Model Transformation

The interpreter described above is helpful for testing and debugging environments. In the long term, however, *model transformation* and *source code generation* will become the preferred way of generating deployable overlay implementations from OverML specifications.

In the context of model driven engineering (see also section 9.7.3 on related work), model transformation denotes the transformation of a higher-level platform independent model (PIM) into lower-level platform independent or platform specific models (PSM). The generation of a PSM usually involves some form of source code generation for a (general purpose) programming language. See also section 9.7.4 of the related work chapter. This section describes a transformation of OverML to object oriented languages in multiple steps.

First, the OverML specification is transformed into a platform independent XML object model by a language agnostic XSL transformation. Then, the active database and its event-driven query execution paths are constructed. In the last step, a language specific generator transforms this representation into source code for a programming language.

This section describes the complete transformation process. However, the current implementation does not cover it in its entirety. The platform independent model transformations are available, but no SLOSL optimisers were implemented and the platform specific source code generators stay at a very preliminary level. The goal was not to provide a reference system, but rather to make the architecture itself visible and accessible. It is the nature of a Model Driven Architecture that different target environments require (or encourage) very specialised implementations. A single reference implementation would be of limited help in this context.

8.3.1 Transforming OverML to an Object Model

In the first step, the four languages are transformed into a representation suitable for creating object oriented source code. NALA types are used during the transformation process to provide data type mappings into the target language. NALA attributes are combined into a node class for the database.

SLOSL views become view objects with corresponding view node classes. The clauses stay mainly unchanged, as their transformation depends mostly on platform specific parameters such as the database implementation.

HIMDEL is expanded into separate message representations containing object hierarchies of containers and headers. This step deploys the same transformation as for building the field access paths (see section 6.4.1). The XSL stylesheet for this transformation is provided in appendix A.6).

EDSL already represents state objects but still requires some transformation for simplification. First, the EDSM graph is flattened to remove the subgraph structures. Then, the transitions are moved into the corresponding source states to honour their normal execution from within the active states.

EDGAR requires no further transformation, as it has a straight forward mapping to conditional statements in procedural code.

The transformations are done independently for each language. The only exception is SLOSL, which depends on NALA for type inference. This transformation of SLOSL into view objects is a little more complex than for the other languages. Appendix A.5 has the complete XSL template. The main part of it follows:

Listing 8.3: Main XSL Template for Object Representation of SLOSL

```
<xsl:template match="slosl:statement">
  <xsl:variable name="classname">
    <xsl:call-template name="capitalize">
      <xsl:with-param name="name" select="@name"/>
    </xsl:call-template>
  </xsl:variable>

  <class access="public" name="{concat($classname, 'View')}" inherits
    ="View">
    <xsl:if test="slosl:buckets[@inherit = 'true']">
      <xsl:attribute name="inherit_buckets">true</xsl:attribute>
    </xsl:if>

    <constructor>
      <param name="views"/>
      <xsl:for-each select="slosl:parent">
        <parent><xsl:value-of select="string()"/></parent>
      </xsl:for-each>
      <xsl:for-each select="slosl:with[math:*]">
        <assign field="{@name}">
          <xsl:apply-templates select="math:*" mode="copy-expression"
            />
        </assign>
      </xsl:for-each>
    </constructor>

    <xsl:for-each select="slosl:with">
      <xsl:variable name="option_name">
        <xsl:call-template name="capitalize">
          <xsl:with-param name="name" select="@name"/>
        </xsl:call-template>
      </xsl:variable>

      <viewoption name="{@name}">
        <xsl:apply-templates select="math:*" mode="copy-expression"/>
      </viewoption>
    </xsl:for-each>

    <class access="private" name="ViewNode" extends="Node">
      <constructor>
        <param name="node"/>
        <xsl:for-each select="slosl:buckets/slosl:foreach/@name">
          <xsl:sort select="string()"/>
          <param name="{string()}"/>
        </xsl:for-each>
        <xsl:apply-templates select="slosl:select" mode="
          classgen-constructor"/>
      </constructor>
      <xsl:apply-templates select="slosl:select" mode="classgen"/>
    </class>

    <method name="select_nodes">
      <xsl:apply-templates select="slosl:where[math:*]"
        mode="copy-expression"/>
      <xsl:apply-templates
         select="slosl:buckets[@inherit != 'true' and slosl:foreach]"
        mode="copy-expression"/>
      <xsl:apply-templates select="slosl:having[math:*]"
        mode="copy-expression"/>
      <xsl:apply-templates select="slosl:ranked[string(@function)]"
        mode="copy-expression"/>
    </method>
    </class>
  </xsl:template>
```

Note the *select_nodes* method at the end that basically copies the main SLOSL clauses to the result. There is no further transformation done at this point as implementations may have very different ways of handling the clauses. Also, there are two distinct use cases of these clauses that frameworks will likely implement differently. One is the computation of the complete view content and the second is about efficient update propagation, as explained in section 8.3.2.

When we apply the above template to the SLOSL statement that was presented for the Chord fingertable view in chapter 5, it outputs an XML result like the following. For readability, some of the longer MathML expressions were replaced by their shorter infix term equivalents.

Listing 8.4: Example for the Object Representation of a SLOSL Statement

```
<classes
    xmlns="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/codegen"
    xmlns:slosl="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/slosl"
    xmlns:m="http://www.w3.org/1998/Math/MathML">
  <class access="public" name="ChordFingertableView" inherits="View">
    <constructor>
      <param name="views"/>
      <parent>db</parent>
      <assign field="log_k">
        <expression>
          <m:math><m:cn type="integer">5</m:cn></m:math>
        </expression>
      </assign>
    </constructor>

    <viewoption name="log_k">
      <expression>
        <m:math><m:cn type="integer">5</m:cn></m:math>
      </expression>
    </viewoption>

    <class access="private" name="ViewNode" extends="Node">
      <constructor>
        <param name="node"/>
        <param name="l"/>
        <assign field="id">
          <expression>
            <depends type="attribute" nala_type="id256">id</depends>
            <m:math><m:ci>node.id</m:ci></m:math>
          </expression>
        </assign>
        <assign field="local_chorddist">
          <expression>
            <depends type="attribute" nala_type="id256"
                     >local_chorddist</depends>
            <m:math><m:ci>node.local_chorddist</m:ci></m:math>
          </expression>
        </assign>
      </constructor>
      <nodeproperty name="id"               nala_type="id256"/>
      <nodeproperty name="local_chorddist"  nala_type="id256"/>
    </class>

    <node_selection>
      <slosl:where>
        <expression>
          <depends type="attribute" nala_type="boolean"
                  >alive</depends>
          <depends type="attribute" nala_type="boolean"
                  >knows_chord</depends>
          <!-- MathML: node.knows_chord and node.alive -->
        </expression>
      </slosl:where>
      <slosl:buckets inherit="false">
        <slosl:foreach name="l">
          <expression>
            <depends type="identifier">log_k</depends>
            <m:math>
              <m:interval closure="closed-open">
                <m:cn type="integer">0</m:cn>
                <m:ci>log_k</m:ci>
              </m:interval>
            </m:math>
          </expression>
        </slosl:foreach>
      </slosl:buckets>
      <slosl:having>
        <expression>
          <depends type="identifier">l</depends>
          <depends type="attribute" nala_type="id256"
                  >local_chorddist</depends>
          <!-- MathML: node.local_chorddist ∈ [2^i, 2^(i+1)) -->
        </expression>
      </slosl:having>
      <slosl:ranked function="lowest">
        <slosl:parameter>
          <expression>
            <m:math><m:cn type="integer">1</m:cn></m:math>
          </expression>
        </slosl:parameter>
        <slosl:parameter>
          <expression>
            <depends type="attribute" nala_type="id256"
                    >local_chorddist</depends>
            <m:math><m:ci>node.local_chorddist</m:ci></m:math>
          </expression>
        </slosl:parameter>
      </slosl:ranked>
    </node_selection>
  </class>
</classes>
```

Once again, this extract is rather verbose (mainly due to the usage of MathML), but it is only used as an intermediate result of the transformation process and thus handed from one translator program to another. The listing shows an object-oriented, but platform-independent representation of the SLOSL view implementation. The expressions were annotated with dependency information. This helps in type inference and simplifies the implementation of language specific generators and optimisers.

As the example shows, each of the generated view objects has a specific

ViewNode class associated with it. It represents the selection and calculation of attributes specified by the SLOSL SELECT clause.

Normally, node objects of views are read-only. The reason is that node attributes of views can be calculated in SELECT expressions on their way from the database. There is not necessarily an inverse to that expression (or function), so attribute writes cannot always be propagated to the database. Attributes must therefore be updated directly in the database.

8.3.2 Efficient Event-driven SLOSL View Updates

One of the major goals of SLOSL and the Node Views architecture is to support the efficient integration of multiple overlays. The shared database provides the main foundation for this integration through locally consistent views. However, the local evaluation of these views allows for further integration that can substantially increase the run-time performance.

The execution of SLOSL statements is naturally event-driven. When harvesters update attributes and add or discard nodes, the architecture must determine which views are affected and reevaluate them, both to provide consistent views to controllers and routers, and to generate the related view events. 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 size.

SLOSL, however, is a declarative language that provides high-level semantics. Its evaluation can be implemented in different ways, depending on the requirements. Chapter 5 already presented a number of example statements and the reader may have noticed that there is a certain overlap between them. They all rely on an *id* attribute, for example, but otherwise differ in their specific dependencies. They all deploy a similar WHERE clause. These redundancies and particularities can be exploited to further integrate their evaluation and to minimise the impact of changes in the database.

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 some of these attributes are usually declared by NALA 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

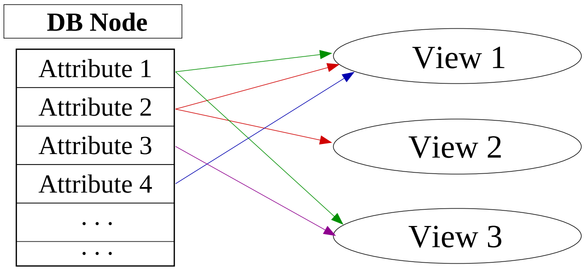


Figure 8.5: Example of an Attribute Dependency Graph for Multiple Views

therefore start by building a dependency graph like in figure 8.5. 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 8.6. As in the previous examples, different 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 dependencies. While the WHERE clause can only depend on the parents of the view (FROM) and on view options (WITH), 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 respective 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

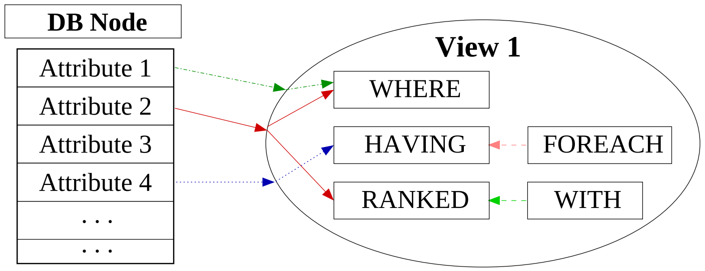


Figure 8.6: Example of an Attribute Dependency Graph between SLOSL Clauses

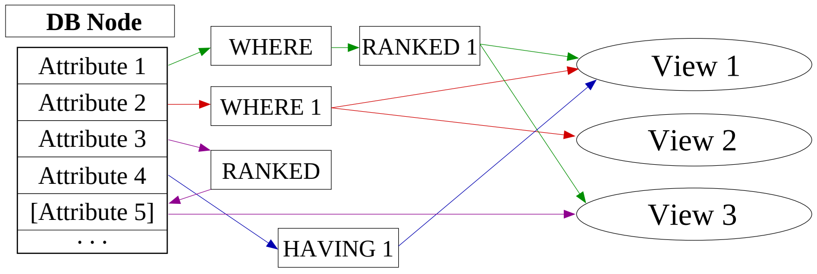


Figure 8.7: Example of a merged Attribute Dependency Graph with an additional dependent attribute in the database

entirely if the result of the WHERE clause stays unchanged. The same applies to the second attribute in the figure. Its update enforces the evaluation of the RANKED expression only if the WHERE clause succeeds or changes.

As mentioned before, independent expressions (even partial expressions) become candidates for pre-evaluation. Their results can be stored in the database as *dependent attributes* (see 6.1). The previously presented examples all allow for pre-evaluation of the ranking expression, as the additional attribute in figure 8.7 illustrates. Another possible pre-evaluation can occur when updating multiple views. Common clauses, subexpressions or evaluation steps can be extracted from the specifications, merged and executed in a different order. This reduces the redundancy between views and therefore the cost of updating them. Figure 8.7 shows an example where the evaluation of clauses and subexpressions was partially merged between views.

In SLOSL implemented overlay software, the detailedness of the dependency graph and the application of possible optimisation techniques 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 cost of finding them does not effect the run time performance. SLOSL optimisers will therefore be able to generate optimal evaluation plans for each specific update event used in a model and source code generators can build efficient execution paths from each of them.

The semantics of SLOSL allow for various other approaches. Optimisers may decide to deploy indexes on certain subexpressions. The bucket structure can be layed out and tested for holes and overlaps. This allows to see if nodes are uniquely mapped to buckets, in which case hashing or indexing become very efficient evaluation strategies for the HAVING–FOREACH clauses.

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 tradeoffs into configurations and parametrisations of frameworks and model transformations, away from source code implementations of each single overlay system. Profiling one overlay implementation can thus yield new optimisations that all other SLOSL implemented overlays can immediately benefit from.

8.3.3 Transforming the OverML Object Model to Source Code

The generation of source code is obviously a very platform specific transformation. It completely depends on the specific framework and programming language. However, the object representation that was generated so far allows for a rather straight forward mapping to object oriented languages like Java, C++ or Python. The following describes the generic infrastructure that frameworks must provide and names possible mappings from the generated object representation of the OverML languages to framework specific implementations.

HIMDEL requires framework support for message serialization and a mapping to language specific data types. Both were briefly discussed in 6.4.2.

EDSL requires the implementation of a generic event-driven state machine. It must support message based networking (see HIMDEL above) and event passing between state components. This infrastructure is often implemented on top of variants of the *select* or *poll* system calls [Ste03] for Unix operating systems. Several programming languages and virtual machines provide similar capabilities, for example the NIO package for the Java virtual machine [R^+02].

Event passing between states is commonly implemented via an indirection through the framework. This is needed to support the non-blocking I/O of the above system calls. A convenient side effect is the decoupling of states, which, in combination with EDSL generated glue code, leads to decoupled components.

Many higher-level frameworks already exist that implement the majority of features required by EDSL. Examples are the Staged Event Driven Architecture for the Java virtual machine (see 3.2.1) or the Twisted framework for the Python language[3]. The remaining layer needed to support EDSL source code generation for these frameworks is rather thin, as EDSL was designed with very simple, generic abstractions.

Another interesting target environment for EDSL is the Gridkit [GCB^+04, GCBP05] (see 8.4). Its component environment, OpenCOM [CBG^+04], is designed for run-time reconfiguration. The state interfaces defined by EDSL can be mapped to glue code for OpenCOM components. Such an environment lifts the restrictions of static design-time protocols and allows for run-time software adaptation at the component layer.

[3] `http://www.twistedmatrix.com/`

Edgar is a rather simple language. It is easily transformed into conditional statements in any general-purpose programming language. Only the exclude_last element requires additional information about node identifiers defined in Nala.

Nala defines a data schema for a local data storage point, which may be represented as a database or a local object store or in any other form that seems appropriate. This requires a platform specific component that is configurable by Nala or a transformed representation (e.g. an SQL table creation statement). The second requirement is a mapping from Nala types to platform specific data types. Today, most platforms can handle SQL data types directly, but more specialised mappings will likely yield better performance.

Explicitly typed languages, like C or Java, use type declarations and casts in the generated source code. Nala provides the required type information only for the node database. Model transformers must then infer the types of node attributes in views from their dependencies on expressions in the respective Slosl statements. This step involves common algorithms known from compilers and query engines. However, it depends partially on language specific semantics of expression evaluation, which makes it a platform specific transformation.

Slosl describes the evaluation of views and their updates. There are different ways to implement views, depending on the database infrastructure. This can be as simple as a linear scan through a list of nodes followed by a sorting step to rank the nodes, or it can deploy materialised views and incremental updates. The possible implementations are as diverse as the target platforms. The first implementation of a Slosl evaluator is part of the interpreter described in section 8.2.2.

First steps towards a Slosl optimiser were described in section 8.3.2. Future implementations of OverML in different target environments will help to identify and improve platform independent optimisations, that can then become part of the model transformation process. One promising candidate for such a target environment is the Gridkit overlay framework.

8.4 OverGrid: OverML on the Gridkit

In joint work with Paul Grace at the University of Lancaster, the author developed a scheme [BBG$^+$06] for implementing OverML support in the Gridkit. Gridkit [GCB$^+$04, GCBP05] is a cross-cutting middleware for overlay deployment. Based on the dynamic OpenCOM component model [CBG$^+$04], it provides abstractions and interfaces for layering various kinds of overlays and interaction paradigms on top of each other, as illustrated in figure 8.8. Its main intention is to provide layered networking and service components that allow the middleware to support any environment from sensor or ad-hoc

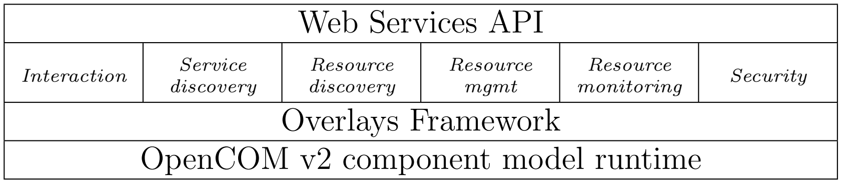


Figure 8.8: The per-node Gridkit software framework

networks to Internet-scale applications through simple reconfiguration at the component level.

The Gridkit architecture is based on two forms of layering: vertical layers for network protocols, overlays and middleware services, and a horizontal separation of concern between forward, control and state components within each overlay layer. The *control* element cooperates with its peers on other hosts to build and maintain the virtual network topology. The *forwarding* element appropriately routes messages over this topology. The *state* element then keeps per-overlay node state such as a next-neighbours list. This allows for a sufficiently fine-grained decomposition to freely stack the resulting implementations into runnable systems.

In Gridkit, the state component is specific to the overlay, and communicates through an overlay specific state interface within its horizontal layer. The control and forwarding components implement the common interfaces *IControl* and *IForward* listed below. They allow for vertical stacking of the overlay layers. The control operations are: *Create* a specified overlay, *Join* an overlay or *Leave* an overlay. The forward operations are: *Send* forwards messages to an identified destination, *Receive* blocks awaiting messages from the overlay, and *EventReceive* does the same in a non-blocking style.

```
interface IControl {
  ResultCode Create(String netId, Object params);
  ResultCode Join(String netId, Object params);
  ResultCode Leave(String netId);
}
interface IForward {
  public byte[] Send(String destID, byte[] msg, int param);
  public byte[] Receive(String netId);
  public void EventReceive(String netId, IDeliver evHandler);
}
```

According to the developers of Gridkit, this clean split is hard to achieve in practice for source code implementations. There are certain cross-cutting concerns, like the routing algorithm, that create hard interdependencies between the components. Note also that the state component is overlay specific and therefore closely tied to the implementation.

Table 8.1, provided by Paul Grace, sums up the lines of code and man days required for implementing various systems and layers, while trying to adhere to the necessary separation of concerns. It becomes clear that even within an infrastructure like Gridkit, and even for well-known overlays, the source code implementation of these systems is hard work, including reverse engineering of existing systems and source level debugging of the new implementation.

Overlay Type	LoC	Man Days
Chord Key-based Routing [SMK+01]	790	42
Chord-based Distributed Hash Table [SMK+01]	570	35
Scribe[RKCD01]	880	42
Tree-Building Control Protocol [MCH01]	736	35
SCAMP [GKM01]	1000	35
Minimum Spanning Tree	1300	20
Gossip-based Failure Monitor [vRMH98]	450	28

Table 8.1: Source code implementation of well-known overlays in Gridkit: lines of code and effort involved

The work presented in this thesis was found to be very much complementary to the Gridkit. The Node Views approach aims to simplify the design and implementation of overlays through high-level, platform-independent models. It further helps in decoupling generic components within overlay implementations and provides an integrative data layer as their basis. The languages EDSL and EDGAR and the generally Model-View-Controller based design allow for an easy mapping to the state-forward-control separation of the Gridkit.

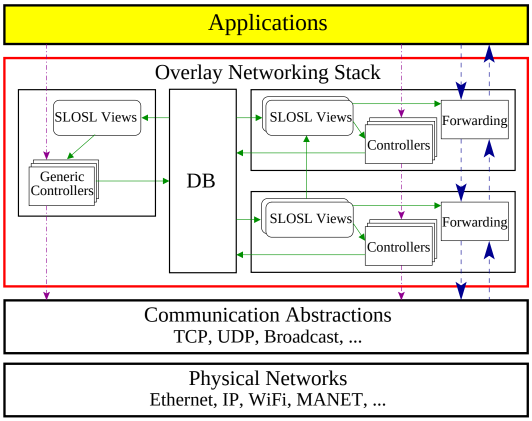


Figure 8.9: Combined architecture of Gridkit and Node Views

Figure 8.9 illustrates a combined architecture of OverML and Gridkit, named *OverGrid*. The vertical interfaces and the basic controllers are provided by Gridkit, the underlying reconfiguration architecture deploys OpenCOM, and the horizontally arranged components in the overlay stack are modelled, integrated and configured using the Node Views architecture and the OverML languages. The introduction of a database and views simplifies the separation of the control and forwarding components, that was previously found hard to achieve in Gridkit.

The main idea is to use OverML for the platform-independent design of overlay topologies, routing strategies etc., and then generate very specialised Gridkit components and OpenCOM glue code from the model. We will now briefly overview major parts of the infrastructure that OverML requires in Gridkit: database and views, control and forwarding components, and event handling.

8.4.1 Database and Slosl Views

OverGrid replaces the individual overlay state components in Gridkit overlays with a generic database exposing views to the control and forwarder components. Hence, the number of executing components is reduced, and state is more easily shared across overlay implementations. This obviously requires a platform specific Slosl infrastructure in Gridkit, namely a database and a view evaluator. Both of them are parametrised by the OverML models. The node attribute language (Nala) describes the data schema and Slosl describes the evaluation of nodes and node attributes into views (or sets of nodes).

Depending on the capabilities of the target environment, the database can be anything from a simple hash table of nodes to an object-relational database. This allows for a tradeoff between the run-time performance, the complexity of the static infrastructure and that of the code generation process. Gridkit can easily support different implementations and select between them at deployment time.

8.4.2 Control

Control components are generated mainly from the Slosl and Edsl models. The resulting event graph implementation is wrapped in an OpenCOM component that implements the IControl interface and interacts with the database through the modelled views.

Currently, Gridkit provides general, re-usable overlay control components (termed *generic controllers* in Gridkit or *harvesters* in Node Views), that provide repair and backup strategies for overlays [PC06]. These can be remodelled using Edsl to make their internal structure visible at the model level and subject to adaptation in OverGrid. A number of other possible harvester schemes are presented in chapter 7.

8.4.3 Forwarding

Forwarding is entirely driven by code generation. EDGAR is easily mapped to conditions in source code. SLOSL views, however, require the decision code to be executed in a SLOSL infrastructure or by code generated for SLOSL. As described in section 5.4, this is mainly identical to normal SLOSL evaluation and thus integrates with the database implementation. OverGrid wraps the generated code within an individual OpenCOM component that implements the IForward interface, and binds to the required views.

8.4.4 Event Infrastructure

EDSL describes the component interaction in terms of events. Implementations will commonly use a generic event-driven state machine engine. It can either interpret the EDSL graph directly or can be specialised by OverML code generators. Such a specialisation can be the generation of unique event IDs that speed up the dispatching process. It can also mean the generation of specific event objects that are forwarded between states, or of event specific handler code. The exact implementation depends on the constrains of the target environment and the effort required for the specialisation of the code generators.

8.4.5 Network Layers

As the FROM clause in SLOSL supports views of views, it can describe a layering of topologies that maps directly to layers in Gridkit as in figure 8.9. In combination with EDSL event flow graphs (and its component subgraphs), this nicely describes the event flow through the network layers and the dependencies of local decisions in one overlay layer on overlays in lower layers.

8.5 OverGrid Case Study: Scribe over Chord

This section describes a case study of designing a complex overlay implementation with OverGrid. It starts by specifying the overlay using the SLOSL Overlay Workbench and then maps the resulting OverML specification to Gridkit components. This simple walk-through does not honour the fact that design usually evolves incrementally. The real-world design process would normally follow edit-compile-test and edit-compile-deploy cycles. For clarity, this section only presents the design steps in summaries. Note, however, that the support for incremental design is a major advantage of the high-level Node Views architecture.

8.5.1 Node Attributes for Scribe/Chord

The first step in the design process is to provide a database schema, expressed as node attributes in NALA. Chord nodes require logical IDs, which are essentially large integers. We define them as having 256 bits. We will see further down that the SLOSL views for Chord and Scribe require the following additional attributes:

ring_dist is a dependent attribute based on the *id* attribute. It contains the locally calculated ring distance towards a node based on the Chord metric.

supports_chord is a boolean flag that states whether a node is known to support the Chord protocol.

alive is a boolean flag that is only true for live nodes.

triggered is a boolean attribute. It is set to true when the node did not respond to a message and is considered to be "possibly no longer alive". Another way of specifying this would be a bounded counter for failed communication attempts (i.e. outgoing messages).

subscribed_groups is a set of group identifiers (256 bit IDs) that a node is known to be subscribed to in Scribe.

8.5.2 SLOSL Implemented Chord Topology

Chord [SMK+01] deploys two different views: the *finger table* defines the major performance characteristics of its topology and the *neighbour set* keeps the predecessor and successor along the ring to assure correctness. As seen in section 5.1, SLOSL implements the finger table as follows.

```
CREATE VIEW chord_fingertable
AS SELECT node.id , node.ring_dist , chord_bucket=i
FROM node_db
WITH log_k = log(|𝒦|)
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(1, node.ring_dist)
```

The neighbour set implementation contains the node with the lowest node ID further along the ring (the successor) and the node with the highest node ID backwards on the ring (the predecessor). For resilience reasons, the view specified below stores a larger number of nodes, as encouraged by the original Chord paper.

```
CREATE VIEW circle_neighbours
AS SELECT node.id , side=sign
FROM node_db
```

```
WITH ncount=10, max_id=|𝒦| − 1
WHERE node.alive = true
HAVING abs(node.id − local.id) <= max_id / 2
        AND sign*(node.id − local.id) < 0
    OR abs(node.id − local.id) >  max_id / 2
        AND sign*(node.id − local.id) > 0
FOREACH sign IN {−1,1}
RANKED lowest(ncount, node.ring_dist)
```

8.5.3 Chord Routing through Slosl Views

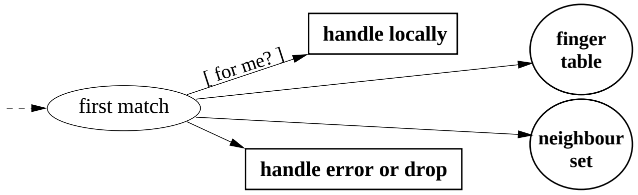


Figure 8.10: Edgar Graph for Chord Routing through Slosl Views

The routing decision graph for Chord is shown in figure 8.10. As generally described for Edgar in section 6.3, messages are pushed through the tree from the left and traverse it in depth-first pre-order. Chord generally decides responsibility for messages through the finger table view, but it can always fall back to using the circle neighbours if that fails.

8.5.4 Slosl Implemented Scribe on Chord

The Scribe [RKCD01] multicast scheme requires an additional view for routing publications. It forwards them towards the rendezvous node of the respective group and at each hop along the path broadcasts it to all subscribed children. Forwarding towards the rendezvous simply deploys Chord routing, but the broadcast requires an additional view on top of the Chord topology that selects only subscribed neighbours.

The selection is based on the active subscriptions of nodes in the finger table. All locally active subscriptions are given by the well defined set containing the subscriptions of the local node or its children towards the parents. It includes the group IDs for which the local node is the rendezvous node and for which children or the local node are subscribed. The Scribe implementation requires a view for each of the locally active groups as follows.

```
CREATE VIEW scribe_subscribed_children
AS SELECT node.id
FROM chord_fingertable
```

```
WITH sub
WHERE sub in node.subscribed_groups
```

8.5.5 Multicast Routing through Slosl Views

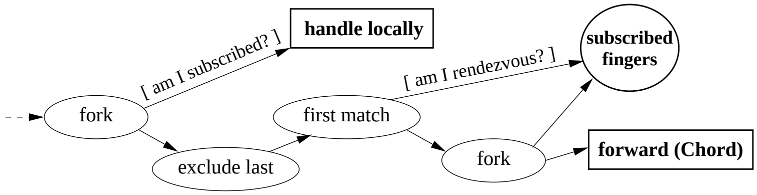


Figure 8.11: Edgar Graph for Multicast Routing through Slosl Views

The routing decision graph for Scribe publications on Chord is shown in figure 8.11. Along the path, the execution is forked into different branches, which treat the message independently for local subscriptions, neighbour subscriptions, and forwarding to the rendezvous node using the lower-level router (Chord in this case). The “exclude last” property prevents the last hop of a received message from appearing amongst the selection of next hops further down the tree.

8.5.6 Message Specifications in Himdel

Instead of using the verbose XML representation of Himdel, the specification is presented in a shorter form here. The names in brackets are the access names used for accessing fields and structure of messages. Remember that the programmatic interface of messages also defines a *last_hop* field (if it is known) and a *next_hop* field (if it was determined by a router). For transmission, OverGrid uses a binary format as described in section 6.4.2.

- Header [chord]
    - Container [ids]
        - id [sender]
        - id [receiver]
    - Message [chord_joined]
    - Message [chord_find_successor]
    - Message [chord_find_predecessor]
    - Message [chord_notify]
    - Message [chord_update_fingertable]
        - View-Data [finger_table_bucket] → chord_fingertable/bucket
    - Header [scribe]

- Message [scribe_create_rendezvous]
- Message [scribe_join]
- Message [scribe_leave]
- Message [scribe_publish]
  - Data [event]

8.5.7 Event Handling in EDSL

This is the main part where Gridkit integrates with the code generation process. Gridkit components implement the controllers that are connected by the EDSL graph specification. Again for clarity, this section does not describe the entire protocol graph of Scribe on Chord that implements the IControl component. It rather presents an example event processing cycle that shows how the system responds to events in a non-trivial way. The leave process in Scribe, which implements the propagation of unsubscriptions from groups, is a good example for this purpose. The complete process is presented in figure 8.12. The vertices are EDSL states (Gridkit implemented controllers), solid lines represent EDSL transitions and dashed lines represent programmatic actions of controllers, that are not covered by EDSL.

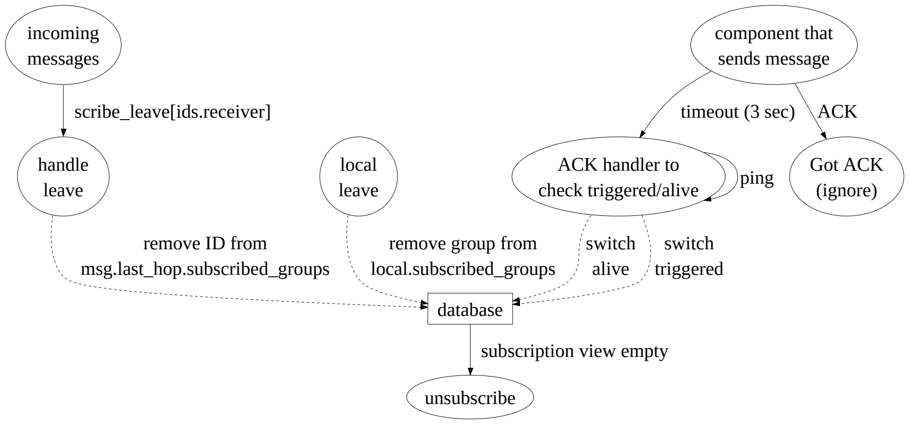

incoming messages
scribe_leave[ids.receiver]
handle leave
local leave
component that sends message
timeout (3 sec)
ACK
ACK handler to check triggered/alive
ping
Got ACK (ignore)
remove ID from msg.last_hop.subscribed_groups
remove group from local.subscribed_groups
switch alive
switch triggered
database
subscription view empty
unsubscribe

Figure 8.12: Event processing for explicit and implicit leaves in Scribe

There are three cases in Scribe that trigger a leave. The first one is the local leave that unsubscribes the local node. The second one is the explicit leave where a neighbour sends an unsubscribe message for a group. The third one is the implicit leave where the local node loses the connection to a subscribed neighbour. The first two cases are mainly identical in handling, the implicit one requires additional logic to decide that a leave must be triggered.

In this case, any component that sends messages and expects some kind of acknowledgement for them has two transitions coming out of it: one for receiving the ACK and one for a timeout. Only one of them will ever be triggered, so whatever comes first will determine the further execution path.

If the ACK is received first, all is fine. If, however, the timeout comes first, it triggers the destination state of the timeout, which is a generic ACK handler controller. This controller looks up the *next_hop* in the respective message and switches its *triggered* attribute through the database API. If it becomes true, the controller triggers the node by sending it a ping and then terminates. The same controller will handle the timeout of this ping. If, however, the attribute was true already and becomes false now, the controller sets the *alive* attribute of that node to false. These changes will trigger events from the database. In our example, the update event of the *triggered* attribute is not used, but all SLOSL views must be updated for the *alive* event.

For simplicity, we will assume that the node was only contained in the finger table view and has open subscriptions in the Scribe subscriptions view. Containment can be decided in different ways depending on the database and view implementations. If materialised views are used (which may be the simplest implementation anyway), it is easy to check if a node is visible in a view. Otherwise, the view has to be evaluated for the node, once with the original attributes and once with the modified attributes, to determine a change. If the node update did not impact the view content, no view update is needed and no events are generated. Note that the SLOSL statements in an OverML specification provide all semantics necessary to determine efficient evaluation plans at compilation time.

In our example, we assumed that the node was contained in the finger table. It will therefore disappear from the view after the *alive* update. This triggers the event that the *node left* the finger table view. The same will happen for the scribe_subscribed_children views that inherit from the finger table. Our current Chord implementation can ignore these events, as its routers only use the consistently updated views and therefore do not require any notifications about updates. Similarly, the Scribe implementation can ignore them as long as there are nodes left in all subscription views. Therefore, in the simplest case, event handling ends just here.

In the case where the failed node was the only subscriber for a group, however, Scribe has to unsubscribe from that group. This is done through the *view empty* event that is triggered whenever a view update leaves the view empty. Note that the same event is triggered when a controller receives an explicit unsubscription from the last subscribed neighbour in a group and deletes the group ID from its *subscribed_groups* attribute. This will delete the last node from the subscription view of that group and thus trigger the event.

The *view empty* event of subscription views is therefore connected to an *unsubscribe* state in the EDSL implementation of Scribe. For each event, it sends out an unsubscribe message towards the rendezvous node of the respective group by using the underlying chord router.

8.5.8 From Specification to Deployment

Once the major characteristics of the overlay stack are defined, we can make the system runnable step by step. We start by running tests within the workbench. The SLOSL visualiser (see 8.1.3) allows the developer to play with simple scenarios. This helps in finding topological problems in the specification before starting to work on the controllers.

Since the router components are generated completely based on SLOSL and EDGAR, the main portion that remains to be implemented is the logic behind the IControl interface. It is internally structured by the EDSL graph. At the beginning of the coding step, it is helpful to use dummies for controllers that are not yet implemented. Their interfaces are defined by their EDSL interaction, so this is simple to do in code generators. Also, as figure 8.12 suggests, controllers generally tend to be very simple and small, which allows for predefined tool sets of components and quick-and-dirty stub implementations to get the system working. From the EDSL graph, it is immediately clear which components are required to make a specific portion of the protocols work.

The following steps obviously depend on the available tool support. Environments for testing, debugging and simulating overlay implementations are not yet available for the OverGrid architecture. Its main achievements in this area, however, are the rich semantics that become available to debuggers and the possibility to write these tools for generic OverML models and the generic Gridkit/OpenCOM component architecture. Once available, these developer tools will not require any adaptation to the specific overlay implementations. Current overlay simulators are very much focused on specific requirements of the specific overlays they were written for. OverGrid implemented overlays, on the other hand, will seamlessly move between different OverGrid compatible simulators, visualisers, debuggers and deployment environments without major redesign or rewrites. Throughout the testing phase, the designer will be free to go back to the design phase, modify the models and regenerate the overlay implementation.

The main difference between environments for testing, simulation and different deployment scenarios is the Gridkit configuration. Different networking stacks (starting with the physical protocols), different IControl components, different subgraphs in the EDSL model and different controller implementations can be used to adapt the system to the current environment at an arbitrary granularity. Due to the OpenCOM component model, all of this can be configured at deployment time.

Chapter 9

Related Work

Contents

Related work exists in a number of areas. First of all, there is a considerable amount of peer-to-peer systems and overlay systems that were proposed and/or implemented. Some of these were already mentioned in the previous chapters, especially in 2.3. A broader overview is given in 9.1 and 9.2. Section 9.3 refers to evaluation studies that compare these systems.

There are two major books that provide a broad overview of the area of overlay networks and peer-to-peer systems. The popular book edited by Andrew Oram in 2001 [Ora01] provides a collection of articles written by a variety of well known people, both from research and industry. The second book, edited by Steinmetz and Wehrle in 2005 [SW05], contains a newer collection of more research oriented articles. The overall breadth of the collection provides a well readable overview of various research directions and an excellent foundation for future work.

One of the intentions behind this thesis was the integration of different overlay networks. Such an integrative approach must prevent exhaustive resource usage when maintaining multiple overlays on top of a single physical network. Related work in this area is presented in section 9.4.

Section 3.2 provided an overview of previous approaches in the area of frameworks and middleware systems for the design of these overlay networks. Further systems and other areas of network and topology design are presented in section 9.5. The general Software Engineering approaches that underly the Model Driven Architecture and CASE tools are briefly overviewed in 9.7.

9.1 Unstructured Overlay Networks

The field of overlay networks is commonly divided into *unstructured* and *structured* networks. There is, however, a substantial number of hybrid approaches and those that reduce the unstructuredness to achieve certain characteristics, so the differences are steadily diminishing since their first distinction.

Structured or Key-based Routing networks, as described in 9.2, explicitly target a well defined topology that is actively maintained by each participant. Unstructured networks, on the other hand, do not actively maintain a specific topology. Their properties rather result from the use patterns in their deployment.

9.1.1 Power-law Networks

Most unstructured overlay networks form Power-law topologies, also known as Small-World Graphs [Mil67, WS98]. The oldest and most well-known of them was the original Gnutella[1]. Their random nature provides relatively high resilience combined with a small network diameter, which has also been popularised as the *six degrees of separation*[2].

The currently deployed overlays are commonly used for file sharing and decentralised keyword search. Their main advantage is that potentially large documents are statically stored on the source nodes, so that only the small queries need to traverse the network. However, due to their size, complete coverage is extremely expensive. Hence, these systems do not provide any guarantees regarding consistency or availability of data, since queries may not be forwarded to all potential sources and nodes may fail arbitrarily.

Evaluations of Gnutella have shown that its random nature makes it highly resilient against random failures [SGG02]. This was found by crawling a live Gnutella network of about 1800 nodes. If the simulated failures are targeted, however, the authors find that it is sufficient to destroy only the most highly connected 4% of the nodes to leave the graph disconnected. A visualisation of

[1]`http://en.wikipedia.org/w/index.php?title=Gnutella&oldid=72696718`

[2]`http://en.wikipedia.org/w/index.php?title=Small_world_phenomenon&oldid=72567335`

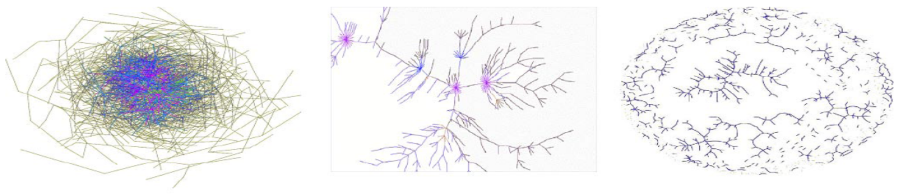


Figure 9.1: Resilience of a Gnutella graph: topology as found by crawler (left), connected after 30% random failures (middle), disconnected after 4% targeted failures (right). Source: [SGG02] (authorised)

these finding is presented in figure 9.1. This fraction obviously depends on the total number of nodes and the distribution of out degrees, so the resilience can be expected to grow with the network. However, this still shows that it is worth putting consideration into the selection of neighbours and the maintenance of the overlay, even if the general topology is randomised.

There were a number of proposals for semantic grouping in unstructured networks, including [Jos02, NS02, BLZK04]. Their advantage is the topological vicinity of related documents, which simplifies searches. However, one of the problems here is the definition of semantic vicinity. These systems require the availability of meta data, which in turn may require global knowledge of document properties or larger data exchanges between random nodes. This can be a substantial barrier for newly joining nodes.

A number of other approaches were proposed to improve the search characteristics of early unstructured networks. Chawathe etal. [CRB$^+$03] evaluate a number of proposals and finally describe *Gia*, a system based on Gnutella that combines the most promising tweaks.

Such an approach is substantially more promising than the implementation of keyword search on top of Distributed Hash Tables as in [LLH$^+$03] (see 9.2). Due to their hash table nature, these systems must rely on distributed joins of potentially large data sets as well as global knowledge about the available documents, e.g. to compute the Inverse Document Frequency (IDF) as known from Information Retrieval. This intuition was also expressed in [CRB$^+$03].

A hybrid solution is presented in [LHSH04]. The authors combine the rather exhaustive search capabilities of unstructured networks with the lookup semantics of Distributed Hash Tables. This enables broad searches for well replicated data as well as lookups of rare (and therefore indexable) data.

9.1.2 Square-root Networks

While power-law topologies mimic a common pattern found in various forms of networks (including social networking), networks and replication schemes based on square-root characteristics were found to exhibit substantially better search capabilities [CS02]. As part of the work on this thesis, a routing scheme over structured overlay networks was developed [TBF$^+$04]. It is called the Bit Zipper Rendezvous and briefly described in section 2.3.6.

Kelips [GBL$^+$03] represents a variation of unstructured overlays. The participants are split into k groups (called *affinity groups*). Each node must connect to some (or all) nodes in its own group and a small number of nodes in each other group. Data is eventually replicated within the group of the source node by epidemic communication. The authors determine $k = \sqrt{N}$ as a reasonable and efficient number of groups. However, the problem of estimating N or $\sqrt{N}$ without global knowledge is not touched. Similarly, a balanced distribution of nodes over all groups may turn out to be hard to achieve and to maintain.

Another example for an unstructured implementation was recently presented in [Coo05]. The author proposes a highly connected topology where each node keeps $O(\sqrt{N})$ neighbours, based on a local estimate. The resulting network shows significantly better search performance than power-law topologies and proves to be optimal for random walk searches.

All of these systems give interesting evidence that square-root topologies provide a number of interesting performance characteristics. Their application can make peer-to-peer search systems generally more efficient.

9.2 Key-Based Routing (KBR) Networks

Key-Based Routing networks form a very interesting class of overlays. Since scalable distributed data structures were initially proposed by Litwin and others [LNS93, KLR96], a large number of different systems has emerged that has seen a substantial growth since the year 2000.

9.2.1 Key-Based Routing

The general idea is as simple as hashing: Given a logical identifier (or key), lookup the node that is responsible for it. The problems, however, are manifold. How do we efficiently map identifiers to nodes? How do we assure a reasonable load distribution without sacrificing the lookup efficiency? How can we assure availability in the face of node failures? This section describes the general approach of KBR systems.

Routing and Responsibility

The main characteristic of all KBR networks is the imposition of a topology that is maintained by a distributed, decentralised algorithm. The topology distributes a large, numeric key space $\mathscr{K}$ (commonly 2^{128} to 2^{256} keys) over all of the participating $\mathscr{N}$ nodes. Each node is therefore assigned the responsibility for a subset of this key space. It then maintains its range jointly with a small set of neighbours.

In KBR networks, routing is equivalent to the distributed evaluation of the responsibility function. If a certain key is to be found, the topology is traversed to determine the node that is responsible for the respective range. Normally, one node is responsible for many keys, but each key is uniquely assigned to one node.

The various KBR systems differ in their responsibility function, i.e. in the way they assign responsibility ranges to nodes. Most of them use key prefixes for the distribution and then route messages based on the Euclidean distance towards the target range. Other possible metrics include the XOR distance between two keys, like in Kademlia [MM02].

Neighbours and Out-Degrees

Overlay networks mainly differ in their topology, which determines the required out-degree of each node, the freedom in the choice of its neighbours, the resilience against node or edge failures, the overhead required for topology maintenance, etc.

As described in 4.1.1, topologies commonly divide the connections of their nodes into different categories. Many KBR overlays provide some form of "short" and "long" connections. The long ones provide faster shortcuts through the topology and the short ones assure its correctness. Direct neighbours must know about each other and must keep up short connections to assure the correctness of responsibility assignments in their area. The long connections help in traversing long paths through the topology and commonly cut down the diameter to logarithmic depth. Most networks also let the number of outgoing connections depend on the total number of nodes in the network, for both short and long connections. This number is commonly proportional to $\log N$, which provides for reasonable resilience against arbitrary node failures [SMK$^+$01].

9.2.2 LH* family of Scalable Distributed Data Structures

The LH* [LNS93, KLR96] family of distributed linear hash functions are the oldest representatives of KBR-like systems. They were originally proposed by Litwin and others as a distributed computing equivalent to Linear Hashing. Since then, they have seen modifications to support a number of features for enhanced resilience, such as replication and erasure coding.

Linear Hashing deploys a set of hash functions to handle collisions and rescaling of a hash table. LH* uses this scheme to map keys to a set of nodes. It assumes partially consistent global knowledge of the participants, which means that every node can directly determine the recipient of a message. If a node erroneously receives a message that was delivered from a node with an outdated mapping, it can simply use its own hash functions and retry. Hash functions change when nodes decide to split their responsibility or to take over the (partial) responsibility of another node, commonly for load balancing reasons.

Later extensions provide support for mirroring (LH*m) and striping (LH*s), similar to hard disk RAID systems. Another interesting achievement was the combination with Reed-Solomon codes in LH*RS. They provide this system family with efficient recovery mechanisms in the face of failing nodes. Still, the general scheme shows that the LH* family was originally designed for data storage in rather static server clusters or LAN environments. Especially their support for self-maintenance is rather limited.

9.2.3 Ring Topologies

Most overlay networks are based on ring topologies. This may be due to symmetry reasons, but also for simplicity. The more complex a topology becomes, the harder it is to maintain it in a decentralised way.

Chord

Chord [SMK+01] uses a very regular ring topology, which makes it one of the more beautiful representatives of the KBR class. Short connections follow the one-way ring, long connections span ranges of $1/2^k$ for $k \in 1..\log N$. This results in binomial tree structures and $O(\log N)$ depth.

The original Chord paper is generally worth reading. It contains proofs for certain graph properties, such as the depth of $\log N$, which it even maintains with high probability under a 50% chance of arbitrary node failure. Under the same assumption, it is provably advisable to maintain $\log N$ short connections to assure connectivity. Many of these properties apply in similar form to other overlays.

There is a considerable number of extensions to Chord. They include anonymity [HW02], sub-group building [KR04] or degree optimality [NW04].

Pastry

Pastry [RD01] is similar to Chord (and Tapestry) in that it uses a ring topology for short connections. The long connections, however, are determined using a prefix routing table. Every Pastry node has its own key, which it stores as a fixed-base number (commonly hexadecimal). For each prefix of that number, it tries to find nodes with keys that differ in the next digit, to make them long connections corresponding to this prefix plus the digit. The topology depth depends on the prefix length for which nodes can be found. Given an equal node distribution over the key space, the depth becomes $O(\log N)$.

A variant of Pastry, called Bamboo [RGRK04], replaces parts of its maintenance algorithm by epidemic communication [DGH+87, Bir02, JGKvS04]. In an evaluation, the authors show that Chord and Pastry require a very high maintenance overhead under frequent node failures and joins (high churn rates). Bamboo aims to decouple the frequency of maintenance messages from this rate, so that it only depends on the number of nodes in the system. Measurements show that this yields considerably better scalability.

On the down side, Bamboo does not achieve a complete decoupling, since nodes still have to ping other nodes that they heard of, to assure their actual availability. It is also worth mentioning that Bamboo trades correctness for scalability, argument being that correctness is not achievable under high churn rates.

Tapestry

Tapestry [ZKJ01] is often cited together with Pastry. Both have mainly identical topologies and similar properties. Tapestry emerged from the OceanStore project[3] [KBC+00].

SkipNet

SkipNet [HJS+03] inherits its name from skip lists [Pug89], a linked list data structure that speeds up traversals by adding long links that skip over a number of entries. SkipNet is similar to a double-sided Chord. Its short connections follow a two way ring, the long connections are selected based on bit prefixes. An interesting feature is that the long connections result in a recursive ring structure.

A difference to Chord is the node order along the ring. SkipList uses a twofold addressing scheme based on literal domain names and numeric identifiers. The names are used to distribute nodes along the ring, which potentially improves locality of neighbours and can support domain internal routing. Bit prefixes of the numeric identifiers are then used to join scattered nodes into sub-rings that skip over the ring.

HyperRing [AS04] provides a similar topology that extends skip graphs to support provable properties for advanced queries, including range queries.

9.2.4 Trees

P-Grid

P-Grid [Abe01] is one of the very few overlays that build directly on a tree structure. Short connections form an unbalanced tree, long connections into

[3] `http://www.oceanstore.org/`

other subtrees support the efficient navigation across the tree. The expected depth of the overall graph lies within $O(\log N)$, which renders balancing the tree unnecessary.

P-Grid has a long tradition in being ignored by citations, although its tree structure inherently provides many interesting properties that other overlays lack or struggle to provide. The original publication is also worth reading. It gives some insights in the self-organisation aspects of the system and was one of the few papers at that time that cited closely related work in the early 1990s.

Trie Overlay

Freedman etal. describe another of the few tree structures in the overlay area, a distributed trie [FV02]. While this is generally an interesting idea, the maintenance costs are relatively high and it is not clear how well the correctness of the topology can be maintained under frequent failures.

9.2.5 De-Bruijn Graphs

There are a number of overlays that are based on de-Bruijn graphs [dB46]. They do not require any long connections, as their topology provides them with logarithmic depth while keeping a fixed out-degree at each node. Examples are ODRI [LKRG03], Koorde [KK03] and Omicron [DMS04]. In the simplest case, these networks only require a constant out-degree of two per node in their directed graph, which obviously makes them degree-optimal. De-Bruijn graphs are frequently used in static low-latency networks [II81], e.g. in clusters and super computers.

All de-Bruijn overlays have the problem of being hard to maintain under frequent node failures. If the node count is not close to a power of two, they become irregular, which means that they have to take measures to replace missing nodes by means that are external to the topology. This often requires non-local actions. Their main problem, however, is the low degree, which requires them to proactively take additional measures to prevent topology disconnection when neighbours fail. Also, neighbour connections are exactly predicted by the topology, there is no freedom of choice, which effectively prevents optimisations through topology adaptation.

Koorde

Koorde [KK03] is implemented based on Chord, but it replaces the topology by a de-Bruijn graph. The paper presents two different configurations, one with a fixed number of neighbours (e.g. 2), where the depth is optimal and the maintenance is as complex as in other de-Bruijn implementations. The second configuration keeps $O(\log N)$ neighbours, which increases the resilience and meats the lower bound of $O(\log N/\log\log N)$ depth for this case. There is no evaluation comparing resilience and maintenance overhead against other systems.

ODRI

ODRI [LKRG03], the Optimal Diameter Routing Infrastructure, similarly deploys de-Bruijn graphs. It is evaluated by a graph theoretical comparison with previous KBR topologies that shows the major advantages of de-Bruijn graphs. The maintenance issues are not part of the comparison.

Omicron

Omicron [DMS04] is another system where the authors have realized the problems regarding topology maintenance. They try to circumvent them by replacing the potentially unreliable participants by more resilient clusters of nodes. The idea is to split the monolithic overlay service that each node normally has to provide into separate tasks that are then distributed within the cluster. Node heterogeneity is supported by assigning different roles based on a node's capabilities. The available roles are taken from a DHT application: Routing, Caching, Indexing and Maintenance. The demands increase in this order and form a service hierarchy.

The authors propose incentives for nodes to take over higher-demanding services. Their revenue is given by a higher position in the hierarchy that frees them from lower-level tasks. On the other hand, lower nodes have an incentive to provide better service to nodes above them in the hierarchy as they benefit from their service. It is somewhat unclear if these incentives are beneficial. They work as long as all participants follow the rules (in which case their actual need is questionable), but when nodes break the rules, it is unlikely that the incentives will be maintained by the system. The overall idea of providing heterogeneous services for heterogeneous nodes, however, can help in the design of more scalable systems.

9.2.6 Other Topologies

CAN

CAN [RFH+01] has one of the most complex topologies amongst the KBR overlays. It is based on a d-dimensional Torus in which each node is responsible for a part of its surface. Routing does not use any long connections. It is based on the Euclidean distance at the surface, which leaves the diameter of the topology within $O(N^{1/d})$. Note that d is fixed at deployment time. There is an extension that provides CAN with a logarithmic diameter [XZ02] by adding long ranging connections.

Kademlia

Kademlia [MM02], as opposed to most other KBR networks, does not apply the Euclidean metrics, but uses the XOR-metric for its responsibility function. Therefore, its key space it not a ring but a flat interval. Apart from that, long connections are established just as in Chord, which gives it basically the same SLOSL implementation. Nodes are chosen from the interval $[2^i, 2^{i+1}], i \in [0, \log|\mathscr{K}|)$, resulting in logarithmic depth. As an interesting property, Kademlia can vary the number of bits that are resolved at each routing step.

HyperCup

HyperCup [SSDN02] was the first system to deploy a distributed hypercube. As with de-Bruijn graphs, this topology is commonly deployed in static interconnection networks of super computers. Consequently, it requires a considerable amount of maintenance in a dynamic environment and the resulting hypercubes degenerate if the number of nodes diverges from a power of 2.

9.2.7 Hierarchical Overlays and Efficient Grouping

Hierarchical overlays were proposed both for structured and unstructured networks. The latter often deploy so-called *Super-Peers* or *Ultra-Peers*, which form a subgroup of more powerful participants. They support the network by providing extensive services like general network maintenance, indexing or (long-range) forwarding. Normal nodes then connect to this infrastructure and provide data or run queries. Despite their potentially large number, the top-level of these hierarchies is somewhat similar to server farms in the client-server model. Current Gnutella versions are entirely built on top of this approach, which has lead to a remarkable increase in performance.

Hierarchies in structured overlays commonly deploy variants of flat topologies like Chord or Tapestry. A common approach is to build additional rings that accommodate more powerful nodes. They become the preferred routers in the network, which can sensibly reduce the average network distance.

The incentives are a common problem in all of these systems. Why should a node do more than it absolutely has to? Countless evaluations have observed the problem of free-riders that refuse to contribute [AH00, YZLD05]. In certain scenarios, this can become a critical problem. On the other hand, even multi-server systems like the original E-Donkey [HSS03] worked surprisingly well and apparently achieved to motivate enough contributors. The long-term success of a heterogeneous system seems to depend very much on the specific applications, the emerging use patterns and the social impact.

Brocade

Brocade [ZDH+02] was one of the first implementations of hierarchical overlays. It extends the Tapestry overlay to provide "landmarks" inside each autonomous system (or subnet) that help in optimising routing decisions. These special nodes connect to each other in a way similar to the super-peer approach of unstructured networks. The goal is to avoid long or slow routes by providing alternative paths through the faster landmark routes. The main problems of the system regard the bootstrap of the landmark overlay and the initial lookup of the nearest landmark node by the other participants.

Structured Superpeers

The Structured Superpeers approach [MCKS03] describes a two-ring extension to Chord in which $\sqrt{N}$ super nodes form an additional super peer ring. Every super node is responsible for maintaining an arc of the main ring and keeping track of all nodes in that range, in addition to the global knowledge of all super peers. Since the number of nodes in a given arc can vary over time and can temporarily differ considerably from other arcs, a load balancing scheme is deployed amongst the super peers. It allows them to split arcs and to delegate responsibilities to maintain a reasonably balanced load. To allow for super node failures, they maintain a global list of normal nodes that volunteer, sorted by the amount of offered contribution.

It is worth noting that the super peer ring only represents an efficient, but redundant extension to the normal Chord ring. This leaves the latter as the default fall-back even in the case of massive super peer failures. The authors do not investigate the possibility of epidemic communication within the super peer ring. It would therefore be interesting to see if such an approach could help in lifting the burden of maintaining the global knowledge amongst the super peers.

Coral

Coral [FM03] extends the previous approach to providing three redundant levels of rings. The innermost ring is reserved for nodes that experience a very low latency of at most 30 ms amongst all participants, whereas the middle ring allows 100 ms and includes all nodes from the inner ring. The outer most ring is formed by all nodes that participate in the system. As the best connected nodes link the rings, forwarded messages can switch to faster rings whenever they hit one of them. The faster rings have fewer participants, which both speeds up hop-by-hop latency and reduces the number of hops along the forwarding paths. It does, however, increase the load inside the inner circles.

From the publication, it is somewhat unclear how big the overhead of determining the appropriate rings is for a node. The decision requires the node to measure its latency towards a certain number of other nodes within the respective circle. It is also unclear how nodes are supposed to decide that they are in the wrong ring, as opposed to a neighbour being the wrong one. The underlying question is how the best quorum of closely connected nodes can be extracted from the total number of participants, without requiring global knowledge. The authors expect nodes to jump back and forth between the rings in order to find an optimal position. This should eventually encourage the old neighbours of the node to join its new ring. A prove for eventual stabilisation is missing.

Hierarchical Chord

Hierarchical Chord [GEBF$^+$03] is another multi-ring approach based on the Chord system. The authors require the more powerful participants in the system to form a global ring that is then used to address the entry points into sub-networks. An advantage of the system is the support for arbitrary topologies within the autonomous sub-systems. Global addressing is based on group identifiers, but the lower topologies can be anything from a star network to another Chord overlay. The global overlay can exploit redundant entry points if more than one member of a sub-networks participates in it. On the other hand, the global network is smaller than the total number of participants in the system. This shortens the paths between the different groups.

Autonomous Sub-Networks

The same idea is later presented in [MD04] for the Pastry network. The publication is very focused on locality within organisations, which allows the resulting system to be deployed in the POST application [MPR$^+$03] for decentralised email infrastructures.

Diminished Chord

Diminished Chord [KR04] presents an interesting extension to the Chord system that provides it with embedded low-overhead group topologies. It supports multicast and can be seen as an alternative to heavy weight group hierarchies.

9.3 Topologies – Comparison and Evaluation

It is generally difficult to provide analytical proves for the characteristics of decentralised overlay networks. Some properties were proven for Chord [SMK$^+$01] and Koorde [KK03] in the original publications. However, analytical comparisons between overlays are still rare.

To simplify comparisons, [CC04] presents a cost model on a per node basis. It models latency and message overhead and allows to compare different DHT topologies. Obviously, star networks yield the lowest costs. A more interesting outcome is that Plaxton trees (as used in Chord and Pastry) are relatively efficient, as is a six dimensional torus (as provided by CAN). Lower dimensional tori and de-Bruijn graphs are less efficient. The results are based on simplified assumptions such as an equal distribution of nodes and data over the topology.

Another interesting comparison of topologies is provided by [GGG+03]. The metrics include the freedom of choosing a neighbour, static resilience and support for latency adaptation. Except for de-Bruijn graphs (Koorde, Omicron) and trees (P-Grid), the authors consider all major overlay topologies that were available at the time. The paper is therefore worth reading for an overview of well known systems.

The major finding is that rings never have worse properties than the other topologies, but achieve the highest flexibility and fault tolerance. The authors conclude that ring topologies are the most promising designs for overlay networks. A drawback of the evaluation is the missing consideration of the maintenance overhead. Also, the next publication shows that de-Bruijn graphs would have been another very interesting candidate for this comparison.

[LKRG03] motivates the design of ODRI (see 9.2.5) by comparing its de-Bruijn topology to Chord, CAN and (less extensively) Pastry. The evaluation takes a graph theoretical perspective. One of the results is the finding of interesting similarities between Chord and CAN, but only under the (unrealistic) assumption of a logarithmic dimensionality of CAN. Also, the comparison with Chord is somewhat constrained as Chord is fixed at a base of 2. Higher values decrease the diameter of the graph at the cost of a higher degree.

Under the given assumptions, the evaluation yields some compelling advantages of de-Bruin based systems. They yield the lowest diameter, the lowest node degree and the highest path independence for the topology. These properties also result in a high resilience against random node failure. The optimal diameter assures that even fall-back routes are not longer than the shortest path. A drawback of the comparison is the lack of consideration of the maintenance overhead, which is considerably higher in de-Bruijn networks.

In [BNAG04], Bergström and others present preliminary work on a formalism for the static verification of KBR overlays. For the state being, they do not consider any dynamic adaptation or topology reconfiguration under which the verification holds. However, following this path further, one could imagine a similar approach based on OverML models.

9.4 Message Overhead and Physical Networks

When multiple overlays run on top of a single physical network, the aggregated overhead for their maintenance may become exhaustive. The Node Views architecture attacks this problem by integrating the data layer used by these overlays. It provides locally consistent views to different components which decouples maintenance from the frequency of routing decisions and allows to merge maintenance tasks from different overlays.

The resource sparing integration of different overlays at design time is a new approach enabled by this work. However, the general problem of reducing maintenance overhead for multiple overlays has been tackled before. The main area of improvements is the mapping between virtual topologies and the physical network, which is mainly orthogonal to the design time approach.

9.4.1 Physical Mappings

Jannotti [Jan02] presented an extension to the physical routing infrastructure. Two new primitives, *packet reflection* and *path painting*, allow nodes to incrementally improve their local part of the mapping between the virtual topologies and the physical network.

Nakao etal. [NPB03] extend this idea towards a so-called "routing underlay" to provide overlays with a more global view of the physical network. Currently, overlay nodes commonly try to determine the availability, hop-count and latency of other nodes using pings, thus possibly issuing several pings for a single neighbour selection. The authors propose to reduce the number of necessary pings by adding services to physical routers that allow estimating distances in several granularities based on BGP data (Border Gateway Protocol [RL95, Hui99]). New primitives expose BGP's view on the global graph of autonomous systems (AS) and estimate distances based on router hop-counts or AS traversals. This effectively establishes a simple data acquisition layer in the physical network, which shows a certain similarity with the more general and higher-level Node Views approach.

Birck and others developed a tool set for simulating peer-to-peer traffic and its effect on physical networks at a large scale [BHMS04]. The proposed approach is to run the simulation in two steps, beginning with an application layer simulation of the peer-to-peer traffic itself. The message logs of the test run are then fed into a network simulator (such as ns2[4]) that can simulate the resulting traffic at the physical network level. This allows a higher scalability of the simulations, as fewer layers are simulated at the same time. It is interesting to note that the Node Views approach provides an inherent level of abstraction for the design of overlay networks that might be particularly well suited for the application level simulation step.

[4]`http://www.isi.edu/nsnam/ns` (Oct. 15, 2006)

9.4.2 Message Overhead and Stability

Rodrigues etal. [RB04] evaluate the message overhead in DHT storage systems. Most of these systems have a logarithmic diameter, which allows them to keep the node degree and therefore the local maintenance overhead low. The opposite approach is a system with global knowledge which is very efficient in static networks or when data is highly replicated. The authors evaluate the overhead of swapping data between nodes when the responsibility changes. Their finding is that DHT based mass storage only make sense in relatively static networks or in extremely large systems of several million users. Global knowledge is the most efficient solution for the first case. In large systems, an increased diameter is only efficient for low replication, which is rather unlikely given the current sizes of hard disks. These findings are very specific to storage systems and do not hold for other overlay applications.

Castro etal. [CJK^{+}03] compare embedded multicast trees in overlays with external overlays per group (see 2.3.1). Embedded multicast proves to be considerably more efficient in terms of message overhead, mainly due to the increased maintenance overhead in multiple overlays.

[SGMZ04] compares different multicast designs for overlays under realistic deployment scenarios. The results show a clear advantage of low-diameter graphs for the stability of the system. On the performance side, multi-tree designs like SplitStream [CDK^{+}03] prove to provide a considerably higher throughput. Finally, clustering based on latency has a positive effect on the overall performance.

[BSV03] presents measurements of the Overnet file swapping network which is based on the Kademlia overlay. The main emphasis is on the availability of nodes. The authors can show that many nodes change their IP address over time without disappearing from the network. This effect can falsify measurement studies and usually results in an increased maintenance overhead for the overlay. Furthermore, the general availability of nodes varies considerably over the day and nodes tend to participate on a sporadic basis. While these findings may not easily map to other types of overlay applications, they show well that the meaning of availability is not easily defined in a peer-to-peer environment.

9.5 Design and Implementation of Overlay Networks and Topologies

The design and implementation of overlay networks has received interest from both the overlay and middleware communities. The major approaches that preceded and motivated this thesis were already presented in 3.2.

So far, there have been very few efforts for standardisation in this area.

JXTA[5] [JXT03] was such an effort. As described in 3.2, however, its strong focus on unstructured broadcast networks has prevented it from providing sufficiently high-level abstractions for simplifying the design of overlay systems and efficient topologies.

Two years later, a common API for KBR overlay networks was proposed by Dabek etal. [DZDS03], that has since been adopted in a substantial number of implementations. It was extracted from a comparison study of CAN, Chord, Pastry and Tapestry (see section 9.2). The paper is driven by the specific requirements of a number of different applications, namely Distributed Hash Tables (DHT), Overlay-internal Anycast and Multicast (CAST), Decentralised Object Location and Routing (DOLR) and Caching. Their requirements fall into two categories: message forwarding (or routing) and access to the local routing state, which are reflected by the API.

Given this background, the work presented in this thesis is the first to provide standardised, domain specific languages and high-level abstractions for the design of overlay networks and their topologies. There is a small set of recent work that is very closely related to the ideas presented here. This comprises two recent projects, namely P2 [LCH$^+$05] at the University of Berkeley and Gridkit [GCBP05] at the University of Lancaster, as well as a publication by Karl Aberer and others [AOAG$^+$05]. The latter and P2 are described in the following sections. Gridkit has its own section in the implementation chapter 8.4.

9.5.1 General Concepts in P2P Overlay Networks

A very recent publication [AOAG$^+$05] by Aberer and others presented a wrap up of general concepts in overlay systems. The authors identify and describe six basic design decisions in current overlays. They must agree on

- a virtual identifier space
- a mapping of resources and nodes to the identifier space
- a management scheme for the identifier space, executed by the nodes
- a topology graph that embeds the identifier space
- a routing strategy
- a maintenance strategy

The ideas behind this separation follow the known concepts of KBR networks (see 9.2). That makes the paper generally worth reading for a comprehensive overview of the theoretical concepts in the field. As a single drawback, the integration of unstructured networks like Gnutella is somewhat artificial, as it

[5]`http://www.jxta.org`

requires the introduction of unused identifiers and empty mappings to these systems.

The authors also mention their interest in folding the presented concepts back into the available overlay implementations to straighten their design. It would therefore be interesting to see if this allows for a separation of concerns also for the initial design process and hence for helpful abstractions in new frameworks.

In comparison, this thesis presents a well founded middleware architecture that takes a different focus on the design itself. Its goal is to capture overlay concepts from the point of view of topology design. The topology graph, the semantics of identifiers (if available) and the deployed routing strategy find their abstract representation in SLOSL and EDGAR. Maintenance is situated in the harvester components.

The remaining concepts (identifier mapping and management) are of a rather theoretical nature. While important for the topological design, they do not sensibly contribute to the actual implementation. SLOSL motivates that they are already covered by the design of the graph. It is therefore currently unclear what additional advantages such a further separation of concerns provides for the design of overlay frameworks.

9.5.2 Declarative Overlay Routing

Modelling overlay topologies in declarative queries is a new idea that is presented in this thesis and in previous publications of the author [BB05a, BB05b]. During the course of this work, however, a similar idea [LHSR05, LCG$^+$06] emerged independently in another recent project, called P2[6].

P2 uses distributed, declarative queries written in a dialect of the Datalog query language [CGT89] to recursively build up routing state at the network nodes. While this was initially targeted at lower level routing layers, later publications extend this approach to the design of overlay networks [LCH$^+$05]. Here, the original dialect is extended with domain specific predicates to adapt it to the more specific requirements of overlay routing and maintenance mechanisms. The resulting language was named Overlog.

The general idea of using a declarative query language dialect is the same in SLOSL and Overlog. However, the differences are also interesting to see. Overlog exploits the natural recursion of the Datalog language to model recursive routing as a distributed query execution process. This eventually results in aggregating the necessary routing state at each node. Overlay maintenance is therefore implemented as distributed query processing.

SLOSL, on the other hand, refuses to produce such a tight coupling of routing and maintenance. In fact, one of its design goals was the separation of routing

[6]`http://p2.cs.berkeley.edu/`

components on the one hand from maintenance components on the other hand, as well as different components from each other. This allows the Node Views architecture to support different ways of maintaining overlays (like static or gossip networks) through pluggable components and without modifications to the data representation or routing layer. It also enables sharing controllers and state between overlays to reduce the aggregated maintenance overhead of multiple overlays.

Both SLOSL/OverML and Overlog allow for the declarative, high-level design of overlay systems. They show that the new, declarative, data-driven design paradigm for routing decisions and overlay topologies opens up a broad and very promising new field of research in the modelling and engineering of overlay software.

9.6 Quality-of-Service in Publish-Subscribe

Following a workshop contribution of the author in [BFM06], chapter 2.2 presented an overview of quality-of-service metrics in the publish-subscribe area. There have been few publications on this topic so far. Arajo and Rodrigues [AR02] present a distinction between a *content profile* (such as the precision of a sensor) and a *QoS profile*, allowing to request periodic or sporadic delivery, as well as bandwidth requirements and latency bounds. No other metrics are considered.

Eugster etal. [EFGK03] refer to persistence, transactions and priorities as relevant metrics. In this thesis, persistence is grouped under the more general concept of delivery guarantees. While message priorities have found their place, transactions are not regarded. They place very high requirements on the broker infrastructure and it is generally questionable if they are suitable for a decoupled system model like publish-subscribe.

Outside the field of publish-subscribe, there is a large body of literature on quality-of-service in the networking and Internet environment [Hui99, ZOS00, ZDE^{+}93], especially regarding differentiated services and resource reservation. Another area of interest is messaging middleware, including CORBA [Obj02] and JMS [Sun02].

The set of metrics presented in chapter 2.2 was largely taken from previous work in these areas. The application to the publish-subscribe model and its distributed implementations, however, has not been considered before in a comparatively extensive way.

9.7 Modelling and Generating Software

Modelling has been part of the software development process for decades. In todays world of object oriented software design, the most well known general purpose modelling language is UML. Additionally, various domain specific languages are available for the high-level design of applications or their components.

9.7.1 UML

Today, object-oriented software is commonly modelled in the OMG's Unified Modelling Language (UML [Obj01]), a set of general purpose, diagrammatic modelling languages that describe various views on a design. UML supports functional models for use cases, object models for the code structure and dynamic models for the behaviour of the system. Figure 9.2 provides a complete overview of the available diagrams, figure 9.3 shows the relation of UML to a number of other software modelling techniques.

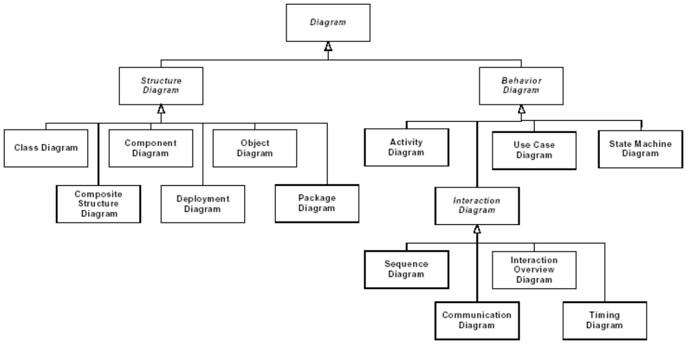


Figure 9.2: Diagrams in UML (source: [Kel05], authorised)

It is worth noting that UML is not limited to the design of software. It has been used in various other modelling scenarios including business processes and organisational structures.

One kind of dynamic diagrams in UML are *state machine diagrams*, which have clearly influenced the work on EDSL. They describe software functionality through named states and textually defined transitions. Just as EDSL, the diagrams have support for subgraphs that structure the diagram into a hierarchy of *superstates* and *substates*.

Compared to the generic UML state machine diagrams, EDSL provides additional domain specific semantics. Its integration into OverML makes it aware of HIMDEL and SLOSL, which allows the definition of very specific events and subscriptions. While a mapping of EDSL graphs down to a representation in UML is straight forward, it ignores the machine readable semantics of transitions that are available in Edsl.

9.7.2 CASE tools

Computer-Aided Software Engineering (CASE) tools are one class of widely deployed tools that support the diagrammatic design of applications through visual editors, mainly based on UML. Many of the available CASE tools further support some form of source code generation (see below) that outputs code fragments for the software under design. These tools are in wide spread use in commercial software design. They help in defining and visualising the design of software before the actual implementation.

One of the problems with the CASE approach is the lack of abstraction. The design process is very tightly coupled to a specific software solution and the design of source code for its components. Although code generation from CASE tools can help in supporting language independence, this is not an intrinsic feature of the CASE approach itself. The tight coupling to source code design leaves decisions about the abstract design granularity and the completeness of models almost entirely to the developers. The CASE approach therefore requires a considerable skill level to be effective.

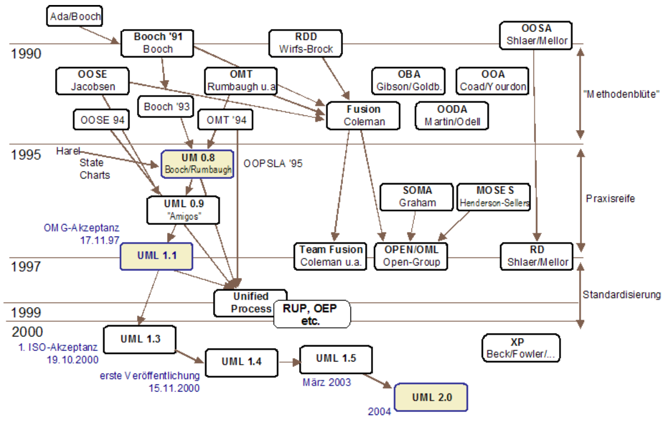


Figure 9.3: The history of UML and other Object-Oriented Methods and Notations (source: [Zoc04], authorised)

9.7.3 The OMG Model Driven Architecture

The *Model Driven Architecture* [MM03] is a software design approach proposed by the Object Management Group (OMG[7]). It is founded on the developments around UML, but explicitly describes a more general approach.

The main idea is to enforce the power of abstract models over platform specific implementations. MDA applications are designed in platform independent models (PIM) that are then transformed (possibly through further intermediate PIMs) into platform specific models (PSM) and finally augmented with platform specific code to provide a complete implementation of a system. Coming from the UML area, the OMG's MDA is mainly targeted at business logic and large-scale enterprise software design, although generally applicable to most software design processes.

This thesis presents a Model Driven Architecture specifically for the design of overlay networks. The deliberately narrowed focus provides the Node Views model and its OverML languages with very rich semantics for the target area. These semantics can parametrise the transformation process, which ultimately leads to more specialised PSMs that are optimised for the specific model and its target platform.

9.7.4 Generative Programming and Domain Specific Languages

As opposed to general purpose modelling languages like UML, domain specific languages aim to provide semantically richer models for a more restricted class of software systems. Sometimes, these languages are executed by interpreters. More commonly, Generative Programming [CE00] translates these language into source code written in general purpose programming languages. This can either result in directly compilable code or in code fragments that must be extended by hand. Often, code is automatically merged with hand-written code or yields abstract classes that are meant to be subclassed in an object-oriented style. Arguably, the most common use cases are object interface descriptions and data binding, where customised implementations for APIs, network protocols and data classes are generated from higher-level descriptions and data schemas.

Examples are CORBA IDL [Pop97, IDL99] or WSDL [W3C01] for object interfaces and Castor[8] or JAXB [$V^{+}06$] for data binding. The latter makes a good example for the main advantages of this approach. Data schemas are rich in that they capture exactly the structure and type of data blocks. They are programming language independent and allow for various mappings to different implementations and environments. Writing data classes by hand, as

[7]`http://www.omg.org/`
[8]`http://www.castor.org/`

well as readers, writers and validators for the data is lengthy, unamusing work, that is best done by generators once the semantics are defined in a compact and semantically rich language.

Another area where domain specific XML languages have recently found broad deployment are web application servers and web services. The most popular examples are arguably deployment descriptor languages in the Java Enterprise Edition framework. The main use case in this context is software component configuration. The Spring framework for Java [WB05] is a prominent and well established project that shows how domain specific configuration languages can help in decoupling software components. The underlying *Inversion of Control* paradigm moves the glue code between components out of the source code and into an external XML language. This approach makes the deployment highly configurable and adaptable to different environments.

In the context of domain specific XML languages, OverML represents a new, integrated set of languages for the domain of overlay implementations. The five languages are based on the Node Views architecture and capture the semantics of node attributes, message data, topology views, routing decisions and event-driven processing. They are independent of programming languages and frameworks and allow for different mappings to concrete implementations.

Especially EDSL is influenced by languages for component configuration. It moves the event-driven interaction out of the components themselves and models them as a domain specific, platform independent event graph.

9.8 Modelling Distributed Systems and Protocols

The long history of research in distributed computing systems yielded a number of modelling techniques. One of the most interesting formal models is the π-calculus [Mil99], a minimal process calculus. Its primitives are concurrency, input/output communication and the creation, replication and termination of processes. Their combination allows to express the behaviour of distributed systems in a way that enables system analysis and reasoning about the equivalence of different systems.

The π-calculus has been augmented with cryptographic primitives (the Spi calculus [AG97]) and other syntactic extensions to support its application to various fields of process modelling. One of its more recent applications is the Business Process Modelling Language (BPML [A$^+$02]).

More commonly, distributed systems are modelled in graphs as finite automata or Petri Nets [Pet63] which are equally powerful. The latter provide very convenient support for concurrency and synchronisation, which are important concepts in the distributed systems and parallel processing communities.

EDSL follows the automaton approach to protocol modelling which better supports its close integration into OverML. Communication protocols are commonly specified and modelled as event-driven finite state-machines, based on the historically well established theory of automata. Even a general purpose modelling language like UML supports state machines as a way to specify system behaviour and component interactions (see section 9.7.1). Countless finite state-machine languages exist for various different special purposes and domains, examples being Ragel[9], FSM-Lang[10] or ASML[11].

Frameworks like SEDA [WCB01] or State Threads[12] show the benefits of the state machine approach in terms of performance, scalability and design aspects for distributed systems. They can be used to build highly concurrent server architectures[13] [PDZ99]. Another advantage of this approach is the near-ubiquitous support of the underlying event-based networking primitives *select* or *poll* on virtually any platform, which makes these frameworks easily portable or re-writable across POSIX-compatible systems [Ste03].

As with SLOSL and SQL, the major advantage of EDSL over other finite state-machine languages is the augmented domain specific semantics and the integration with OverML and the SLOSL events. Due to the common theoretical background, EDSL graphs can be mapped to other state machine languages without major difficulties.

9.9 Databases and Data-Driven Software

Database research has a long tradition in computer science. Data abstractions, query languages, database management systems and multi-tier architectures had major impacts on various other fields, above all software design and distributed systems. Today, the most common database abstraction is based on tuple relations stored in tables, although XML databases are gaining ground.

A number of different query languages were developed for different purposes and different applications. The Structured Query Language for relational database management systems (SQL [SQL03a]) is by far the most widely used query language in database applications today, one of the most widely applied programming languages and a major requirement in job offers for software developers[14]. Other important languages are the logical query language Datalog [CGT89] and the relatively young W3C standard XQuery [XQu06] for querying XML data.

[9]`http://www.elude.ca/ragel/` (Aug. 30, 2006)
[10]`http://fsmlang.sourceforge.net/` (Aug. 30, 2006)
[11]`http://research.microsoft.com/fse/asml/default.aspx` (Aug. 30, 2006)
[12]`http://state-threads.sourceforge.net/` (Oct. 17, 2006)
[13]`http://aap.sourceforge.net/` (Oct. 20, 2006)
[14]`http://www.dedasys.com/articles/language_popularity.html` (Oct. 15, 2006)

9.9.1 Database Views

Views are a concept that became part of the SQL'92 standard [SQL92]. They provide pre-defined queries that look like normal database tables to the user and recursively support sub-views and other queries. Their main advantage is the layered support for abstractions on top of the basic database schema. Another benefit is their definition inside the database which allows pre-optimised evaluation as with stored procedures.

The work presented in this thesis found a new application of databases and views in the area of distributed systems. SLOSL, a new view definition language based on SQL, is used to specify topology rules in a declarative and machine readable way. The extensions of SLOSL over SQL provide more specific semantics for the target domain of topologies and overlay networks. Section 8.2.1 describes a mapping for generating SQL view creation statements from SLOSL statements. In comparison, the original SLOSL statement is much easier to read and provides a more specific description of the topological features.

Data Warehousing is a field in database research that makes extensive use of views. It lead to the invention of *materialised views* that physically store the content of views for faster read access. This is obviously payed with higher update costs. The maintenance of views is therefore a topic that received major interest in the database community [LW95]. Especially in distributed database management systems, view updates must be communicated efficiently to avoid excessive overhead.

Many achievements in this field can be adopted for SLOSL views. In fact, the introduction of views into the area of overlay networks and topologies makes it possible to map distributed view maintenance algorithms directly to overlay maintenance schemes. The distributed harvesters of section 7.4 are one example where efficient view maintenance has a major impact on the overall system performance. Different view maintenance algorithms can be deployed to achieve the required tradeoff between the speed of update dissemination and the communication overhead. The support for SLOSL views in EDSL and HIMDEL allows their platform independent, modular specification in OverML.

9.9.2 Data-Driven Software Architectures

The Node Views architecture is clearly influenced by the *Model-View-Controller* pattern (MVC [Ree79, BMR$^+$96]). It stands for a separation of concern between three main types of system components responsible for processing, presenting and updating data:

- a data model that stores and processes data
- view components that have read-only access to the model and use its data for presentation and decisions (views, routers, visualisers, etc.)

- control components that update data (harvesters and controllers)

The main idea behind this pattern is the instantiation of a central data store that decouples controllers from views. It is well established in GUI frameworks for the Smalltalk programming language[15] or in the Java Swing GUI library[16], but also in web frameworks based on Java Servlets [M+03] and Java Server Pages (JSP [D+06]).

In the Node Views architecture, the model is an *active database*, a concept that was first described in [DBM88]. Active databases provide event-driven callbacks, commonly based on triggers and event-driven rules. Rules are processed on data updates and cause user defined code to be executed. Today, most database management systems have active features, such as triggers and (procedural) trigger languages. MVC frameworks often deploy the Observer or Listener pattern [BMR+96] to provide a similar functionality to view or control components.

The Node Views architecture borrows this idea to support SLOSL view events in EDSL. Being a domain specific language, SLOSL is restricted to the generation of semantically relevant and semantically rich event types (see 5.5). It further extends the MVC idea by introducing platform independent languages for data-driven decisions in views and controllers. The database model, topology rules, routing decisions and maintenance triggers are all specified in OverML, which makes them machine readable and available to MDA translators.

[15]`http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html` (Oct. 15, 2006)

[16]`http://java.sun.com/j2se/1.5.0/docs/api/index.html?javax/swing/package-tree.html` (Oct. 15, 2006)

Chapter 10

Conclusion and Future Work

The growing interest in decentralised overlay infrastructures for Internet services shows a noticeable paradigm shift in distributed systems research. Numerous projects from several research communities and commercial organisations have started building and deploying self-organising, server-free overlay systems. Adoption, however, has been restricted to tightly selected areas, dominated by few applications, such as EDonkey/Overnet and KaZaA for file search, or BitTorrent for the distribution of large files.

Amongst the technical reasons for the limited availability of deployable systems is the complexity of their design and the incompatibility of systems and frameworks. Applications are tightly coupled to a specific overlay implementation and the framework it uses. This leaves developers with two choices: implementing their application based on the fixed combination of overlay, networking framework and programming language, or reimplementing the overlay based on the desired application framework and language.

Implementing an overlay, even as a reimplementation, is a task that exhibits considerable challenges. Protocols have to be adapted, completed and partially reverse engineered from the original system to implement them correctly. Message serialisations and interfaces have to be rewritten for the new framework. State maintenance and event handling are programmed in very different ways in different environments, which typically requires their redesign. Reimplementing an overlay for a new environment is therefore not necessarily less work than writing a new one.

Even if an existing overlay implementation meets the application level requirements on the framework, the examples in chapter 2.2 motivate that distributed applications can well benefit from the diversity of choices between different overlays. The current designs, however, tie the implementation of applications to one specific overlay, straight from the design to test and deployment.

The current techniques used for overlay implementation turn out to become limiting factors in the development cycle of overlay applications. The work presented in this thesis provides distributed system developers with novel techniques for the platform-independent, integrative design of overlay networks.

Contribution of the Thesis

This thesis motivated the search for new ways in overlay software design and presented a novel methodology for the integrative, platform-independent, high-level design of adaptable overlay networks. This approach provides overlay designers with high-level modelling techniques and application developers with portable, integrative overlay implementations.

Quality-of-Service in an Overlay Application

The case study in chapter 2 motivated the need for integrative, portable, adaptable overlay implementations. In section 2.3, it compared a number of well-known overlay systems and system types that were designed for publish-subscribe applications.

To this end, a prior section (2.2) devised meaningful criteria for the comparison of overlay implementations for publish-subscribe applications. It presented the first broad evaluation of quality-of-service measures in the specific context of publish-subscribe applications. The metrics it discussed for the infrastructure level are latency, bandwidth and message priorities. For the level of notifications and subscriptions, the relevant metrics were found to be delivery guarantees, selectivity of subscriptions, periodic/sporadic delivery, order of notifications, validity intervals, source redundancy, confidentiality, authentication and integrity.

The system comparisons showed that there is no clear winner for the publish-subscribe area. None of the current designs can efficiently solve all problems that publish-subscribe applications may face. Different types of overlay systems are optimised for different requirements and are not easily merged into all-win solutions.

This outcome makes a convincing case for combining simple overlay implementations into more complex building blocks for quality-of-service aware real-world applications. Such a bundle of choices allows them to adapt to varying requirements by selecting optimised implementations for their current system state. As chapter 3 explains, however, this requires new approaches for the integrative, platform-independent, adaptable design of overlay networks.

A Visual Modelling Approach for Overlay Networks

The Overlay Modelling Language OverML, presented in chapter 6, is a set of domain specific languages for modelling overlay networks. Where current overlay implementations are hand-written and hand-tuned for specific frameworks, overlay specifications in OverML are platform independent and can be automatically translated into different platform specific implementations. This has a number of advantages for the designers of both overlays and applications.

Platform independent modelling. The result of the modelling phase is a platform independent representation of an overlay implementation. With a single modelling effort, the overlay designer creates an entire set of possible implementations.

Machine readable semantics. The high-level model of an overlay system becomes accessible by tools for model transformation, system validation and system analysis. This enables the development of highly context aware tools for system design and automated model testing.

Visual design. Overlay designers can visually construct their overlay specifications in design tools such as the SLOSL Overlay Workbench (see 8.1). This reduces the perceived complexity of the system and commonly leads to a better understanding and a better overview for the designers. Design tools can exploit the rich semantics of OverML models to provide context-sensitive support to developers.

Increased readability. The readability of graphical overlay specifications is considerably higher than for source code. Specifications become understandable to other developers without external documentation. Where current source code implementations require reverse engineering to understand or even re-implement a system, OverML specifications are easily re-usable in different contexts and environments.

Reduced code complexity. The expressive event models and the subsequent generation of source code reduce the complexity of the remaining external components. Their interfaces become generic and the size of their implementation benefits from their specifically modelled task.

Design time testing. The high modelling level allows frameworks to execute and test partial specifications, as exemplified in section 8.1.3. Where a source code implemented system requires major components of the target implementation to be available, OverML specifications only need a generic software infrastructure for testing, visualising and benchmarking.

OverML forms the intermediate language that connects readable specifications and generated implementations. Its applicability to different kinds of overlay topologies was motivated in section 5.3, routing schemes were exemplified in section 5.4 and possible overlay maintenance patterns were presented in chapter 7. A complete specification-to-deployment walk-through was presented in sections 8.4 and 8.5, including insights into implementation details.

Data-driven Topology Design and Adaptation

The SQL-Like Overlay Specification Language SLOSL, presented in chapter 5, is a data-driven language for the specification of rules and adaptation strategies in the design of overlay topologies. Its short and readable expressions implement the local decisions that determine the main characteristics of an overlay network. This makes the implementation of complex routing strategies trivial and easy to understand.

The data-driven approach encapsulates the local state of a node and enables generic interfaces to state keeping and local decisions. It avoids additional state keeping in hand-written components and increases the chance for making them reusable in different overlay designs.

Section 5.3 shows a number of examples how different topologies and adaptation strategies can be implemented in SLOSL. Chapter 7 then describes a number of generic update schemes for the local node database.

A Model Driven Architecture for Overlay Networks

OverML and the Node Views architecture represent a Model Driven Architecture for the visual design and automated implementation of overlay networks. They introduce a major paradigm shift from low-level, hand-written, framework specific implementations towards visual, platform-independent models, rapid implementation, adaptation and deployment based on source code generation, and automated optimisation strategies based on query optimisation.

It is a side effect of modelling approaches to encourage the delay of otherwise premature optimisations to later design phases. In Model Driven Architectures, much of this can be left to optimising translators.

The high readability of the visual models and visual tools like the SLOSL Overlay Workbench will enable designers to deal with a considerable increase in complexity of their systems. This will allow very interesting new overlay designs in the future.

Future Work

The Node Views Model Driven Architecture and the OverML languages open up an interesting field of future work. Future framework implementations can be built directly around this architecture and thus harness the new level of freedom in overlay design. The decreased complexity of overlay implementations and the now possible reuse of components allows developers and researchers to focus on new, more complex challenges in the overlay design area.

Inter-Overlay Optimisation

The Node Views architecture integrates overlay implementations at the data level. This merges the state of different running systems and thus reduces the maintenance overhead involved.

Furthermore, OverML makes various aspects of the deployed algorithms and protocols machine readable. This opens up the path towards more sophisticated inter-overlay optimisation strategies to reduce the redundancy when different overlay specifications are deployed. Future work should further investigate this field to come up with generic strategies. Starting points are query optimisation techniques from the database area as well as code optimisation strategies from the compiler area.

Model Analysis

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.

The Node Views architecture proposed in this thesis is the first to move these rules into machine readable overlay models. This makes them available for automated model transformation and analysis. Future work should examine the semantics of OverML and SLOSL under system analysis aspects and try to infer global properties from the models.

An important aspect in this context is topology validation. Given an OverML specification of a topology, validation tools could check the correctness of topology and routing algorithm, possibly considering different node distributions.

Framework Toolsets and Harvester Patterns

At the current state, it is hard to predict which design patterns will emerge from the long-term utilisation of SLOSL and OverML (or similar languages) in real-world overlay design. Only the wide-spread use of these languages and of frameworks that support them will allow us to extract generic components and patterns from the resulting overlay designs. It is therefore a long-term task to fold the emerging generic design helpers back into models and framework toolsets.

Chapter 7 outlined a number of interesting harvester types. As described, OverML supports their specification in terms of node data schemas, views and protocols. However, it would be interesting to augment OverML with more far-reaching models that capture the specific semantics of these harvester design patterns. These patterns would become part of the platform-independent model to apply more easily to different overlay implementations. Such an approach would further reduce the overall design effort and could lead to a number of interesting higher-level optimisations for the platform specific implementations.

Portable Controllers and Harvesters

OverML currently assumes harvesters and controllers to be part of platform specific frameworks or user-provided code. In many cases, however, their algorithms could be expressed in a platform independent language, which would allow a mapping to different platform specific implementations. Future work could investigate different controller types and try to come up with extensions to EDSL that move a larger part of the currently platform specific components to the modelling level. Some of the ideas behind Overlog [LCH+05] and Macedon [RKB+04] may provide starting points.

A possible alternative would be a common scripting language similar to the procedural trigger languages found in current database implementations. However, this approach would apply at the framework level rather than the modelling level. It would require this language to be available in all frameworks and thus reduce the platform-independence of the specifications. As an intermediate solution, a scripting language can be used to prototype the controllers that could then be re-implemented for specific frameworks.

Execution Environments

A platform-independent model, as provided by OverML, can naturally be implemented in different ways to target different environments. In the deployment phase, the dynamic aspects of an application can be restricted to the envisioned scenarios, which allows environments to trade generality against speed. In the debugging and testing phase, it is most important to provide substantial insights into the running system. Simulators and debuggers can therefore take a more analytical approach, visualise different parts of the system and provide statistics about various decisions taken during the execution of SLOSL and EDSL. Future work in these areas should investigate different types of environments and find well-adapted framework implementations for their specific requirements.

A particularly interesting idea in the simulation context was presented in [BHMS04]. The proposed environment splits the entire simulation into two steps: an application level simulation of the overlay deployment itself and a network level simulation of the resulting message traffic. This separation allows for a higher scalability of each step. The Node Views approach provides a considerably high and semantically rich level of abstraction for overlay designs that might be very well suited for application level simulations. Future work should look at ways to efficiently deploy OverML specifications in application level simulators and to map the resulting logs to network level simulators.

Mobile and Ad-Hoc Environments

OverML is a set of domain specific languages designed for the requirements and capabilities of Internet overlay networks. However, the application of similar ideas to other networking environments, especially mobile ad-hoc networks, is a very interesting field of future work. The clean separation of routing and maintenance in the Node Views architecture should keep the necessary modifications small.

The main differences between overlays for the Internet and mobile networks lie within the efficient broadcast capabilities of mobile installations and the limited reachability of remote nodes in wireless multi-hop networks. The former is of low importance for the modelling techniques themselves. The latter requires a representation in the node data model. Therefore, modifications for mobile environments can be expected at the level of frameworks and specifications rather than the modelling language or the general architecture. Future work should investigate the application of OverML to mobile environments and possible extensions to the language to support the specific requirements of mobile applications.

Appendix A

XML Reference Implementations

Contents

A.1 The RelaxNG Schema of **OverML**

The following is the formal RelaxNG [CM01] schema specification for the OverML XML languages.

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar
   xmlns:doc="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/annotate"
   xmlns:edsl="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/edsl"
   xmlns:edgar="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/edgar"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema-datatypes"
   xmlns:himdel="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/himdel"
   xmlns:slosl="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/slosl"
   xmlns:slowgui="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/slow-gui"
   xmlns:slow="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/slow"
   xmlns:nala="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/nala"
   xmlns:overml="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML"
   xmlns:math="http://www.w3.org/1998/Math/MathML"
   xmlns="http://relaxng.org/ns/structure/1.0"
   datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <define name="identifier_string">
    <data type="Name"/>
  </define>
  <define name="type_string">
    <data type="QName"/>
  </define>
  <define name="identifier_attribute">
    <attribute name="access_name">
      <ref name="identifier_string"/>
    </attribute>
  </define>
  <define name="name_attribute">
    <attribute name="name">
      <ref name="identifier_string"/>
    </attribute>
  </define>
  <define name="type_attribute">
    <attribute name="type_name">
      <ref name="type_string"/>
    </attribute>
  </define>
  <define name="readable_name_attribute">
    <attribute name="readable_name"/>
  </define>
  <define name="anyOtherElement">
    <element>
      <anyName>
        <except>
          <nsName
            ns="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/slow"/>
          <nsName
            ns="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/nala"/>
          <nsName
            ns="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/slosl"/>
          <nsName
            ns="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/himdel"/>
          <nsName
            ns="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/edsl"/>
        </except>
      </anyName>
      <zeroOrMore>
        <choice>
          <attribute>
            <anyName/>
          </attribute>
          <text/>
          <ref name="anyOtherElement"/>
        </choice>
      </zeroOrMore>
    </element>
  </define>
  <define name="math_expression">
    <choice>
      <element name="math:math">
        <ref name="math_expression_content"/>
      </element>
      <ref name="math_expression_content"/>
    </choice>
  </define>
  <define name="math_expression_content">
    <choice>
      <ref name="math_apply"/>
```

```
        <ref name="math_number"/>
        <ref name="math_constant"/>
      </choice>
    </define>
    <define name="math_interval">
      <choice>
        <element name="math:math">
          <ref name="math_interval_content"/>
        </element>
        <ref name="math_interval_content"/>
      </choice>
    </define>
    <define name="math_interval_content">
      <element name="math:interval">
        <attribute name="closure">
          <choice>
            <value>open</value>
            <value>closed</value>
            <value>open-closed</value>
            <value>closed-open</value>
          </choice>
        </attribute>
        <ref name="math_expression_content"/>
        <ref name="math_expression_content"/>
      </element>
    </define>
    <define name="math_list">
      <choice>
        <element name="math:math">
          <ref name="math_list_content"/>
        </element>
        <ref name="math_list_content"/>
      </choice>
    </define>
    <define name="math_list_content">
      <element name="math:list">
        <zeroOrMore>
          <attribute>
            <anyName/>
          </attribute>
        </zeroOrMore>
        <zeroOrMore>
          <ref name="math_expression_content"/>
        </zeroOrMore>
      </element>
    </define>
    <define name="math_apply">
      <element name="math:apply">
        <ref name="anyMathML"/>
        <oneOrMore>
          <ref name="anyMathML"/>
        </oneOrMore>
      </element>
    </define>
    <define name="math_number">
      <element name="math:cn">
        <zeroOrMore>
          <attribute>
            <anyName/>
          </attribute>
        </zeroOrMore>
        <text/>
      </element>
    </define>
    <define name="math_constant">
      <element name="math:ci">
        <zeroOrMore>
          <attribute>
            <anyName/>
          </attribute>
        </zeroOrMore>
        <text/>
      </element>
    </define>
    <define name="anyMathML">
      <choice>
        <element>
          <nsName ns="http://www.w3.org/1998/Math/MathML"/>
          <empty/>
        </element>
        <ref name="math_list"/>
        <ref name="math_interval"/>
        <ref name="math_apply"/>
        <ref name="math_number"/>
        <ref name="math_constant"/>
      </choice>
    </define>
    <start>
      <doc:documentation xml:lang="en">OverML Schema 0.2 - SUBJECT TO
         CHANGES!</doc:documentation>
      <ref name="slow_file"/>
    </start>
    <define name="slow_file">
      <element name="slow:file">
        <interleave>
          <optional>
            <ref name="nala_types"/>
          </optional>
          <optional>
            <choice>
              <zeroOrMore>
                <ref name="nala_attribute"/>
              </zeroOrMore>
              <ref name="nala_attributes"/>
            </choice>
          </optional>
          <optional>
            <choice>
              <zeroOrMore>
                <ref name="slosl_statement"/>
              </zeroOrMore>
              <ref name="slosl_statements"/>
            </choice>
          </optional>
          <optional>
            <ref name="edgar_statements"/>
          </optional>
          <optional>
            <ref name="message_hierarchy"/>
          </optional>
          <optional>
            <ref name="edsl_graph"/>
          </optional>
          <optional>
            <ref name="slow_gui"/>
          </optional>
        </interleave>
      </element>
    </define>
    <define name="slow_gui">
      <element name="slowgui:gui">
        <zeroOrMore>
          <ref name="anyGuiElement"/>
        </zeroOrMore>
      </element>
    </define>
    <define name="anyGuiElement">
      <element>
        <nsName
           ns="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/slow-gui"/>
        <zeroOrMore>
          <choice>
            <attribute>
              <anyName/>
            </attribute>
            <text/>
            <ref name="anyGuiElement"/>
          </choice>
        </zeroOrMore>
      </element>
    </define>
    <define name="nala_types">
      <element name="nala:types">
        <zeroOrMore>
          <ref name="anyOtherElement"/>
        </zeroOrMore>
      </element>
    </define>
    <define name="nala_attributes">
      <element name="nala:attributes">
        <zeroOrMore>
          <ref name="nala_attribute"/>
        </zeroOrMore>
      </element>
    </define>
    <define name="nala_attribute">
      <zeroOrMore>
        <element name="nala:attribute">
          <attribute name="name">
            <ref name="identifier_string"/>
          </attribute>
          <attribute name="type_name">
            <ref name="type_string"/>
          </attribute>
          <optional>
            <attribute name="selected">
              <data type="boolean"/>
            </attribute>
          </optional>
          <interleave>
            <optional>
              <element name="nala:static">
                <empty/>
              </element>
            </optional>
            <optional>
              <element name="nala:transferable">
                <empty/>
              </element>
            </optional>
            <optional>
              <element name="nala:identifier">
                <empty/>
              </element>
            </optional>
            <optional>
              <ref name="nala_depends"/>
            </optional>
          </interleave>
        </element>
      </zeroOrMore>
    </define>
    <define name="nala_depends">
      <element name="nala:depends">
        <choice>
          <interleave>
            <attribute name="type">
              <value>math</value>
            </attribute>
            <ref name="nala_attribute_refs"/>
            <ref name="math_expression"/>
          </interleave>
          <interleave>
            <attribute name="type">
              <value>external</value>
            </attribute>
            <ref name="nala_attribute_refs"/>
            <optional>
              <element name="nala:call">
                <ref name="identifier_string"/>
              </element>
            </optional>
          </interleave>
        </choice>
      </element>
    </define>
    <define name="nala_attribute_refs">
      <ref name="nala_attribute_ref"/>
      <zeroOrMore>
        <ref name="nala_attribute_ref"/>
      </zeroOrMore>
    </define>
    <define name="nala_attribute_ref">
      <element name="attribute-ref">
        <attribute name="name">
          <ref name="identifier_string"/>
        </attribute>
      </element>
    </define>
    <define name="message_hierarchy">
      <element name="himdel:message_hierarchy">
        <interleave>
          <zeroOrMore>
            <ref name="toplevel_container"/>
          </zeroOrMore>
          <zeroOrMore>
            <ref name="header"/>
          </zeroOrMore>
          <zeroOrMore>
            <ref name="protocol"/>
          </zeroOrMore>
        </interleave>
      </element>
    </define>
    <define name="toplevel_container">
    <element name="himdel:container">
      <optional>
        <ref name="readable_name_attribute"/>
      </optional>
      <ref name="type_attribute"/>
      <zeroOrMore>
        <ref name="content_field"/>
      </zeroOrMore>
    </element>
  </define>
  <define name="container">
    <element name="himdel:container">
      <optional>
        <ref name="readable_name_attribute"/>
      </optional>
      <optional>
        <ref name="type_attribute"/>
      </optional>
      <zeroOrMore>
        <ref name="content_field"/>
      </zeroOrMore>
    </element>
  </define>
  <define name="header">
    <element name="himdel:header">
      <optional>
        <ref name="readable_name_attribute"/>
      </optional>
      <optional>
        <ref name="identifier_attribute"/>
      </optional>
      <interleave>
        <zeroOrMore>
          <ref name="header"/>
        </zeroOrMore>
        <zeroOrMore>
          <ref name="content_field"/>
        </zeroOrMore>
        <oneOrMore>
          <ref name="message"/>
        </oneOrMore>
      </interleave>
    </element>
  </define>
  <define name="message">
    <element name="himdel:message">
      <optional>
        <ref name="readable_name_attribute"/>
      </optional>
      <ref name="type_attribute"/>
      <zeroOrMore>
        <ref name="content_field"/>
      </zeroOrMore>
    </element>
  </define>
  <define name="protocol">
    <element name="himdel:protocol">
      <ref name="content_attributes"/>
      <zeroOrMore>
        <element name="himdel:message-ref">
          <ref name="type_attribute"/>
        </element>
      </zeroOrMore>
    </element>
  </define>
  <define name="content_field">
    <choice>
      <element name="himdel:attribute">
        <ref name="content_attributes"/>
      </element>
      <element name="himdel:content">
        <ref name="content_attributes"/>
      </element>
      <ref name="container"/>
      <element name="himdel:container-ref">
        <ref name="content_attributes"/>
      </element>
      <element name="himdel:viewdata">
        <attribute name="structured">
          <data type="boolean"/>
        </attribute>
        <ref name="content_attributes"/>
      </element>
    </choice>
  </define>
  <define name="content_attributes">
    <ref name="identifier_attribute"/>
    <ref name="type_attribute"/>
    <optional>
      <ref name="readable_name_attribute"/>
    </optional>
  </define>
  <define name="slosl_statements">
    <element name="slosl:statements">
      <zeroOrMore>
        <ref name="slosl_statement"/>
      </zeroOrMore>
    </element>
  </define>
  <define name="slosl_statement">
    <element name="slosl:statement">
      <ref name="name_attribute"/>
      <optional>
        <attribute name="selected">
          <data type="boolean"/>
        </attribute>
      </optional>
      <interleave>
        <oneOrMore>
          <ref name="slosl_select"/>
        </oneOrMore>
        <optional>
          <ref name="slosl_ranked"/>
        </optional>
        <oneOrMore>
          <ref name="slosl_parent"/>
        </oneOrMore>
        <zeroOrMore>
          <ref name="slosl_with"/>
        </zeroOrMore>
        <optional>
          <ref name="slosl_where"/>
        </optional>
        <optional>
          <choice>
            <ref name="slosl_inherit_buckets"/>
            <interleave>
              <optional>
                <ref name="slosl_having"/>
              </optional>
              <ref name="slosl_buckets"/>
            </interleave>
          </choice>
        </optional>
      </interleave>
    </element>
  </define>
  <define name="slosl_select">
    <element name="slosl:select">
      <ref name="name_attribute"/>
      <ref name="type_attribute"/>
      <optional>
        <choice>
          <text/>
          <ref name="math_expression"/>
        </choice>
      </optional>
    </element>
  </define>
  <define name="slosl_parent">
    <element name="slosl:parent">
      <ref name="identifier_string"/>
    </element>
  </define>
  <define name="slosl_ranked">
    <element name="slosl:ranked">
      <ref name="f_parameter"/>
      <ref name="f_parameter"/>
      <choice>
        <attribute name="function">
          <choice>
            <value>lowest</value>
            <value>highest</value>
          </choice>
        </attribute>
        <group>
          <attribute name="function">
            <choice>
              <value>closest</value>
              <value>furthest</value>
          </choice>
        </attribute>
        <ref name="f_parameter"/>
      </group>
    </choice>
  </element>
</define>
<define name="f_parameter">
  <element name="slosl:parameter">
    <ref name="math_expression"/>
  </element>
</define>
<define name="slosl_with">
  <element name="slosl:with">
    <ref name="name_attribute"/>
    <optional>
      <ref name="math_expression"/>
    </optional>
  </element>
</define>
<define name="slosl_where">
  <element name="slosl:where">
    <ref name="math_expression"/>
  </element>
</define>
<define name="slosl_having">
  <element name="slosl:having">
    <ref name="math_expression"/>
  </element>
</define>
<define name="slosl_inherit_buckets">
  <element name="slosl:buckets">
    <attribute name="inherit">
      <value>true</value>
    </attribute>
  </element>
</define>
<define name="slosl_buckets">
  <element name="slosl:buckets">
    <optional>
      <attribute name="inherit">
        <value>false</value>
      </attribute>
    </optional>
    <oneOrMore>
      <element name="slosl:foreach">
        <ref name="name_attribute"/>
        <choice>
          <ref name="math_list"/>
          <ref name="math_interval"/>
        </choice>
      </element>
    </oneOrMore>
  </element>
</define>
<define name="edgar_statements">
  <element name="edgar:routers">
    <zeroOrMore>
      <ref name="edgar_start"/>
    </zeroOrMore>
  </element>
</define>
<define name="edgar_start">
  <element name="edgar:router">
    <ref name="name_attribute"/>
    <optional>
      <attribute name="selected">
        <data type="boolean"/>
      </attribute>
    </optional>
    <ref name="edgar_content"/>
  </element>
</define>
<define name="edgar_content">
  <choice>
    <ref name="edgar_firstmatch"/>
    <ref name="edgar_fork"/>
    <ref name="edgar_tag"/>
    <ref name="edgar_predicate"/>
    <ref name="edgar_exit"/>
  </choice>
</define>
<define name="edgar_firstmatch">
  <element name="edgar:firstmatch">
    <oneOrMore>
```

```
          <ref name="edgar_content"/>
        </oneOrMore>
      </element>
    </define>
    <define name="edgar_fork">
      <element name="edgar:fork">
        <oneOrMore>
          <ref name="edgar_content"/>
        </oneOrMore>
      </element>
    </define>
    <define name="edgar_tag">
      <element name="edgar:tag">
        <attribute name="type">
          <ref name="type_string"/>
        </attribute>
        <ref name="edgar_content"/>
      </element>
    </define>
    <define name="edgar_predicate">
      <element name="edgar:predicate">
        <ref name="name_attribute"/>
        <ref name="edgar_content"/>
      </element>
    </define>
    <define name="edgar_exit">
      <element name="edgar:exit">
        <attribute name="target">
          <ref name="type_string"/>
        </attribute>
      </element>
    </define>
    <define name="edsl_graph">
      <element name="edsl:edsm">
        <interleave>
          <ref name="edsl_states"/>
          <ref name="edsl_transitions"/>
        </interleave>
      </element>
    </define>
    <define name="edsl_states">
      <element name="edsl:states">
        <interleave>
          <zeroOrMore>
            <ref name="edsl_state"/>
          </zeroOrMore>
          <zeroOrMore>
            <ref name="edsl_subgraph"/>
          </zeroOrMore>
        </interleave>
      </element>
    </define>
    <define name="edsl_transitions">
      <element name="edsl:transitions">
        <zeroOrMore>
          <ref name="edsl_transition"/>
        </zeroOrMore>
      </element>
    </define>
    <define name="edsl_subgraph">
      <element name="edsl:subgraph">
        <interleave>
          <ref name="name_attribute"/>
          <attribute name="id">
            <ref name="edsl_state_id"/>
          </attribute>
          <attribute name="entry_state">
            <ref name="edsl_state_id"/>
          </attribute>
          <attribute name="exit_state">
            <ref name="edsl_state_id"/>
          </attribute>
          <optional>
            <ref name="edsl_readable_name"/>
          </optional>
          <ref name="edsl_states"/>
          <ref name="edsl_transitions"/>
        </interleave>
      </element>
    </define>
    <define name="edsl_state">
      <element name="edsl:state">
        <interleave>
          <ref name="name_attribute"/>
          <attribute name="id">
          <ref name="edsl_state_id"/>
        </attribute>
        <attribute name="inherit_context">
          <data type="boolean"/>
        </attribute>
        <attribute name="long_running">
          <data type="boolean"/>
        </attribute>
        <optional>
          <ref name="edsl_readable_name"/>
        </optional>
        <zeroOrMore>
          <ref name="edsl_code"/>
        </zeroOrMore>
        <zeroOrMore>
          <element name="input">
            <ref name="edsl_queue_name"/>
          </element>
        </zeroOrMore>
        <zeroOrMore>
          <element name="output">
            <ref name="edsl_queue_name"/>
          </element>
        </zeroOrMore>
      </interleave>
    </element>
  </define>
  <define name="edsl_transition">
    <element name="edsl:transition">
      <interleave>
        <optional>
          <ref name="edsl_readable_name"/>
        </optional>
        <zeroOrMore>
          <ref name="edsl_code"/>
        </zeroOrMore>
        <choice>
          <group>
            <attribute name="type">
              <value>message</value>
            </attribute>
            <ref name="edsl_message_transition"/>
          </group>
          <group>
            <attribute name="type">
              <value>event</value>
            </attribute>
            <ref name="edsl_event_transition"/>
          </group>
          <group>
            <attribute name="type">
              <value>outputchain</value>
            </attribute>
            <ref name="edsl_outputchain_transition"/>
          </group>
          <group>
            <attribute name="type">
              <value>transition</value>
            </attribute>
            <ref name="edsl_immediate_transition"/>
          </group>
          <group>
            <attribute name="type">
              <value>timer</value>
            </attribute>
            <ref name="edsl_timer_transition"/>
          </group>
        </choice>
      </interleave>
    </element>
  </define>
  <define name="edsl_message_transition">
    <interleave>
      <optional>
        <element name="edsl:messagetype">
          <text/>
        </element>
      </optional>
      <ref name="edsl_from_state"/>
      <ref name="edsl_to_queue"/>
    </interleave>
  </define>
  <define name="edsl_event_transition">
    <interleave>
      <optional>
```

```
          <element name="edsl:subscription">
            <text/>
          </element>
        </optional>
        <ref name="edsl_from_state"/>
        <ref name="edsl_to_queue"/>
      </interleave>
    </define>
    <define name="edsl_timer_transition">
      <interleave>
        <element name="edsl:timerdelay">
          <data type="integer"/>
        </element>
        <ref name="edsl_from_state"/>
        <ref name="edsl_to_queue"/>
      </interleave>
    </define>
    <define name="edsl_outputchain_transition">
      <interleave>
        <ref name="edsl_from_queue"/>
        <ref name="edsl_to_queue"/>
      </interleave>
    </define>
    <define name="edsl_immediate_transition">
      <interleave>
        <ref name="edsl_from_state"/>
        <ref name="edsl_to_state"/>
      </interleave>
    </define>
    <define name="edsl_code">
      <element name="edsl:code">
        <optional>
          <attribute name="classname">
            <ref name="identifier_string"/>
          </attribute>
        </optional>
        <optional>
          <attribute name="methodname">
            <ref name="identifier_string"/>
          </attribute>
        </optional>
        <optional>
          <attribute name="language"/>
        </optional>
        <text>
          <doc:documentation xml:lang="en">the text is
             base64-encoded(zlib-compressed(code))</doc:documentation>
        </text>
      </element>
    </define>
    <define name="edsl_from_state">
      <element name="edsl:from_state">
        <ref name="edsl_state_ref"/>
      </element>
    </define>
    <define name="edsl_to_state">
      <element name="edsl:to_state">
        <ref name="edsl_state_ref"/>
      </element>
    </define>
    <define name="edsl_from_queue">
      <element name="edsl:from_state">
        <ref name="edsl_state_queue_ref"/>
      </element>
    </define>
    <define name="edsl_to_queue">
      <element name="edsl:to_state">
        <ref name="edsl_state_queue_ref"/>
      </element>
    </define>
    <define name="edsl_queue_name">
      <ref name="identifier_string"/>
    </define>
    <define name="edsl_state_ref">
      <attribute name="ref">
        <ref name="edsl_state_id"/>
      </attribute>
    </define>
    <define name="edsl_state_queue_ref">
      <ref name="edsl_state_ref"/>
      <optional>
        <attribute name="queue">
          <ref name="edsl_queue_name"/>
        </attribute>
      </optional>
```

```
  </define>
  <define name="edsl_state_id">
    <text/>
  </define>
  <define name="edsl_readable_name">
    <element name="edsl:readablename">
      <text/>
    </element>
  </define>
</grammar>
```

A.2 XSL Transformation from EDSL to the DOT Language

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:edsl="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/edsl"
    xmlns:l="local"
    >
  <xsl:output method="text" encoding="UTF-8" />

  <xsl:param name="graph_name">edsm_graph</xsl:param>

  <l:colours>
    <l:colour type="transition" >#000000</l:colour><!-- black   -->
    <l:colour type="message"    >#0000ff</l:colour><!-- blue    -->
    <l:colour type="event"      >#ff0000</l:colour><!-- red     -->
    <l:colour type="outputchain">#00ff00</l:colour><!-- green   -->
    <l:colour type="timer"      >#ff00ff</l:colour><!-- magenta -->
  </l:colours>

  <xsl:variable name="colours" select="document('')/*/l:colours">

  <xsl:template match="edsl:edsm">
    <xsl:text>digraph </xsl:text>
    <xsl:value-of select="$graph_name"/>
    <xsl:text> {</xsl:text>
    <xsl:call-template name="cr"/>

    <xsl:apply-templates
        select="./edsl:states/edsl:state[@name = 'start']"
        mode="start-end-state"/>
    <xsl:apply-templates
        select="./edsl:states/edsl:state[@name != 'start']"/>
    <xsl:apply-templates
        select="./edsl:states/edsl:subgraph"/>
    <xsl:apply-templates
        select="./edsl:transitions/edsl:transition"/>

    <xsl:call-template name="cr"/>
    <xsl:text>}</xsl:text>
    <xsl:call-template name="cr"/>
  </xsl:template>

  <xsl:template match="edsl:subgraph">
    <xsl:variable name="entry" select="string(@entry_state)"/>
    <xsl:variable name="exit"  select="string(@exit_state)"/>

    <xsl:text>subgraph cluster_</xsl:text>
    <xsl:value-of select="@id"/>
    <xsl:text> {</xsl:text>
    <xsl:call-template name="cr"/>
    <xsl:apply-templates select="." mode="label"/>
    <xsl:call-template name="cr"/>

    <xsl:apply-templates
        select="./edsl:states/edsl:state[@id = $entry or @id = $exit]"
        mode="start-end-state"/>
    <xsl:apply-templates
        select="./edsl:states/edsl:state[@id != $entry and @id != $exit]"/>
    <xsl:apply-templates
        select="./edsl:transitions/edsl:transition"/>

    <xsl:call-template name="cr"/>
    <xsl:text>}</xsl:text>
    <xsl:call-template name="cr"/>
  </xsl:template>

  <xsl:template match="edsl:state" mode="start-end-state">
    <xsl:text>"</xsl:text><xsl:value-of select="@id"/><xsl:text>"</xsl:text>
    <xsl:text> [ shape="box"</xsl:text>
    <xsl:apply-templates select="." mode="state-attributes"/>
    <xsl:text>]</xsl:text>
    <xsl:call-template name="cr"/>
  </xsl:template>

  <xsl:template match="edsl:state">
    <xsl:text>"</xsl:text><xsl:value-of select="@id"/><xsl:text>"</xsl:text>
    <xsl:text> [ shape="circle"</xsl:text>
    <xsl:apply-templates select="." mode="state-attributes"/>
    <xsl:text> ]</xsl:text>
    <xsl:call-template name="cr"/>
  </xsl:template>

  <xsl:template match="edsl:state" mode="state-attributes">
    <xsl:text>, </xsl:text>
    <xsl:apply-templates select="." mode="label"/>
  </xsl:template>

  <xsl:template match="edsl:transition">
    <xsl:variable name="type" select="@type"/>

    <xsl:text>"</xsl:text>
    <xsl:value-of select="string(./from_state/@ref)"/>
    <xsl:text>" -> "</xsl:text>
    <xsl:value-of select="string(./to_state/@ref)"/>
    <xsl:text>" [ color="</xsl:text>
    <xsl:value-of select="string($colours/l:colour[@type = $type])"/>
    <xsl:text>"</xsl:text>

    <xsl:if test="normalize-space(./readablename)">
      <xsl:text>, label="</xsl:text>
      <xsl:value-of select="normalize-space(./readablename)"/>
      <xsl:text>"</xsl:text>
    </xsl:if>

    <xsl:text> ]</xsl:text>
    <xsl:call-template name="cr"/>
  </xsl:template>

  <xsl:template match="edsl:*" mode="label">
    <xsl:text>label="</xsl:text>
    <xsl:choose>
      <xsl:when test="normalize-space(./readablename)">
        <xsl:value-of select="normalize-space(./readablename)"/>
      </xsl:when>
      <xsl:otherwise><xsl:value-of select="normalize-space(@name)"/></
        xsl:otherwise>
    </xsl:choose>
    <xsl:text>"</xsl:text>
  </xsl:template>

  <xsl:template name="cr">
    <xsl:text>&#10;</xsl:text>
  </xsl:template>
</xsl:stylesheet>
```

A.3 XSL Transformation from EDSL to a flat EDSM Graph

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:edsl="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/edsl">

  <xsl:import href="common.xsl"/>

  <xsl:output method="xml" encoding="UTF-8" indent="no" />

  <xsl:template match="edsl:edsm">
    <xsl:copy>
      <edsl:states>
        <xsl:apply-templates select="./edsl:states/edsl:state" mode="edsm"/>
        <xsl:apply-templates select=".//edsl:subgraph" mode="edsm"/>
      </edsl:states>
      <edsl:transitions>
        <xsl:apply-templates select=".//edsl:transition" mode="edsm"/>
      </edsl:transitions>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="edsl:subgraph" mode="edsm">
    <xsl:variable name="id" select="@id"/>
    <xsl:for-each select="edsl:states/edsl:state">
      <xsl:copy>
        <xsl:attribute name="name">
          <xsl:value-of select="concat($id, '_', @name)"/>
        </xsl:attribute>
        <xsl:apply-templates select="@*[local-name() != 'name']" mode="copyattr"/>
        <xsl:apply-templates select="edsl:*" mode="edsm"/>
      </xsl:copy>
    </xsl:for-each>
    <xsl:apply-templates select="edsl:states/edsl:subgraph" mode="edsm"/>
  </xsl:template>

  <!-- strip empty elements -->
  <xsl:template match="edsl:*[not (@* or * or normalize-space())]" mode="edsm"/>

  <!-- copy everything else -->
  <xsl:template match="edsl:*" mode="edsm">
    <xsl:copy>
      <xsl:apply-templates select="@*" mode="copyattr"/>
      <xsl:apply-templates select="edsl:*|text()" mode="edsm"/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="*">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates select="*"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

A.4 XSL Transformation from EDSL to a merged State Graph

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:edsl="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/edsl">

  <xsl:output method="xml" encoding="UTF-8" indent="no" />

  <xsl:template match="edsl:edsm">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates select="edsl:states" mode="merge_edsl"/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="edsl:states" mode="merge_edsl">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates select="edsl:state" mode="merge_edsl"/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="edsl:state" mode="merge_edsl">
    <xsl:variable name="state" select="@id"/>
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:copy-of select="*"/>
      <xsl:copy-of
        select="../../edsl:transitions/edsl:transition[ edsl:from_state/@ref = $
          state ]"/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="*">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates select="*"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

A.5 XSL Transformation from Slosl to View Objects

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:code="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/codegen
      "
    xmlns:slosl="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/slosl"
    xmlns:nala="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/nala"
    xmlns:math="http://www.w3.org/1998/Math/MathML"
    xmlns:exslt="http://exslt.org/common"
    xmlns:l="local"
    exclude-result-prefixes="l exslt"
    >

  <xsl:import href="nala2objects.xsl"/>
  <xsl:import href="names.xsl"/>      <!-- name handling: capitalization, etc. -->

  <xsl:key name="attributes" match="//nala:attribute" use="@name"/>

  <xsl:template match="/">
    <code:classes>
      <xsl:apply-templates select="*/slosl:statements/slosl:statement"/>
    </code:classes>
  </xsl:template>

  <xsl:template match="slosl:statement[@selected != 'true']"/>
  <xsl:template match="slosl:statement">
    <!-- create classes for a view statement -->
    <xsl:variable name="classname">
      <xsl:call-template name="capitalize">
        <xsl:with-param name="name" select="@name"/>
      </xsl:call-template>
    </xsl:variable>

    <xsl:variable name="deps">
      <xsl:apply-templates mode="attribute-depends"/>
    </xsl:variable>

    <code:viewclass access="public" name="{concat($classname, 'View')}" extends="
       AbstractView">
      <xsl:if test="slosl:buckets[@inherit = 'true']">
        <xsl:attribute name="inherit_buckets">true</xsl:attribute>
      </xsl:if>

      <code:constructor>
        <code:param name="views"/>
        <xsl:for-each select="slosl:parent">
          <code:parent><xsl:value-of select="string()"/></code:parent>
        </xsl:for-each>
        <xsl:for-each select="slosl:with[math:*]">
          <code:assign field="{@name}">
            <xsl:apply-templates select="math:*" mode="copy-expression"/>
          </code:assign>
        </xsl:for-each>
      </code:constructor>

      <xsl:for-each select="slosl:with">
        <xsl:variable name="option_name">
          <xsl:call-template name="capitalize">
            <xsl:with-param name="name" select="@name"/>
          </xsl:call-template>
        </xsl:variable>

        <code:viewoption name="{@name}">
          <xsl:apply-templates select="math:*" mode="copy-expression"/>
        </code:viewoption>
      </xsl:for-each>

      <code:viewnodeclass access="private" name="ViewNode" extends="AbstractNode">
        <code:constructor>
          <code:param name="node"/>
          <xsl:for-each select="slosl:buckets/slosl:foreach/@name">
            <xsl:sort select="string()"/>
            <code:param name="{string()}"/>
          </xsl:for-each>
          <xsl:apply-templates select="slosl:select" mode="classgen-constructor"/>
        </code:constructor>
        <xsl:apply-templates select="slosl:select" mode="classgen"/>
      </code:viewnodeclass>
```

```
        <code:node_selection>
          <xsl:apply-templates select="slosl:where[math:*]"
            mode="copy-expression"/>
          <xsl:apply-templates
            select="slosl:buckets[@inherit != 'true' and slosl:foreach]"
            mode="copy-expression"/>
          <xsl:apply-templates select="slosl:having[math:*]"
            mode="copy-expression"/>
          <xsl:apply-templates select="slosl:ranked[string(@function) and
             slosl:parameter]"
            mode="copy-expression"/>
        </code:node_selection>
      </code:viewclass>
    </xsl:template>

    <xsl:template match="slosl:select" mode="classgen-constructor">
      <code:assign field="{@name}">
        <xsl:apply-templates select="math:*" mode="copy-expression"/>
      </code:assign>
    </xsl:template>

    <xsl:template match="slosl:select" mode="classgen">
      <!-- infer attribute types, call 'view_attribute' to write 'nodeproperty' tag
         -->
      <xsl:choose>
        <xsl:when test="key('attributes', @name)">
          <xsl:call-template name="view_attribute">
            <xsl:with-param name="name"      select="@name"/>
            <xsl:with-param name="attribute" select="key('attributes', @name)[1]"/>
          </xsl:call-template>
        </xsl:when>
        <xsl:when test=".//math:ci[substring(.,1,5) = 'node.' and key('attributes',
           substring(.,6))]">
          <xsl:call-template name="view_attribute">
            <xsl:with-param name="name" select="@name"/>
            <xsl:with-param name="attribute"
                             select=".//math:ci[substring(.,1,5) = 'node.' and key('
                               attributes', substring(.,6))][1]"/>
          </xsl:call-template>
        </xsl:when>
        <xsl:otherwise>
          <xsl:call-template name="view_attribute">
            <xsl:with-param name="name" select="@name"/>
          </xsl:call-template>
          <xsl:message>Cannot infer attribute type of '<xsl:value-of select="@name
            "/>'.</xsl:message>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:template>

    <xsl:template name="view_attribute">
      <!-- write 'nodeproperty' tag for an attribute -->
      <xsl:param name="name"/>
      <xsl:param name="attribute"/>

      <xsl:variable name="type_name">
        <xsl:if test="$attribute">
          <xsl:call-template name="sqltype">
            <xsl:with-param name="type" select="$attribute/@type_name"/>
          </xsl:call-template>
        </xsl:if>
      </xsl:variable>

      <code:nodeproperty name="{$name}" nala_type="{$type_name}"/>
    </xsl:template>


    <!-- add dependency information to MathML tree -->

    <!--
    <xsl:template match="node()" mode="attribute-depends">
      <xsl:for-each select=".//math:ci[substring(.,1,5) = 'node.' and not(. =
         preceding::math:ci)]">
        <xsl:sort select="string()"/>
        <l:name><xsl:copy-of select="substring(.,6)"/></l:name>
      </xsl:for-each>
    </xsl:template>

    <xsl:template match="node()" mode="non-attribute-depends">
      <xsl:for-each select=".//math:ci[substring(.,1,5) != 'node.' and not(. =
         preceding::math:ci)]">
        <xsl:sort select="string()"/>
        <l:name><xsl:copy-of select="string()"/></l:name>
      </xsl:for-each>
```

```
  </xsl:template>

  <xsl:template match="slosl:*" mode="copy-expression">
    <!-- default for 'slosl:*': just copy and continue the traversal -->
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates mode="copy-expression"/>
    </xsl:copy>
  </xsl:template>
  -->

  <xsl:template match="math:ci" mode="copy-expression">
    <!-- add 'code:type' and 'code:nala_type' attributes to all math:ci tags -->
    <xsl:copy>
      <xsl:choose>
        <xsl:when test="substring(.,1,5) = 'node.'">
          <xsl:attribute name="code:type">attribute</xsl:attribute>
          <xsl:attribute name="code:nala_type">
            <xsl:call-template name="sqltype">
              <xsl:with-param name="type"
                select="key('attributes', substring(.,6))/@type_name"/>
            </xsl:call-template>
          </xsl:attribute>
          <xsl:value-of select="substring(.,6)"/>
        </xsl:when>
        <xsl:when test="substring(.,1,6) = 'local.'">
          <xsl:attribute name="code:type">local_attribute</xsl:attribute>
          <xsl:attribute name="code:nala_type">
            <xsl:call-template name="sqltype">
              <xsl:with-param name="type"
                select="key('attributes', substring(.,7))/@type_name"/>
            </xsl:call-template>
          </xsl:attribute>
          <xsl:value-of select="substring(.,7)"/>
        </xsl:when>
        <xsl:otherwise>
          <xsl:attribute name="code:type">identifier</xsl:attribute>
          <xsl:value-of select="string()"/>
        </xsl:otherwise>
      </xsl:choose>
      <xsl:apply-templates select="*" mode="copy-expression"/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="math:*[.//math:ci]" mode="copy-expression">
    <!-- add 'code:depends' tags to all levels of the math tree -->
    <xsl:variable name="test" select=".//math:ci"/>

    <xsl:copy>
      <xsl:copy-of select="@*"/>

      <xsl:for-each select="$test[not(. = preceding::*[1])]">
        <xsl:sort />

        <xsl:choose>
          <xsl:when test="substring(.,1,5) = 'node.'">
            <code:depends type="attribute">
              <xsl:attribute name="nala_type">
                <xsl:call-template name="sqltype">
                  <xsl:with-param name="type"
                    select="key('attributes', substring(.,6))/@type_name"/>
                </xsl:call-template>
              </xsl:attribute>
              <xsl:value-of select="substring(.,6)"/>
            </code:depends>
          </xsl:when>
          <xsl:when test="substring(.,1,6) = 'local.'">
            <code:depends type="local_attribute">
              <xsl:attribute name="nala_type">
                <xsl:call-template name="sqltype">
                  <xsl:with-param name="type"
                    select="key('attributes', substring(.,7))/@type_name"/>
                </xsl:call-template>
              </xsl:attribute>
              <xsl:value-of select="substring(.,7)"/>
            </code:depends>
          </xsl:when>
          <xsl:otherwise>
            <code:depends type="identifier"><xsl:value-of select="string()"/></
              code:depends>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:for-each>

      <xsl:apply-templates mode="copy-expression"/>
      </xsl:copy>
    </xsl:template>

    <xsl:template match="math:*" mode="copy-expression">
      <!-- default fallback: just copy and continue the traversal -->
      <xsl:copy>
        <xsl:copy-of select="@*"/>
        <xsl:apply-templates mode="copy-expression"/>
      </xsl:copy>
    </xsl:template>

</xsl:stylesheet>
```

A.6 XSL Transformation from HIMDEL to Plain Messages

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:msg="http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/himdel"
  >

  <xsl:import href="common.xsl"/>
  <xsl:output method="xml" encoding="UTF-8" indent="no" />
  <xsl:strip-space elements="*"/>

  <xsl:param name="copy_toplevel">false</xsl:param>

  <xsl:template match="msg:message_hierarchy">
    <msg:messages>
      <xsl:apply-templates mode="messages"/>
      <xsl:if test="$copy_toplevel != 'false'">
        <xsl:copy-of select="msg:container"/>
        <xsl:copy-of select="msg:protocol"/>
      </xsl:if>
    </msg:messages>
  </xsl:template>

  <xsl:template match="msg:message">
    <msg:messages>
      <xsl:apply-templates select="." mode="messages"/>
    </msg:messages>
  </xsl:template>

  <xsl:template match="/*">
    <xsl:apply-templates select="msg:message_hierarchy" />
  </xsl:template>

  <xsl:template match="msg:message" mode="messages">
    <xsl:variable name="message" select="."/>
    <xsl:variable name="children">
      <xsl:apply-templates select="msg:*" mode="message"/>
      <xsl:apply-templates select="ancestor::msg:header[string(@access_name)]"
          mode="message"/>
    </xsl:variable>

    <xsl:variable name="typename" select="@type_name"/>
    <xsl:variable
        name="protocols"
        select="ancestor::msg:message_hierarchy/msg:protocol[msg:message-ref/
          @type_name = $typename]"/>

    <xsl:choose>
      <xsl:when test="$protocols">
        <xsl:for-each select="$protocols">
          <msg:message>
            <xsl:apply-templates select="$message/@access_name|$message/@type_name
                " mode="copyattr"/>
            <xsl:copy-of select="$children"/>
            <xsl:copy>
              <xsl:apply-templates select="@access_name|@type_name" mode="copyattr
                "/>
            </xsl:copy>
          </msg:message>
        </xsl:for-each>
      </xsl:when>
      <xsl:otherwise>
        <xsl:copy>
          <xsl:apply-templates select="@access_name|@type_name" mode="copyattr"/>
          <xsl:copy-of select="$children"/>
        </xsl:copy>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

  <!-- headers -->
  <xsl:template match="msg:header[string(@access_name)]" mode="message">
    <xsl:copy>
      <xsl:apply-templates select="@*" mode="copyattr"/>
      <xsl:apply-templates select="msg:*" mode="header"/>
    </xsl:copy>
  </xsl:template>
  <xsl:template match="msg:header" mode="message" />
  <xsl:template match="msg:header[not(string(@access_name))]" mode="header">
    <xsl:apply-templates select="msg:*" mode="header"/>
  </xsl:template>

  <xsl:template match="msg:content|msg:viewdata|msg:container|msg:container-ref"
     mode="header">
    <xsl:apply-templates select="." mode="message"/>
  </xsl:template>

  <xsl:template match="msg:*" mode="header" />

  <!-- message content -->
  <xsl:template match="msg:content|msg:viewdata" mode="message">
    <xsl:copy>
      <xsl:apply-templates select="@*" mode="copyattr"/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="msg:container[string(@access_name)]" mode="message">
    <xsl:copy>
      <xsl:apply-templates select="@access_name|@type_name" mode="copyattr"/>
      <xsl:apply-templates mode="message"/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="msg:container" mode="message">
    <xsl:apply-templates mode="message"/>
  </xsl:template>

  <xsl:template match="msg:container-ref[string(@access_name)]" mode="message">
    <msg:container>
      <xsl:apply-templates select="@access_name|@type_name" mode="copyattr"/>
      <xsl:variable name="typename" select="@type_name"/>
      <xsl:apply-templates
          select="ancestor::msg:message_hierarchy/msg:container[@type_name = $
            typename]/msg:*"
          mode="message"/>
    </msg:container>
  </xsl:template>

  <xsl:template match="msg:container-ref" mode="message">
    <xsl:variable name="typename" select="@type_name"/>
    <xsl:apply-templates
        select="ancestor::msg:message_hierarchy/msg:container[@type_name = $
          typename]/msg:*"
        mode="message"/>
  </xsl:template>
</xsl:stylesheet>
```

Bibliography

[A$^+$02] Assaf Arkin et al. *Business Process Modeling Language (BPML) Version 1.0.* Business Process Management Initiative (BPMI), November 2002.

[Abe01] Karl Aberer. 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*, September 2001.

[AG97] Martìn Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proc. of the 4th ACM Conference on Computer and Communications Security*, Zürich, Switzerland, April 1997.

[AH00] Eytan Adar and Bernardo A. Huberman. Free Riding on Gnutella. Technical report, Xerox PARC, August 2000.

[AKD03] Mikhail J. Atallah, Florian Kerschbaum, and Wenliang Du. Secure and private sequence comparisons. In *WPES '03: Proc. of the 2003 ACM workshop on Privacy in the electronic society*, Washington, DC, USA, October 2003.

[AKSX04] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *Proc. of the ACM SIGMOD Int. Conference on Management of Data*, Paris, France, June 2004.

[AOAG$^+$05] Karl Aberer, Luc Onana Alima, Ali Ghodsi, Sarunas Girdzijauskas, Seif Haridi, and Manfred Hauswirth. The essence of P2P: A reference architecture for overlay networks. In *Proc. of the 5th IEEE Intl. Conference on Peer-to-Peer Computing (P2P2005)*, Konstanz, Germany, September 2005.

[AR02] Filipe Araujo and Luis Rodrigues. On QoS-aware publish subscribe. In *Proc. of the 1st Int. Workshop on Distributed Event-based Systems (DEBS'02)*, Vienna, Austria, July 2002.

[AS04] Baruch Awerbuch and Christian Scheideler. The hyperring: A low-congestion deterministic data structure for distributed environments. In *Proc. of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, New Orleans, Louisiana, USA, January 2004.

[BB05a] Stefan Behnel and Alejandro Buchmann. 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, August 2005.

[BB05b] Stefan Behnel and Alejandro Buchmann. Overlay Networks – Implementation by Specification. In *Proc. of the Int. Middleware Conference (Middleware2005)*, Grenoble, France, November 2005.

[BBG$^+$06] Stefan Behnel, Alejandro Buchmann, Paul Grace, Barry Porter, and Geoff Coulson. A specification-to-deployment architecture for overlay networks. In *Proc. of the Int. Symposium on Distributed Objects and Applications (DOA)*, Montpellier, France, October 2006.

[Beh05a] Stefan Behnel. MathDOM – A Content MathML Implementation for the Python Programming Language. `http://mathdom.sourceforge.net/`, 2005.

[Beh05b] Stefan Behnel. The SLOSL Overlay Workbench for Visual Overlay Design. `http://developer.berlios.de/projects/slow/`, 2005.

[Beh05c] Stefan Behnel. Tin Topologies – Designing Overlay Networks in Databases. In *Proc. of the Int. Middleware Conference (Middleware2005)*, Grenoble, France, November 2005. Demonstration and Poster Presentation.

[Beh05d] Stefan Behnel. Topologien aus der Dose – Ein Datenbank-Ansatz zum Overlay-Design. In Paul Müller, Reinhard Gotzhein, and Jens B. Schmitt, editors, *KiVS Kurzbeiträge und Workshop*, volume 61, Kaiserslautern, Germany, March 2005. Poster.

[Beh07] Stefan Behnel. SLOSL - a modelling language for topologies and routing in overlay networks. In *Proc. of the 1st Int. Workshop on Modeling, Simulation and Optimization of Peer-to-peer environments (MSOP2P)*, Naples, Italy, February 2007.

[BEP$^+$03] Andras Belokosztolszki, David M. Eyers, Peter R. Pietzuch, Jean Bacon, and Ken Moody. Role-based access control for publish/subscribe middleware architectures. In *In Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, San Diego, CA, USA, June 2003.

[BFM06] Stefan Behnel, Ludger Fiege, and Gero Mühl. On quality-of-service and publish-subscribe. In *Proc. of the 5th Int. Workshop on Distributed Event-based Systems (DEBS'06)*, Lisbon, Portugal, July 2006.

[BHMS04] Hannes Birck, Oliver Heckmann, Andreas Mauthe, and Ralf Steinmetz. The two-step overlay network simulation approach. In *Proc. of SoftCOM, Split, Croatia*, October 2004.

[Bir02] Kenneth Birman. The surprising power of epidemic communication. In A. Schiper, A.A. Shvartsman, H. Weatherspoon, and B.Y. Zhao, editors, *International Workshop on Future Directions in Distributed Computing (FuDiCo 2002)*, volume 2584 of *LNCS*, Bertinoro, Italy, 2002. Springer-Verlag. Revised papers of the FuDiCo 2002 Workshop.

[BLZK04] Xiaole Bai, Shuping Liu, Peng Zhang, and Raimo Kantola. ICN: Interest-based clustering network. In *Proc. of the 4th IEEE Intl. Conference on Peer-to-Peer Computing (P2P2004)*, Zürich, Switzerland, August 2004.

[BMR$^+$96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley & Sons, 1996.

[BNAG04] Johannes Borgström, Uwe Nestmann, Luc Onana Alima, and Dilian Gurov. Verifying a structured peer-to-peer overlay network: The static case. In *Proc. of the Int. Workshop on Global Computing*, 2004.

[BSV03] Ranjita Bhagwan, Stefan Savage, and Geoffrey Voelker. Understanding availability. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, USA, February 2003.

[CAR05] Nuno Carvalho, Filipe Araujo, and Luis Rodrigues. Scalable QoS-based event routing in publish-subscribe systems. Technical report, Departamento de Informática, Faculdade de Ciências da Universidade de Lisboa, Lisbon, Portugal, February 2005.

[CBG^{+}04] Geoff Coulson, Gordon Blair, Paul Grace, Ackbar Joolia, Kevon Lee, and Jo Ueyama. A Component Model for Building Systems Software. In *Proc. of the IASTED Conference on Software Engineering and Applications (SEA'04)*, Cambridge, MA, USA, 2004.

[CC04] Nicolas Christin and John Chuang. On the cost of participating in a peer-to-peer network. In *Proc. of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS04)*, San Diego, CA, USA, February 2004.

[CDK^{+}03] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: High-bandwidth content distribution in a cooperative environment. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, USA, February 2003.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, June 2000.

[CFH^{+}03] Mariano Cilia, Ludger Fiege, Christian Haul, Andreas Zeidler, and Alejandro Buchmann. Looking into the past: Enhancing mobile publish/subscribe middleware. In *Proc. of the 2nd Int. Workshop on Distributed Event-based Systems (DEBS'03)*, San Diego, CA, USA, June 2003.

[CGT89] Stefano Ceri, Goerg Gottlob, and Letizia Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[CJK^{+}03] Miguel Castro, Michael B. Jones, Anne-Marie Kermarrec, Antony Rowstron, Marvin Theimer, Helen Wang, and Alec Wolman. An evaluation of scalable application-level multicast using peer-to-peer overlays. In *Proc. of INFOCOM 2003*, San Francisco, CA, USA, March 2003.

[CM01] James Clark and Makoto Murata. *RELAX NG Specification.* The Organization for the Advancement of Structured

Information Standards (OASIS), December 2001. http://relaxng.org/spec-20011203.html.

[Coo05] Brian F. Cooper. An optimal overlay topology for routing peer-to-peer searches. In *Proc. of the Int. Middleware Conference (Middleware2005)*, Grenoble, France, November 2005.

[CRB+03] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making Gnutella-like P2P systems scalable. In *Proc. of the 2003 ACM SIGCOMM Conference*, Karlsruhe, Germany, August 2003.

[CS02] Edith Cohen and Scott Shenker. Replication strategies in unstructured peer-to-peer networks. In *Proc. of the 2002 ACM SIGCOMM Conference*, Pittsburgh, PA, USA, August 2002.

[D+06] Pierre Delisle et al. *Java Specification Request 245: JavaServer Pages 2.1.* Sun Microsystems, Inc., May 2006.

[DAK00] Wenliang Du, Mikhail J. Atallah, and Florian Kerschbaum. Protocols for secure remote database access with approximate matching. In *Proc. of the First ACMCCS Workshop on Security and Privacy in E-Commerce*, Athens, Greece, November 2000.

[dB46] Nicolaas Govert de Bruijn. A combinatorial problem. *Koninklijke Nederlandse Akademie van Wetenschappen*, 49:758–764, 1946.

[DBM88] U. Dayal, A. Buchmann, and D. McCarthy. Rules are objects too: a knowledge model for an active, object-oriented database system. In *Proc. of the 2nd International Workshop on Object-Oriented Database Systems*, Bad Münster am Stein-Ebernburg, Germany, 1988.

[DCKM04] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. In *Proc. of the 2004 ACM SIGCOMM Conference*, Portland, Oregon, USA, August 2004.

[Dee91] Steve Deering. *Multicast Routing in a Datagram Internetwork.* PhD thesis, Stanford University, 1991.

[DGH+87] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, and John Larson. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM*

Symposium on Principles of Distributed Computing, pages 1–12. ACM Press, 1987.

[DMS04] Vasilios Darlagiannis, Andreas Mauthe, and Ralf Steinmetz. Overlay design mechanisms for heterogeneous, large scale, dynamic P2P systems. *Journal of Network and Systems Management, Special Issue on Distributed Management*, 12(3), September 2004.

[DOT70] George A. Donohue, Clarice N. Olien, and Phillip J. Tichenor. Mass media flow and differential growth in knowledge. *Public Opinion Quarterly*, 32(2):159–170, 1970.

[DZDS03] Frank Dabek, Ben Zhao, Peter Druschel, and Ion Stoica. 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, February 2003.

[EAABH03] Sameh El-Ansary, Luc Onana Alima, Per Brand, and Seif Haridi. Efficient broadcast in structured P2P networks. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, USA, February 2003.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[Fie05] Ludger Fiege. *Visibility in Event-Based Systems.* PhD thesis, Technical University of Darmstadt, Darmstadt, Germany, 2005.

[FM03] Michael J. Freedman and David Mazières. Sloppy hashing and self-organizing clusters. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, USA, February 2003.

[FMG03] Ludger Fiege, Gero Mühl, and Felix C. Gärtner. Modular event-based systems. *The Knowledge Engineering Review*, 17(4):359–388, 2003.

[Fow04] Martin Fowler. *UML Distilled.* Addison-Wesley, 2004.

[FV02] Michael J. Freedman and Radek Vingralek. Efficient peer-to-peer lookup based on a distributed trie. In *Proc. of the 1st International Workshop on Peer-to-Peer Systems (IPTPS02)*, MIT Faculty Club, Cambridge, MA, USA, March 2002.

[FZB$^+$04] Ludger Fiege, Andreas Zeidler, Alejandro Buchmann, Roger Kilian-Kehr, and Gero Mühl. Security aspects in publish/subscribe systems. In Antonio Carzaniga and Pascal Fenkam, editors, *Third Intl. Workshop on Distributed Event-based Systems (DEBS'04)*, Edinburgh, Scotland, UK, May 2004. IEEE.

[GBL$^+$03] Indranil Gupta, Ken Birman, Prakash Linga, Al Demers, and Robbert van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, USA, February 2003.

[GCB$^+$04] Paul Grace, Geoff Coulson, Gordon Blair, Laurent Mathy, David Duce, Chris Cooper, Wai Kit Yeung, and Wei Cai. GRIDKIT: Pluggable overlay networks for grid computing. In *Proc. of the Int. Symposium on Distributed Objects and Applications (DOA)*, Larnaca, Cyprus, October 2004.

[GCBP05] Paul Grace, Geoff Coulson, Gordon S. Blair, and Barry Porter. Deep middleware for the divergent grid. In *Proc. of the Int. Middleware Conference (Middleware2005)*, Grenoble, France, November 2005.

[GEBF$^+$03] Luis Garcés-Erice, Ernst W. Biersack, Pascal A. Felber, Keith W. Ross, and Guillaume Urvoy-Keller. Hierarchical peer-to-peer systems. In *Proc. of the Int. Conference on Parallel and Distributed Computing (Euro-Par)*, Klagenfurt, Austria, August 2003.

[GGG$^+$03] Krishna Gummadi, Ramakrishna Gummadi, Steven Gribble, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The impact of dht routing geometry on resilience and proximity. In *Proc. of the 2003 ACM SIGCOMM Conference*, Karlsruhe, Germany, August 2003.

[GKM01] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulie. SCAMP: Peer-to-peer lightweight membership service for large-scale group communication. In *Proc. of the 3rd Int. Workshop on Networked Group Communications (NGC'01)*, London, UK, November 2001.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[HCW04] Daniel Hughes, Geoff Coulson, and Ian Warren. A framework for developing reflective and dynamic P2P networks (RaDP2P). In *Proc. of the 4th IEEE Intl. Conference on Peer-to-Peer Computing (P2P2004)*, Zürich, Switzerland, August 2004.

[HJS$^+$03] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proc. of the 4th USENIX Symposium on Internet Technologies and Systems*, Seattly, WA, USA, March 2003.

[HP98] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts*. McGraw-Hill, 1998.

[HSS03] Oliver Heckmann, Jens Schmidt, and Ralf Steinmetz. Peer-to-peer Tauschbörsen - Eine Protokollübersicht. *Wirtschaftsinformatik, Schwerpunktthema Peer-to-Peer (P2P): Technologien, Architekturen und Anwendungen*, 3, March 2003.

[Hui99] Christian Huitema. *Routing in the Internet*. Prentice Hall, 1999.

[HW02] Steven Hazel and Brandon Wiley. Achord: A variant of the chord lookup service for use in censorship resistant peer-to-peer publishing systems. In *Proc. of the 1st International Workshop on Peer-to-Peer Systems (IPTPS02)*, MIT Faculty Club, Cambridge, MA, USA, March 2002.

[IDL99] ISO, Document ISO/IEC 14750:1999. *Interface Definition Language (IDL)*, 1999.

[II81] Makoto Imase and Masaki Itoh. Design to minimize diameter on building-block networks. *IEEE Transactions on Computers*, 30, 1981.

[Jan02] John Jannotti. *Network Layer Support for Overlay Networks*. PhD thesis, Massachusetts Institute of Technology, August 2002.

[JGKvS04] Mark Jelasity, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In *Proc. of the Int. Middleware Conference (Middleware2004)*, Toronto, Canada, October 2004.

[Jos02] Sam Joseph. NeuroGrid: Semantically routing queries in Peer-to-Peer networks. In *Proc. of the Second IEEE Int. Conference on Peer-to-Peer Computing (P2P2002)*, Linköping, Sweden, September 2002.

[JXT] JXTA. `http://www.jxta.org`.

[JXT03] Sun Microsystems, Inc. *JXTA v2.0 Protocols Specification*, February 2003.

[KBC+00] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gumadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*, November 2000.

[Kel05] David Keller. Uml hierarchie des diagrammes. `http://en.wikipedia.org/wiki/Image:Uml_hierarchie_des_diagrammes.png`, November 2005. authorised.

[KK03] M. Frans Kaashœk and David R. Karger. Koorde: A simple degree-optimal distributed hash table. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, USA, February 2003.

[KLR96] Jonas S. Karlsson, Witold Litwin, and Tore Risch. LH*lh: A scalable high performance data structure for switched multicomputers. In *Proc. of the Intl. Conference on Extending Database Technology (EDBT)*, Avignon, France, March 1996.

[KR04] David R. Karger and Matthias Ruhl. Diminished Chord: A protocol for heterogeneous subgroup formation in peer-to-peer networks. In *Proc. of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS04)*, San Diego, CA, USA, February 2004.

[KRT03] Jussi Kangasharju, Keith W. Ross, and David A. Turner. Secure and resilient peer-to-peer e-mail: Design and implementation. In *Proc. of the 3rd IEEE Int. Conference on Peer-to-Peer Computing (P2P2003)*, Linköping, Sweden, September 2003.

[LCG+06] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: Language, execution and optimization. In *Proc. of the ACM SIGMOD Int. Conference on Management of Data*, Chicago, IL, USA, June 2006.

[LCH+05] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. *SIGOPS Operating Systems Review*, October 2005.

[LGW04] Baochun Li, Jiang Guo, and Mea Wan. iOverlay: A lightweight middleware infrastructure for overlay application implementations. In *Proc. of the Int. Middleware Conference (Middleware2004)*, Toronto, Canada, October 2004.

[LHSH04] Boon Thau Loo, Ryan Huebsch, Ion Stoica, and Joseph Hellerstein. The case for a hybrid P2P search infrastructure. In *Proc. of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS04)*, San Diego, CA, USA, February 2004.

[LHSR05] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: Extensible routing with declarative queries. In *Proc. of the 2005 ACM SIGCOMM Conference*, Philadelphia, PA, USA, August 2005.

[LKRG03] Dmitri Loguinov, Anuj Kumar, Vivek Rai, and Sai Ganesh. Graph-theoretic analysis of structured peer-to-peer systems: Routing distances and fault resilience. In *Proc. of the 2003 ACM SIGCOMM Conference*, Karlsruhe, Germany, August 2003.

[LLH$^+$03] Jinyang Li, Boon Thau Loo, Joe Hellerstein, Frans Kaashœk, David R. Karger, and Robert Morris. On the feasibility of peer-to-peer web indexing and search. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, USA, February 2003.

[LNS93] Witold Litwin, Marie-Anne Neimat, and Donovan A. Schneider. LH* - linear hashing for distributed files. In *Proc. of the ACM SIGMOD Int. Conference on Management of Data*, Washington, D.C., USA, May 1993.

[LW95] David Lomet and Jennifer Widom, editors. *Special Issue on Materialized Views and Data Warehousing*, volume 18 of *IEEE Data Engineering Bulletin.* June 1995.

[M$^+$03] Gregory Murray et al. *Java Specification Request 154: Java Servlet 2.4 Specification.* Sun Microsystems, Inc., November 2003.

[Mat03] The World Wide Web Consortium. *Mathematical Markup Language (MathML) Version 2.0 (Second Edition)*, October 2003. `http://www.w3.org/TR/MathML2/`.

[MCH01] Laurent Mathy, Roberto Canonico, and David Hutchinson. An overlay tree building control protocol. In *Proc. of the 3rd Int. Workshop on Networked Group Communications (NGC'01)*, London, UK, November 2001.

[MCKS03] Alper Tugay Mizrak, Yuchung Cheng, Vineet Kumar, and Stefan Savage. Structured superpeers: leveraging heterogeneity to provide constant-time lookup. In *Proc. of the Third IEEE Workshop on Internet Applications (WIAPP03)*, San Jose, CA, USA, June 2003.

[MD04] Alan Mislove and Peter Druschel. Providing administrative control and autonomy in peer-to-peer overlays. In *Proc. of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS04)*, San Diego, CA, USA, February 2004.

[MFB02] Gero Mühl, Ludger Fiege, and Alejandro P. Buchmann. Filter similarities in content-based publish/subscribe systems. In H. Schmeck, T. Ungerer, and L. Wolf, editors, *International Conference on Architecture of Computing Systems (ARCS)*, pages 224–238, Karlsruhe, Germany, 2002.

[Mil67] Stanley Milgram. The small world problem. *Psychology Today*, pages 60–67, May 1967.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus.* Cambridge University Press, 1st edition, June 1999.

[MM02] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. of the 1st International Workshop on Peer-to-Peer Systems (IPTPS02)*, MIT Faculty Club, Cambridge, MA, USA, March 2002.

[MM03] Jishnu Mukerji and Joaquin Miller. MDA Guide, v1.0.1. omg/03-06-01, June 2003.

[MPR$^+$03] Alan Mislove, Ansley Post, Charles Reis, Paul Willmann, Peter Druschel, Dan S. Wallach, Xavier Bonnaire, Pierre Sens, Jean-Michel Busca, and Luciana Arantes-Bezerra. POST: A secure, resilient, cooperative messaging system. In *Proc. of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*, Lihue, Hawaii, May 2003.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*, chapter 13, pages 377–422. Morgan Kaufmann, 1st edition, August 1997.

[Müh02] Gero Mühl. *Large-Scale Content-Based Publish/Subscribe Systems.* PhD thesis, Darmstadt University of Technology, 2002. http://elib.tu-darmstadt.de/diss/000274/.

[MUHW04] Gero Mühl, Andreas Ulbrich, Klaus Herrmann, and Torben Weis. Disseminating information to mobile clients using publish/subscribe. *IEEE Internet Computing*, 8(3), May 2004.

[NPB03] Akihiro Nakao, Larry Peterson, and Andy Bavier. A routing underlay for overlay networks. In *Proc. of the 2003 ACM SIGCOMM Conference*, Karlsruhe, Germany, August 2003.

[NS02] Cheuk Hang Ng and Ka Cheung Sia. Peer clustering and firework query model. In *Poster Proc. of 11th World Wide Web Conference (WWW2002)*, Honolulu, Hawaii, May 2002.

[NW04] Moni Naor and Udi Wieder. Know thy neighbor's neighbor: Better routing for skip-graphs and small worlds. In *Proc. of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS04)*, San Diego, CA, USA, February 2004.

[Obj01] Object Management Group. Complete UML 1.4 specification, 2001.

[Obj02] Object Management Group. CORBA notification service, version 1.0.1. OMG Document formal/2002-08-04, 2002.

[Ora01] Andrew Oram, editor. *Peer-to-Peer - Harnessing the Power of Disruptive Technologies.* O'Reilly, March 2001.

[Pat99] Norman W. Paton, editor. *Active Rules in Database Systems.* Monographs in Computer Science. Springer, 1999. ISBN 0-387-98529-8.

[PB02] Peter Pietzuch and Jean Bacon. Hermes: A distributed event-based middleware architecture. In *Proc. of the 1st Int. Workshop on Distributed Event-based Systems (DEBS'02)*, Vienna, Austria, July 2002.

[PC06] Barry Porter and Geoff Coulson. Intelligent Dependability Services for Overlay Networks. In *Proc. of the 6th IFIP WG International Conference on Distributed Applications and Interoperable Systems*, Bologna, Italy, June 2006.

[PCW$^+$03] Marcelo Pias, Jon Crowcroft, Steve Wilbur, Tim Harris, and Saleem Bhatti. Lighthouses for scalable distributed location. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, USA, February 2003.

[PDZ99] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *Proc. of the USENIX Annual Technical Conference*, Monterey, CA, USA, June 1999.

[Pet63] Carl Adam Petri. Fundamentals of a theory of asynchronous information flow. In *Proc. of the 1962 IFIP Congress*, Amsterdam, Netherlands, August 1963.

[Pop97] Alan Pope. *The CORBA Reference Guide.* Addison-Wesley, Reading, MA, USA, 1997.

[Pug89] William Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Proceedings of the Workshop on Algorithms and Data Structures*, Ottawa, Ontario, Canada, August 1989.

[QS04] Dongyu Qiu and Rayadurgam Srikant. Modeling and performance analysis of bit torrent-like peer-to-peer networks. In *Proc. of the 2004 ACM SIGCOMM Conference*, Portland, Oregon, USA, August 2004.

[R$^+$02] Mark Reinhold et al. *Java Specification Request 51: New I/O APIs for the Java Platform.* Sun Microsystems, Inc., May 2002.

[RB04] Rodrigo Rodrigues and Charles Blake. When multi-hop peer-to-peer routing matters. In *Proc. of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS04)*, San Diego, CA, USA, February 2004.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale Peer-to-Peer systems. In *Proc. of the Int. Middleware Conference (Middleware2001)*, November 2001.

[Reb] Rebeca Event-Based Electronic Commerce Architecture. http://www.gkec.informatik.tu-darmstadt.de/rebeca/.

[Ree79] Trygve Reenskaug. Thing-Model-View-Editor - an example from a planningsystem. Technical report, Xerox PARC, May 1979.

[RFH$^+$01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content Addressable Network. In *Proc. of the 2001 ACM SIGCOMM Conference*, San Diego, CA, USA, August 2001.

[RGRK04] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *Proc. of the USENIX Annual Technical Conference*, Boston, MA, USA, June 2004.

[RHKS01] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application-level multicast using content-addressable networks. In *Proc. of the Third Int. COST264 Workshop (NGC 2001)*, London, UK, November 2001.

[RKB$^+$04] Adolfo Rodriguez, Charles Killian, Sooraj Bhat, Dejan Kostić, and Amin Vahdat. 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, March 2004.

[RKCD01] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Proc. of the 3rd Int. Workshop on Networked Group Communications (NGC'01)*, London, UK, November 2001.

[RL95] Y. Rekhter and T. Li. A Border Gateway Protocol 4 (BGP-4). RFC 1771 (Draft Standard), March 1995.

[SGG02] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of Peer-to-Peer file sharing systems. In *Proc. of Multimedia Computing and Networking 2002 (MMCN '02)*, San Jose, CA, USA, January 2002.

[SGMZ04] Kunwadee Sripanidkulchai, Aditya Ganjam, Bruce Maggs, and Hui Zhang. The feasibility of supporting large-scale live streaming applications with dynamic application end-points. In *Proc. of the 2004 ACM SIGCOMM Conference*, Portland, Oregon, USA, August 2004.

[SMK$^+$01] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of the 2001 ACM SIGCOMM Conference*, San Diego, CA, USA, August 2001.

[SMPD05] Daniel Sandler, Alan Mislove, Ansley Post, and Peter Druschel. FeedTree: Sharing web micronews with peer-to-peer event notification. In *Proc. of the 4th International Workshop on*

Peer-to-Peer Systems (IPTPS05), Ithaca, New York, USA, February 2005.

[SQL92] ISO, Document ISO/IEC 9075:1992. *Database Language SQL*, 1992.

[SQL03a] ISO, Document ISO/IEC 9075:2003. *Database Language SQL*, 2003.

[SQL03b] ISO, Document ISO/IEC 9075:14-2003. *Database Language SQL - Part 14: XML-Related Specifications (SQL/XML)*, 2003.

[Sri95] Raj Srinivasan. XDR: External Data Representation Standard. RFC 1832 (Draft Standard), August 1995.

[SSDN02] Mario Schlosser, Michael Sintek, Stefan Decker, and Wolfgang Nejdl. HyperCuP - shaping up Peer-to-Peer networks. Technical report, Stanford University, USA, 2002.

[Ste03] W. Richard Stevens. *UNIX Network Programming: The Sockets Networking API*, volume 1, chapter I/O Multiplexing: The select and poll Functions, pages 153–190. Addison-Wesley, 3rd edition, October 2003.

[Sun87] Sun Microsystems, Inc. XDR: External Data Representation standard. RFC 1014, June 1987.

[Sun88] Sun Microsystems, Inc. RPC: Remote Procedure Call Protocol specification: Version 2. RFC 1057 (Informational), June 1988.

[Sun02] Sun Microsystems, Inc. Java Message Service (JMS) Specification 1.1, 2002.

[SW05] Ralf Steinmetz and Klaus Wehrle, editors. *Peer-to-Peer Systems and Applications.* Springer Verlag, October 2005.

[TB04] Wesley W. Terpstra and Stefan Behnel. Bit Zipper Rendezvous - Optimal data placement for general P2P queries. In *International Dagstuhl Seminar 04111 on Peer-to-Peer-Systems and Applications*, Schloss Dagstuhl, Germany, March 2004.

[TBF$^+$03] Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Andreas Zeidler, and Alejandro Buchmann. A Peer-to-Peer Approach to Content-Based Publish/Subscribe. In *Proc. of the 2nd Int. Workshop on Distributed Event-based Systems (DEBS'03)*, San Diego, CA, USA, June 2003.

[TBF$^+$04] Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Jussi Kangasharju, and Alejandro Buchmann. Bit Zipper Rendezvous - Optimal Data Placement for General P2P Queries. In *Proc. of the 1st Int. Workshop on Peer-to-peer Computing and Databases*, Heraklion, Crete, March 2004.

[V$^+$06] Sekhar Vajjhala et al. *Java Specification Request 222: Java Architecture for XML Binding (JAXB) 2.0.* Sun Microsystems, Inc., May 2006.

[vRMH98] Robert van Renesse, Yaron Minsky, and Mark Hayden. A gossip-based failure detection service. In *Proc. of the Int. Middleware Conference (Middleware1998)*, Lake District, UK, September 1998.

[VvRB03] Werner Vogels, Robbert van Renesse, and Ken Birman. The power of epidemics: robust communication for large-scale distributed systems. *ACM SIGCOMM Computer Communication Review*, 33(1):131–135, 2003.

[W3C01] W3C. Web services description language (wsdl) 1.1. Technical Report, March 2001. http://www.w3.org/TR/wsdl.

[WB05] Craig Walls and Ryan Breidenbach. *Spring in Action.* Manning Publications Co., February 2005.

[WCB01] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proc. of the 18th ACM symposium on operating systems principles*, Banff, Alberta, Canada, 2001.

[WS98] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of small-world networks. *Nature*, 393:440–442, June 1998.

[XPa99] The World Wide Web Consortium. *XML Path Language (XPath) Version 1.0*, November 1999. `http://www.w3.org/TR/xpath/`.

[XQu06] The World Wide Web Consortium. *XML Query Language (XQuery) Version 1.0*, June 2006. `http://www.w3.org/TR/xquery/`.

[XSD04] The World Wide Web Consortium. *XML Schema Part 2: Datatypes Second Edition*, October 2004. `http://www.w3.org/TR/xmlschema-2/`.

[XSL99] The World Wide Web Consortium. *XSL Transformations (XSLT) Version 1.0*, November 1999. `http://www.w3.org/TR/xslt/`.

[XZ02] Zhichen Xu and Zheng Zhang. Building low-maintenance expressways for P2P systems. Technical report, Hewlett-Packard Labs, Palo Alto, CA, USA, 2002.

[YG03] Beverly Yang and Hector Garcia-Molina. Designing a Super-Peer network. In *Proc. of the 19th Int. Conference on Data Engineering (ICDE'03)*, Bangalore, India, March 2003.

[YZLD05] Mao Yang, Zheng Zhang, Xiaoming Li, and Yafei Dai. An empirical study of free-riding behavior in the maze P2P file-sharing system. In *Proc. of the 4th International Workshop on Peer-to-Peer Systems (IPTPS05)*, pages 182–192, Ithaca, New York, USA, February 2005.

[ZCLC05] Zhan Zhangy, Shigang Cheny, Yibei Lingz, and Randy Chow. Resilient capacity-aware multicast based on overlay networks. In *Proc. of the 25th Int. Conference on Distributed Computing Systems*, Columbus, Ohio, USA, June 2005.

[ZDE+93] Lixia Zhang, Steve Deering, Deborah Estrin, Scott Shenker, and Daniel Zappala. RSVP: A new resource ReSerVation Protocol. *IEEE Network*, 7(5):8–18, September 1993.

[ZDH+02] Ben Y. Zhao, Yitao Duan, Ling Huang, Anthony D. Joseph, and John D. Kubiatowicz. Brocade: Landmark routing on overlay networks. In *Proc. of the 1st International Workshop on Peer-to-Peer Systems (IPTPS02)*, MIT Faculty Club, Cambridge, MA, USA, March 2002.

[ZH03] Rongmei Zhang and Y. Charlie Hu. Borg: a hybrid protocol for scalable multicast in peer-to-peer networks. In *Proc. of the 13th Int. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'03)*, Monterey, CA, USA, June 2003.

[ZHD03] Rongmei Zhang, Y. Charlie Hu, and Peter Druschel. Optimizing routing in structured peer-to-peer overlay networks using routing table redundancy. In *Proc. of the 9th Int. Workshop on Future Trends of Distributed Computing Systems (FTDCS'03)*, San Juan, Puerto Rico, May 2003.

[ZKJ01] Ben Zhao, John Kubiatowicz, and Anthony Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical report, Computer Science Division, U. C. Berkeley, April 2001.

[Zoc04] Guido Zockoll. History of object-oriented methods and notations. `http://de.wikipedia.org/wiki/Bild:Oo-historie.png`, July 2004. authorised.

[ZOS00] Weibin Zhao, David Olshefski, and Henning Schulzrinne. Internet quality of service: An overview. Technical Report CUCS-003-00, Columbia University, February 2000.

[ZZJ$^+$01] Shelly Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proc. of the 11th Int. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'01)*, Port Jefferson, NY, USA, June 2001.