Vrije Universiteit Brussel

Faculty of Science
Department of Computer Science
and Applied Computer Science

Identification and Distribution of Cross-Media Entities in a Peer-to-Peer Environment

Graduation thesis submitted in partial fulfillment of the
requirements for the degree of Master in Computer Science

Barry d'Hoine

Promoter: Prof. Dr. Beat Signer
Advisors: Bruno Dumas

June 2014

Abstract

The goal of this thesis is to design and implement a general distributed cross-media server. The cross-media environment contains entities that need to be identified and distributed in a peer-to-peer environment.

Before providing the design and the implementation of the distributed cross-media server, the state of the art to design distributed cross-media systems is explored.

The cross-media server is designed using a hypermedia model. The features and properties of existing hypermedia models are compared and then evaluated. The hypermedia model that is used for the cross-media server is a domain independent and open model.

To support data distribution, existing distribution protocols are explored that can be used in a hypermedia environment. The distribution protocol focusses on content distribution. The distribution is done by using a distributed hash table.

A mechanism for identifying data in the distributed cross-media server is also needed. Different identification mechanisms are compared and then evaluated. A suitable identification mechanism for a distributed cross-media server is selected.

Using the features and properties of the models and protocols explored, the requirements of the distributed cross-media server are identified. The contribution of this thesis is that a design is provided for the distributed cross-media server in a domain independent way that is also extendable. After the design phase, the distributed cross-media server is implemented.

Samenvatting

Het doel van deze thesis is om een generieke gedistribueerde cross-media server te ontwerpen en te implementeren. De cross-media omgeving bevat entiteiten dat geïdentificeerd moeten worden in een gedistribueerd peer-to-peer omgeving.

Voor er een ontwerp en een implementatie gegeven wordt zullen bestaande oplossingen om een gedistribueerde cross-media server te ontwerpen verkend worden.

De cross-media server is ontworpen door gebruik te maken van een hypermedia model. De functies en eigenschappen van bestaande hypermedia modellen worden vergeleken en vervolgens geëvalueerd. Het hypermedia model dat gebruikt wordt voor de cross-media server is een domein onafhankelijk en open model.

Om data distributie te ondersteunen worden bestaande distributie protocollen dat in een hypermedia omgeving kunnen gebruikt worden verkend. Het distributie protocol zal verantwoordelijk zijn voor de distributie van de inhoud. Het distributie protocol maakt gebruik van een gedistribueerde hash tabel.

Een mechanisme om data te identificeren in een gedistribueerde cross-media server is ook onderzocht. Verschillende identificatie mechanismen worden vergeleken en vervolgens geëvalueerd. Na de vergelijking en de evaluatie wordt een gepast identificatie mechanisme dat kan gebruikt worden in een gedistribueerde cross-media server gekozen.

Door gebruik te maken van de functies en eigenschappen van de verkende modellen en protocollen worden de vereisten van de cross-media server bepaald. De bijdrage van deze thesis is dat een domein onafhankelijk en uitbreidbaar ontwerp word gegeven voor een gedistribueerde cross-media server. Na de ontwerp fase is de gedistribueerde cross-media server geïmplementeerd.

Acknowledgments

This thesis could not be written without the support and help of various people. In this section I want to express my gratitude to these people.

First of all I want to thank the VUB for giving me the opportunity to write this thesis and in particular the members of the WISE research group. A special thank word goes out to Bruno Dumas, my supervisor during the whole process. He provided fresh insights, many helping hands and a good guidance. I also want to thank Prof. Dr. Beat Signer to promote this thesis.

I also want to thank my family for all the support they gave me to come to this point in my life. In particular I want to thank my parents for giving me the opportunity and motivation to study.

Last but not least, I want to thank my friends and girlfriend for standing by me whenever I needed them.

Barry d'Hoine
Mechelen, 2014

"In the middle of difficulty lies opportunity" - Albert Einstein

Contents

1 Introduction

This thesis will describe how one can design and implement a distributed cross-media server. A distributed cross-media server is a hypermedia system that is distributed across several heterogenous nodes. The desirable features of a hypermedia system will be explored and a discussion about the impact on the design of the distributed cross-media server will be pointed out. Taking into consideration the design of the targeted distributed cross-media server, a study about identification features and the distribution of data will be provided.

The distribution of the system is done by using nodes that cooperate. Every node should have the same functionality. No central authority should be needed to make the nodes cooperate. Every node has equal functionality but not every node should be identical. Nodes can differ in some node specific characteristics like memory, CPU, bandwidth, etc. The distributed system is deployed as an overlay network of nodes and should not be visible to the end user.

While designing a distributed system one needs to take into account scalability, availability and load balancing. The base model for the cross-media server is the *Resource-Selector-Link (RSL)* model that is described in [44]. The *RSL* model was designed as a general meta-model for object-oriented hypermedia systems that enables cross-media linking. To achieve the cross-media linking, concepts of hypermedia systems are applied [43].

To implement a cross-media server, an exploration of the state of the art is needed. The main challenges for the design are choosing the identification mechanism and defining the algorithm for the distribution, with respect to a distributed hypermedia system and without changing the main properties of the targeted distributed cross-media server. While designing the system, there is also the need to preserve all the core features of the used *RSL* model without losing the generality and openness of the model.

1.1 Structure

After the introduction chapter, the second chapter explores the state of the art for designing a distributed cross-media server. In the first section of the state of the art a full coverage of hypermedia systems is given followed by a section about distributed systems. A small section about load balancing concludes the first chapter.

In the hypermedia systems section, previous efforts to design a hypermedia system are covered. We also explore why many new hypermedia systems are designed based on existing hypermedia systems and why some of the systems were not further developed. Then, the motivation for designing the *RSL* model is given and the *RSL* model itself is described briefly.

In the distributed systems section a short overview of distributed systems, with respect to hypermedia systems, is given. Finally, prefix and key-based routing mechanisms are covered.

After the state of the art is explored, the design choices for the distributed cross-media are summed up in the third chapter. An overview is given about identification of data. Existing identification mechanisms that can be used for our distributed cross-media server are discussed. The identification of data, in the context of the *RSL* model, is also discussed.

The fourth chapter gives all the details regarding the implementation of the distributed cross-media server. First, in the introduction, an overview of the existing implementations of *RSL* are discussed. Afterwards, existing technologies and frameworks are selected to speed up the implementation. The details about the actual implementation, test for the implementation and some simulations are described afterwards.

Finally, in the fifth chapter a conclusion of the thesis is given, together with some remarks about the distributed cross-media server and the implementation in general. The future work is pointed out in the last section of the fifth chapter.

1.2 Focus

The focus throughout the thesis is on distributed hypermedia systems that are open and domain independent. When designing such systems we need to identify what the wished features of such systems are and what the consequences are when we are combining existing state of the art solutions. Another problem that needs to be

solved is how identification and distribution of data should be done in distributed hypermedia systems. The research question of this thesis is: ”How should we design a distributed cross-media server and how can data be identified in such a system”.

1.3 Example of Cross-Media Data

An example that can help to have a mental picture of how to model linked data in a cross-media environment is the concept of a news site that contains news articles. A news article has many properties, functional as well as structural, that can be used as an example of data that is stored in the distributed cross-media server. A news article for example has a logical document structure with a title, some paragraphs and some references. Links to related news articles can be embedded in the news article as well as some images and videos. Another property of a news article is that it can be commented by other external users.

2 State of the Art

2.1 Hypermedia Systems

The first section will provide a definition of a hypermedia system. A hypermedia system will be used as a framework to enable cross-media linking in the distributed cross-media server implementation. In the second section, an overview of historical and still existing hypermedia systems is given. The third section will introduce the *Dexter* model, the first effort towards a general and open hypermedia system. The fourth section will evaluate and compare all the previously explored hypermedia systems.

In the last section, the most important features and properties of the *RSL* model, a new effort towards a general and open hypermedia system using meta modelling principles, will be explored. A full coverage about the details of the *RSL* model can be found in [44]. Later in the thesis, more details will follow about *RSL* that are relevant for the distributed cross-media server implementation.

2.1.1 Definition

Due to the lack of a generally accepted definition of what a hypermedia system is, we define it, based on the definition given in [1], as a system with the following features:

- The information space of a hypermedia system is built up using several small units of information, usually called cards, frames, nodes, etc.
- Some of the information units are interconnected by the concept of links. Traversing the information from unit to unit is possible by following links.
- Information structures are built up from the creation, editing and linking of information units.

- For hypermedia systems it should also be possible to access, edit, create or delete information in a collaborative way. We call these kind of systems shared hypermedia systems and they can be designed as a distributed system.

2.1.2 History

When designing a hypermedia system one must first have a good idea what a hypermedia system is and where the concept of a hypermedia system came from. A hypermedia system is the generalisation of the concepts of hypertext with features like audio, video, graphics, etc. [1]

Hypermedia systems are accessed in a non-linear way, like newspapers or phone books. This is the key point where a hypermedia system is distinguished from a multimedia system.

The first signs of the concept of hypertext, which led to the research and development of hypermedia systems, was described by Vannevar Bush [7] by introducing the concept of the *memex* in 1945. The *memex* is a hypothetical system that can be used to browse linked data. The *memex* uses the concept of trails to link data from different scientific libraries. The ability to create notes for the linked data is also described for the *memex*.

A first effort towards the generalisation of the hypertext concept was done by Ted Nelson. In [34], hypertext is defined by Ted Nelson as a linked, written or pictorial entity in such a complex way that it is not convenient to easily present or represent it as a physical document. The entity itself could contain some other entities like annotations, additions or footnotes from other authors. The global hypertext system should gradually grow in time to contain more and more data from the current knowledge that is written down in documents.

A survey of Jeff Conklin [12], that was made much later than the work of Ted Nelson, states that a hypertext system should have support for two types of links, *within* and *between* document links. Conklin also states that there should be objects in a database and links between those objects should exist on the conceptual and the database level. Conklin claims that the concept of hypertext is rather simple but that it can have a huge impact on the use of plain text documents, commonly used those days.

As computer systems evolved to more complex systems, the hypertext concept was not general enough to cover the concept of linked data any more. The newer systems started to have support for a larger variety of data types such as video,

graphics, etc. The natural extension of the hypertext concept to support linking of the newer data types was called hypermedia and should also be interpreted as a non-linear system.

In the next subsections a few historical hypertext and hypermedia systems will be briefly discussed. They all had some contribution to the current state of hypermedia systems. To conclude this section an overview and comparison of the discussed systems is given.

2.1.2.1 oN-Line System

After the introduction of the *memex* of Vannevar Bush, it took a while until a framework was designed and implemented that is related to the hypermedia concept. In 1963, 18 years after the introduction of the *memex*, Douglas Engelbart designed a framework [17] that is referenced as the *oN-Line System (NLS)* or *Augment*.

Five years later the *NLS* system was implemented and demonstrated at the Fall Joint Computer Conference in San Francisco. The demo is these days called "The Mother of all Demos" because it demonstrated most of the fundamental concepts of modern personal computing, including hypertext.

NLS was designed to have an interaction between a human user and a computer system. A human user could for example introduce a file into the system. The new file is divided in different segments by the system. Every file as well as every segment, which is called a statement, has an identifier. The identifiers can be used to create some reference links between both files and statements, and thus also parts of other files. The system supported both hierarchical and non-hierarchical links.

The *NLS* system provided a database to store information, a mechanism to select some parts of the stored information and a view of the targeted information. Another important concept of the *NLS* system was that it provided a mechanism to edit or view information in a collaborative way. Unlike most of the early implementations of hypermedia systems, the *NLS* system could be distributed on the early Arpanet, one of the first package switched networks using TCP/IP.

2.1.2.2 Project Xanadu

In parallel with the work of Douglas Engelbart, Ted Nelson, another hypertext visionary, was designing another hypertext system. The core concept of the system,

the comparison of side-by-side interconnected documents, was first described in 1960. After some years of research on the new system, it was only in 1967 that the system was called *Project Xanadu* [34].

Some believe that *Project Xanadu* was designed as an attempt to create the World Wide Web but the goal of the project was far more ambitious [35]. The goal of *Project Xanadu* was to create a system where linking of documents is not broken throughout different versions and where side-by-side comparison and navigation is possible. Also the support for annotations in built into the system. The annotations and information about those annotations, can be accessed by the *Project Xanadu* system. Another important requirement in the design of the system is that copyright protection is also taken into account.

The backend of the *Project Xanadu* system was eventually implemented after a few years. The focus of the project was mostly on the back-end. During the design phase, a strong separation between the front-end and the back-end was made, this was due to the fact that the ambition of *Project Xanadu* was to store and link all the literary works of the whole world [12].

2.1.2.3 Knowledge Management System

Another important system throughout the history of distributed hypermedia systems is *Knowledge Management System (KMS)* [1]. *KMS* was implemented and the implementation is based on the *ZOG* system that was developed in 1972 at Carnegie-Mellon University. *KMS* was developed as a collaborative large-scale hypermedia system based on a conceptual data model. The implementation of *KMS* was released in 1983 and is actually the commercial version of *ZOG*. The implementation was accompanied with a graphical interface.

In the *KMS* system the data set that needs to be stored and linked is divided into different frames. Every frame consists of a title. Below the title a description of the topic of the frame is found, and a set of numbered menu items of text called selections are defined. To provide a linking mechanism between frames, selections are used. A user can select an item by calling the corresponding menu item number and the selected frame will appear on the screen. In *ZOG* and *KMS* it is only possible to view one frame at the time.

The conclusion of the developers of *ZOG* and *KMS* is that the underlying data model has a serious impact on the visualisation of the model. They experienced that changes in the data model reflected on changes in the visualisation of the model.

2.1.2.4 Intermedia

At Brown university, where a lot of research is done on hypertext, the *Intermedia* [51] system was designed. It is built on the know-how of 20 years of research and three other previously designed systems. One of those system is the *Hypertext Editing System (HES)* [9] that was designed by Ted Nelson and Andries van Dam in 1968. Another system that was used as basis for the design is the *File Retrieval and Editing System (FRESS)* [48] that was implemented in 1969.

Using all the information of hypertext systems that were previously designed and implemented, a new attempt was made to create a collaborative system to manage linked data. Intermedia was designed as both an authoring and reading tool without making a distinction of specific types of users. The system was implemented as a multiuser environment.

Not only the back-end was designed carefully to provide a tool for linked data to use on a large scale, also a lot of attention was given on the visualisation of the linked data and browsing through the large data space. A mechanism was introduced to filter parts of the linked data. On the visualisation level a mechanism was introduced to support context-dependent visualisations called webs. Two different visualisations, a global map and a local map, can be used to browse the linked data. The global map targets an entire linked data object and has a within data navigation mechanism. The local map provides a view of a single data object together with the closest links in the hyperspace within the selected webs.

2.1.2.5 NoteCards

The *NoteCards* [22] system was developed at *Xerox PARC* in 1987. The initial goal of the *NoteCards* system was to design a tool for information analysis. The designers of the *NoteCards* system found out that analysts followed a recurring methodology to produce analytic reports. The *NoteCards* system should help analysts to make a better analysis of existing collection of ideas.

In the *NoteCards* system, electronic notecards are interconnected by using typed links. The system was designed for users to represent collections of related ideas. The system should also help users to create a structured and searchable information space.

Because of the open architecture it is possible to build applications written in Lisp on top of the *NoteCards* system. In this way a customised browser can be created for the notecards. Another way to extend the *NoteCards* system is to create

new types of nodes that can be represented as an information chunk in a digital notecards.

2.1.2.6 Overview

In Fig. 2.1 a timeline of all the previously discussed hypermedia systems is shown. The first item on the timeline is the introduction of the *memex* concept. About two decades later the first generation hypermedia systems were released and followed each other up in the next 10 years. After the release of the first generation hypermedia systems, a gap of about a decade is seen between the release of second generation hypermedia systems. The hypermedia systems on the timeline are considered state of the art systems that had an enormous impact on similar systems that were released during the same period or later.

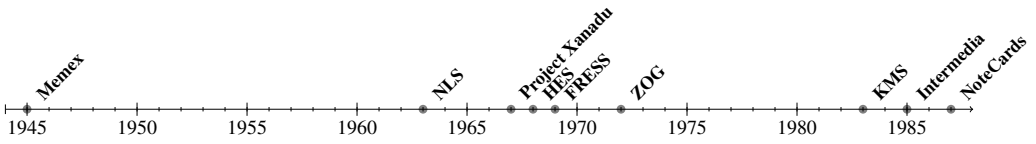


Fig. 2.1: Timeline of hypermedia systems

An overview of some important features of the discussed hypermedia systems are summarised in Table 2.1. The first column, the *system* column, shows which hypermedia system we are considering. In the next column, the *hierarchy* column, the support for hierarchical structures is given. The *graph-based (graph)* column defines the support for non-hierarchical links. The *attributes (attr.)* column shows if the system has support to add user-defined attribute/value pairs to a node or link. The support for typing of links is found in the column *link type*. If the system supports more than one version for a node or link it can be found in the *versions* column. In the last column, the *concurrency (concur.)* column, the support for users to concurrently edit or access data is shown. In [12] a broader overview is given of the history of hypermedia systems.

Table 2.1 shows that every historical hypermedia systems focussed on certain features and lacked at least one of the other features. No system leaves room to support all of the features that can be used in a hypermedia system. This makes it particularly hard to compare them by using a general paradigm.

System	Hierarchy	Graph	Attr.	Link type	Versions	Concur.
NLS/Augment	Yes	Yes	Yes	Yes	Yes	Yes
Project Xanadu	No	Yes	Yes	Yes	Yes	No
ZOG	Yes	No	No	No	No	Yes
KMS	Multiple	Yes	No	Fixed	Yes	Yes
Intermedia	Yes	Yes	Yes	Yes	No	Yes
NoteCards	Multiple	Yes	Nodes	Yes	No	Yes

Table 2.1: Comparison of the features of historical hypermedia systems

2.1.3 Dexter Reference Model

The first effort towards the design of a general and open hypermedia system was found in the *Dexter* model [20] that was published in 1990. The design of the *Dexter* reference model is a part of the *DeVise Hypermedia System* project. The intent of the *Dexter* model was to provide a reference model that is used as a common basis for several hypermedia systems. A standard terminology is introduced and the minimal functionality is described for the model. Furthermore, a basis is provided to work towards open hypermedia systems that can exchange information. The openness was introduced to compare different hypermedia systems. It is hard to compare for example, *NLS* with *KMS*, because the first system supports an arbitrary length for documents and the second system has a fixed frame size.

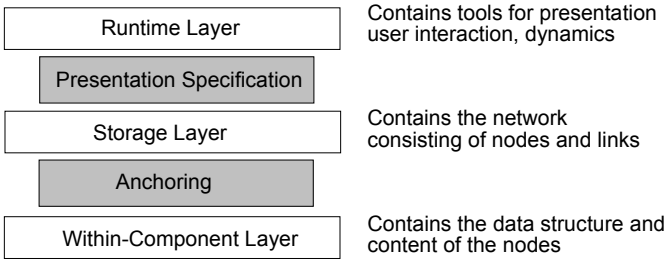


Fig. 2.2: Dexter reference model

The *Dexter* model is defined by three layers, the within-component layer, the storage layer and the runtime layer. The within-component layer contains the data structure and the content of the nodes. The storage layer defines the network of nodes and links. The runtime layer provides the presentation of the components towards the user. Hypermedia objects that are stored in the storage layer are called components.

In the *Dexter* model, two interfaces are defined, the presentation specification and

the anchoring interface. The anchoring mechanism is responsible for the linking. The presentation specification defines how the components are presented in the runtime layer. A visual representation of the *Dexter* model can be found in Fig. 2.2.

The *Dexter* reference model was a good first effort but it also has some issues, as described in [18] where some of the limitations of the model and design issues are pointed out. The main issue in the *Dexter* model is the fact that the data outside the components could not be used inside the hypermedia system itself. There is also no support for extensibility in the model. The mechanism of anchors is not well defined, they introduce a conceptual weakness for the model itself. The *Dexter* model defines a strict separation of the three layers but this separation is broken by the anchors.

2.1.4 Evaluation

Because both the *NLS* and the *Project Xanadu* systems were developed in times where the targeted hypermedia system was actually a hypertext system, some limitations and domain specific constraints are embedded in the core of the systems themselves. This makes it particular hard to have support for newer hypermedia features.

The *KMS* implementation lacks a well-defined general data model even though the focus was on prividing a solid data model. Although they wanted to have a simple data model because otherwise the system would become too complex, they made some restrictions in the core data model. When the data model needed to change to support new features, it also had an impact on the visualisation and this made it hard to have an evolutionary system that should be up and running on a large scale for a long period.

For the *Intermedia* system, many features were designed and implemented but a feature that is lacking is the support for versioning. Another drawback of the *Intermedia* system is that the links themselves are not part of the data, so links cannot be embedded into a new object.

Some issues of the *NoteCards* system are described in [21]. An important issue is the lack of a hypermedia construction. Only nodes in the form of cards and links are supported by the system. Another issue is that the system can be accessed in a concurrent manner but is was designed as a single-user system without the support of collaboration.

The *Dexter* model, that can be seen as the start of third generation hypermedia

systems, was a good first effort towards a general and open hypermedia system, but as stated before some issues are inside the core of the model itself.

Although there were many efforts after the introduction of the *Dexter* model to design a solid model for hypermedia systems, there is still the need for a better model. The *Adaptive Hypermedia Application Model (AHAM)* [13] for example added the lacking support of a user abstraction to the *Dexter* model. The *AHAM* system enables adaption based on a user model that is persistent throughout the system.

Another step toward a general and open hypermedia system is found at the introduction of the *Fundamental Open Hypertext Model (FOHM)* [32]. Because hypermedia systems were evolving toward more open and interoperable systems, the focus shifted toward component based frameworks. A problem with the existing component based systems is that most of the components of those systems are incompatible between other systems. The *FOHM* was introduced after investigating the common features between all the domain specific components. A common data model was introduced together with some domain-independent operations.

An important feature to come to an open hypermedia server is to separate resources from links on the conceptual level. When links can be managed on their own abstraction level it will make the system more flexible and enables the support of new link features. Those features are for example bidirectional links, multi-source and multi-target links. One of the historical hypermedia systems that uses the concept of external links is the *Hyper-G* system [2].

The main problem of many of the hypermedia systems is that they are designed for a specific domain. Another problem is that some other hypermedia systems include technical details in the core of the model and lose in that way their generality and uniformity. Another fact is that there is a strong need for a mechanism to support extensibility when designing hypermedia systems. The user abstraction and data ownership concept is also mostly absent inside the core of the hypermedia model.

There are a variety of hypermedia models proposed, implemented and even still in development. Some of those are extensions of older hypermedia systems and others try to differentiate from the existing hypermedia systems. There are a lot of abandoned as well as running projects regarding the design and implementation of hypermedia systems. This is a strong indication that there still is no solid reference model that can be used as a base for a hypermedia system.

2.1.5 RSL Model

All the problems described in the evaluation of the previous section are the main driving force to introduce a new model that can be used for hypermedia systems. A new approach was used to design a model that is open and general. This model is the *RSL* model and is explained in the following sections.

The *RSL* model is a link model for hypermedia systems. The model is actually a metamodel inspired by metamodelling concepts of the database community. The model of *RSL* is defined using the semantic, object-oriented data model OM [36]. This allows us to use it as an operational model for data management and for the design of the target system.

Because *RSL* is a well-defined conceptual metamodel, it leaves room for generality and flexibility. This is not only true for the model but also for the targeted system. The core of the *RSL* model tries to be as general as possible. The model provides a plug-in mechanism to support extensibility and this leaves room to build domain specific systems using *RSL* in the core.

2.1.5.1 Core

The core concept of the *RSL* model are the entities. An entity of the *RSL* model is an abstract concept and defines a partition of three subtypes: links, selectors and resources. The core link metamodel can be found in Fig. 2.3.

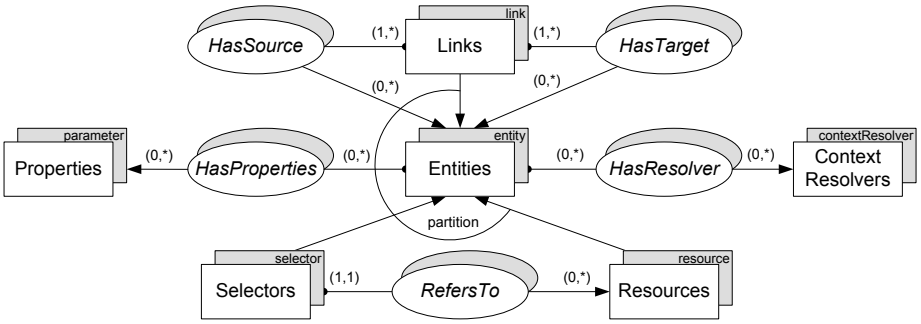


Fig. 2.3: Core link metamodel of RSL

A resource is also an abstract concept. All the media types handled by a hypermedia system are defined as an extension of an resource by using the plug-in mechanism. To address specific parts of resources that can be linked, selectors are introduced. A selector is another abstract concept and should be designed for

every type of resource. To support the linking of data, the abstract concept of links are introduced. A link is always between entities and is directed from one or more sources to one or more targets. Two types of links are supported, navigational and structural links.

To keep the core of the *RSL* model general and flexible, different properties can be associated with entities. Context-dependent handling of entities is another mechanism that is supported inside the core of *RSL*.

2.1.5.2 User Abstraction

Another important concept that should be included for hypermedia systems is a user abstraction. This enables sharing, collaboration and data ownership. In *RSL*, the user model provides data ownership for every entity. Every user is a part of a partition of individuals and groups. A group contains users but can also be empty. Some entities are defined to be accessible or inaccessible to certain users. This enables collaborative authoring of links and resources. An entity is always owned by an individual who created the entity. The model for the user abstraction can be found in Fig. 2.4.

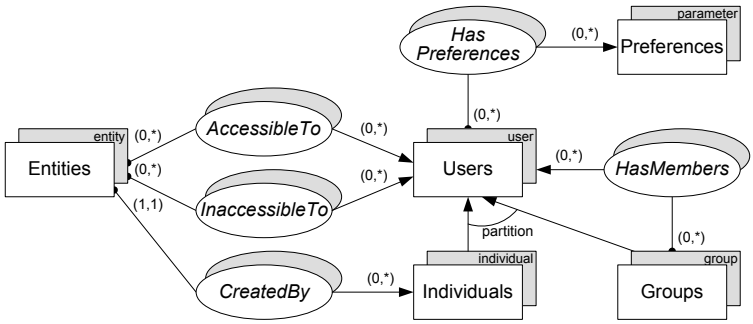


Fig. 2.4: User abstraction in RSL core

2.1.5.3 Layers

Due to the fact that the *RSL* model has selectors over resources and that links can be nested, problems can arise when a link over a resource is to be activated. There should also be support for a link resolution mechanism. That mechanism is provided by the concept of layers in the *RSL* model. The model for the *RSL* layers is shown in Fig. 2.5.

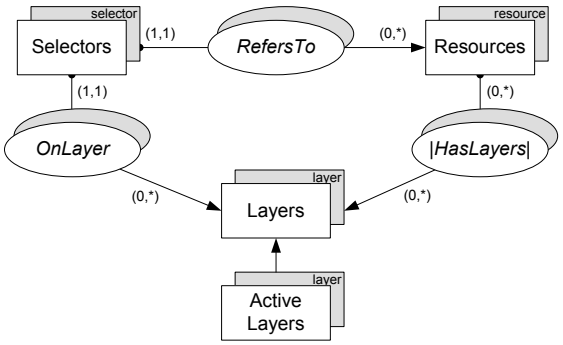


Fig. 2.5: Layer model for RSL

To support the link resolution mechanism, each selector in on exactly one layer. No overlapping selectors can be on the same layer. When several links are associated with an entity, the link that will be handled is the selector that is on the top layer. This is possible because layers maintain an order. As you can see in Fig. 2.5, a resource consists of a totally ordered sequence of layers. Some of the layers can be active, meaning that it is also possible to deactivate some layers as well. Note that reordering of layers can take place even at runtime.

2.2 Distributed Systems

2.2.1 Overview

When taking a look at the state of the art in distributed computing and in particular, systems labelled as peer-to-peer systems, a broad overview is given in [3]. Most of the concepts and architectures found in this section are based on that survey. The label ”peer-to-peer system”, as they use in the survey, can actually be dropped because we only look at fully decentralised, structured, distributed systems.

We can divide distributed systems in three categories based on the following use cases: communication and collaboration, distributed computation and content distribution. The focus for the distributed cross-media server is on the content distribution type of systems because these kind of distributed system are best suited for the core features of a hypermedia system. In the survey, a framework is provided to analyse different content distribution architectures.

The survey studies existing distribution mechanisms in terms of distributed object location, routing mechanisms, replication, caching, migration, encryption, access

control, authentication, identity, anonymity, deniability, accountability, reputation and resource management. Some nonfunctional requirements are also investigated. Characteristics such as security, scalability, performance, fairness and resource management are considered.

The field of distributed systems is rapidly evolving and has gained a lot of attention by researchers. Many new architectures are proposed and researched but many core concepts are maintained and stay relevant for many architectures.

Using a distributed architecture is a good design pattern when designing systems that can and will grow. Distributed systems tend to scale well. They also provide the possibility to have a self organising architecture in a highly transient node population. All these properties are not hard requirements but they are natural properties that should be found in a distributed architecture.

All the nodes in the system should be heterogenous in terms of functionality. There are also distributed systems that take other approaches but these are not discussed here. Another requirement of the distributed architecture for our cross-media server is that it should be fully decentralised. The main reason for this is that there are no bottlenecks or single points of failure introduced when using a fully decentralised system.

Because the linked data should be handled by the *RSL* model, the focus of the distribution is in the routing of requests and locating data in the distributed system. In distributed systems that focus on routing and location, the global system is built up as an overlay network of nodes. Between the nodes in the network, messages and requests are routed in an efficient manner to locate content.

If we look at the state of the art of these systems we find two major infrastructures that use routing and location-based architectures. All these systems implement a Distributed Hash Table (DHT). The first is a prefix-based routing mechanism that is used in *Chord* [46]. The second is a key-based routing mechanism, based on the XOR metric as found in *Kademlia* [30].

2.2.2 Prefix-based Routing

Prefix based routing is found in systems like *Pastry* [40]. As described by *Pastry*, the identifiers of nodes and keys are considered as a sequence of digits with a base 2^m bits. The routing is done by passing messages to the nodes that are numerically closer to closer to the key that should be found. While passing messages, the routing algorithm tries to forward a message to a node that has a prefix that is longer than the prefix of the current node and the key that should be found. If there

is no such node it tries to forward it to an equally long prefix that is numerically closer. This routing mechanism is only possible when every node maintains some routing information.

The *Chord* protocol, which is similar to *Pastry*, is a very simple protocol that only provides one operation, i.e. mapping a key to a node. The protocol does not describe a put or get command, which is common in distributed systems for storing and locating data. Obviously these commands can be built easily on top of the *Chord* protocol. *Chord* is designed to adapt to nodes joining and leaving the system and the system should stay operational in a continuously changing environment.

A variant of consistent hashing, like described in [27], is used in *Chord*. No restriction is made on the type of the consistent hashing function. A standard consistent hashing function like SHA-1 can be used as well. The consistent hashing function is used to assign keys to nodes. This mechanism provides a simple and basic load balancing mechanism because each node will get roughly the same number of keys assigned. When a node joins or leaves the system only a small fraction of the keys will be moved.

In the algorithm, a *Chord* node needs information about $O(logN)$ nodes for efficient routing, where N is the total number of nodes. Lookups are handled in $O(logN)$ messages. A join or a leave command will result in a production of $O(log^2N)$ messages. *Chord* has guaranteed but slow success rate for node lookups when node information is in an inconsistent state. The guarantee can be made as long as a every node has at least one piece of information. In *Chord*, an algorithm is defined for keeping the routing information of the nodes in a consistent state when the system changes over time.

Chord was designed for systems that need load balancing, decentralisation, scalability and availability. The *Chord* protocol does not provide any means of authentication, caching or replication. These functions are left to the upper layer that uses the *Chord* protocol.

The *Chord* protocol was further developed and this led to a new protocol, *Koorde* [26]. *Koorde* is based on a combination of the *Chord* protocol and properties of the Bruijn graphs [14]. The *Koorde* protocol has lower bounds for lookup requests and should store less routing information. In *Koorde* for example, when a node has two neighbours it will find a key in $O(logN)$ steps. The problem with this protocol is that it does not perform well in a highly transient node population. In hypermedia systems it is often the case that nodes leave and join the system constantly while others remain longer in the system. For this reason *Koorde* is not explored deeper.

2.2.2.1 Chord Protocol

In the *Chord* protocol, every node is assigned an identifier by hashing its IP address. All the assigned identifiers are ordered in an identifier circle. Every node will have keys assigned to it, starting from its own identifier until the identifier of his successor. The example in Fig. 2.6 shows an identifier circle that has 10 nodes. There are 5 keys stored inside the identifier circle. The examples for the *Chord* protocol are taken from [46].

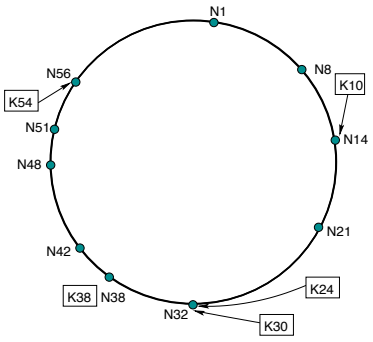
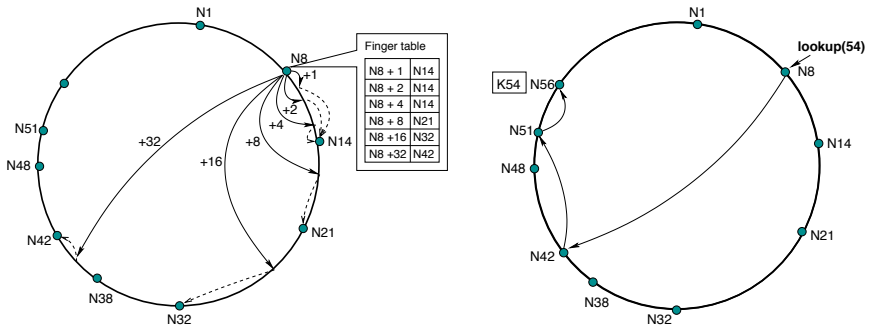


Fig. 2.6: Identifier circle in Chord

To speed up node lookups and make the protocol more scalable, nodes in *Chord* will store additional routing information. The routing information will be stored in a finger table. If the identifier space consists of m bit identifiers, then every node will maintain a finger table with at most m entries. Entries in the finger table will contain *Chord* identifiers and the endpoint of the node. An example of a finger table for a node is shown in Fig. 2.7(a). In Fig. 2.7(b) the lookup path is shown using the finger table routing information.

To ensure that a lookup can take place in $O(logN)$ steps, the routing information needs to be refreshed when the system changes over time. There are three possibilities when a key is looked up:

- The routing information is relatively fresh and the lookup takes $O(logN)$ steps.
- The successor pointers are correct but the entries in the finger table are inaccurate. Lookups will still be correct but slower.
- The successor pointers are incorrect or keys are not yet transferred to the correct nodes. In this case the lookup can fail and it is left to the upper layer using *Chord* to take actions.



(a) Chord finger table for node 8 (b) Lookup path on node 8 for key 54

Fig. 2.7: Finger table and lookup path in Chord

To ensure that lookups will succeed with a high probability in $O(logN)$ steps, the stabilisation algorithm was introduced in *Chord*. It is resistant when nodes simultaneously join the system and when messages are lost in transmission. The stabilisation algorithm will be run periodically on every node. The node asks his successors for their predecessors. The node then has to decides if the retrieved predecessors should actually be successors of the node itself. It will also announce his own presence in the network every time the stabilisation algorithm is executed. More details about the design and performance of the *Chord* protocol can be found in [46].

2.2.3 Key-Based Routing Using the XOR Metric

In *Kademlia*, which is a key-based routing protocol that uses the XOR metric, DHT features are combined that are not commonly combined in other protocols. Nodes learn about the system automatically when messages travel through the system because messages always originate from nodes in the routing table of a receiving node and thus can exchange useful information in any message. The messages travel in a parallel and asynchronous way to avoid timeouts and large delays. Messages can be passed trough low-latency paths using the routing information stored inside the nodes.

Like in most DHT implementations, *Kademlia* also uses an identifier space and every node and data has a key that will map to a corresponding node or the data on the node. The nodes will store data assigned to a key that is close to its own key into the identifier space. The notion of closeness will be determined by the XOR metric. The node identifier will be used in the routing algorithm to locate

nodes that contain specific keys.

The *Kademlia* protocol uses only one algorithm for all the routing and is thus less complex than other DHT implementations like *Pastry* [40]. *Pastry* is also a well-know key-based routing protocol but it uses two algorithms to lookup a key. A problem with using these two algorithms is that the metrics used in both algorithms are not similar. This degrades the performance and makes the protocol more complex.

2.2.3.1 XOR Metric

The XOR metric is used as a measure of distance between two points in the key space. If we talk about closeness this will be reflected in the distance defined by the XOR metric in this section. The XOR metric is symmetric and is the origin of the learning ability of the system as messages pass trough the system. This is because *Kademlia* can route information to any node within a specified interval. *Chord* for example is not symmetric, so it cannot learn useful information while routing messages.

In *Kademlia* we define the distance d between two identifiers, x and y, as the bitwise exclusive or (XOR) interpreted as an integer, $d(x, y) = x \oplus y$. You can see that the XOR topology is symmetric because $\forall x, y : d(x, y) = d(y, x)$.

The XOR metric is also unidirectional. This means that for a given point x in the key space there is a distance $\Delta > 0$ and exactly one point y, such that $d(x, y) = \Delta$. The unidirectionality implies that all the lookups for a key from any node converge along the same path. This means that key lookups can be cached along those paths.

2.2.3.2 Kademlia Protocol

In the *Kademlia* protocol, every node is considered as a leaf of a binary tree. An internal node in the binary tree determines a node with a specific prefix. For a node with some prefix, the binary tree is divided into several subtrees not containing the node and thus having a different prefix. This partitioning is used in the routing algorithm. Every *Kademlia* node will have information about at least one node in its subtrees in the partition.

For every prefix with length i, a list of triples $\langle$IP address, UDP port, Node ID$\rangle$ is stored for nodes at a distance between 2^i and 2^{i+1}. These lists are called k-buckets.

Those lists can contain maximum k triples, k can be interpreted as the system-wide replication parameter. Nodes are assigned to an appropriate k-bucket by using the shortest unique prefix of their identifier. By using this mechanism, several disjoint paths are spanned between the overlay network of nodes.

The k-buckets are sorted by time a node was last seen. Least recently seen nodes are located at the top of a k-bucket. The least recently seen nodes remain, as long as they are alive, longer in the k-buckets than newer nodes. This makes it hard to flood the routing tables that are stored in the nodes. In this manner, the replacement policy provides protection against some types of Denial-of-Service (DoS) attacks. Every time a node receives a message, it updates the corresponding k-bucket.

In Fig. 2.8 the location of a node with prefix 0011 in the binary tree is shown as a black dot. The partitioning where the node with prefix 0011 needs contacts is also shown. In Fig. 2.9 the lookup for a key is explained. The node with prefix 0011 will lookup a node with prefix 1110 by contacting a node that is closer, according to the XOR metric, in the identifier space. The lookup will always converge to the correct node because of the partitioning and the routing information present in the k-buckets. The examples for the *Kademlia* protocol are taken from [30].

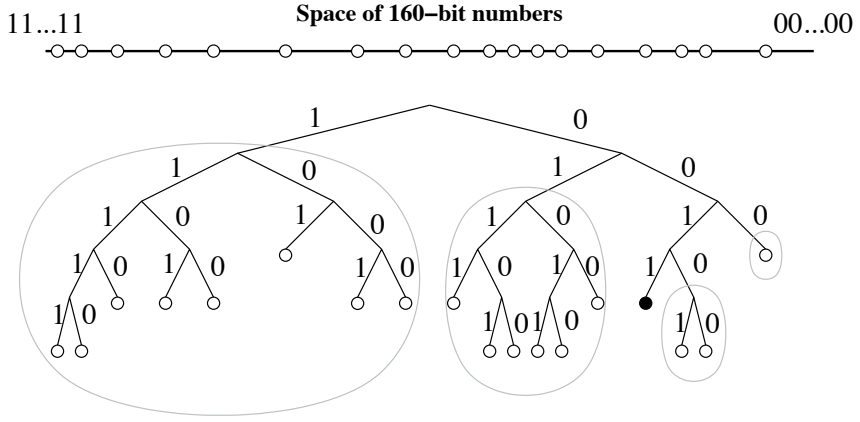


Fig. 2.8: Partitioning of a binary tree in Kademlia

The motivation for the replacement policy for the routing information is based on analysis of the Gnutella network in [19]. A measurement study of peer-to-peer file sharing systems was done and those results can be used for hypermedia systems as well. They conclude for example that nodes that are alive for more than one hour will stay alive for a long time with a very high probability.

The *Kademlia* protocol is built by using four commands: $ping$, $store$, $findNode$

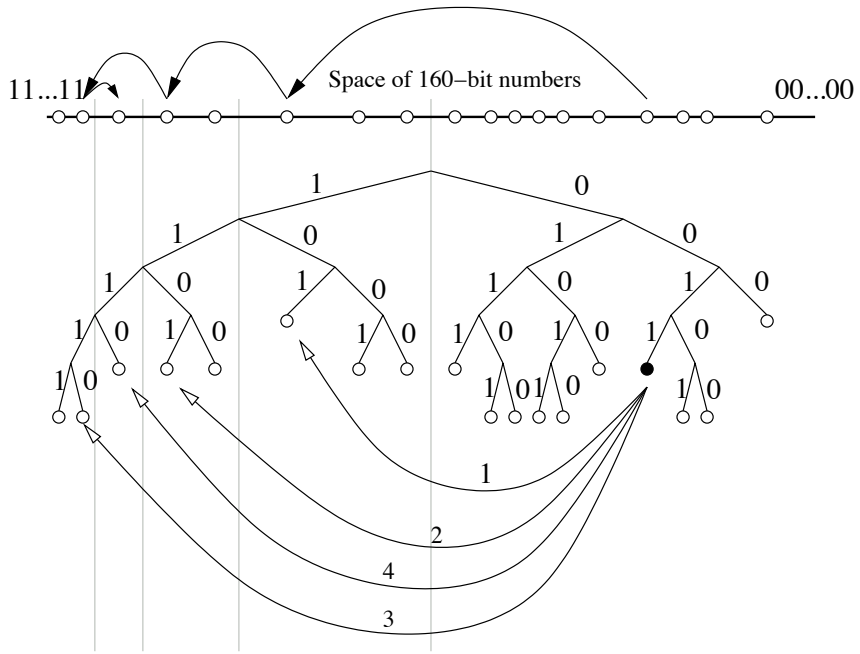


Fig. 2.9: Looking up a node with prefix 1111 in Kademlia

and $findValue$. The $ping$ command checks if a node is still online. The $store$ command will tell a node to store a ⟨key, value⟩ pair. The $findNode$ command will return ⟨IP address, UDP port, Node ID⟩ triples for a specific key for the k closest nodes it is aware of. The $findValue$ command will return exactly the same as $findNode$, unless it has received a $store$ command, then it returns the stored value for the given key.

A recursive algorithm is used to locate the k closest nodes. The initiator of the lookup will send α parallel, asynchronous $findNode$ requests to the α closest nodes it is aware of. In the recursive step the initiator will resend the $findNode$ command to nodes it learned about in the previous steps. If all the α lookups fail the request will be sent to the k closest nodes that were not already contacted. The parameter α is the concurrency parameter that allows a tradeoff between bandwidth, lowest-latency paths and delay-free fault recovery. If $\alpha = 1$ the *Kademlia* algorithm resembles the algorithm of the *Chord* protocol.

To store a ⟨key, value⟩ pair, the $store$ command is sent to the k closest nodes using the key that needs to be stored. Every node also republishes the stored ⟨key, value⟩ pairs to keep them alive. Every β hour the original publisher of a ⟨key, value⟩ pair needs to republish the pair. After β hours, ⟨key, value⟩ pairs that are not republished are removed. This is done to keep information in the caches up to

date and that no unnecessary information is stored for a long period. β is typically 24 as in [30] but it can tweaked for specific domains.

When a node needs to lookup a key, it will look for the k closest nodes of the key by using the $findValue$ command. When a value is returned the lookup will end. If a lookup succeeds, the requesting node will send a $store$ command to the closest node it saw that does not already contain the value. This is done for caching purposes. More details, optimisations and implementation details about the *Kademlia* protocol can be found in [30].

2.2.4 Secure Key-Based Routing

The *Chord* and *Kademlia* protocol are two of the many protocols that implement a DHT. The problem with most DHT implementations is the lack of security measurements. Most of the protocols trust all nodes in the overlay network. Many attacks use this assumption or they abuse the algorithm that is responsible for the routing. The most important security problems for DHT implementations are identified and described in [45] and [10].

As stated in the summary of [30], the *Kademlia* key-based routing protocol has provable consistency and performance, latency-minimising routing and a symmetric unidirectional topology. If we consider these properties and the fact that *Kademlia* provides some parameters that can be tweaked for domain specific systems, we conclude that it is a very promising protocol that can be used for the distributed cross-media server but it is still not perfect.

Most of the issues that arise for completely decentralised distributed systems are security concerns. In [5], a practical approach toward secure key-based routing is given for the *Kademlia* protocol. By using the problems identified in [45] and [10] a more secure version of the *Kademlia* protocol is proposed. Most of the known problems have a theoretical solution but the protocol that is proposed in [5] provides concrete solutions. The attacks that are covered are attacks both on the key-based routing mechanism and the storage layer. The secure key-based routing protocol is called *S/Kademlia*.

2.2.4.1 Type of Attacks

First of all, no assumptions are made regarding the security measures for the underlying network. This means that there are several attacks possible on the underlay network. These attacks will lead to a DoS for the overlay network. Attacks

on the underlay network are possible by using IP address spoofing, lack of data authentication, packet sniffing and packet modification. To authenticate nodes and to ensure the integrity of messages traveling inside the network, a cryptographic or supervised signature can be used.

On the overlay network on the other hand, several attacks are also possible. We distinguish the following type of attacks.

- Eclipse attacks: Adversarial nodes are placed inside the overlay network in such a way that messages pass through at least one adversarial node. In this way the attacker can gain control over a specific part of the overlay network.
- Sybil attacks: An attacker tries to insert many nodes inside the overlay network until he controls a part of the overlay. The difference with the eclipse attack is that the routing tables are not the driving force of the attack but the number of nodes inserted into the network.
- Churn attacks: An attacker will try to destabilise the overlay by continuously and simultaneously letting nodes join and leave the network. The amount of nodes joining an leaving the overlay is called the degree of churn inside the network.
- Adversarial routing: Instead of abusing the routing algorithm, adversarial routing will let a node return adversarial routing information. By passing wrong routing information, the structure of the overlay network stored inside the routing tables will not be correct.

If we disallow nodes to freely choose their identifier, it will be harder to perform eclipse attacks. As investigated and proven in [16], it is not possible to completely prevent sybil attacks but it is possible to make them hard to perform. The problem is that in completely decentralised systems, it is hard to limit the number of nodes that can join the network. Only system resources can be used as a measure to prevent many micro nodes joining the network to perform sybil attacks. The impact of a churn attack is minimised due to the replacement policy in the k-buckets in the *Kademlia* protocol. Adversarial routing depends on the concurrency factor α and the impact will be inversely proportional to α. The algorithm can, depending on α, detect adversarial routing if the network still has a portion of nodes that can be trusted.

Another important attack is the DoS attack where an adversarial node tries to consume all the resources of another node. The secure protocol should provide a way to allocate resources avoiding that all resources of a node will be used and thus lead to a DoS. Inside the DHT $\langle$key, value$\rangle$ pairs will be stored and it should not be possible, or at least very difficult, for adversarial nodes to modify the stored pairs.

This can be achieved by replicating the stored pairs in a secure neighborhood. The secure node neighborhood should be provided by the protocol.

2.3 Load Balancing

2.3.1 Introduction

The protocols that were covered for the distribution are all implementations of a DHT. Those algorithms typically tend to spread the load but they will introduce a $O(logN)$ imbalance. Furthermore, the distributed system we target can contain many heterogeneous nodes. This will lead to a further imbalance of the system. These statements were identified in [39]. Three schemes are provided that will balance the load from 80% to 95% compared to the optimal value, in structured distributed systems that use a DHT abstraction. In [39] they also state that their proposed load balancing schemes are orthogonal and complementary to caching mechanisms.

Other load balancing algorithms are already proposed, like in *Chord* or [8], but all those load balancing algorithms make assumptions that the system is static and that node keys are uniformly distributed. In [8] they claim that their algorithm should also work in dynamic systems but their algorithm is designed for a static environment and no proof if given for a dynamic environment.

The core concept of the load balancing mechanism is the existence of virtual servers. For the DHT protocol, a virtual server looks like a node but every physical node can contain multiple virtual servers. The storage and routing will happen at the virtual server level. By introducing the virtual servers and balancing the load among them, the path length for lookups will also increase. Load balancing is important when you target a scalable and responsive system. The path length for lookups can be shortened by letting the routing tables contain more information as stated in [30]. This makes it possible to make systems targeted to a certain bound on the path length by aligning the size of the routing information with the load balancing algorithm.

Three assumptions were made when the load balancing schemes were designed:

- The type of the node resource that is balanced is not restricted but can be only one. This means that one can balance the storage or the bandwidth for example, but not both.
- There is only load moved from heavily loaded nodes to lightly loaded nodes.

- The load on a virtual server is stable or the load of a virtual server can be predicted while the load balancing algorithm is active.

Taking into consideration the assumptions listed before, three schemes are proposed to achieve load balancing:

- One-to-one scheme: A light node will periodically contact random nodes. If the contacted node is heavy then the virtual servers of the heavy node are transferred to the contacting node without making that node heavy.
- One-to-many scheme: A heavy node will contact light nodes and will transfer some of its virtual servers to the lightest nodes.
- Many-to-many scheme: Information about light and heavy nodes is collected and maintained in different directories inside the system. An algorithm then runs on the different directories to transfer load from heavy nodes to light nodes. This is a solution that will run in a centralised way for all the different directories.

2.3.2 Load Balancing in Dynamic Systems

As stated in the future work in [39], load balancing should also be possible in dynamic systems. In [33], a load balancing algorithm is proposed for dynamic structured distributed systems. The load balancing algorithm proposed is negligibly worse than a centralised algorithm. The dynamic version of the algorithm is more suitable for distributed systems that possibly implement a hypermedia system.

As stated before most of the distributed systems that use a DHT abstraction tend to have an $O(logN)$ imbalance. The imbalance implied by the DHT is not the only imbalance that can be introduced in the overlay network. A non uniform distribution of the objects in the identifier space will further increase the imbalance.

Further, no assumptions should be made regarding the load of a node. Systems can be highly dynamic and will have some degree of churn. Another thing to notice is that there can be a skewed arrival pattern for objects. All the previous properties lead to a further imbalance of the system. The heterogeneity of nodes on the other hand, will make the proposed algorithm more scalable and will contribute to the solution provided.

The dynamic load balancing algorithm targets structured distributed systems that should have the following properties:

- A dynamic storage is used by continuously inserting and deleting data.
- A notable degree of churn is present.
- Keys for identifying data can be skewed.

The load balancing algorithms state that they need to transfer virtual servers inside the DHT but this can be done in two ways. The first is that the keys of objects are changed. This is not a good approach because the keys are also used for lookups. So the second and better option is to change the keys of the virtual servers that act as the nodes inside the DHT. This can be achieved by using the leave and join operations. More details about the algorithm, that uses a combination of the one-to-many and many-to-many schemes defined in [39], can be found in [33]. Note that this is in contrast with the security measurement that nodes should not be able to freely choose their identifier.

Both the static and dynamic load balancing scheme can only balance one type of resource. It would be better if we could balance multiple resources. As stated in [25], multiple resource balancing is a multidimensional-vector packing problem. That problem is described in [11] and is proven to be NP hard. Another fact is that sometimes balancing multiple types of resources can lead to contradictions. In [25] a solution is provided to achieve multiple resource load balancing for cloud cache systems. The findings in [25] can contribute to find a multiple resource load balancing algorithm for a dynamic structured distributed system.

3 Distributed Cross-Media Server

3.1 Requirements

Based on the requirements of previously designed distributed hypermedia systems [41], a summary of all the key features of the distributed cross-media server are pointed out in this section. The targeted distributed cross-media server should provide a way to support the following concepts:

- **Links**: The key feature of the cross-media server is to support the concept of linking data.
- **Scalability**: No limits should be implied by the choice of the architecture on the scale of the system. The system should remain to operate well as the dataset grows. A *DHT* will be used to provide a distribution of the data. Distributed systems tend to scale well. The size the of nodes and the data that can be stored on the nodes should also not be restricted. The nodes will be placed in an overlay network that will make all the data of the system accessible.
- **Persistency**: The data that is stored in the *DHT* should be stored permanently throughout the lifetime of the data. The persistency of the data should be handled by the core of the system. When data needs to be stored the concept of a put command will be used. When retrieving data a get command will be used.
- **Extensibility**: The design of the system should allow a way to extend the supported hypermedia types. In this way the system will be future proof. This is important if we want to keep using the system when new media types need to be supported.
- **Openness**: The system should be able to communicate with any other hypermedia system by providing a standardised way to interpret the data. To

achieve this we need to design the hypermedia system in a domain independent way.

- **Availability**: When data is stored in the system, every node or user that has access to that data should be able to locate it. Because we are dealing with a dynamic system the availability of the data is not trivial. The system should be stable enough when it is running to locate data. In dynamic systems, mechanisms to handle requests than can fail when the system changes should be provided.
- **Performance**: When the system scales it should alway remain performant and responsive.
- **Concurrency**: Data should be stored in a multi-user context. This also means that concurrent access of data should be supported.
- **Balancing**: When the system grows, one way to achieve a performant system with high availability is to balance the load of the system. This does not need to be done by the *DHT* only, there needs to be support to apply load balancing schemes.
- **Versioning**: Objects that contain data should be able to support different versions. While changing an object to another version, the linked items should be retained. The versioning should be done on the data as well as the structural entities.
- **Ownership**: Because we are operating in a concurrent, multi-user context, data ownership should be supported in the core of the system. This is done by providing a user model that allows sharing of data.
- **Portability**: The system should be portable in such a way that the design does not make any assumptions for the nodes that will store the data. An overlay network of heterogenous nodes should be supported and every used algorithm should take this into account.
- **Security**: Security measurements should also be provided but these must not all occur in the core. As stated before, some attacks can occur in distributed systems and most of the attacks can be solved or made harder.

3.2 Object Identity

If we want to store objects in a database, the object-oriented database system manifesto [4] states that identity needs to be supported. As stated before the

RSL model is defined using a semantic, object-oriented data model. In an object-oriented system every existing object needs the ability to be identified regardless of the attributes and properties of the object itself. This means that there are two different questions one needs to ask about objects:

- How are we going to identify objects? If we want to compare objects we need a way to determine if they are identical. This means that we are checking if we are really talking about the same objects or that the objects are equal.
- How can we share objects? This is a direct consequence of the identification question. If you are sharing object information you also need to support the updating of objects by only manipulating the object itself.

When we choose an identifier it needs to be a singular reference and naming for an object. We also need to take into consideration that an identifier will never be reused and is created and deleted together with the creation and deletion of the object [29]. If we specify an identifier we also need to specify the semantics of the identifier regarding versioning, distribution and location information. This does not imply that these semantics need to be integrated into the identifier but one must leave room on the implementation part to support it in later phases.

The location information for an identifier is shifting away from a physical location. In decentralised distributed systems that support replication there needs to be support for location-independent routing [24]. Replication is used to increase availability, durability and locality. Queries will be routed using location information that is not bound to a physical server.

3.3 Object Identification

3.3.1 Identification Concepts

In the literature the terms object identifier (oid), keys and surrogates are frequently used. It is important to clear out the conceptual and practical differences between them. In some cases they are just referred to as an identifier in general. Different types of identifiers in the context of object-oriented databases are described in [50]. In the following sections the findings of [50] will serve as a lead and will be explored further in the context of decentralised distributed systems.

3.3.1.1 Object Identifiers

As stated before, when we design a system that will store objects, object identity needs to be supported. When we store objects these objects will always be abstractions on a certain level, that are represented in a conceptual model. We need a proper name for an object to distinguish every object that is introduced. The proper name should be unique. Proper names should not change when the state of the objects is changed. This leads to two assumptions:

- Each object will have a unique identity that differs from the identity of any other object, now, in the past and in the future.
- The identity of an object always remains the same, independent of the state of the object.

The two assumptions above are not always satisfied because not all objects are discrete. When some objects are merged identity cannot be preserved but it is not practical to introduce these kind of objects inside a database. An example of this is if you have two piles of sand and you put them together you get a new pile of sand. Mostly the two assumptions will be satisfied and when designing objects one must take into consideration to meet the assumptions.

To identify objects, object identities need to be simulated by using a globally unique proper name. These proper names are called object identifiers (oids). Every object can have only one oid and the oid is preserved for that object only. Oids are found in the field of conceptual modelling and are technically speaking very hard to implement because every implementation is only a subset of all possible data without the need to prevent overlaps. Even if there should not be an overlap, no overview is possible for all past, present and future systems. A constraint that one must keep in mind is that when using oids, only a finite amount of objects can be represented. Oids have the single purpose to differentiate between objects and allow persistence over time.

Objects do not come naturally with an oid. The mechanism that will take care of assigning oids to objects is called an oid distributor. An oid distributor should be a unique instance but the responsibility of the oid distributor can be divided and delegated as is needed in distributed systems. However the fraction of the oids assigned should be divided, the different delegates together form the oid distributor. When an oid distributor assigns an oid to an object it should also ensure that no oid is already assigned to it.

A problem with oids and other identification mechanisms is that it is impossible to ensure authentication in such systems. When an object contains some un-

changeable properties, a link can be made between these properties and the oid will provide some form of authentication. The problem with these unchangeable properties is that it also assumes in some way that objects are introduced with an already existing oid. Thus solving the authentication problem leads to a contradiction regarding oids.

3.3.1.2 Keys

If we talk about databases, the concept of oid's is too general even when it is a necessity to design a good database. Oid's are found more frequently in the field of modelling than in the database field or as an implementation concept. A key consists of one or more attributes of an object that should be unique when they are combined, when considering objects that are relevant to it. Also note that a key is a database concept that is information carrying and oids do not carry any form of information. A key can be updatable but this can be prohibited by the definition of the key inside the database scope. A key should only be unique within the database concept unlike an oid which should be globally unique, thus also between different databases.

3.3.1.3 Surrogates

Surrogates are also identifiers but they are always assigned by the system on object creation. The surrogate is kept invisible towards the user and is maintained by the system. When an object is updated, a surrogate stays unchanged. All the surrogates assigned should be unique across all possible states of the database. A comparison of oids, keys and surrogates can be found in Table 3.1.

oid's	**keys**	**surrogates**
no information	contains information	no information
not updatable	updatable	not updatable
globally unique	unique in single db state	unique in all db states
visible to user	visible to user	invisible to user
assigned by oid distributor	can be assigned by user	assigned by db

Table 3.1: Comparison of oids, keys and surrogates

3.3.2 Identifiers for Decentralised Distributed Systems

Now that different types of conceptual identifiers are defined for objects, we can start looking for a suitable identifier that can be used in a decentralised structured distributed system. Keys and surrogates are less suitable for decentralised systems. It is more natural to mimic the concept of oids in decentralised systems. Also note that the oid distributor will be composed of many different delegates. This is because we target a fully decentralised distributed system. If there is a single oid distributor a centralised authority is introduced. Some mechanisms have already been proposed to identify objects or resources and are investigated further in a distributed context.

3.3.2.1 Uniform Resource Identifiers

In [6], Uniform Resource Identifiers (URI) are defined. URIs are introduced to provide a simple and extensible way to identify a resource by means of a compact sequence of characters. The notion of a resource can be taken in the most general way. A resource is anything, abstract or physical, that can be identified by a URI. A resource identifier does not imply accessibility. Uniformity is introduced to be able to use different types of URIs in the same context. It allows embedding semantics in different types of resource identifiers.

The notion of identifier for a URI is used to identify something with the scope of identification. It is thus a definition of a general concept that will serve in a specific identification context. The identification mechanism will be used to distinguish a resource from any other resource. The identity of a resource is not revealed by a URI but it can be. The URI specification does not define identifiers to identify the same resource over time but most of the URI schemes do provide persistency over time for a URI. A URI can be interpreted in a user defined context but the interpretation of the URI should always be consistent regardless of the context.

A URI also does not make a restriction to singular identification, it can also be used to identify sets and provide mappings. If we consider other identifiers, it should also be possible to come up with a construction for composed or mapped identification. The identifiers of the objects that are singular references are then used for a new resource that is a compound of the other resources. If an object is then updated the compound object will be updated also if we model object identity as described in the previous sections.

Every URI starts with a scheme name that will define the scope of the identifiers

that are assigned. For each scheme, syntactic constructions and restrictions can be defined that do not collide with the general URI syntax. A general syntax is defined so that every URI can be parsed in a scheme independent way. The handling of the scheme specific URI syntax should be handled on the layer of the scheme itself. If a URI is used in some specification it will be able to support different URI schemes because of the general syntax but scheme dependent handling should be provided as new schemes arise. Some examples of URIs are given in Table 3.2.

`http://www.ietf.org/rfc/rfc2396.txt`
`ldap://[2001:db8::7]/c=GB?objectClass?one`
`news:comp.infosystems.www.server.unix`
`tel:+32-3-827-12-11`
`urn:oasis:names:specifications:docbook:dtd:xml:4.1.2`

Table 3.2: Examples of URIs

URIs can also be divided into two subclasses based on their uses. A URI can be used as a locator, a name or both. When we want to identify and locate a resource we will talk about Uniform Resource Locators (URL). The other subclass of URIs consists of Uniform Resource Names (URN). These type of URIs use the urn scheme. URNs are used to identify a globally unique and persistent resource. Note that the class of URIs must not be bound to a specific scheme. Different URIs in a specific scheme can be used to locate, name or both locate and name a resource.

A URI is built from a specific set of characters. These character are the letters of the Latin alphabet, digits and some limited set of special characters. How a URI is interpreted only depends on the used characters. The main reason URIs are built using characters is because a URI could be constructed by different kind of systems and thus also by using a keyboard for example. In a URI, one can embed useful information to make the URI more readable and learnable. Even though a limited set of characters is used, other characters can be encoded into a URI by using percent-encoded octets, if the scheme allows it. More about the details on the syntax can be found in [6].

We are targeting a system in which the objects are distributed and where an identifier is location independent so only the subclass of the URNs will be useful for the identification of the objects. The URN mechanism can be used by the routing algorithm of the DHT algorithm to locate an object. Some parts of the URI specification should not be used. An example is the use of fragments by using the '#' character. This has a historical use to refer to specific parts of a resource.

However this should be handled by the *RSL* selectors. When using URIs, some specific encodings can be used that are platform dependent. Every node that will use URIs will have to take this platform dependency into account.

3.3.2.2 Object Identifiers

In the class of the URN scheme for URIs a special namespace is proposed for object identifiers. We will make the division between the scheme and the concept by respectively using the uppercase abbreviation OID for the scheme and the lowercase abbreviation oid for the concept. As stated before an oid is a general modelling concept. The OIDs defined here are more of an implementation concept. In [31] an OID is defined as a tree of nodes where each node is a sequence of digits and entities are assigned to a node in the tree. The representation is a sequence of characters that is limited to digits and a dot. A dot is used to separate between subtrees. No leading zeros should be used in the digit sequences. The most significant digit is on the left side and the least significant on the right side. Some examples of OIDs are given in Table 3.3. The scheme name `urn` and the `oid` namespace are defined case insensitive.

`urn:oid:1.2`
`urn:oid:1.2.62.1`
`urn:oid:0.9.2323.192232.100`

Table 3.3: Examples of OIDs using URIs

In [31] is stated that the assignment of OIDs should be unique for every resource and that OIDs cannot be reused. This uniqueness is parallel to the conceptual oids. There are standards defined for how OIDs should be assigned and some OIDs are reserved. Once an entity receives and OID it can assign OIDs in its own subtree for that particular OID. For example if we register *RSL* as an ISO assigned OID we can assign `urn:oid:1.28` as OID. If we then want to create *RSL* entities, every entity will have that prefix in its own OID. An entity for *RSL* will for example have an OID like `urn:oid:1.28.0.19873`. Thus once we have received a registered node for our use, we are free to assign any node to an object in our subtree.

It is possible to embed some semantics in OIDs but it is not recommended. Every collection of objects in *RSL* for example can represent a subtree. There are many possibilities when regarding OIDs but one thing that should certainly not be done is embed location information in an OID. Objects can be on different locations or the physical location can change on certain points in time.

3.3.2.3 Digital Object Identifiers

Another effort in providing a system to identify objects is described in [38]. The objects are identified by using Digital Object Identifiers (DOI). Not only DOIs but a whole DOI system to create and work with DOIs is defined. A DOI is a location-independent name for an entity in a digital network. DOIs are used to identify objects, allow persistence of objects and provide means to exchange information inside digital networks. DOIs are managed by the International DOI Foundation and are accepted since 2012 as the ISO 26324:2012[1] standard. DOIs are in general used for combining persistent identification with current information about objects.

The DOI system consists of four components: A specified numbering syntax, a resolution service, a data model and implementation procedures. The DOI system allows reusing existing numbering standards and metadata schemes within the DOI system. The DOI system itself is built by combining several existing standards. The combination of the standards is further developed to provide a consistent system.

The first component in the DOI system is the syntax. A DOI is represented as an opaque string and contains a unique naming authority and a delegation part. The delegation part allows embedding existing identifiers. The naming authority occurs first in the string followed by a '/' character. After the '/' character the delegation part follows. Both parts are also referred to as the DOI prefix and the DOI suffix. Some examples of DOIs are given in Table 3.4.

`doi:10.1234/NP8348`
`doi:10.5678/ISBN-0-7623-2323-5`
`doi:10.2224/2004-10-ISO-DOI`

Table 3.4: Examples of DOIs

The second component in the DOI system is the DOI resolution mechanism. In the resolution component a DOI identifier will be used to request the information that is bound to the identifier. A resolution can return multiple results. The resolution component is based on the Handle System [47]. The Handle System is a secure name resolution and administration service. A client can use a DOI identifier and send it to the DOI system. The DOI system then returns the URI, XML, Data, etc., that is associated with the DOI to the requesting client.

[1] http://www.iso.org/iso/catalogue_detail.htm?csnumber=43506

The third component in the DOI system is the DOI data model. In this component a data dictionary is provided as well as a framework to apply the dictionary to the data. The dictionary contains entries to describe relations between DOIs. The data model was introduced to support interoperability and reusability inside the model. The *interoperability data dictionary* provides the relations between different systems. The reusability is driven by ontology-based dictionaries.

The fourth component in the DOI system are the collections of DOI implementation procedures. The DOI system is implemented as a general and flexible identification and resolution service. The implementation procedures are provided to enable users to create domain specific implementations. The implementation component provides the low level identification and resolution. The data model component should be used also to add more semantics to the data for reusability and interoperability when using this component.

The DOI system is a complete system to identify data, resolve identifiers and provide semantics to the data. The DOI system is a standardised system but the resolution component could also be handled by the distributed system itself by means of a DHT. The DOI system represents a complete system that uses an OID like system to build and assign identifiers. The mechanism to identifying the data and the architecture of the DOI system is a good starting point to learn how an identification mechanism for distributed systems can work.

3.3.2.4 Universally Unique Identifiers

Universally Unique IDentifiers (UUID) were first defined as a DCE standard and later as an ITU-T recommendation and got accepted as an ISO/IEC standard. Both are revised to be compatible. In [28] on the other hand, UUIDs are declared in the URN namespace and are sometimes referred to as Globally Unique IDentifiers (GUID). In this section those kind of UUIDs will be further investigated.

A UUID is a fixed size 128 bit identifier that can be assigned without requiring a central authority. The 128 bit fixed size is relatively small compared to some other unique identifiers and makes it usable for sorting, ordering, hashing, storing and other programming purposes. Even without the presence of central authority, up to 10 million UUIDs can be generated by the UUID generation algorithm, per machine, per second. The URN UUIDs are an ITU-T recommendation and are accepted as the ISO/IEC 9834-8:2005 standard and is revised in the ISO/IEC 9834-8:2008 standard[2] standard.

[2]http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=53416

A UUID can be represented by the URN scheme by converting the 128 bit sequence into a string. Every field in the UUID is treated as an integer and represented as a zero-filled hexadecimal digit string. An example of a UUID represented in the URN scheme is `urn:uuid:573fd550-f5f5-11e2-b778--0800200c9a66`.

When a UUID is generated, it should be unique across space and time inside the UUID space. A UUID contains a time field so it is only valid in a specific time interval. Somewhere around the year 3400, UUIDs will rollover. Because of the nature of UUIDs, they can be used for both objects that have a short lifetime like sensor data but also for long living objects like documents.

The UUID algorithm can use three different approaches to generate UUIDs. The first approach is to use the MAC address of a machine to guarantee uniqueness, the second approach is to use a random or pseudo-random number generator and the third approach uses cryptographic hashing in combination with specific text strings defined by the UUID requestor. Apart from the validation using the timestamp portion of a UUID it is not possible to validate UUIDs. Two algorithms cannot produce the same UUIDs, with the assumption that each algorithm generates unique UUIDs.

A UUID consists of 16 octets, numbered from zero to fifteen. The most significant bits of a UUID are always encoded in the leftmost positions. Octets zero to three contain the low field of the timestamp, octets four and five the middle field of the timestamp, octets six and seven the high field of the timestamp multiplexed with the version number of the UUID, octet eight the high field of the clock sequence multiplexed with the variant and octet nine the low field of the clock sequence. Octets ten to fifteen are used for the spatially unique node identifier. In Fig. 3.1 the octet format of a UUID is shown with four octets per line.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td></tr>
<tr><td colspan="4">Time Low</td></tr>
<tr><td colspan="2">Time Mid</td><td colspan="2">Time High & Version</td></tr>
<tr><td>Clock High & Variant</td><td>Clock Low</td><td colspan="2">Node Identifier</td></tr>
<tr><td colspan="4">Node Identifier</td></tr>
</table>

Fig. 3.1: General UUID octet format

The variant defines how a UUID is formatted. The variant is set in the most significant bits (msb) of octet eight. The variants defined by [28] will have bits zero to two set to `10x`, where `x` can be any binary value. The variant values `0xx`, `110` and `111` are reserved for backward compatibility and future definition.

The version number on the other hand defines the algorithm used for the UUID generation. The version number is set on the 4 msb of the timestamp field. The possible values for the version are shown in Table 3.5. For the hashing based algorithms and the random or pseudorandom algorithm, the bits will be set using the hashed name or will be filled in randomly. For the time-based variant more details will follow in this section. A full specification that covers all versions except the DCE Security version can be found in [28].

Bits	Version	Description
0001	1	Time-based version
0010	2	DCE Security version
0011	3	Name-based version using MD5 hashing
0100	4	Random or pseudo-random version
0101	5	Name-based version using SHA-1 hashing

Table 3.5: Overview of UUID versions

The timestamp of a UUID is represented in the first 60 bits of the bit sequence. For the time-based algorithm the UTC time is stored in the bit sequence, as a count of 100 nanosecond intervals since 00:00:00.00, 15 October 1582. Another time format is possible but the used time format must be used consistently inside the whole UUID space. For the random or pseudo-random algorithms the timestamp is filled in as a 60 bit random generated value. For the hashing based algorithms the timestamp is set using the octets of a hashed name space identifier.

The clock sequence in a UUID is used to avoid generating duplicate items when a clock is set backwards in time or if a node identifier changes. A new clock sequence should be generated if no guarantee can be given by the system that the timestamp it will use will be set later than previously generated timestamps. If this is not the case the clock sequence should be a random value. Otherwise the information about the previous clock sequence can be used. For a node identifier change, a similar approach is used. The first clock sequence for a node should always be initialised as a random value, independent of the node identifier. As for the timestamps, the clock sequence is set by a similar approach for the algorithms that use hashing and random numbers.

The node identifier part in the time-based version will be constructed using a IEEE 802 MAC address that is available on the machine. If a real MAC address is not available a random or pseudo random generated value can be used. To ensure that there is no overlapping between both mechanisms the two leftmost bits will be used. The leftmost bit will differentiate between global and local addresses and the second leftmost bit will differentiate between unicast or multicast. A schema

for a UUID using a real MAC address can be found in Fig. 3.2. If we use the time-based variant for a UUID we can conclude that every node will be able to generate in total 2^{72} unique UUIDs if we allow the UUID's to roll over. In practice this number will be a lot smaller due to the time dependancy.

Time UTC	Version				Time UTC	Variant		Clock	MAC
48 bits	0	0	0	1	12 bits	0	1	12 bits	48 bits

Fig. 3.2: Time-based UUID schema using a real MAC address

If we take the UUID `573fd550-f5f5-11e2-b778-0800200c9a66`, notice that in the third group the first hexadecimal digit is `1`, which has binary representation `0001` and thus informs us that it is a time-based generated UUID. The first hexadecimal value of the fourth group is `b`, which has binary representation `1011` and thus informs us in the two first bits that it is the IETF variant specified in [28]. The last group will tell us the MAC address that generated this UUID is `08:00:20:0c:9a:66`. If the MAC address embedded in a UUID should not be visible due to security or privacy concerns it is possible to bypass this by reserving new MAC addresses or to apply cryptography.

3.3.2.5 Comparison

When looking at the different identifiers, URIs, OIDs and DOIs have a similar approach but they all differ on some key concepts. To compare the different types of identifiers, an overview about the length of the identifier, the degree of centralisation to use the identifiers, the guaranteed uniqueness and the semantics that are embedded in the identifier itself will be given. The summary of the comparisons can be found in Table 3.6.

If we look at the identifier length of URIs, OIDs and DOIs we see that they can have an arbitrary length. The variable length is good when you need support for many identifiers but if you take a look at the length from a programmer's point of view, the variable length can introduce some issues. First of all, because the length is not fixed, the size to store the identifiers is also not bound. Sorting and ordering of URIs and DOIs is not trivial. No natural or standardised sorting is defined for the URIs and DOIs. For the OIDs one can use the tree abstraction for ordering and sorting. The UUIDs are a 128 fixed size bit sequence. As stated before, this lends itself well to support storing, sorting and ordering.

When comparing the degree of centralisation, we notice that for the URIs, OIDs and DOIs, some centralised authority should be used to reserve specific parts of

URIs	OIDs	DOIs	UUIDs
Arbitrary length	Arbitrary length	Arbitrary length	Fixed length
Centralised scheme reservation	Centralised OID reservation	Centralised prefix reservation	Fully decentralised
Reserved scheme uniqueness	Reserved OID tree uniqueness	Reserved prefix uniqueness	Unique by algorithm
Parsed	Parsed	Resolved	Interpreted
Embedded scheme semantics	Embedded subtree semantics	Embedded prefix semantics	No semantics

Table 3.6: Comparison of identifiers

the identifiers. These reservations of a URI scheme, an OID tree node and a DOI prefix are needed to give a scope for the identifiers where there can be a guarantee of uniqueness. The uniqueness is only guaranteed for the reserved parts of the identifier. The uniqueness of the other part of the identifier must be guaranteed when an identifier is assigned to an object. In fully decentralised systems the use of reserved parts in the identifiers can lead to another centralisation but it does not imply it. One can for example take an UUID as the numbering used in the suffix, for a prefix in the DOI system that was reserved before the system is used.

Because UUIDs have a fixed length, UUIDs must not be parsed and can be interpreted. This is also the case when a UUID is represented as a string. URIs and OIDs on the other hand are defined in such a way that they can always be parsed, no matter what the identifiers are used for or which scheme or object tree is used. For the URIs, the syntax is defined in a general way such that a URI can always be parsed. For the OIDs, the syntax is also strictly defined and the sequence of digits and dots that represents a OID can always be parsed. For the DOI system a resolver component is provided that should be used to resolve DOIs.

Every identifier discussed has some form of semantics embedded in the identifier. When we talk about embedded semantics, we refer to object specific information that is placed into the identifier. The MAC address for the UUIDs for example has no semantic meaning for the object that needs to be identified itself. A MAC address is not even required for all algorithms that generate UUIDs and should not be referred to as semantic embedded information of the identifier. For URIs, every scheme has his own semantics. By using a defined scheme, some classification for the objects is embedded into the identifier itself. For the OIDs and DOIs the reserved subtree and the reserved prefixes are also a way of embedding semantic

classification of objects into the identifier. Furthermore it is possible for URIs, OIDs and DOIs to embed more semantic information in the identifiers but it is not recommended to do it due to security considerations.

When discussing all the identifiers, it looks like UUIDs are more suitable to be used in distributed systems. This is because every node in the distributed system can generate a UUID in a standardised way. A UUID can be represented as a URN, and thus also as a URI. One can argue to use the URI system but then we lose the strong points of using a UUID itself. The URIs, OIDs and DOIs are more targeted towards systems that allow some degree of centralisation but they can be used in a fully decentralised way also. The strength of the DOI system is that every step for the identification can be done by using the four components but this is also the weakness when we want to use it in a distributed way.

The time-based UUIDs are actually very useful because a cross-media server is a networked system and should have at least one MAC address available for every node. When using the time-based UUIDs as identifiers every node in the distributed systems mimics a delegate of an oid distributor. A UUID is not actually an oid but a key. In the cross-media server we assign keys and do not allow keys to be changed.

The fact that UUIDs are not designed for a specific domain is not a problem. A UUID can be used in any context. A machine can for example generate five UUIDs and four of those can be used in another system that runs on the same node.

One of the concerns of using UUIDs is a scalability concern. Because of the fixed size and the limit on UUID generation in combination with the rollover period, using UUIDs can become problematic in the future. In practice, 10 million UUIDs per MAC address, per second, will be for most distributed systems now and in the future, a good upper bound. If this limit will become too small it is always possible to extend the algorithm to support UUIDs that contain more bits as seen for example in the IPv4 to IPv6 transition [15]. There a transition of 32 bit identifiers to 128 bit identifiers is described and a similar approach can be used if necessary.

3.3.3 Object Identity in the RSL Model

In the distributed cross-media server a UUID will be generated on a node when an *RSL* object is created. The UUID is thus assigned on object creation and that UUID will not be used again if the object is deleted from the system. For UUIDs

one needs to take into consideration the rollover property because of the timestamp field. Otherwise UUIDs will be regenerated and objects will have the same UUID.

When considering objects that will be created when using the *RSL* metamodel, the creator of an object will have a large impact on determining the uniqueness of an object when we look at it at the semantic level. If two different users create an object with the same attributes and properties the objects should not be the same. For example if somebody creates a resource and another user creates the same resource those two objects should not be the same. On the system level on the other hand, the state of the object should not contribute to the identifier.

There are different kinds of equality that can be discussed. On the user level for example, objects can also be equal. Let us say two users both create a resource with the same data. Both objects can be equal but they are certainly not the same object on the system level. To compare objects in *RSL* we need to introduce algorithms depending on the object classes. For equality of a general entity the properties of that entity influence the equality. If we look at links for example, the source and target properties are important when considering equality.

If we use the UUIDs as suggested previously we can identify every object. When we embed information of one object inside another and we update the original embedded object, only the original object needs to be updated because we use identifiers that can be used as a reference to another object. For example, if we have two different links to a resource and we change some properties of the resource, the links pointing to those resources do not need to be updated but only the resource itself.

4 Implementation

Our goal is to implement a distributed cross-media server. In the implementation phase existing implementations of protocols and frameworks will be combined to speed up the implementation. The use and reuse of already tested code should lead to a more stable system.

As the research pointed out, some interesting work is already done in the field of hypermedia systems and DHT protocols. To support many of the desired features of the distributed cross-media server, the *RSL* model can be used in combination with the existing DHT protocol *S/Kademlia*. Some properties or requirements are fulfilled as a direct consequence of the overall design of the distributed cross-media server that is provided in this thesis.

The goal is not to design a new state of the art system or to completely re-implement existing algorithms, protocols or models but to combine state of the art systems without losing some of the properties of the standalone systems. Some changes need to be made to the existing implementations to combine existing frameworks and protocols to implement the distributed cross-media server. These changes are also necessary to support all the desired features and properties of the targeted system or to leave room for desired features in future work.

4.1 Previous Implementations

In the sections about the state of the art, important first and second generation hypermedia system implementations are explored and reviewed. For the third generation hypermedia systems, which came after the *Dexter* reference model, no overview is given for the implementations. Those implementations will not be considered because the *RSL* model is a successor of those third generation hypermedia models. The hypermedia systems we consider in this chapter are previous implementations that are designed using the *RSL* model.

As stated before, this thesis is not the first implementation of a distributed hypermedia system but also not the first implementation that uses the *RSL* model in the core to implement a hypermedia system. There were two important implementations of the *RSL* model. The first implementation is the *iServer* [44] platform, the second implementation is a cooperative *iServer* extension [42] that is based on the *iServer* platform.

4.1.1 iServer

The *RSL* model was not only published as a model but it was also implemented as *iServer*, a platform for cross-media information management. The *iServer* platform was used as the core framework of some hypermedia projects that need mechanisms to link digital and physical data. One of the most interesting use cases was the use of the *iServer* platform in the *iPaper* [37] framework. The *iPaper* framework provides a solution to support interactive paper. Promising results were achieved with the *iPaper* framework.

The *iServer* implementation has support for extensibility through the plug-in mechanism but also the *iServer* implementation itself has been extended through time. The changes that needed to be done to extend the functionality of the *iServer* platform itself, even though some of them were extensions of the core model, did not have an impact on the applications that were using the *iServer* framework. The applications evolved in parallel with the *iServer* framework and kept working without any changes at the application level.

4.1.2 Distributed Link Server

Because the *iServer* platform did not provide any mechanisms that enable collaborative authoring, a distributed link service was designed and implemented [42]. In the newly designed cross-media environment, it is possible to share links as a collaborative information space. To ensure that links have a certain quality, a mechanism of user and link rating is designed for the system.

As stated in [42], the peer-to-peer-based distributed link service is not designed to be a distributed hypermedia system based on peer-to-peer technology. The goal was to design an architecture that allows the sharing of link metadata. This makes the design of the cross-media environment not very useful for the targeted cross-media server but a similar link rating mechanism can be used to rate links in the distributed cross-media server. However it is considered as future work. Some

research needs to be done on the consequences of a user model in combination with a link rating system.

4.2 Architecture

4.2.1 RSL

The implementation of the distributed cross-media server uses an already existing cross-media server implementation that is based on the *iServer* platform. The existing implementation is an object-oriented Java implementation that uses the class $AbstractRslElement$ as a parent for every *RSL* object. All the *RSL*-related classes can be found in the *org.mobricant.iserver.rsl* package.

As discussed in the section about object identity, the *RSL* implementation should be modified to support UUIDs for the identification of objects. This change is made in the $AbstractRslElement$ class. The class will generate a new UUID when a new *RSL* object is generated and stores it in the $uuid$ field which can be accessed by the $getUUID$ method. The UUID of an *RSL* object can not be changed, is assigned on object creation and is never reused even when the object is deleted. The UML class diagram of the $AbstractRslElement$ class can be found in Fig. 4.1.

The UUIDs that are generated are version 1 UUIDs and thus a MAC address needs to be available. Because we are working in a networked environment a MAC address should be available on the system that creates a node. A new class is created to generate UUIDs. The $UUID$ class can be found in the *org.mobricant.iserver.distributed* package. The UML class diagram is shown in Fig. 4.2

Before a UUID is generated the $UUID$ class will check if there is a MAC address available and if it is not available an exception is thrown. For generating the UUIDs, an external library is used because the built in Java version can only generate version 4 random UUIDs. As discussed before we don't want version 4 UUIDs, what we need are version 1 UUIDs. The library for generating version 1 UUIDs that is used in the implementation is developed by Johann Burkard. The library allows to generate, parse, clone and compare UUIDs. The UUID library is available at the developers website[1].

To support the identification mechanism using UUIDs the $equals$ method of the $AbstractRslElement$ class needs to be implemented using UUIDs. To compare

[1] http://johannburkard.de/software/uuid/

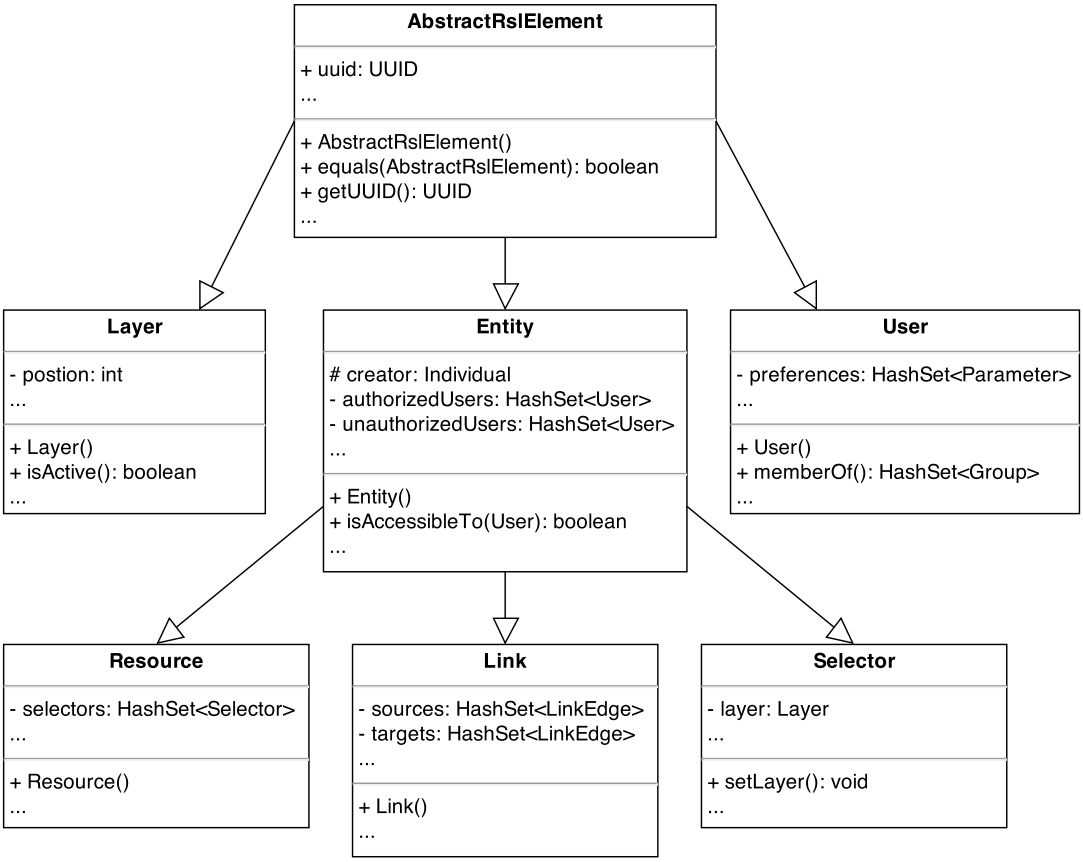


Fig. 4.1: RSL UML class diagram

two *RSL* objects their UUIDs need to be compared. This is done by using the $equals$ method of the $UUID$ class that compares the UUIDs.

The most important feature of the distributed cross-media server is that a mechanism for linking data should be implemented. This feature is supported in the *RSL* model by the concept of links over entities. A link, a selector and a resource are subclasses of entities. The provided mechanism of linking data will be a linking mechanism for links, resources and selectors. The classes that represent these core concepts are in the implementation present as the $Entity$, $Resource$, $Selector$ and $Link$ classes. The classes are shown in the UML class diagram in Fig. 4.1.

More details about the linking mechanism can be found in section 2.1.5 that covers the *RSL* model in detail. All the corresponding classes that are present in the metamodel are also represented in the implementation.

To support a multi-user environment, the user model that is present in the *RSL*

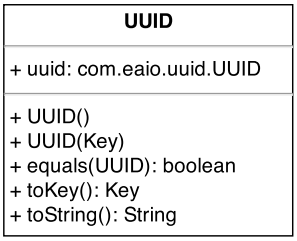


Fig. 4.2: UUID UML class diagram

model is used. The implemented user model will provide an abstraction for a user and user groups that provides mechanisms for data ownership and data sharing. As can be seen in the UML class diagram in Fig. 4.1, no further changes should be made to the other RSL classes to support key-based routing because all the changes are done in the $AbstractRslElement$ class.

Two other properties that should be supported by the system are found in the core of the *RSL* model. The first property is extensibility and is provided by using the plug-in mechanism. The second property is the openness and is a property that results from the design of the *RSL* model.

4.2.2 S/Kademlia

The implementation of the DHT is done by using the Kademlia protocol in combination with some of the security features described earlier in this document. An existing implementation of *S/Kademlia*[2], that is based on the *openkad*[3] *Kademlia* implementation is modified to support the key-based routing algorithm using UUIDs as keys and to support the features of *RSL* in the lookup phase.

The *S/Kademlia* implementation was done in Java and uses the open source dependency injection framework *Guice* [49] that is developed by *Google*. Dependency injection enables developers to inject new dependencies into existing classes by using modules, injectors and providers.

The security problems that were described in previous sections regarding the Kademlia protocol are solved in the *S/Kademlia* protocol design. The system is still not waterproof for attacks but it should be rather hard to abuse the properties and operations of the system to perform attacks on the underlay or overlay network.

[2]https://code.google.com/p/skademlia/
[3]https://code.google.com/p/openkad/

The *S/Kademlia* implementation that is used adds a node authentication mechanism to prevent eclipse attacks by hashing the UUIDs used to identify nodes. Lookups in the network will be routed through k disjoint paths where k is the replication parameter of the *Kademlia* protocol. This will ensure that lookups succeed with up to k-1 adversarial neighbours for a node. The data that travels through the overlay network will be validated by comparing the hashed key in the message with the hashed key of the data. An identifier is assigned to the storage layer and checked at delivery to make sure that messages that travel through the overlay network are destined for a specific storage layer.

The caching of data along the lookup paths in *Kademlia* will increase the availability of data and the performance of the cross-media server by speeding up data retrieval. In combination with the ability to lookup nodes in an asynchronous way the system will have a good responsiveness compared to other DHT protocols.

4.2.2.1 Storage

The storage is done by using the *S/Kademlia* interface. The *S/Kademlia* interface is defined in the $SkademliaInterface$ class that can be found in the *il.technion.ewolf.kbr.SKademlia* package. The interface is implemented in the $SKademliaNet$ class that is inside the *net* subpackage. The UML class diagram can be found in Fig. 4.3

As can be seen in the class diagram in Fig. 4.3 the Key class is used as an abstraction of a key. Objects of the Key class represent the keys that are used in the routing algorithm provided by the *S/Kademlia* implementation. If we take a look at Fig. 4.2 you can see that the $UUID$ class has a method $toKey$ to convert a $UUID$ object to a Key object.

To store data in the overlay network the $securedPutValue$ method is defined. This method executes a put operation that is implemented in the $PutOperation$ class. All the operations of *S/Kademlia* can be found in the *il.technion.ewolf.kbr.SKademlia.op* package.

When a put operation is performed the existing implementation does not store the information in the overlay network as described in [5]. In the modified implementation the secure siblings are used to generate a list of secure neighbour nodes. A store message will be send to those nodes as well.

The method $securedGetValues$ that takes a Key and an $User$ object as arguments is added to the interface of *S/Kademlia*. This is done to enable looking up

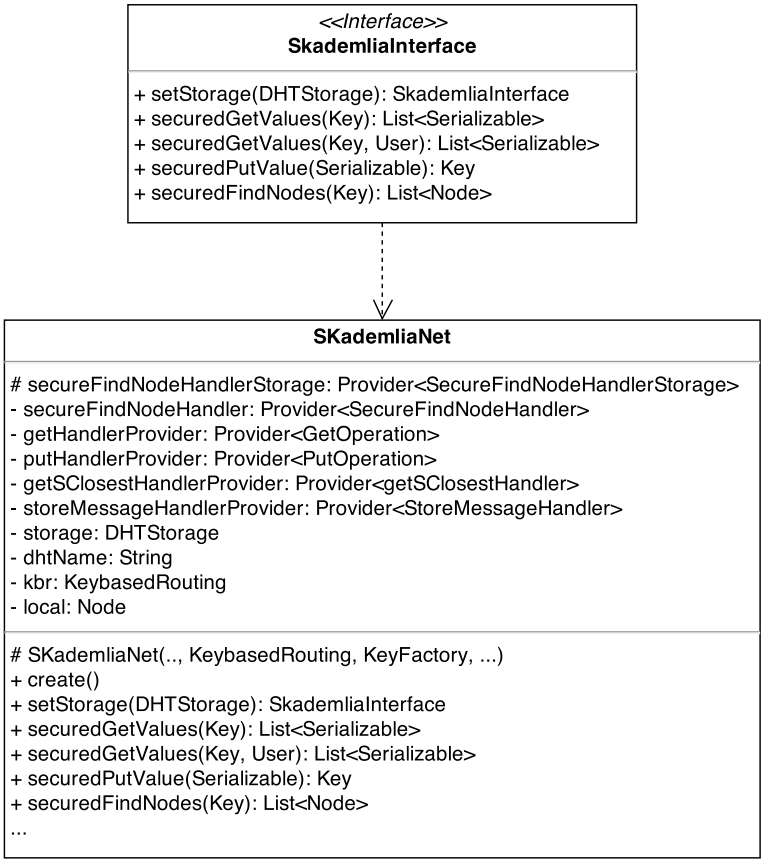


Fig. 4.3: S/Kademlia interface UML class diagram

RSL elements for a certain user. The $securedGetValues$ method will check the identifier assigned to the storage layer before performing a get operation. If the storage identifiers match then the get operation is executed.

For the $GetOperation$ class, which is used for executing get operations, a new field is added to the class. The added field is the $user$ field. The $user$ field contains the $User$ object of the requesting user and needs to be set by using the $setUser$ method.

The storage of data is handled by the $AgeLimitedDHTStorage$ class. This class implements the $DHTStorage$ interface. All the storage related classes can be found in the *il.technion.ewolf.kbr.SKademlia.storage* package. The UML class diagram for the interface and the implementation can be found in Fig. 4.4.

The $DHTStorage$ interface is also changed to perform searches based on incoming get messages that support *RSL* objects. These messages will provide a key for the *RSL* object that needs to be found and the $User$ object for the user that need to

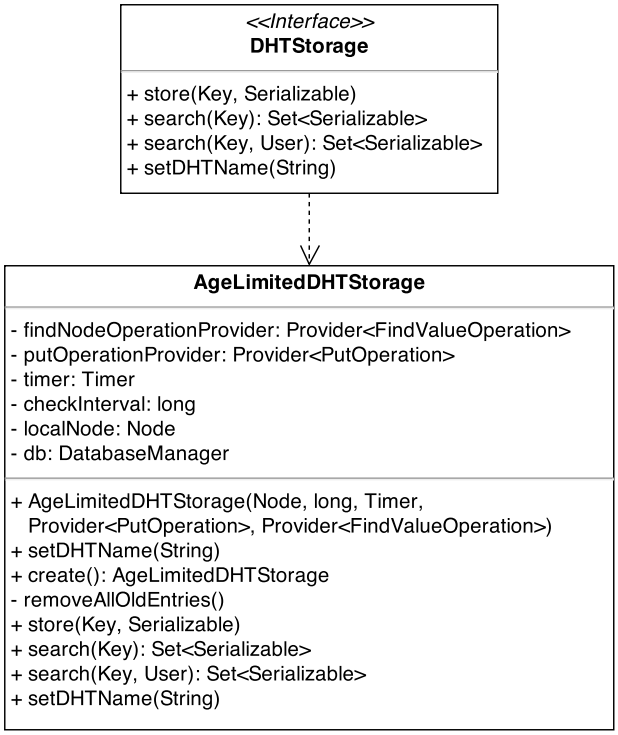


Fig. 4.4: DHT storage UML class diagram in S/Kademlia

access the object. When a $search$ method is called for a specific user, the storage is searched for an object that maps to the key that is accessible for that user.

The *RSL* implementation uses abstract interfaces in the database layer and could also be rewritten to support dependency injection. In this way new database implementations could be added without modifying existing code. In the current distributed cross-media server implementation the storage is implemented in the $AgeLimitedDHTStorage$ class without using any database abstraction. The implementations uses an in memory database.

4.2.2.2 UUID

To support UUIDs for nodes some changes need to be made to the $SecureKademliaModule$ class. This class is the core class of the *S/Kademlia* implementation. It provides most of the dependencies and binds all of the components that enable the key-based routing and storage. The $SecureKademliaModule$ can be found in the *il.technion.ewolf.kbr.SKademlia.net* package.

The $SecureKademliaModule$ class provides the node object that is used in the

S/Kademlia implementation to represent contact information. The node provider is modified in such a way that every node is assigned a new UUID on node creation.

4.2.3 RLS Node Features

Every RLS node in the system has equal functionality. This node abstraction is not to be confused with the node abstraction that is provided in the *S/Kademlia* implementation by the $Node$ class. An RSL node should be seen as a building block in the overlay network.

The node functionality is implemented in the $RSLNode$ class. The $RLSNode$ class can be found in the *org.mobricant.iserver.distributed* package. In Fig. 4.5 the UML class diagram is shown for the $RSLNode$ class.

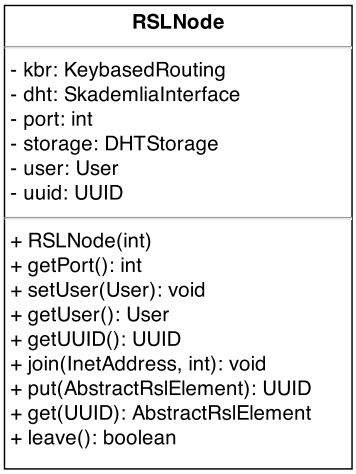


Fig. 4.5: Node UML diagram

Every node has four fundamental operations: a $join$, a put, a get and a $leave$ command. The $join$ command will contact a bootstrap node in the overlay network. The two nodes will exchange the information that is necessary to join the overlay network.

When constructing a node, the UDP port on which the node listens for incoming traffic should be provided. When a node is created a new $SecureKademlia-Module$ object is created. The $SKademliaNet$ class is used for the key-based routing. The DHT will be handled by the $SkademliaInterface$ and the storage layer is handled by the $AgeLimitedDHTStorage$ class. When creating an

$RSLNode$ object a UUID to identify the node is generated. This UUID can be retrieved using the $getUUID$ method.

For every node, a user must be set to create and lookup *RSL* objects. This is because every *RSL* object needs to have a creator set and not every *RSL* object must be accessible to every *RSL* user. The $setUser$ method can be used with a $User$ object as parameter to set the current user operating on an *RSL* node.

As in most DHT implementations, a put and get command are provided to store and locate data in the overlay network. The put command will store an instance of a subclass of the $AbstractRslElement$ class in the overlay network. The put command uses the *S/Kademlia* interface to store data in the DHT. A get command will lookup an *RSL* object by locating it in the network using the key-based routing algorithm. The UUID of the object is converted to a Key object and that key is used to perform the lookup. Like the put command, the get command also uses the *S/Kademlia* interface to lookup data based on a UUID key.

A $leave$ method is provided to close all the sockets that the RSL node uses for communication in the network. When a $leave$ command is executed in the network no data is moved. In some scenario's it is possible that the data on a node that leaves the network needs to be transferred to other nodes.

4.2.3.1 Parameters

As explained in [30] and [5] some properties of the system can be changed by adjusting some parameters of the *Kademlia* and *S/Kademlia* protocol. These properties can be found in the $SecureKademliaModule$ class. By using named properties provided by the *Guice* framework some parameters are set.

The α parameter is set using the named property *openkad.net.concurrency*. The k parameter is set using the named property *openkad.bucket.kbuckets.maxsize*. For the storage layer, the β replication parameter is set as the named property *dht.storage.checkInterval*. In the $SecureKademliaModule$ class more named properties can be found. The named properties *openkad.net.udp.port* and *openkad.keyfactory.keysize* that represent respectively the UDP port and the key size are overridden in the $RSLNode$ constructor to set the right UDP port the node should listen to and the correct key size for UUIDs.

4.2.4 Test Cases

To have a good overview on the internals of the distributed cross-media server a logger is used. The logging is done by using a lightweight Java library called *Minlog*[4] developed by EsotericSoftware. There are different levels of logging available. The error log will output critical errors that break the application. Information messages are shown in the info log. There is also a debug and trace log available that can be used during the further development and debugging of the application. More information about the *Minlog* library can be found on the developers website.

Some test case are implemented using the JUnit testing framework. An overlay network is simulated and some operations are performed. These tests can be found in the $RSLTests$ class in the *org.mobricant..iserver.distributed* package. The tests defined in the $RSLTests$ class are basic unit tests that always need to succeed.

The first test is that the system should be able to create an *RSL* node. After the test to create a node, some networks are built up using an arbitrary number of nodes by performing $join$ operations using a bootstrap node. Afterwards some networks are built up using the previous approach and data is put inside the network using the put command. After putting the data in the network the data is queried in the network using the get command. All data that was put in the network should be found afterwards. The last test will check if only data can be retrieved from the network that should be accessible for a specific user.

Not only unit tests are run. Also some statistical data is collected regarding the implementation by performing some simulations. The simulations are run in an overlay network consisting of 100 nodes with a replication parameter of 8. The network is always built up in a random order. This means that a new node will contact a random bootstrap node to join the network.

A first simulation is run to see how the load is balanced in the overlay network. The load we investigate is implied by the DHT implementation and no further balancing is done using a load balancing scheme. The simulation will start with the network previously described and the results of the simulation are shown in Fig. 4.6. An equal number of resources are put in the overlay network by using random nodes to perform the put command.

For every node a dot is plotted in Fig. 4.6. This dot represents how much load is present on the node. The load is defined as the number of resources on the node

[4]http://github.com/EsotericSoftware/minlog

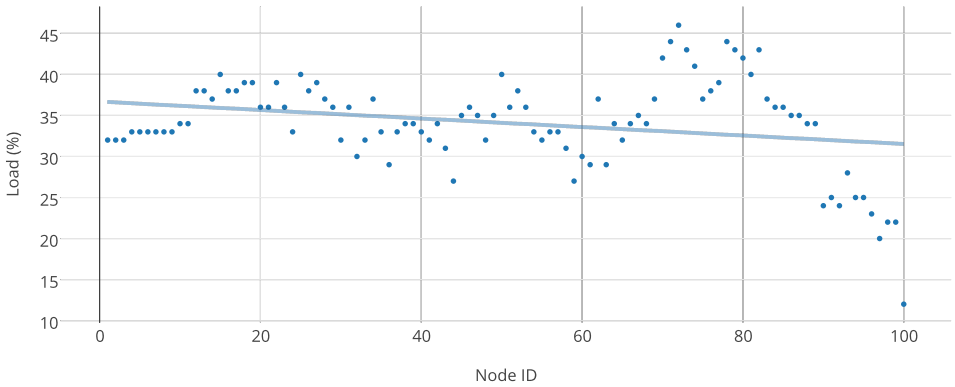


Fig. 4.6: Average load on every node with caching

divided by the total number of nodes in the overlay network. A line is fitted for the scatter plot to see the balance in the system. The used numbers for the load are the average load of a node after running five simulations.

As can be seen in Fig. 4.6, the load is spread with a variation of 5% from the linear fit. This is a good starting point to achieve a balanced system. The load balancing schemes given in section 2.3 can be used to further balance the load in the system. This is considered as future work. The distributed cross-media server was designed in such a way that the virtual node approach can be implemented on top of the current design.

The second simulation is run to check how much of the resources can still be found when nodes concurrently leave the network. The initial setup is like in the simulation to determine the load. The resources are also put in the overlay network in a similar way. In Fig. 4.7 the success rate of the get commands is shown when a fraction of the total number of nodes leave the system.

The random get commands are generated in a similar way like the data is put in the overlay network. Random nodes that are still in the overlay network are used to get the resources that were put in the overlay network. The success rate is shown as a fitted polynomial that results from the average success rate when the simulation is run five times.

In the resulting plot shown in Fig. 4.7 we see that the network will find only a fraction less than the percentage of nodes that left the network. This is normal because no resources are moved when a leave command is performed. Also when the fraction that concurrently leaves system increases, the system will not completely fail

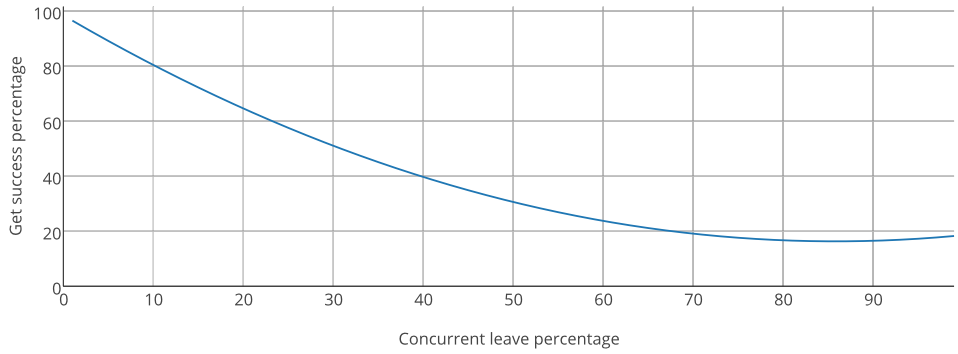


Fig. 4.7: Plot of get command success rate when nodes leave the network

to route requests.

The tests and simulations show that the implementation can serve as good base for a general implementation of a distributed cross-media server. The operations that need to succeed in the test cases succeed with a success rate of 100%. The first simulation shows that the load is balanced rather good without any extra load balancing schemes and that concurrent leaving of nodes will not introduce a high error rate on the routing mechanism. A remark that need to be made regarding the tests is that the testing environment is rather small and the simulations should also be done in overlay networks that contain much more nodes. This is anticipated by changing the concurrency parameter of the *Kademlia* protocol.

4.2.5 Versioning

Versioning was not implemented in the system but the design did not make any restrictions or assumptions about one version systems. The identifiers could be mapped to the latest version of a resource and identifiers for other versions can be constructed by appending a new ID to the UUID. Also for the caching features of *Kademlia*, the introduction of versioning should generate problems with backward compatibility. The changes that need to be made are in the storage layer and this will handle the announcement of fresh data and renewal of the caching along the paths.

As motivated by David Hicks, the need for a version control framework for hypermedia systems is very important. In [23], Hicks provides a design for a version control framework for open hypermedia environments. The version control frame-

works supports versioning on both the data level and the structural level.

5 Conclusions

Even though the research about hypermedia systems started in the sixties, there is still no globally accepted definition and solution for hypermedia systems and in particular for distributed hypermedia systems. Many efforts have been made to design state-of-the-art systems but most of them focussed on the data model only or on the visualisation only. This thesis provides a clear domain independent view on what a hypermedia system is and which features it should support.

Some important core features described in the design phase of this thesis are not globally present on all the systems that were designed or implemented. Even on the link level, not all systems support multi-source and target links for example. Other features that are lacking in many systems are the existence of a user model and this also has great implications on concurrency and collaboration. Many implementations provide a domain specific approach and leave no room for evolution.

When all the problems regarding the existing and historical hypermedia systems were identified, a new approach was used to design a model that can be used in a domain independent way. This was done by using concepts of the metamodelling domain for the design of a general hypermedia model. What is still lacking in the *RSL* model is a well defined mechanism for the identification of data that will have no impact on the openness and generality of the model.

After exploring the state of the art of distributed systems, the conclusion is that this domain has more mature solutions for the problems of distributing data with respect to hypermedia systems. Many efforts were made with success to define protocols to enable a distributed system without making assumptions about the domain it will be applied to. This thesis provides a design for a distributed system hypermedia system by identifying all the features and properties it should support. On overview of how the design requirements are met can be found in Table 5.1

One thing we need to note is that the investigation of the protocols for distribution

Requirement	Solution
Links	RSL Link concept
Scalability	Property of distributed systems, use of UUIDs
Persistency	Sending store message to k-closest nodes on put command, age limited DHT storage layer
Extensibility	RSL plug-in mechanism
Openness	Property of the RSL metamodel
Availability	Lookup over disjoint paths, update routing information when messages travel through the system
Performance	Caching on k-closest nodes when find value did not generate response, lookup over disjoint path
Concurrency	Implementation allows concurrent querying
Balancing	Natural load balancing of a DHT, node design leaves room for versioning in future work
Versioning	UUIDs leave room for versioning in future work
Ownership	RSL User model
Portability	No assumptions are made about nodes or the systems where a node should be run on.
Security	S/Kademlia security measure, least recently used nodes replacement policy in k-buckets

Table 5.1: Overview of solutions to requirements

are targeted towards content distribution even though there are also state of the art protocols for communication and collaboration. The core of a hypermedia system is content distribution but also collaboration is an important feature. However it should be built on top of the data distribution.

The focus on content distribution is done without taking into consideration the concept of linked data. Further research should be done to identify the place where the load balancing should be modified or inserted with respect to the links. This is due to the fact that content distribution typically does not link data together. Mostly only data with a specific key is looked up and no related data is defined or retrieved. This can be integrated in the distribution protocol as well as the load balancer.

Another contribution of the thesis is that a mechanism is provided to identify data in distributed hypermedia system. The identification of data is investigated in such a way that it does not make assumptions about the implementation or domain of the distributed hypermedia system. Also no assumptions are made about the nodes that exist in the distributed hypermedia system.

After combining the state of the art solutions, promising results were achieved. The impact of combining and slightly adapting the *RSL* model and the S/Kademlia protocol had no impact on the desired features of the targeted distributed cross-media server. Both of the used solutions did not introduce any restrictions on each other.

When integrating the UUIDs into *RSL* no restrictions are implied for the model. Only a way of identifying data is introduced in the implementation. Because *S/Kademlia* does not make any restrictions on the key space where identifiers are assigned from, integrating the UUIDs as keys for the key-based routing mechanism does not break the algorithms of *S/Kademlia*.

To support the concepts of *RSL* no dramatic changes needed to be made in the core of the *S/Kademlia* implementation. All the changes that are made are actually mechanisms to support or enable *RSL* in a convenient way. This means that a generic solution is provided and that any existing extension to *RSL* can use the distributed system that is implemented.

All the points summed up in this section can make us conclude that the design of the distributed hypermedia system provided in this thesis can be used as a starting point for any other distributed hypermedia system and that the identification in such a system can be done in such a way that it does not make restrictions for future work or domain specific implementations.

As a summary, these are the contributions of this thesis:

- Definition of a hypermedia system.
- Design for a domain independent distributed cross-media server.
- Identification mechanism that can be used in a distributed cross-media environment.
- Bug resolved in the *S/Kademlia* implementation.

When reading the summery of the contributions we can see state that the research question ”How should we design a distributed cross-media server and how can data be identified in such a system” got solved throughout the thesis.

5.1 Future Work

A link rating mechanism, as found in the previous implementations of *RSL*, could also be integrated into the distributed cross-media server to provide some quality

mechanisms for linked data in a collaborative cross-media environment.

As stated before, the versioning and load balancing of data is not present in the current implementation. In future work, versioning could be supported by the mechanism proposed in section 4.2.5. A load balancing scheme described in section 2.3 can be used to further balance the load in the system.

Another feature that should be explored is the load balancing mechanisms that make use of virtual servers. In a dynamic heterogenous system the load balancing should be further explored to integrate hypermedia and *RSL* specific concepts into the load balancing scheme.

In contrast to load balancing where the load is spread throughout the system, further research needs to be done to clutter parts of data on specific nodes to have a more performant and responsive system.

Bibliography

[1] R. Akscyn, D. McCracken, and E. Yoder. KMS: A Distributed Hypermedia System for Managing Knowledge in Organizations. In *Proceedings of Hypertext 1987*, pages 1–20, Chapel Hill, USA, November 1987.

[2] K. Andrews, F. Kappe, and H. Maurer. Hyper-G and Harmony: Towards the Next Generation of Networked Information Technology. In *Proceedings of CHI 1995*, pages 33–34, Denver, USA, May 1995.

[3] S. Androutsellis-Theotokis and D. Spinellis. A Survey of Peer-to-Peer Content Distribution Technologies. *ACM Computing Surveys*, 36(4):335–371, December 2004.

[4] M. Atkinson, D. DeWitt, D. Maier, F. Bancilhon, K. Dittrich, and S. Zdonik. The Object-Oriented Database System Manifesto. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an Object-Oriented Database System*, pages 3–20. Morgan Kaufmann Publishers Inc., San Francisco, USA, June 1992.

[5] I. Baumgart and S. Mies. S/Kademlia: A Practicable Approach Towards Secure Key-based Routing. In *Proceedings of ICPADS 2007*, volume 2, pages 1–8, Hsinchu, Taiwan, December 2007.

[6] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. IETF RFC 3986, January 2005.

[7] V. Bush. As We May Think. *Atlantic Monthly*, 176(1):101–108, July 1945.

[8] J. Byers, J. Considine, and M. Mitzenmacher. Simple Load Balancing for Distributed Hash Tables. In M. F. Kaashoek and I. Stoica, editors, *Peer-to-Peer Systems II*, volume 2735 of *Lecture Notes in Computer Science*, pages 80–87. Springer, August 2003.

[9] S. Carmody, W. Gross, T. H. Nelson, D. Rice, and A. Van Dam. A Hypertext Editing System for the /360. In M. Faiman and J. Nievergelt, editors, *Per-*

tinent concepts in computer graphics, pages 291–330. University of Illinois Press, December 1969.

[10] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure Routing for Structured Peer-to-Peer Overlay Networks. *SIGOPS Operating Systems Review*, 36(SI):299–314, December 2002.

[11] C. Chekuri and S. Khanna. On Multidimensional Packing Problems. *SIAM Journal on Computing*, 33(4):837–851, April 2004.

[12] J. Conklin. Hypertext: An Introduction and Survey. *IEEE Computer*, 20(9):17–41, September 1987.

[13] P. De Bra, G.-J. Houben, and H. Wu. AHAM: A Dexter-based Reference Model for Adaptive Hypermedia. In *Proceedings of Hypertext 1999*, pages 147–156, Darmstadt, Germany, February 1999.

[14] N. G. de Bruijn and P. Erdos. A Combinatorial Problem. *Koninklijke Nederlandse Akademie van Wetenschappen*, 49(49):758–764, June 1946.

[15] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6). IETF RFC 2460, December 1998.

[16] J. R. Douceur. The Sybil Attack. In P. Druschel, M. F. Kaashoek, and A. Rowstron, editors, *Peer-to-Peer Systems*, volume 2429 of *Lecture Notes in Computer Science*, pages 251–260. Springer, October 2002.

[17] D. C. Engelbart and W. K. English. A Research Center for Augmenting Human Intellect. In *Proceedings of AFIPS 1968, Part I*, pages 395–410, San Francisco, California, December 1968.

[18] K. Grønbæk and R. H. Trigg. Design Issues for a Dexter-based Hypermedia System. In *Proceedings of ECHT 1992*, pages 191–200, Milan, Italy, November–December 1992.

[19] P. K. Gummadi, S. Saroiu, and S. D. Gribble. A Measurement Study of Napster and Gnutella as Examples of Peer-to-Peer File Sharing Systems. *SIGCOMM Computer Communication Review*, 32(1):82–82, January 2002.

[20] F. Halasz and M. Schwartz. The Dexter Hypertext Reference Model. *Communications of the ACM*, 37(2):30–39, February 1994.

[21] F. G. Halasz. Reflections on NoteCards: Seven Issues for the Next Generation of Hypermedia Systems. In *Proceedings of Hypertext 1987*, pages 345–365, Chapel Hill, North Carolina, USA, November 1987.

[22] F. G. Halasz, T. P. Moran, and R. H. Trigg. Notecards in a Nutshell. *SIGCHI Bulletin*, 17(SI):45–52, May 1986.

[23] D. L. Hicks, J. J. Leggett, P. J. Nürnberg, and J. L. Schnase. A Hypermedia Version Control Framework. *ACM Transactions on Information Systems*, 16(2):127–160, April 1998.

[24] K. Hildrum, J. D. Kubiatowicz, S. Rao, and B. Y. Zhao. Distributed Object Location in a Dynamic Network. *Theory of Computing Systems*, 37(3):405–440, January 2004.

[25] Y. Jia, I. Brondino, R. J. Peris, M. P. n. Martínez, and D. Ma. A Multi-resource Load Balancing Algorithm for Cloud Cache Systems. In *Proceedings of SAC 2013*, pages 463–470, Coimbra, Portugal, March 2013.

[26] M. F. Kaashoek and D. R. Karger. Koorde: A Simple Degree-Optimal Distributed Hash Table. In M. F. Kaashoek and I. Stoica, editors, *Peer-to-Peer Systems II*, volume 2735 of *Lecture Notes in Computer Science*, pages 98–107. Springer, August 2003.

[27] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of STOC 1997*, pages 654–663, El Paso, USA, May 1997.

[28] P. Leach and M. Mealling. A Universally Unique IDentifier (UUID) URN Namespace. IETF RFC 4122, July 2005.

[29] A. Lombardoni. *Towards a Universal Information Platform: An Object-Oriented, Multi-user, Information Store*, volume 68. ETH Zurich, June 2007.

[30] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In P. Druschel, F. Kaashoek, and A. Rowstron, editors, *Peer-to-Peer Systems*, volume 2429 of *Lecture Notes in Computer Science*, pages 53–65. Springer, October 2002.

[31] M. Mealling. A URN Namespace of Object Identifiers. IETF RFC 3061, February 2001.

[32] D. E. Millard, L. Moreau, H. C. Davis, and S. Reich. FOHM: A Fundamental Open Hypertext Model for Investigating Interoperability Between Hypertext Domains. In *Proceedings of Hypertext 2000*, pages 93–102, San Antonio, Texas, USA, May 2000.

[33] Y. Mu, C. Yu, T. Ma, C. Zhang, W. Zheng, and X. Zhang. Dynamic Load Balancing with Multiple Hash Functions in Structured P2P Systems. In

Proceedings of WiCOM 2009, pages 5364–5367, Beijing, China, September 2009.

[34] T. H. Nelson. Complex Information Processing: A File Structure for the Complex, the Changing and the Indeterminate. In *Proceedings of ACM 1965*, pages 84–100, Cleveland, USA, 1965.

[35] T. H. Nelson. Xanalogical Structure, Needed Now More Than Ever: Parallel Documents, Deep Links to Content, Deep Versioning, and Deep Re-use. *ACM Computing Surveys*, 31(4es), December 1999.

[36] M. C. Norrie. An Extended Entity-Relationship Approach to Data Management in Object-Oriented Systems. In R. A. Elmasri, V. Kouramajian, and B. Thalheim, editors, *Entity-Relationship Approach - ER '93*, volume 823 of *Lecture Notes in Computer Science*, pages 390–401. Springer, July 1994.

[37] M. C. Norrie, B. Signer, and N. Weibel. General framework for the rapid development of interactive paper applications. In *Proceedings of CoPADD 2006*, volume 6, Banff, Canada, November 2006.

[38] N. Paskin. Digital Object Identifiers for Scientific Data. *Data Science Journal*, 4:12–20, March 2005.

[39] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load Balancing in Structured P2P Systems. In M. F. Kaashoek and I. Stoica, editors, *Peer-to-Peer Systems II*, volume 2735 of *Lecture Notes in Computer Science*, pages 68–79. Springer, August 2003.

[40] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In R. Guerraoui, editor, *Middleware 2001*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer, October 2001.

[41] D. E. Shackelford, J. B. Smith, and F. D. Smith. The Architecture and Implementation of a Distributed Hypermedia Storage System. In *Proceedings of Hypertext 1993*, pages 1–13, Seattle, USA, November 1993.

[42] B. Signer, A. de Spindler, and M. C. Norrie. A Peer-to-Peer-based Distributed Link Service Architecture. Technical Report TR636, ETH Zurich, August 2009.

[43] B. Signer and M. C. Norrie. A Framework for Cross-media Information Management. In *Proceedings of EuroIMSA 2005*, pages 318–323, Grindewald, Switzerland, February 2005. IASTED/ACTA Press.

[44] B. Signer and M. C. Norrie. As We May Link: A General Metamodel for Hypermedia Systems. In C. Parent, K.-D. Schewe, V. C. Storey, and B. Thalheim, editors, *Conceptual Modeling - ER 2007*, volume 4801 of *Lecture Notes in Computer Science*, pages 359–374. Springer, October 2007.

[45] E. Sit and R. Morris. Security Considerations for Peer-to-Peer Distributed Hash Tables. In P. Druschel, F. Kaashoek, and A. Rowstron, editors, *Peer-to-Peer Systems*, volume 2429 of *Lecture Notes in Computer Science*, pages 261–269. Springer, October 2002.

[46] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *SIGCOMM Computer Communication Review*, 31(4):149–160, August 2001.

[47] S. Sun, L. Lannom, and B. Boesch. Handle System Overview. IETF RFC 3650, November 2003.

[48] A. van Dam and D. E. Rice. On-line Text Editing: A Survey. *ACM Computing Surveys*, 3(3):93–114, September 1971.

[49] R. Vanbrabant. *Google Guice: Agile Lightweight Dependency Injection Framework*. Apress, April 2008.

[50] R. J. Wieringa and W. de Jonge. The Identification of Objects and Roles - Object Identifiers Revisited. Technical Report IR-267, Faculty of Mathematics and Computer Science, Vrije Universiteit, December 1991.

[51] N. Yankelovich, B. J. Haan, N. K. Meyrowitz, and S. M. Drucker. Intermedia: The Concept and the Construction of a Seamless Information Environment. *IEEE Computer*, 21(1):81–96, January 1988.

List of Figures

List of Tables