Models and Languages for Overlay Networks

Stefan Behnel, Alejandro Buchmann

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

Abstract. Implementing overlay software is non-trivial. In current projects, overlays or frameworks are built on top of low-level networking abstractions. This leaves the implementation of topologies, their maintenance and optimisation strategies, and the routing entirely to the developer. Consequently, topology characteristics are woven deaply into the source code and the tight coupling with low-level frameworks prevents code reuse when other frameworks prove a better match for the evolving requirements.
This paper presents *OverML*, a high-level overlay specification language that is independent of specific frameworks. The underlying system model, named "Node Views", abstracts from low-level issues such as I/O and message handling and instead moves *ranking nodes* and *selecting neighbours* into the heart of the overlay software development process. The abstraction decouples maintenance components in overlay software, considerably reduces their need for framework dependent source code and enables their generic, configurable implementation in pluggable EDSM frameworks.

1 Introduction

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

Contrary to this expectation, current overlay implementations are built with incompatible, language specific frameworks on top of low-level networking abstractions. This complicates their design and prohibits code-reuse in different frameworks. It also hinders the comparison and exchangeability of different topologies within an application.

Our recent work [5] promotes a data abstraction as a much cleaner foundation for the implementation of overlay software. It decouples components for routing and maintenance and enables abstract, framework independent specifications.

This paper presents OverML, a new set of languages that were specifically designed for the framework independent specification and implementation of adaptable overlay networks. Section 2 briefly overviews the underlying data driven system model that is more thoroughly explained and motivated in [5]. The remaining sections then describe OverML, a set of abstract XML specification languages for the major parts of overlay implementations: topology specifications, event flows, messages and node attributes.

2 Node Views, the System Model

We model overlay software as data management systems by applying the well-known Model-View-Controller pattern [6]. It aims to decouple software components by separating the roles for data storage (model), data presentation (views) and data manipulation (controllers).

In our case, the *model* is an active local database on each node, a central storage place for all data that a node knows about remote nodes. The major characteristics of the overlay topology are then defined in *node views* of the database. They represent sets of nodes that are of interest to the local node (such as its neighbours). Different views provide different ways of selecting and categorising nodes, and therefore different ways of adapting topology characteristics to application requirements.

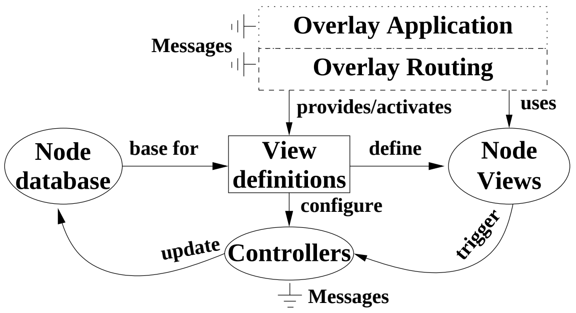

Fig. 1. The system model

The *controllers* are tiny EDSM states that are triggered by events like timeouts, incoming or leaving messages or changes in the views. They perform simple maintenance tasks like updating node attributes when new data becomes available or sending out messages to search new nodes that match the view definitions.

Controllers and other overlay components like message handlers or routers use node views for their decisions. Database and views decouple them from each other and simplify their design considerably. Even more so, as this architecture can provide powerful operations like selecting and adapting topologies with a single view selection command. The abstract view definition becomes the central point of control for the characteristics of the overlay.

3 OverML, the XML Overlay Modelling Language

We propose the XML Overlay Modelling Language *OverML* for specifying the four portable parts of overlay software: node attributes, messages, view definitions and EDSM graphs. Because of space limitations, only the first three are presented here. Schema definitions are provided at `http://www.dvs1.informatik.tu-darmstadt.de/research/OverML/`, which is also the XML namespace that we abbreviated in the examples.

3.1 Slosl, the View Specification Language

The view definitions implement adaptable topologies which makes them the key components in overlay software. They are expressed in Slosl, the SQL-Like Overlay Specification Language. As the XML representation of Slosl is relatively straight forward (using Content MathML [7]), we will stick to the more readable representation. We start with a simple example, an implementation of an extended Chord graph [3].

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

While most clauses behave as in SQL, the new clauses RANKED and HAVING–FOREACH were added to provide simple statements for highly expressive overlay specifications. A more detailed description of the example follows, leaving out the obvious clauses CREATE VIEW and FROM.

SELECT The interface of this view contains the attributes *id* and *ring_dist* of its nodes (ID and distance along the ring), as well as a newly calculated attribute *bucket_dist*.

RANKED To support **topology adaptation**, the nodes in the created view are chosen by the ranking function *lowest* as the $nodes + i$ top node(s) 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 or even user defined functions can be used.

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

WHERE Any SQL boolean expression based on node attributes can be used in this clause. It constrains nodes that are valid candidates for this view (**node selection**). Here we use an attribute *supports_chord* that is true for all nodes that know the Chord protocol. The second attribute, *alive*, is true for nodes that the local node considers alive.

HAVING–FOREACH This pair of clauses aggregates valid candidates into buckets for **node categorisation**. In the example, the HAVING part states that the ID distance 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 can define a single bucket of nodes, but also 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 HAVING expression (which is optional and defaults to true). A node can naturally appear in multiple buckets if the HAVING expression allows it.

The bucket abstraction is enough to implement graphs like Chord, Pastry, Kademlia or de-Bruijn in less than a dozen lines and should be just as useful for a large number of other cases. It is not limited to numbers and ranges, buckets can be defined on any list or even on a database table. Numbers are commonly used in structured overlays (which are based on numeric identifiers), while strings could be used for topic-clustering in unstructured networks.

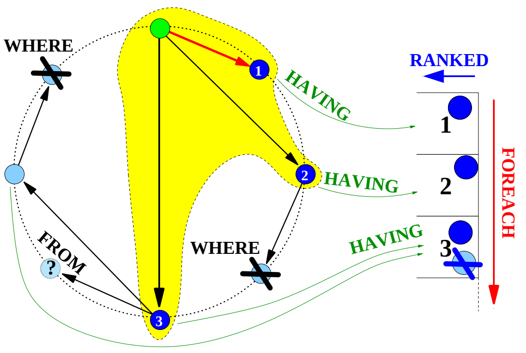

Fig. 2. Implementing the chord topology in SLOSL

The clauses FROM, WHERE, HAVING–FOREACH, RANKED and SELECT directly impact the nodes and attributes in the view. We will therefore explain their interaction semantics in terms of an execution order as illustrated in figure 2. Note, however, that SLOSL is a declarative language and that query optimisers and source code generators may handle specific SLOSL statements quite differently.

1. FROM selects all nodes from the parent view. Note that the database may be globally incomplete or outdated (as the missing node in figure 2 suggests).
2. WHERE restricts this set to nodes matching a boolean expression.
3. FOREACH sorts the selected nodes into buckets by evaluating the HAVING expression for each node and each value of the bucket variable(s).
4. RANKED restricts the maximum number of nodes in each bucket and selects only those nodes for which the ranking expression yields the best results.
5. SELECT finally selects the attributes of each node that are visible in this view. Note that SLOSL inherits SQL's powerful capability of calculating attribute values based on arbitrary SQL expressions. If bucket variables are used in these expressions, the same node can carry different attribute values in different buckets of the created view.

3.2 NALA, the Node Attribute Language

Attribute definitions can currently utilise the data types of XML Schema [9] or SQL [10].[1] For SQL types, OverML allows the definition of custom data types based on the predefined types as follows.

```
<types xmlns:sql="OverML/sql">
  <sql:composite   type_name="tcpaddress" />  <!--composite data type-->
    <sql:inet      name="address" />
    <sql:shortint  name="port" />
  </sql:composite>
  <sql:decimal type_name="id128"  bits="128"/> <!--restricted decimals-->
  <sql:decimal type_name="id256"  bits="256"/>
</types>
```

[1] The 2003 SQL/XML standard [11] defines a mapping between the two.

Line 6 and 7 define customised descendents of the normal **sql:decimal** data type that are restricted to a fixed size to represent node IDs. Note that for specific restrictions, some implementations may not support an exact mapping to storage types and may need to do range checking. The **sql:composite** meta type allows composing multiple simple types into one new structured type.

Any base type or custom type can be used for node attributes. Attributes have a name and a number of flags as shown in the following example.

```
<nala:attributes xmlns:nala="OverML/nala" xmlns:sql="OverML/sql">
  <nala:attribute name="id" type_name="id256" selected="true">
    <nala:static/> <nala:transferable/> <nala:identifier/>
  </nala:attribute>
  <nala:attribute name="knows_chord" type_name="sql:boolean"
                  selected="true">
    <nala:static/> <nala:transferable/>
  </nala:attribute>
  <nala:attribute name="latency" type_name="sql:interval"
                  selected="true" />
</nala:attributes>
```

For easier extensibility, all flags except '*selected*' are represented as XML elements. Their meaning is as follows:

identifier The attribute uniquely identifies a 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 combined into a single new type, which can then be used as identifier.

static The attribute is static and does not change once it is known about a node. All identifiers are implicitly static, but not all static attributes fulfil the uniqueness requirement of an identifier.

transferable The attribute can be sent in messages. Some attributes (like network latency) only make sense locally and should be marked non-transferable.

selected Selects the attribute for use in the database. Unselected attributes can reside in the specification without actually being used during execution. They can be dynamically activated at need, just like SLOSL statements.

3.3 HIMDEL, the Hierarchical Message Description Language

Messages combine attributes and other content into well defined data units for transmission. Their definition follows a hierarchy rooted in the top-level header, followed by a sequence of other headers and finally a sequence of content fields. Being an XML language, OverML presents this hierarchy in a natural way.

```
<msg:message_hierarchy xmlns:msg="OverML/msg" xmlns:sql="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, a message becomes 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 hierarchy of headers that preceed it along 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 channels and to decide the best protocol at runtime.

Multiple independent messages can be defined within the same header. Messages and headers branching away from a message path are completely ignored, i.e. the 'joined_message' in the example is not part of the 'view_message' and the second header is not part of any of them.

This means that the tag order on the message path is 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 preceeding messages will *not* contain the successor fields, as they 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 as children of the **message_hierarchy** tag. Here, 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 their **type_name** attribute.

The Source code interface to messages and their fields is defined using the **access_name** attribute. Accessing the fields of a message from an object oriented language should look like the following Python snippet.

```python
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 following rules define the access paths. They allow for a concise, but nevertheless structured and well defined path to each element.

1. As the basic unit of network traffic, a message is always the top-level element.
2. Everything defined within the message becomes 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 the protocol become second-level elements, referenced by access name. Their child fields and container elements are referenced recursively as before. Children of nameless headers become elements of the message itself.

The message specification allows EDSM states to be triggered by framework-independent subscriptions to message names or header hierarchies. A simple subset of the XPath language [8] lends itself for defining these subscriptions. Note that even expensive abbreviations like '`//`' can be resolved at compile time or deployment time using the message specifications.

The network serialisation of messages depends on framework and language, whereas the specification above does not. There is a huge number of network representations for messages that are in more or less wide-spread use. In any case, the specified message hierarchy can directly be mapped to an XML serialisation format. But also in flat serialisations like XDR [12], mapping the message specification is straight forward when laying out the data fields depth-first along the message path. Depending on the attribute '*structured*', Views are serialised either as bucket structure (XML subtrees or XDR arrays) or as plain node data. The latter can avoid duplicate data if nodes appear unchanged in multiple buckets.

4 Conclusion, Current and Future Work

This paper presented OverML, a language for abstract overlay specification. Based on a Model-View-Controller architecture, it provides portable, framework-independent abstractions for the major components in overlay software. The achieved modularisation facilitates the development of generic components which enables pluggable development and integration of overlay systems.

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

OverML and SLOSL make the design of the main characteristics of overlay software simple, fast, and independent of languages and frameworks. Our prototype implementation comprises a graphical editor for OverML specifications as well as a proof-of-concept runtime environment. Their combination moves the development of the remaining framework specific software components to the very end of the design process and supports it with a powerful and generic high-level API.

Future work will include better mechanisms for view and query optimisation. Our current PostgreSQL implementation maps SLOSL statements to rather complex, generic SQL queries. Building on the large body of literature on query modification and optimisation, we can imagine a number of ways to investigate for pre-optimising these statements. This is most interesting for recursive views and for merging view definitions when sending them over the wire.

We believe that high-level, integrative overlay design is an interesting new field that builds upon major achievements in the areas of databases, networking and software engineering. We would be glad to seed interest in new implementations of OverML compatible frameworks, SLOSL optimisers, source code generators, as well as possible mappings to existing frameworks.

References

1. Aberer, K.: P-Grid: A Self-Organizing access structure for P2P information systems. In: Proc. of the Sixth Int. Conference on Cooperative Information Systems (CoopIS 2001), Trento, Italy. (2001)
2. Loguinov, D., Kumar, A., Rai, V., Ganesh, S.: Graph-theoretic analysis of structured peer-to-peer systems: Routing distances and fault resilience. [13]
3. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proc. of the 2001 ACM SIGCOMM Conference, San Diego, California, USA (2001)
4. Chawathe, Y., Ratnasamy, S., Breslau, L., Lanham, N., Shenker, S.: Making gnutella-like p2p systems scalable. [13]
5. Behnel, S., Buchmann, A.: Overlay networks - implementation by specification. In: Proc. of the Int. Middleware Conference (Middleware2005), Grenoble, France (2005)
6. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons (1996)
7. The World Wide Web Consortium: Mathematical Markup Language (MathML) Version 2.0 (Second Edition). (2003)
8. The World Wide Web Consortium: XML Path Language (XPath) Version 1.0. (1999)
9. The World Wide Web Consortium: XML Schema Part 2: Datatypes Second Edition. (2004)
10. ISO Document ISO/IEC 9075:2003: Database Language SQL. (2003)
11. ISO Document ISO/IEC 9075:14-2003: Database Language SQL - Part 14: XML-Related Specifications (SQL/XML). (2003)
12. Srinivasan, R.: XDR: External Data Representation Standard. RFC 1832 (Draft Standard) (1995)
13. The 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM). (2003)