A Specification-to-Deployment Architecture for Overlay Networks

Stefan Behnel[1], Alejandro Buchmann[1],
Paul Grace[2], Barry Porter[2], and Geoff Coulson[2]

[1] Databases and Distributed Systems Group,
Darmstadt University of Technology (TUD), Germany
{behnel, buchmann}@dvs1.informatik.tu-darmstadt.de
[2] Computing Department, Lancaster University, UK
{gracep, porterbf, geoff}@comp.lancs.ac.uk

Abstract. Implementing overlay software is non-trivial and time-consuming. Current projects build overlays or intermediate frameworks on top of low-level networking abstractions. This leads to far reaching incompatibilities between overlay implementations, tight coupling to frameworks and limited adaptability to different deployment environments.

We present a new approach to rapid overlay implementation that combines a modelling framework for overlay design with a dynamic component architecture for run-time adaptation. It is the first architecture in the overlay area that tackles the complete design process from modelling, through code generation and implementation down to adaptive deployment. To demonstrate the effectiveness of this architecture, we describe the step-by-step procedure of designing an overlay and deploying it within an adaptive middleware framework.

1 Introduction

Recent years have seen a large body of research in decentralised, self-maintaining overlay networks like Chord [1], ODRI [2] or Gia [3]. They are commonly regarded as building blocks for Internet-scale distributed applications. Current overlay implementations are built with incompatible, language specific frameworks on top of low level networking abstractions. This complicates their design and hinders the comparison and integration of different topologies. Apart from a recently proposed API for the specific use case of structured overlay networks [4], there is little standardisation effort in the rest of the overlay area. And a common API by no means simplifies the design of the overlay implementation itself.

Currently, programmers who want to use overlays for their applications must decide in advance, at a very early design phase, which of the distinct overlay implementations they want to use and must invest time to understand its specific usage. This effectively prohibits testing the final product with different topologies or delivering versions with specialised overlays. There is no support for deployment time or run time adaptation to changing environments or quality-of-service requirements. Consequently, the actual usefulness of overlays for application design is currently very limited.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1522–1540, 2006.

This exposes a clear requirement for the high-level specification, development and automated deployment of overlay networks. In this paper, we investigate a specific case of this. Gridkit [5] is a middleware framework that underpins different middleware services: event notification, group communication, media streaming, etc. with an array of overlay network types; this is because the deployment of these services can be across highly diverse environments, and hence they require different properties from an overlay network, e.g. in terms of QoS attributes like robustness, scalability, ability to handle particular traffic patterns, etc. Hence, Gridkit developers need to rapidly generate and evaluate overlays that will best support particular middleware configurations; for example, which overlay will best support multicast in an ad-hoc network; and which will best support large-scale data storage in the Internet.

Therefore, this paper investigates a high-level specification-to-deployment architecture for overlay networks. It aims to provide developers with clean models for overlay design, implementation support using a model driven approach, and run-time deployment and dynamic adaptation using a component-based infrastructure. To evaluate our solution, we document an example case study which illustrates how a complex overlay implementation for the Gridkit middleware is specified, developed and deployed by our architecture. We chose to implement the Scribe [6] multicast tree atop the Chord [1] key-based routing overlay.

In the remainder of this paper, we first examine in section 2 the highly complementary work effected by two research groups at Lancaster University and the University of Darmstadt; namely, the Gridkit middleware and the OverML overlay specification tools respectively. In section 3, we then present OverGrid, an architecture that completes the edit-compile-deploy loop for developing overlays that can underpin middleware services. Then in section 4, we exemplify the development of a particular overlay. Finally, in section 5, we describe related work in this area, and draw conclusions in section 6.

2 Background on Gridkit and OverML

2.1 Gridkit

Motivation for Overlay-Based Middleware. Gridkit[1] [5] is a reflective middleware whose key aim is to support application development for pervasive computing in the face of increasing diversity at both the infrastructure level and the "middleware service" level. There is diversity in the types of devices (workstations, mobile devices, sensors, and clusters) and in the types of networks employed (e.g high-speed LAN, infrastructure-based wireless networks, and ad-hoc wireless networks). Furthermore, the requirements of middleware have exploded in the range of "middleware service types" that must be available to application developers; beginning with basic point-to-point interactions (e.g. RPC and

[1] `https://sourceforge.net/projects/gridkit/`

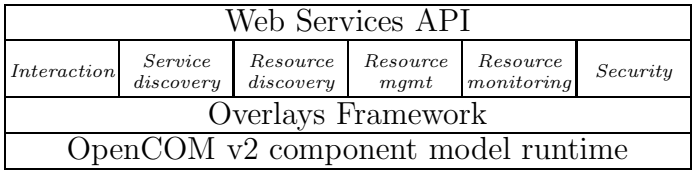


Fig. 1. The per-node Gridkit software framework

SOAP messaging), the range of interaction paradigms is expanding to include (amongst others): reliable and unreliable multicast; work flow; media streaming; publish-subscribe; tuple-space/ generative communication; and peer-to-peer based resource location or file sharing.

Gridkit's key approach to tackle these challenges is to deploy an extensive and extensible set of middleware services over an infrastructure of overlay networks that best support particular interaction types. For example, a publish-subscribe service can be built atop a multicast network service e.g. an application-level multicast overlay; and a tuple-space middleware can by layered atop a DHT key-based routing overlay e.g. Chord. Notably, one of the key research questions posed by Gridkit is how to select appropriate overlay networks to support individual middleware services.

Software Architecture. In terms of software architecture, the Gridkit framework (illustrated in figure 1) is deployed on each participating node of a middleware service. In this paper, we concentrate on the bottom two parts: the OpenCOM runtime and the overlays framework; for further information about the higher level services, see [5].

At the bottom level, Gridkit employs a minimal runtime for the loading and binding of lightweight software components, known as OpenCOM components [7]. Components are language-independent encapsulated units of functionality and deployment that interact with other components exclusively through interfaces and receptacles. Fundamentally, OpenCOM supports both reflection and component frameworks. Reflection is used to reason about component configurations and to dynamically alter configurations at runtime. Component frameworks are scoped compositions of components that accept plug-in components that are validated according to component framework specific constraint rules (cf. the overlay framework described later). A further key feature of the OpenCOM model is the support for component *caplets*. Essentially, these are separate address spaces into which heterogeneous component types can be loaded. The component runtime then manages the bindings between components in different caplets. For example, applications and systems software can easily be composed of Java, C++, python, and other OpenCOM components.

The layer above (the overlays framework) is a distributed framework for the deployment of multiple overlay networks. In practice, this amounts to hosting, in a set of distributed overlay framework instances, a set of per-overlay plug-in

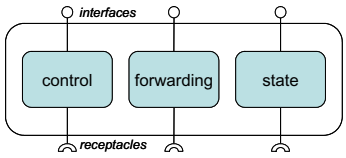


Fig. 2. The overlay framework control-state-forward specification

overlay components (see figure 2). Each of these represents a single overlay network node and consists of a *control* element that cooperates with its peers on other hosts to build and maintain some virtual network topology, a *forwarding* element that appropriately routes messages over its virtual topology, and a *state* element that contains per-overlay-node state such as a next-neighbours list.

The state component is specific to the overlay, and communicates through an overlay-specific state interface. Whereas, the control component and forwarding component implement the common interfaces *IControl* and *IForward* listed below. This is because overlays are plugged vertically into the framework using just the control and forward interfaces; state communication is horizontal. 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.

```
1  interface IControl {
2    ResultCode Create(String netId, Object params);
3    ResultCode Join(String netId, Object params);
4    ResultCode Leave(String netId);
5  }
6  interface IForward {
7    public byte[] Send(String destID, byte[] msg, int param);
8    public byte[] Receive(String netId);
9    public void EventReceive(String netId, IDeliver evHandler);
10 }
```

Cost of Developing Gridkit Overlays. Gridkit requires overlays that best support a middleware service in the current environmental conditions; hence, a key question to be answered is: which overlay meets the requirement set for this environment? To best answer this, multiple overlay types should be developed and tested to determine the right choice. Gridkit represents a considerable improvement on the basic approach of implementing overlays without a software framework. So far, we have implemented 7 types of overlay networks; this work has been carried out by 8 different developers. However, developing overlays in the Gridkit style is still a relatively expensive, time-consuming task. The following table illustrates both the length of code produced, and the time taken for each overlay to be developed. It can be seen that a new individual implementation will require a significant development expense.

Overlay Type	Code Lines	Man Days
Chord Key-based Routing [1]	790	42
Chord-based Distributed Hash Table [1]	570	35
Scribe[6]	880	42
Tree-Building Control Protocol [8]	7406	35
SCAMP [9]	1000	35
Minimum Spanning Tree	1300	20
Gossip-based Failure Monitor [10]	450	28

We further analysed the development process by examining the produced code, and questioning the developers. This exercise discovered these key points:

- Developers easily followed the framework structure e.g. the IForward and IControl interfaces i.e. the code behaved correctly for the create, join, leave, send, and receive operations. Hence, third-party overlay implementations were directly plugged in and out of the Gridkit framework.
- Developers sometimes had difficulty separating the overlay behaviour for control, forward and state into distinct components. It was noticeable that the code for these were often interleaved across the three components. Hence, this hinders the ability for a third party to effectively perform fine-grained dynamic reconfiguration e.g. replacement of the routing algorithm.
- The implementations of all the overlays are in Java. However, certain devices e.g. sensor motes cannot run a Java virtual machine. Hence, overlays would need to be reimplemented in order to operate on such devices. OpenCOM supports this level of portability, however, at present this requires overlay reimplementation and hence significant duplicated effort.

We therefore argue that the high-level development of portable overlay code is needed to overcome the identified problems, and provide the following benefits:

- Reduced development time, will aid experimentation of component configurations, leading to better supported middleware services.
- Generated overlay code will better follow the control-forward-state split, avoiding the high level of programmer skill needed for this task. The resultant implementation will then be easily reconfigurable by third parties.
- Portable overlay implementations can be generated for heterogeneous settings, reducing the duplication of overlay implementations.

2.2 OverML

The Overlay Modelling Language OverML [11] is an integrated set of domain specific XML specification languages for node attribute schemas, messages, view definitions, routing decisions and event graphs. There are obviously many different ways of specifying data schemas, messages and graphs. The main advantage of OverML is the clear focus on the domain of overlay design and the integration between the languages. This allows OverML models to provide a rich set of semantics for generative overlay implementation.

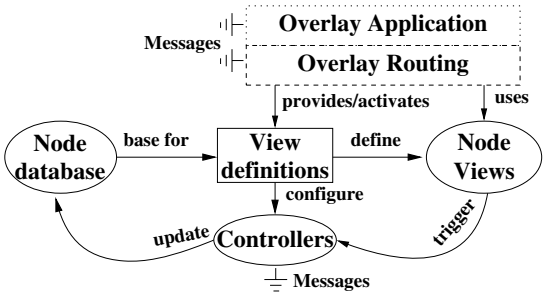


Fig. 3. Components of the OverML System Model

The architectural idea behind OverML is illustrated in figure 3. It clearly borrows from the Model-View-Controller Pattern for software design. The locally available knowledge about remote nodes (node attributes) and about the system state is stored in a local node database. View definitions select nodes from it according to topological rules and overlay adaptation strategies. The resulting database views present these nodes to software components. Note that the local database may well be incomplete or globally inconsistent. No guarantees are made on the data storage and representation side of the architecture.

Routers (or message forwarders) need these views for their forwarding decisions, controllers use them for their maintenance decisions. Controllers are event-triggered maintenance components that are integrated into the system at deployment time or run time. OverML event graphs are used to specify the connection between the events generated from views and messages, and the triggered controller actions that update the database or request new data from remote nodes.

The overlap between Gridkit and OverML should be clear from the figure. Just like Gridkit, OverML targets a clean separation of control, forwarding and state. However, its modelling approach allows it to impose further structure on these components and to leave a major part of their implementation to code generators and generic infrastructure components.

The OverML languages are briefly described here and in more detail in [11]. For brevity, **Nala** and **Himdel** , which specify node attributes and overlay messages respectively, are restricted to an exemplary description later on.

Slosl, the SQL-Like Overlay Specification Language. The view definitions for topology rules and adaptation are expressed in Slosl. We present it here using a simple example, an advanced implementation of the Chord graph [1].

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

The first three clauses behave as in SQL. The WHERE clause specifically selects nodes based on their attributes. Note that SLOSL is not concerned with the source of the information that node attributes contain. This is left entirely to the controllers. SLOSL only constrains and categorises the presentation of locally available data. The remaining clauses do the following:

WITH This clause defines variables or options of this view that can be set at instantiation time and changed at run-time. Here, *log_k* is a global constant for the life-time of an overlay, while *backups* allows adding neighbour redundancy at runtime.

HAVING–FOREACH This pair of clauses aggregates the selected nodes into buckets (or equivalence classes) to categorise them. In the example, the (constant) node attribute *ring_dist* refers to the logical distance between the local node and the remote node. The HAVING expression states that it must lie within the given half-open interval (excluding the highest value) that depends on the bucket variable i. The FOREACH part defines the available node buckets by declaring this bucket variable over a range (or a list, database table, ...) of values. It defines either a single bucket of nodes, or a list, matrix, cube, etc. of buckets. The structure is imposed by the occurrence of zero or more FOREACH clauses, where each clause adds a dimension. Nodes are selected into these buckets by the (optional) HAVING expression. In the example, a Chord node sorts remote nodes into ring intervals of increasing size and distance that designate equivalent neighbours: itself $\rightarrow 2^1 \rightarrow 2^2 \rightarrow 2^3 \ldots$

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

EDGAR, Extensible Decision Graphs for Adaptive Routing. EDGAR is a small graph language that models routing decisions. As opposed to other routing languages, it is not concerned with topology decisions. This is entirely left to SLOSL views. EDGAR only combines views and conditions into a decision graph. A general graph is shown in figure 4.

EDSL, the Event Driven State-Machine Language. EDSL is the language that connects the OverML specification with external controller components. It defines the event flow in the overlay implementation, including the protocol and maintenance strategies. EDSL is a special event-driven state machine language in that it knows about the events specified by the other OverML languages. States

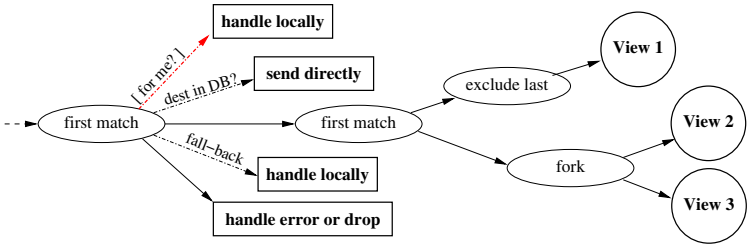


Fig. 4. General graph for EDGAR routing decisions

are defined using a number of generic interfaces that handle incoming messages or that respond to view events. Event flows are visually described as automata, as shown in figure 5. An important feature of EDSL is the support for subgraphs. This allows the creation of EDSL implemented complex components for database update strategies and maintenance, and their integration in different designs.

The SLOSL Overlay Workbench. Major work went into the implementation of a graphical overlay design tool, named the SLOSL Overlay Workbench. The screenshots in figure 5 give an idea about the interfaces for the different languages of OverML [11]. They support the semantically integrated specification of node attributes, messages, SLOSL view statements and component event flow graphs. EDGAR routing support is currently worked on. The workbench exports platform independent XML descriptions of the designed overlay, which can be passed into translators and source code generators. The development of the workbench is now continued as an Open Source project at the Berlios developer site:
`http://developer.berlios.de/projects/slow/`

3 OverGrid – The Best of Both Worlds

OverML and Gridkit are very complementary approaches. OverML focuses on the design aspects of overlay implementations and tries to provide platform-independent models for model driven code generation. Gridkit, on the other hand, provides a dynamic component environment and a layered software architecture for run-time adaptation of overlay systems. Where OverML lacks the support for modelling and designing the controllers in source-level detail, Gridkit provides a dynamic software component infrastructure and generic, layered networking interfaces. Where Gridkit lacks the support for rapid overlay development and code portability, OverML excels with its platform-independent models and the generation of efficient, platform-specific source code.

The resulting architecture is presented in figure 6. The vertical separation follows Gridkit's interface layers. The horizontal separation represents the OverML implemented overlay architecture. 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 glue code from the model.

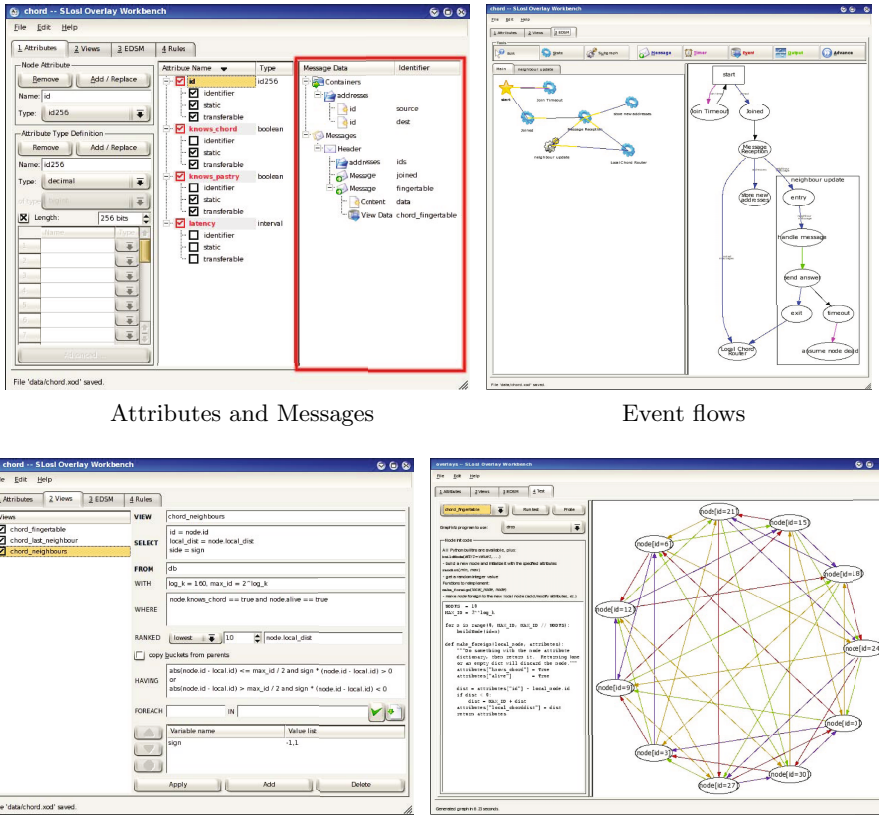


Fig. 5. A Glimpse at the SLOSL Overlay Workbench

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.

3.1 Database and SLOSL Views

OverGrid replaces individual overlay state components in Gridkit overlays with a 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. However, both of them are partially generated from OverML. 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). In its simplest incarnation, the database can even be a hash table with node objects that are

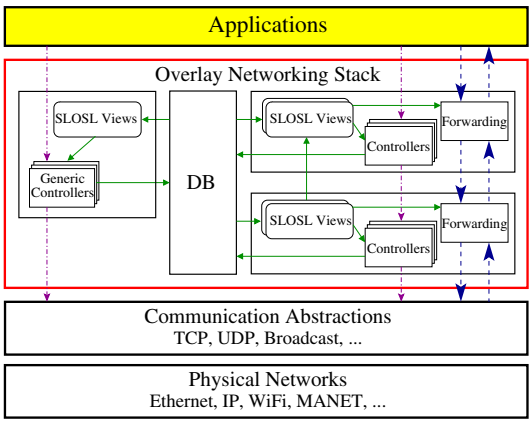


Fig. 6. The OverGrid Architecture

linearly scanned during evaluation. View updates can then be triggered by updates to the node objects. The trigger paths are entirely known through the dependencies of the SLOSL statements, which allows efficient view update code to be generated.

The tradeoff between the complexity of the code generation process and that of static components is up to the specific requirements of the target environment. Gridkit can easily provide different implementations and select between them at deployment time.

3.2 Control

Control components are generated mainly from the SLOSL and EDSL models. The resulting code 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 here), that provide repair and backup strategies for overlays [12]. These can be remodelled using EDSL to make their internal structure visible and subject to adaptation in OverGrid.

3.3 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. 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 state database.

3.4 Event Infrastructure

EDSL describes the component interaction in terms of events. Implementations can follow two possible paths:

EDSM. The implementation uses a generic event-driven state machine implementation. This 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.

Interface code. Implementations can choose to generate glue code for components that directly interconnects them according to the event flow graph. The extreme incarnation is direct interaction of components through their generic interfaces. This requires run time setup code for OpenCOM that connects the correct components. Note that this approach allows for threaded implementations, such as one thread per event path.

3.5 Network Layers

The FROM clause in SLOSL describes a layering of topologies that maps directly to layers in Gridkit as in figure 6. 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 on lower layers.

4 Case Study: Scribe over Chord

This section describes a case study of designing a complex overlay implementation with OverGrid. We specify the overlay using the SLOSL Overlay Workbench and then map 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. Due to space constraints, we only present the design steps in summaries. Note, however, that the support for incremental design is a major advantage of our high-level architecture.

4.1 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 later on 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 another 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 in a state between dead and alive. Another way of specifying this would be a bounded counter.

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

4.2 SLOSL Implemented Chord Topology

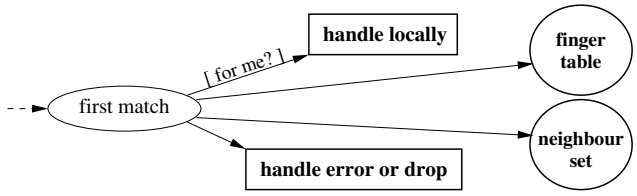
Chord [1] deploys two different views: the *finger table* defines the major characteristics of its topology and the *neighbour set* contains the predecessor and successor along the ring. SLOSL implements the finger table as follows.

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

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

4.3 Chord Routing Through SLOSL Views



The routing decision graph for Chord is shown in the figure. Messages are pushed through the tree from the left. The outcome is either some kind of local

handling (boxes) or message forwarding through views (circles). At the *first match* of a target amongst the sequence of child branches (top-to-bottom order), the execution is terminated and the message is handed to the target. One of the transitions has a predicate associated with it. Its implementation is provided by an external component.

If the target is a view, the message is forwarded via its equivalence classes. The receiver node is locally instantiated as a possibly incomplete node, for which the HAVING–FOREACH and RANKED clauses are evaluated. This yields a number of nodes to which the message is then broadcasted. If the view evaluation fails, the tree traversal continues.

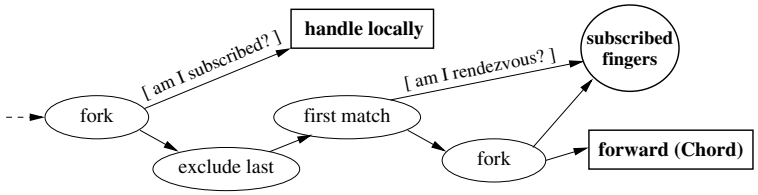
4.4 SLOSL Implemented Scribe on Chord

The Scribe [6] 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.

```
1 CREATE VIEW scribe_subscribed_children
2 AS SELECT node.id
3 FROM chord_fingertable
4 WITH sub
5 WHERE sub in node.subscribed_groups
```

4.5 Multicast Routing Through SLOSL Views



The routing decision graph for Scribe publications on Chord is shown in the figure. Along the path, the execution is forked into different branches, which treat the message independently. The "exclude last" property is a common helper that prevents the last hop of a received message from appearing amongst the selection of next hops further down the tree.

4.6 Message Specifications in HIMDEL

Instead of presenting the XML representation of HIMDEL, we will stick to a shorter form here. The following messages are used in our implementation. The name in brackets is used for accessing fields and structure of messages. Note 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).

- 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]

4.7 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. Due to space constraints, we cannot present the entire protocol graph of Scribe on Chord that implements the IControl component. We must therefore content ourselves with an example event processing cycle that shows how the system responds to events in a non-trivial way. We chose the leave process in Scribe for this purpose, i.e. the propagation of unsubscriptions from groups. The complete process is presented in figure 7. 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.

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 do not require any notifications about updates. Similarly, Scribe can ignore them as long as there are nodes left in all subscription views. Therefore, in the simplest case, event handling ends just here.

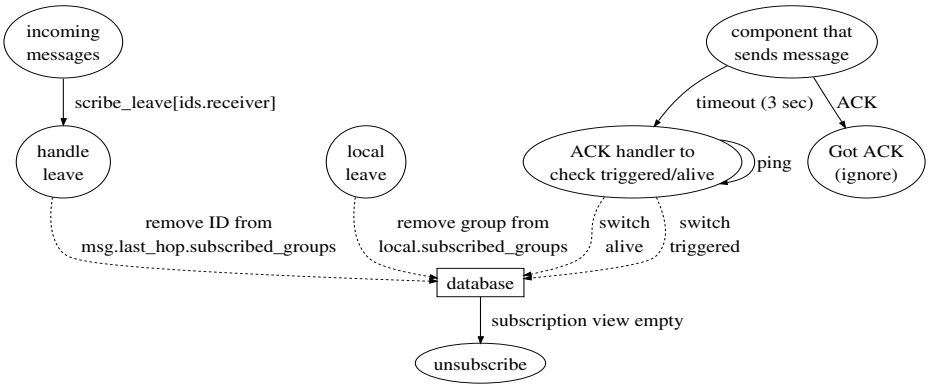


Fig. 7. The event processing cycle for explicit and implicit leaves in Scribe

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* controller in the Scribe implementation. For each event, it sends out an unsubscribe message towards the rendezvous node of the respective group by using the underlying chord router.

4.8 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 in figure 5 allows the developer to play with simple scenarios. Based on the evaluation of SLOSL views against a global database, it supports per-node restrictions on the local view to simulate global differences and inconsistencies in the local state of each node. This helps in finding topological problems in the specification before starting to work on the controllers.

Overlay specifications are then used to generate OpenCOM components that fit the Overlay framework. 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 7 suggests, controllers generally tend to be very simple and small, which allows for pre-implemented 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. Our 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.

5 Related Work

There are middlewares and application toolkits that provide principled support for p2p application development. *JXTA* (`http://www.jxta.org`) is a framework where p2p applications are developed atop a resource search abstraction; this supports grouping and contacting nodes. This abstraction can be implemented using a number of overlay topologies. Hence, like Gridkit the developer must follow the implementation specification.

Furthermore, like OverML there are high-level languages and tools for reducing the development effort of overlay networks. *iOverlay* [13] provides a message switch abstraction for the design of the local routing algorithm. The neighbours of a node are instantiated as local I/O queues between which the user provided implementation switches messages. This simplifies the design of overlay algorithms by hiding the lower networking levels. However, unlike OverML there is no further support for topology rules, maintenance or adaptation.

Macedon [14] is a state machine compiler for overlay protocol design and forms the most interesting approach so far. Event-driven state machines (EDSMs) have been used over decades for protocol design and specification. Macedon extends this approach to an overlay specific, C++ based language from which it generates source code for overlay maintenance and routing. In a number of different proof-of-concept overlay implementations, this was shown to be very useful for implementing and testing algorithms for routing and maintenance.

However, overlays must automatically configure themselves and adapt to a changes; it is not just a matter of routing protocol design. Each node in an overlay needs to take local decisions. The sum of these decisions is the distributed algorithm that maintains the overlay. iOverlay bases these decisions on the currently available connections. It does not provide means for selecting the "right" connections or categorising them, neither does it support ranking connection candidates for adaptation and fall-back mechanisms. Similarly, Macedon does not support candidate nodes or adaptability of topologies. Modelling adaptivity in state machines is likely to be complex and lead to state explosion. Neither framework supports a fine-grained componentisation of the implementation. Consequently, in these incompatible and language dependent frameworks, the designer models local decisions in framework specific source code.

In the P2/Overlog project [15], applications use a declarative logic language to specify their requirements of the overlay network. This is combined with a data flow approach, as opposed to a finite state machine approach, to maintain the overlay at runtime. Like Macedon, this simplifies the development process of overlays in specific cases. Apart from the implementation of overlay protocols, however, P2 offers no support for overlay applications or their deployment.

In this section, we have investigated solutions from both the middleware and overlay development domains. However, as far as we are aware, there has been no work similar to OverGrid that has investigated the combining of the two approaches; whereby the best of both worlds allows the creation of highly tailorable overlays to better support actual deployed applications and middleware services.

6 Conclusions

OverGrid is not yet a ready-to-run overlay suite. It is the main purpose of this paper to outline the possibilities of an integrated specification-to-deployment approach to overlay implementation. The combination of OverML models with the Gridkit infrastructure shows extremely promising results for the simplified design, automated generation and adaptive deployment of overlay applications.

A major advantage of OverGrid is the clean separation into forward, state and control components, that is otherwise hard to achieve by hand. Even more, this separation is done by OverML in a completely platform-independent way. Radical moves to new deployment environments become possible through the reduced dependency on platform-specific source code of overlay implementations. A generic, platform-specific OverGrid implementation can host a broad variety of overlays and topologies. Configuration and adaptation is moved into code generators and deployment- or run-time component architectures.

To back our promises for OverGrid, we have presented specifications of complex overlays. The simplicity of the abstract models that shines from the examples translates into a better understanding of the system and faster development cycles through early edit-compile-run feedback. However, a problem of OverGrid in this early project phase is the lack of tool support. Besides the SLOSL Overlay Workbench prototype there are no simulators or debugging environments that could support developers in their work. However, we expect that these tools can be integrated into the Gridkit/OpenCOM component infrastructure; especially debuggers will benefit from the semantically rich execution environment.

To our knowledge, OverGrid is the first in the area of overlay middleware and frameworks, to tackle the entire design process from the specification in platform-independent models to the adaptable deployment in changing environments.

References

1. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proc. of the 2001 ACM SIGCOMM Conference, San Diego, CA, USA (2001)
2. Loguinov, D., Kumar, A., Rai, V., Ganesh, S.: Graph-theoretic analysis of structured peer-to-peer systems: Routing distances and fault resilience. [16]
3. Chawathe, Y., Ratnasamy, S., Breslau, L., Lanham, N., Shenker, S.: Making gnutella-like p2p systems scalable. [16]
4. Dabek, F., Zhao, B., Druschel, P., Stoica, I.: Towards a common API for structured peer-to-peer overlays. In: Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03), Berkeley, CA, USA (2003)
5. Grace, P., Coulson, G., Blair, G., Porter, B.: Deep middleware for the divergent grid. In: Proc. of the Int. Middleware Conference (Middleware2005), Grenoble, France (2005)
6. Rowstron, A., Kermarrec, A.M., Castro, M., Druschel, P.: 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 (2001)

7. Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., Ueyama, J.: A Component Model for Building Systems Software. In: Proc. of the IASTED Conference on Software Engineering and Applications (SEA'04), Cambridge, MA, USA (2004)
8. Mathy, L., Canonico, R., Hutchinson, D.: An Overlay Tree Building Control Protocol. In: Proc. of the 3rd Int. COST264 Workshop on Networked Group Communication, London, UK (2001) 76–87
9. Ganesh, A.J., Kermarrec, A.M., Massoulie, L.: SCAMP: Peer-to-peer lightweight membership service for large-scale group communication. In: Proc. of the 3rd Int. Workshop on Networked Group Communication, London, UK (2001) 44–55
10. van Renesse, R., Minsky, Y., Hayden, M.: A Gossip-Based Failure Detection Service. In: Proc. of the 1st IFIP International Conference on Middleware, Lake District, UK (1998) 55–70
11. Behnel, S., Buchmann, A.: Models and Languages for Overlay Networks. In: Proc. of the 3rd Int. VLDB Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P 2005), Trondheim, Norway (2005)
12. Porter, B., Coulson, G.: Intelligent Dependability Services for Overlay Networks. In: Proc. of the 6th IFIP WG International Conference on Distributed Applications and Interoperable Systems, Bologna, Italy (2006)
13. Li, B., Guo, J., Wan, M.: iOverlay: A lightweight middleware infrastructure for overlay application implementations. In: Proc. of the Int. Middleware Conference (Middleware2004), Toronto, Canada (2004)
14. Rodriguez, A., Killian, C., Bhat, S., Kostić, D., Vahdat, A.: MACEDON: Methodology for automatically creating, evaluating, and designing overlay networks. In: Proc. of the USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI2004), San Francisco, CA, USA (2004)
15. Loo, B.T., Condie, T., Hellerstein, J.M., Maniatis, P., Roscoe, T., Stoica, I.: Implementing declarative overlays. SIGOPS Operating Systems Review (2005)
16. The 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM). (2003)