Master's Thesis in Telematics
for the Award of the Academic Degree
Diplom Ingenieur
at
Graz University of Technology

Aspects of Interrelations in Distributed Component Systems

---

submitted by:

Stefan Thalauer

May 2004

Institute for Information Systems
and Computer Media

Supervisor: Univ.-Doz., Dipl.-Ing., Dr.techn. Klaus Schmaranz

Diplomarbeit aus Telematik
zur Verleihung des Akademischen Grades
Diplom Ingenieur
an der
Technischen Universität Graz

Aspekte wechselseitiger Komponentenbeziehungen in verteilten Systemen

---

vorgelegt von:

Stefan Thalauer

Mai 2004

Institut für Informationssysteme und Computer Medien (IICM)

Begutachter: Univ.-Doz., Dipl.-Ing., Dr.techn. Klaus Schmaranz

Acknowledgments

First of all I would like to thank Klaus Schmaranz who gave me the opportunity to accomplish this thesis within the limits of the Dinopolis project. He always gave me the support and advice that was needed to finish this work most accurately.

Secondly I would like to honor my teammate and fellow student Thomas Oberhuber for the great teamwork during the work on this project. Furthermore I would like to thank all members of the Dinopolis development team at the IICM (Institute for Information Systems and Computer Media), Graz University of Technology who supported my work on this master's thesis. Special thanks go to Klaus Schmaranz, Edmund Haselwanter, and Jürgen Malin for proofreading this thesis. Further I would like to thank Christof Dallermassl for many useful LaTeX tips which made my life a lot easier.

Very special thanks go to Margot for her love, patience and support during the last years.

Last but not least I want to thank my parents, Brigitte and Karl. They gave me the opportunity to study telematics and I always had their full support through all the years of my studies.

Abstract

One of the main problems of the information society we are living in, is the way how data is retrieved and managed. Thus a facility to model relationships between arbitrary chunks of data and a way to navigate through the information space is needed. These two challenges are solved in the Dinopolis distributed component middleware framework by the use of a highly sophisticated interrelation mechanism. In Dinopolis interrelations are uniquely addressable connections, relations, or dependencies between arbitrarily many objects or components of arbitrary type or parts of them.

This thesis discusses aspects of such an interrelation mechanism. A comparison with existing systems is given and the requirements on interrelations are worked out. It will be shown that the most important requirements are stability and robustness of interrelations against reconstruction of the information space. The key to stability and consistency is a sophisticated addressing mechanism. Therefore some considerations about addressing are presented.

Finally the technical concept and some design details of the interrelation mechanism in Dinopolis are shown.

Zusammenfassung

Eines der größten Probleme der Informationsgesellschaft liegt in der Art und Weise, wie Information verwaltet wird. Die Modellierung von Beziehungen zwischen beliebigen Teilen von Daten und die Navigation im Informationsraum sind deshalb immens wichtige Aufgaben eines verteilten Komponenten Systems. Diese Aufgaben werden in Dinopolis, einem verteilten Komponenten Middleware Framework, mittels sogenannter *Interrelations* gelöst. Solche Interrelations stellen eindeutig adressierbare Verhältnisse bzw. Abhängigkeiten zwischen Objekten oder Komponenten beliebigen Typs dar.

In dieser Arbeit werden Aspekte eines solchen Mechanismus behandelt. Bestehende Systeme werden miteinander verglichen und die Anforderungen an ein Interrelation Modul erarbeitet. Es stellt sich heraus, dass zu den wichtigsten Anforderungen Stabilität und Robustheit bezüglich Rekonstruktion des Informationsraumes zählen. Der Schlüssel zu dieser Stabilität und Konsistenz liegt in einem ausgeklügelten Adressierungsmechanismus, welcher deshalb genauer behandelt wird.

Abschließend werden die technischen Konzepte und einige Design Details des Interrelation Moduls in Dinopolis beschrieben.

I hereby certify that the work reported in this thesis is my own and that work performed by others is appropriately cited.

Signature of the author:

Ich versichere hiermit wahrheitsgemäß, die Arbeit bis auf die dem Aufgabensteller bereits bekannte Hilfe selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten Anderer unverändert oder mit Abänderungen entnommen wurde.

Table of Contents

Acknowledgments i

Abstract ii

Zusammenfassung iii

1 Introduction 1
- 1.1 Chapter Overview . . . 2
- 1.2 Motivation . . . 4
- 1.3 Related Work . . . 5

2 Background 6
- 2.1 Definitions . . . 6
- 2.2 The Dexter Hypertext Reference Model . . . 7
- 2.3 The World Wide Web . . . 8
- 2.4 XLink . . . 9
- 2.5 Hyper-G . . . 9
- 2.6 DINO . . . 10

3 Requirements 12
- 3.1 Types and Semantics . . . 12
- 3.2 Meta-data . . . 13
- 3.3 Stability and Consistency . . . 14
- 3.4 Logical Structure strictly separated from Addressing Structure . . . 15
- 3.5 Multidirectional Interrelations . . . 15
- 3.6 Multidimensional Interrelations . . . 16
- 3.7 Interrelations between Interrelations . . . 17
- 3.8 Adding Interrelations to Systems which do not support them . . . 17
- 3.9 System Immanent and Explicit Interrelations . . . 18
- 3.10 Operations on Interrelations . . . 19

4 Addressing, Naming and Navigation **21**
4.1 Definitions . . . 21
4.2 Addressing vs. Navigation . . . 22
4.3 Uniform Resource Identifiers . . . 24
4.4 PURL (Persistent URL) . . . 25
4.5 The Handle System . . . 26
4.6 DOI (Digital Object Identifier) . . . 26
4.7 DOLSA (Distributed Object Lookup Service Algorithm) . . . 27

5 The Dinopolis System Architecture **29**
5.1 Features of Dinopolis . . . 29
5.2 Dinopolis Objects . . . 30
5.3 Design Overview . . . 32

6 Interrelations in Dinopolis **34**
6.1 The Interrelation Model . . . 34
6.2 The Interrelation Object . . . 35
6.3 Endpoints of Interrelations . . . 36
6.4 Semantics of Endpoints . . . 39
6.5 Meta-data . . . 41
6.6 Persistence . . . 41
6.6.1 System Data Storage . . . 45
6.7 How the interrelation mechanism works . . . 46
6.7.1 Life-cycle Operations . . . 46
6.7.2 Administrative Operations . . . 47
6.7.3 Resolve Operations . . . 48
6.8 Implicit Interrelations . . . 48

7 Design Details **51**
7.1 A Use Case Scenario . . . 51
7.2 Software Requirements . . . 55
7.3 Submodules and Classes . . . 56
7.3.1 Endpoints . . . 57
7.3.2 Endpoint Container . . . 58
7.3.3 Interrelation Handler . . . 59

8 Processes and Algorithms **60**
8.1 Processes concerning the Interrelation Handler . . . 60
8.1.1 Get an Endpoint . . . 60
8.1.2 Get all Endpoints . . . 61
8.1.3 Get number of Endpoints . . . 61
8.2 Internal functions of *Dinopolis Objects* . . . 62

8.2.1 Adding an *Endpoint* to a *Dinopolis Object* 62
8.2.2 Removing an *Endpoint* from a *Dinopolis Object* 62
8.3 Detaching a *Dinopolis Object* from all *Interrelation Objects* 63

9 Conclusion and Outlook **64**
9.1 Conclusion 64
9.2 Outlook 64

References **66**

Glossary **69**

List of Acronyms **70**

Index **73**

List of Figures

5.1 A simple *Dinopolis Object* 31

6.1 Example of Interrelations between Dinopolis Objects 35
6.2 Schematic View of Endpoints 38
6.3 The Interrelations logical view 40
6.4 Interrelation Data treated as content 42
6.5 Interrelation extended by *Interrelation Object* part 43
6.6 Interrelation Data resides in Interrelation part 44
6.7 Example of a filesystem 50

7.1 Use Case Scenario hyperlink-interrelation 52

Chapter 1

Introduction

This master's thesis discusses the modeling and design of *Interrelations* between objects and components in distributed component systems. It results from the development process of the *Interrelation Management* module of the Dinopolis [1] distributed component middleware framework. I worked on this assignment with my fellow worker Thomas Oberhuber who also wrote his master's thesis (see [Oberhuber, 2004]). During the development of the use-case scenarios and at the beginning of the architectural design phase Gernot Höbenreich temporarily joined our team.

Dinopolis is a massively distributed componentware framework. It was been developed at the IICM (Institute for Information Systems and Computer Media)[2] by several researches in small teams under the leadership of Klaus Schmaranz. In [Schmaranz, 2002a] he presents an overall description of the Dinopolis framework and characterizes the main intentions of Dinopolis. These are platform and programming language independency, network transparency, robust and globally unique component addressing and the possibility of an easy integration of existing systems. Therefore the main features of Dinopolis are a slim and extensible kernel architecture, dynamically typeable components, full meta-information support for components and, last but not least, a highly sophisticated multi-directional interrelation mechanism.

The key characteristics of this interrelation mechanism in the Dinopolis framework can be outlined as follows:

- Arbitrary objects and components can be interconnected. There are no restrictions regarding their location on the net. It is also possible to interconnect parts of objects and components with interrelations. Moreover interrelations can also be interconnected with other interrelations.
- In the *Dinopolis System Architecture* addressing is strictly separated from navigation. Therefore *Interrelations* are used for modeling navigation through the object space.
- Since objects and components in the Dinopolis system are addressed by globally unique handles it is guaranteed that interrelations are robust against object

[1] `http://www.dinopolis.org`
[2] `http://www.iicm.edu`

movement. This means that object movement does not break the interrelation mechanism.

- Different types of interrelations are supported. For example simple hyperlinks can be modeled as well as a complex relation of some version controlled documents.
- Interrelations are multidimensional. This means that $n_1 : n_2 : \ldots : n_m$ dimensional interrelations are provided.
- Interrelations are multidirectional. This means that each end of an interrelation can be reached from any other end.
- It is possible to specify any meta-data information about these object interrelations.

1.1 Chapter Overview

Chapter 1, *Introduction:* Gives a short overview about this work, outlines the key features of the interrelation mechanism in the Dinopolis system and provides some reasons why it is necessary to develop a new interrelation mechanism. In addition some publications about related work on distributed systems are referred.

Chapter 2, *Background:* To be able to define the requirements of a sophisticated interrelation mechanism, it is necessary to get introduced to the terminology of this problem domain. Therefore the terms of use are defined in this chapter and existing technologies (hyperlinks in the WWW (World Wide Web)) and standards (XLink) are discussed. Finally two historic systems, named Hyper-G and DINO (Distributed Interactive Network Objects) are introduced.

Chapter 3, *Requirements:* In this chapter the requirements which an interrelation mechanism has to fulfill are outlined. Additional the limitations of traditional hyperlinks concerning this requirements are pointed out.

Chapter 4, *Addressing, Naming and Navigation:* In most of today's systems addressing, naming, and structure are totally mixed up. But separating navigation and addressing is the key to a consistent and robust interrelation mechanism. This chapter gives some considerations on addressing and navigation. The terms URN and object handles are introduced and finally an algorithm named DOLSA (Distributed Object Lookup Service Algorithm) is sketched.

Chapter 5, *The Dinopolis System Architecture:* This chapter gives a brief introduction of the Dinopolis system architecture. First of all the aims Dinopolis intended to

solve are outlined. After this one of the key parts of the Dinopolis system is introduced; the *Dinopolis Object*. Finally the main modules of the system architecture are sketched in this chapter.

Chapter 6, *Interrelations in Dinopolis:* They are uniquely addressable, multi directed connections, relations or dependencies between arbitrarily many *Dinopolis Objects* of arbitrary types or parts of them. In this chapter the technical concept of the interrelation mechanism of Dinopolis is described. It is shown that Interrelations are *Dinopolis Objects* themselves and *Endpoints* are introduced. Finally the background for the need of so called *Implicit Interrelations* is discussed and these *Implicit Interrelations* are specified.

Chapter 7, *Design Details:* Some selected design details of the *Interrelation Management* module of Dinopolis are presented in this chapter. A use case scenario is sketched. In this scenario a hyperlink is modeled by the use of an interrelation. Then software requirements are identified and the classes and submodules of a *Dinopolis Object* concerning interrelations are considered.

Chapter 8, *Processes and Algorithms:* Some selected interrelation specific processes and algorithms are outlined in this chapter. The considerations are centered on processes applying to *Dinopolis Objects*. Therefore operations called on the *Interrelation Handler* are discussed.

Chapter 9, *Conclusion and Outlook:* In the last chapter of this thesis the results of this work are summarized and future developments concerning interrelations are discussed.

As mentioned above, the *interrelation management* module was designed in teamwork by Thomas Oberhuber and me (Stefan Thalauer). The module was developed together, but we split the topics of our master's theses which are the result of the design process.

Thomas in his thesis [Oberhuber, 2004] deals with "Aspects of Structured Component Spaces in Distributed Systems". He focuses on the design of *Interrelation Objects*, especially on possibilities to give semantics to interrelations. Before this discussion he sketches a real world scenario to give a motivation why it is necessary to develop a new, highly sophisticated interrelation mechanism.

I worked out the general requirements (Chapter 3) and an inspection about aspects of addressing vs. navigation concerning interrelations (Chapter 4). Then I describe in detail the interrelation model (Section 6.1), the endpoint mechanism (Section 6.3) and the concept of so called *Implicit Interrelations* (Section 6.8) in Dinopolis. Furthermore I cover the problem how to make this information about connections persistent (Section 6.6).

Since we worked on the same project, some topics of our theses are overlapping. First of all we present similar technologies in the chapters that are dealing with the introduction of the problem domain. These technologies which are sketched in both

theses in some way, are the WWW (World Wide Web), XLink (XML Linking Language) and Hyper-G. Of course a brief introduction of the Dinopolis middleware framework is given and its history is presented in both theses. The design details are adapted from the design document of the interrelation management module [Oberhuber and Thalauer, 2004] and therefore they for sure look similar.

At the end of both theses some important processes and algorithms are presented (see [Oberhuber, 2004] and Chapter 8). They are again adapted from [Oberhuber and Thalauer, 2004] but in these chapters we focus on different topics. Thomas focuses on processes that are called on *Interrelation Objects* whereas my considerations are concentrating on processes applying *Dinopolis Objects*.

1.2 Motivation

One of the main problems of the information society we are living in is the way how data is retrieved and managed. Modern document-, information- and knowledge-management systems promise to handle the rising information flood. Therefore a facility to navigate through the information space is needed. Furthermore it should be possible to model relations between arbitrary chunks of data. In today's systems these two tasks are often performed by the use of links. Thus link management is an essential part of modern hypermedia systems. This fact is stated in [Lennon, 1997]:

> Although in many respects link management is the crux of hypermedia systems design, it is, unfortunately, the weakest aspect of many modern systems. Indiscriminate use of links, particularly in large distributed systems has led to tangles and mismanaged webs, which is frustrating for system managers and users alike.

From this it follows that the design of a sophisticated interrelation mechanism is not an easy but a very important task. Besides the above mentioned frustrating aspects some other fundamental problems of today's hypermedia systems come into play. Hyperlinks are often neither robust against object movement nor stay consistent when an interconnected object is deleted. Schmaranz points out in [Schmaranz, 2002a] one of the main reasons for this problem:

> In case of the Web there exist hyperlinks that represent pointers which are not robust and inline images that share the same problem. These two types of interrelations are not even part of the Web technology. In fact they are part of the HTML definition [...] the XLink recommendation is for sure a step into the right direction. Nevertheless, even in this recommendation interrelations are still treated as parts of documents rather than integral parts of the distribution platform.

Another reason for non robust, inconsistent links is the fact that in most systems addressing and navigation are totally mixed up. A sophisticated addressing mechanism and links which are not embedded in the content of interrelated documents are the key characteristics to solve these already mentioned problems. But additional problems can be identified. Hyperlinks are often one-dimensional and the facility to give them different semantics is often only rudimentarily implemented.

Starting from these considerations the requirements on a sophisticated interrelation mechanism are developed. These needs are covered in detail in Chapter 3. In contrast to this theoretical treating Thomas Oberhuber sketches in his master's thesis [Oberhuber, 2004] an imaginary but realistic real world scenario. Based on the resulting requirements of this scenario he derives the key features of the Dinopolis system. He focuses on aspects of structured component spaces in distributed systems.

After developing the requirements for an interrelation mechanism some observations about addressing and naming of objects are presented. These observations and some considerations about navigation aspects are discussed in Chapter 4. Then the cornerstones of the Dinopolis middleware system are outlined in Chapter 5. In Chapter 6 the interrelation mechanism of Dinopolis is introduced. Chapter 7 presents some selected design details and some interrelation specific processes applying to *Dinopolis Objects* are presented in Chapter 8. Finally the conclusions of this work are discussed in Chapter 9.

1.3 Related Work

The development process of Dinopolis just entered the implementation phase. Klaus Schmaranz identified in [Schmaranz, 2002c] the requirements that a modern distributed system has to satisfy. These requirements and the algorithm described in [Schmaranz, 2002a] for robust object lookup services lead to the current design of Dinopolis. Some other publications from members of the Dinopolis development team cover topics concerning Dinopolis. [Blümlinger et al., 2003c] deals with the object life-cycle management in Dinopolis, [Blümlinger et al., 2003a] describes how controlling access from applications to core modules of Dinopolis works and [Blümlinger et al., 2003b] describes how it is possible to change the subtypes of objects at runtime. And, last but not least, Edmund Haselwanter describes how component composition in Dinopolis works in his master's thesis [Haselwanter, 2003].

Chapter 2

Background

Before the requirements (see Chapter 3) for a highly sophisticated relation mechanism can be worked out, the used terms have to be clarified. Therefore some definitions are given in this chapter. To get an understanding about some problems that may arise during the development process of an interrelation mechanism the existing technologies and standards are outlined. The Dexter Hypermedia Reference Model is introduced. A short description of hyperlinks in the WWW (World Wide Web) is given. After this XLink (XML Linking Language) is discussed and the last two sections of this chapter introduce two systems which have been developed at the IICM, named Hyper-G and DINO.

2.1 Definitions

Hypertext The term Hypertext is introduced by Ted Nelson in the paper "A file structure for the complex, the changing and the indeterminate" [Nelson, 1965] in 1965. According to Nelson hypertext means:

> ... a body of written or pictorial material interconnected in such a complex way that it could not conveniently be presented or represented on paper.

In other words hypertext is non-sequential writing. Different chunks of data are connected in a certain way. This offers different navigation paths through the provided information. The connection of the data chunks is done by the use of hyperlinks.

But the idea of hypertext-systems is much older. It goes back to the article "As we make think" [Bush, 1945] published 1945. In this article Vannevar Bush describes a device called MEMEX (Memory Expander). MEMEX is a mechanical device which is an extension of the human brain.

Hypermedia is the extension of the term Hypertext. Audio and video are supported in addition to pure hypertext.

A good overview about the history of hypermedia and about some systems is given by Lennon in [Lennon, 1997] and Nielsen in [Nielsen, 1995]. The WWW is considered as showcase for link management in hypertext systems in Section 2.3 and Hyper-G in Section 2.5.

Nodes and Links are the essential elements of each hypermedia system. A node represents an atomic information entity. To understand the content of a node, no knowledge about the content of another node is required. This is essential for non-sequential reading. A node may be written text, an image or a video clip etc.

Two related nodes are connected by a link. Links can be directed or directionless. But links often are only pointers from a source anchor to a destination node. It is possible to "resolve" a link. This means that, in case of a directed link, the destination node is fetched for the user.

Transclusion (or Inclusion) is also introduced by Ted Nelson [Nelson, 1992]. Transclusions are one of the key features of Nelson's Project Xanadu. They allow to include objects by reference. This means that the included data is not copied and also does not have to be stored in two places. Therefore transclusions have some advantages compared with common links. Since data is stored only once, storage space is saved. The access to some quoted data is always redirected to the original location and therefore changes are updated automatically. The context of the embedded object is still available and has to be displayed. It is guaranteed that the original author is always quoted and eventually paid correctly.

Transclusions are implemented only rudimentarily in today's systems. One reason for the lack of implementations is the existence of many different heterogeneous systems [Krottmaier and Maurer, 2001]. This problem can be solved by a distributed component system with a sophisticated interrelation mechanism such as Dinopolis.

2.2 The Dexter Hypertext Reference Model

The Dexter Hypertext Reference Model [Halasz and Schwartz, 1994] (short: Dexter Model) has the goal to provide a basis for comparing hypertext systems. The Dexter Model is the result of two workshops. It is named after the location where the first workshop was held in 1988, the Dexter Inn in New Hampshire.

The Dexter Model divides a hypertext system into three layers. The *run-time* layer deals with the presentation of the hypertext and with user interaction. The *storage* layer describes the "data base" of a system. This data base contains a network of nodes and links. The third layer is the *within-component* layer which defines the content structure of the nodes.

The main focus of the model is on the *storage* layer. The "data base" consists of components. A component is an atom (text, image, etc.) or a link or a composition of components. An atomic component is comparable with the term node which is used in common hypertext systems. The term node was avoided by the authors of the Dexter Model because it was not clearly defined before. Components are addressable entities and therefore every component has to be identified by a UID (Unique Identifier). This UID has to be unique within the entire hypertext system. The *storage* layer defines two functions for retrieving components. The *accessor* function returns a component with a given UID. But sometimes the use of UIDs as addressing mechanism is not enough. Therefore a kind of indirect addressing is supported by the *storage* layer. The *resolver* function returns a UID for a certain *specification*. But this is sometimes not possible. Consider the case that a component specified by such a *specification* does not exist because of an editing process or a move operation.

A link in the Dexter Model represents a relationship between components. In general a link connects two or more components. An anchor has two parts. The *anchor id* is an identifier defined only locally concerning a component and stays constant even if a component is edited. The *anchor value* specifies some location or region within a component and is interpreted by applications. It is also changed if a component is edited by an application. The UID and *anchor id* pair guarantees a stable reference mechanism within the content. In addition it is possible in the Dexter model to define a *presentation* and *direction specifier*. The *presentation specifier* is used within the runtime layer to provide a way to make an anchor visible. The *direction specifier* is used to define whether the endpoint is the source or destination of a link. It is also possible to define bidirectional links or endpoints that are neither source nor destination of a link. Since in the Dexter model links are components and therefore they have a UID it is also possible to have a link which interconnects two other links.

In addition to the already mentioned "data base", the storage layer also defines some operations. It is possible to create, delete and modify components. Furthermore it is possible to retrieve a component or to resolve a link.

In the Dexter model multidimensional links are possible. Furthermore links to links are also possible. These two facts and the facility to create arbitrary (non cyclic) compositions of components are the strengths of the model. One of the biggest limitations of the Dexter Model is the fact that it requires links and other system information to be completely available at all time [Dyke and Sollins, 1995]. This is an unrealistic requirement especially for distributed systems.

2.3 The World Wide Web

The WWW [Berners-Lee et al., 1994] is the most commonly used distributed hypertext system of today. It was developed at the CERN in the late 1980s. Its success results from the use of a relatively simple document format (HTML) and a simple network protocol

(HTTP). Hence this provides a simple way to exchange information in a human readable format. Furthermore exchange of information is platform independent.

In addition to the definition of the used format and network protocol the WWW also defines an addressing system (URI). Some considerations about this addressing format and addressing in general can be found in Chapter 4.

In the WWW relations between documents are modeled by links. These hyperlinks are also used for navigation issues between documents and across the document space. A big drawback of the design of the WWW is the fact that hyperlinks are defined in the document format [Berners-Lee and Connolly, 1995]. According to this recommendation a hyperlink interconnects two anchors which are identified by an identifier. Such links in HTML (HyperText Markup Language) are only unidirectional. The results of these circumstances are discussed in detail in Section 3.3 and Section 3.5.

2.4 XLink

XLink (XML Linking Language) is defined in [DeRose et al., 2001]. A detailed description about XLink and XML related topics can be found on the web-page of the World Wide Web Consortium[1].

As the name implies, it is responsible to manage links between XML documents. It is possible to describe links similar to hyperlinks known from the WWW. But is also possible to model more sophisticated links with XLink. Therefore it is possible to create links with more than two endpoints. Links are always bidirectional and they are directed. It is further possible to assign certain meta-data to a link. But the biggest advantage compared with traditional hyperlinks in the WWW is the fact that XLink allows to store links on locations separate from the linked documents.

Interconnected documents are identified by URIs. Note that Chapter 4 gives a detailed discussion about aspects of addressing, including an introduction of the URI schema.

As mentioned in Section 1.2 XLink is an enhancement of hyperlinks known from the WWW, but it also has the main drawback that link information is treated as part of documents.

2.5 Hyper-G

Hyper-G (now Hyperwave[2]) is a distributed hypermedia system. It was developed at the IICM, Graz University of Technology, because all so called "first generation hypermedia systems" (e.g. the WWW) did not fulfill the major requirements for a hypermedia system. [Andrews et al., 1994]. Hyper-G was introduced as a so called "second generation

[1] `http://www.w3.org/`
[2] `http://www.hyperwave.com`

hypermedia system". A detailed description of the system architecture can be found in [Maurer, 1996]. The key features of Hyper-G are outlined in this section. Of course the focus lies on aspects of interrelations between objects.

In Hyper-G links are defined by source and destination anchors. The source anchor is the starting point of a hyperlink. The destination anchor defines the endpoint of a hyperlink. One of the key characteristics of Hyper-G is the fact that links are stored separately from documents. The link information is stored in external databases. The resulting advantages in comparison with stored inks within documents can be summarized as follows: Bidirectional Links are supported by Hyper-G. Therefore it is possible to navigate backwards from destination to source and it is also possible to generate link maps that show all links of a certain document. Link consistency is guaranteed because of bidirectional links. For example, if a document is deleted, links pointing to it will be removed. Links in Hyper-G support attributes. It is possible to query for links that have certain attributes. Hyper-G supports links with different access rules. And last but not least documents do not have to be modified if a link is created or altered.

Another key feature of Hyper-G is the used data model. The node-link model is extended by special link types and a new object class, the *collection* is introduced. A collection may be compared with directories in a file system. It is a composite object and contains other objects. These objects are documents or other collections. With this definition it is possible to form a directed acyclic graph of collections, the collection hierarchy. This collection hierarchy can be used for navigation.

It can be seen that Hyper-G satisfies most of the requirements proposed in Chapter 3. It guarantees consistency of the links and the collection hierarchy in the local case, when both endpoints of a link reside on the same server. But it does not guarantee consistency for links between documents which are stored on different servers. In this case only "weak" consistency is guaranteed. That means the hyperweb may be inconsistent for a certain period of time if a document is deleted [Kappe, 1995].

2.6 DINO

DINO (Distributed Interactive Network Objects) is a Java library which was developed by a team of researchers at the IICM, Graz University of Technology. The development process started in 1997. The inital aim of DINO was to develop a messaging system [Freismuth et al., 1997]. In 1999 DINO was presented at the CEBIT in Hannover as the core of the MTP (Medical Telematics Platform) prototype. The MTP project has been developed in cooperation with the DLR (German Aerospace Center).

But DINO did not scale well for large systems and enhancement of the system was nearly impossible because of the system architecture. This made it necessary to perform a complete redesign of the system. With the lessons learned during the development phase of the first version of DINO and with new intentions in mind the development of Dinopolis began. The new system was not only a simple messaging system. It was

intended to design a massively distributed component middleware framework. Thus Dinopolis has its origin in the DINO library. An introduction of the characteristics of the Dinopolis system architecture, especially focused on aspects needed for interrelations, is given in Chapter 5.

Chapter 3

Requirements

A highly sophisticated interrelation mechanism between objects has additional needs in comparison with a traditional hyperlink mechanism. Furthermore it should be clear that traditional hyperlinks have some fundamental limitations which made them unemployable for a modern distributed component system.

In this chapter these limitations are considered and the requirements that an interrelation mechanism in a modern distributed component system has to satisfy, are developed. Additionally a comparison with the link mechanisms in traditional hypermedia systems is given.

3.1 Types and Semantics

interrelations between objects may have different semantics. For example a parent-child relation between two objects has a different meaning than an annotation of an object. Randal Trigg proposes in his Ph.D thesis [Trigg, 1983] a list of different link types. In Trigg's taxonomy of link types he first distinguishes between two main categories. *Normal* links which are connecting parts of scientific documents and *Commentary* links that are attaching statements to a document. Examples for Trigg's *Normal* links are *Citation source*, *Example* or *Explanation*. *Comment* or *Critics* represent *Commentary* links in Trigg's taxonomy of links. Other interrelation types in hypertext systems such as glossary, footnote, appendix etc. are thinkable. With typed interrelations it is easier to organize information and navigate through the object space. The following requirement can be derived:

⇒ **It must be possible to assign different types to interrelations.**

Semantically, typed links are partly used in the WWW [Berners-Lee et al., 1994].The historic recommendation of HTML 2.0 [Berners-Lee and Connolly, 1995] already included the `<A>` element which denotes an anchor or destination of a link and the `<LINK>` element for defining rudimentary typed document relations:

> The `<LINK>` element is typically used to indicate authorship, related indexes and glossaries, older or more recent versions, document hierarchy, associated resources such as style sheets, etc.

For both elements the W3C recommends the attributes `rel` (relationship to link) and `rev` (relationship from link) to define semantics. Supported link types are `prev` and `next` for sequences, `start` and `up` for hierarchy, `contents` and `index` for navigation. Common link types as `glossary`, `copyright`, `appendix`, `help` and `author` are also supported, as well as the possibility to refer to a related object with the link type `bookmark`. It is a remarkable note, that graphical web browsers did not completely support these attributes of the `LINK` element until the end of 2001 [1] [Tekelenburg, 2004].

3.2 Meta-data

In addition to the already mentioned different types of interrelations it may be useful to attach some other information to an interrelation or to assign certain attributes to it. Such information can be the name of the author, a timestamp, labels or some describing keywords. For example in HTML the `title` attribute exists which may provide a title for a hyperlink. This attribute is commonly displayed as "tooltip" in graphical web browsers. This meta-information and attributes are represented by meta-data, leading to the following requirement:

⇒ **Interrelations have to support meta-data**

In addition meta-data should not only be attachable to an interrelation, but it should also be possible to attach meta-data to single endpoints. Some of this meta-data is attached explicitly (e.g. author), but other meta-data may be generated and attached implicitly. For example such implicit meta-data may be the content-type of an interconnected object.

In addition to the ability to have types of relations, meta-data attached to relations give a supplementary possibility to distinguish different interrelations. Furthermore it also is possible to implement a search facility based on interrelations' meta-data. For example someone would like to find the following:

- All interrelations which are newer than two weeks.
- All interrelations which are created by the same author.

An outline about "link attributes and structure-based query" and some application areas for structure-based query is presented in [Bieber et al., 1997].

[1] *Mozilla* supports `<LINK>` since version 0.9.5, *Opera* since version 7 and the *Internet Explorer* still needs a third-party tool to support it

3.3 Stability and Consistency

One of the most annoying problems in hypertext systems are links, which cannot be fetched. Perhaps you know this situation when you get a "404 Error" response from a web server if the destination document of a hyperlink is not available. The reason for such dangling links can be:

- The destination object has been deleted.
- The destination object has been moved to another place on the net.
- The destination object has been renamed.

Furthermore it may occur that an object is eventually replaced by an other one. In this case users usually have no chance to recognize this fact, because they will not get an error response from the web server.

As a result of these considerations the following two requirements are essential for a highly sophisticated interrelation mechanism:

⇒ **Interrelations have to be robust against object movement**

⇒ **The interrelation mechanism has to be consistent, even if an endpoint is deleted.**

Let us now analyze which design issues of today's systems are responsible for the above mentioned problems.

One problem is that links are often treated as content of an object. For example hyperlinks are integrated in the HTML recommendation (see Section 3.1). If a referenced document which lies on an external server is moved to another place, the content of the document including this link has to be modified. Since often the required access rights are not present it is not possible to alter this document. The solution for this problem would be not to store the links internally in the documents but somewhere externally. For example the recommendation of XLink allows links which are not part of the content of the referring document. Another approach is the use of external link databases to organize relations between objects. Using link databases, some additional work is required to guarantee link consistency [Kappe, 1995]. The Hyperwave Information Server [Maurer, 1996] and an early version of DINO [Blümlinger, 2000] for instance use such external link databases. It is mentioned that externally stored links have to be included in the documents before they are delivered to the user.

Another reason for this "dangling links syndrome" is the fact that links are often only pointers to the destination object. If such one-directional links are used, it is difficult to notice whether a destination object is moved to another place or whether it is deleted. In server-client architectures it may be possible to inform users that the destination

of a link has changed. For example the standard document [Berners-Lee et al., 1999] defining HTTP (HyperText Transfer Protocol) includes some "Status Code Definitions" which are used to inform the user agent about the status of his request. If a document has moved to another place the server may answer an HTTP GET request with 301 (Moved Permanently). The status code 404 (Not Found) and the status code 410 (Gone) indicate that a requested resource is "intentionally unavailable." This approach has the drawback that it only notifies the requesting user about the changed destination of a link. However, the author of the document including this broken link is not informed in this case. Therefore a better solution for this problem is needed. This solution is shown in detail in Section 3.5.

3.4 Logical Structure strictly separated from Addressing Structure

There often isn't a separation between addressing and navigation. Let us have a closer look at this aspect. We will see that this may be a major problem for interrelations concerning the requirements on stability and consistency. Furthermore an important requirement for a modern distributed system will be derived.

Objects can reside on different systems distributed across the net. Besides the fact that it should not make a difference for users if they refer to a local or remote object, the protocol schema would not be part of an object's address too. Consider the case that a document first may reside on an FTP-server and then this server is replaced by an HTTP-server. If the protocol is part of the address the object will not be reachable anymore, even if the object's path has not changed. The same problem occurs if the hierarchical structure of a location is encoded in the address of an object. An object stored in a filesystem is not reachable under its original address if for example it is moved to its parent directory. This leads us to the very important requirement on interrelations:

⇒ **Addressing has to be strictly separated from Navigation**

Since most hypermedia systems do not distinguish between the address of an object and its name, they are not able to provide consistent and stable links. Remember the URL-based address mechanism of the WWW. Consistency and robustness of object-addressing are not only an important requirement on interrelations, these aspects are also essential for a massively distributed system's architecture. Therefore a detailed discussion about aspects of addressing and navigation is given in Chapter 4.

3.5 Multidirectional Interrelations

As seen above, one potential reason for inconsistent interrelations are one-directional interrelations. Thus, by using multidirectional interrelations it is possible to notify all

objects referring to a certain object which have been moved or deleted. Furthermore it is possible to remove interrelations if they are irrelevant because the referred object has been deleted. These steps can be performed automatically. Therefore the following requirement can be proposed:

⇒ **Interrelations have to be multidirectional**

From this requirement follows that it is possible for the system to display all links referring an object. In other words starting from an object it is possible to reach all other objects which are interconnected by an interrelation. This may be essential if navigation is done with interrelations. In this case it is easy to provide required navigation paths (e.g forward, backward, up ...) without performing a search operation over all objects.

Now let us have a look how this is solved in today's systems. Since in most of today systems links are two-dimensional (i.e. 1 : 1) only the situation concerning bidirectional links is considered. The Hyperwave Information Server provides bidirectional links, as well es an early version of DINO does. The currently very popular collaborative software Wiki provides a "back link" mechanism to detect all internal links which point to the current document. But this "back link" mechanism is implemented by searching the underlying database. More precisely, in most Wikis different pages are identified by their titles in CamelCase. Therefore links are differentiated easily from standard spelling and finding all back links is implemented by a simple search for the titles of the current page in all other pages.

3.6 Multidimensional Interrelations

With hyperlinks it is only possible to model 1 : 1 interrelations. This can be seen by the fact that a hyperlink has two ends, the anchor node and the destination node. It should be clear that such 1 : 1 interrelations cover only a small spectrum of thinkable multidimensional relation scenarios. For instance two-dimensional interrelations may be:

1:1 An interrelation with one source and one destination. As already mentioned this case is similar to the link definition known from HTML.

1:n An interrelation with one source and several destinations (e.g. children within a hierarchy, one parent has several children).

n:1 An interrelation with several sources and one destination (e.g. a citation of a fundamental paper).

0:n An interrelation with no source and several destinations (e.g. a bookmark)

n:m This is the general case, an interrelation with several sources and destinations (e.g. interconnecting two version controlled objects).

All these cases can be modeled with an $n : m$ interrelation. Certainly it is possible to emulate $n : m$ interrelations by using many $1 : 1$ interrelations, but this may cause some avoidable overhead. However, sometimes it is even not possible to model all thinkable and meaningful relationships between objects with such an $n : m$ interrelation mechanism. To solve this problem and to be able to model universally valid relationships between arbitrary objects, a distributed component system should provide an $n_1 : n_2 : \ldots : n_m$ interrelation mechanism. Therefore a highly sophisticated interrelation mechanism has to satisfy the following:

⇒ **Multidimensional interrelations have to be provided**

3.7 Interrelations between Interrelations

As seen in the previous section, it should be possible to have interrelations between arbitrary objects. There's more to it than that. Sometimes an interrelation may be of interest. Just think of an annotation of a link in a collaborative system. Therefore it will be very useful to have interrelations which are attached to other interrelations. From this follows the next requirement:

⇒ **Interrelations between interrelations have to be supported**

As a result of this requirement it is possible to model a link to a destination that itself represents a hyperlink between other objects. As seen in Section 2.2 [Halasz and Schwartz, 1994] in the Dexter Model such interrelations between interrelations are supported. Additionally another important aspect comes into play. Interrelations always interconnect some addressable entities. Because of this fact and to be conformant to the requirement that interrelations between interrelations have to be provided, the next fact can be proposed:

⇒ **Interrelations also have to be addressable**

An outcome of this fact is that interrelations have to be addressable objects, respectively components. This circumstance makes sense because a common treatment of addressable objects is guaranteed.

3.8 Adding Interrelations to Systems which do not support them

One major requirement on a modern distributed middleware framework is that it is possible to embed and combine arbitrary existing systems, such as file-systems, databases,

and web-servers. [Schmaranz, 2002c]. Thus interrelations are not supported by all embedded systems. Consider a database that does not allow setting links between the records in it. Another example are links between audio and video-streams. This has the following consequency:

⇒ **Systems that do not support a certain kind of interrelation or no interrelations at all may be enhanced with this functionality**

It should be clear that this can be achieved storing such interrelations somewhere external. Additional the middleware system has to provide a possibility to store this external managed interrelations.

3.9 System Immanent and Explicit Interrelations

As mentioned above it has to be possible to embed existing systems in a middleware framework. In addition to the last requirement, further aspects come into play.

Consider a filesystem which organizes files in directories.

- It is responsible for the hierarchical organization of files and provides a way for navigation. But addressing and navigation are often totally mixed up
- Some kind of symbolic links are supported by most filesystems.

These system internal dependencies of files and directories should also be modeled by interrelations. Since these relationships are already present in the existing system and not generated explicit within the middleware framework these interrelations are called *Implicit Interrelations* and it follows:

⇒ ***Implicit* Interrelations have to be supported.**

On the other hand there are interrelations which can be set explicitly from users or applications. These interrelations are called *Explicit Interrelations*. When speaking about interrelations, explicit interrelations are meant. It should be clear that there has to be a way to distinguish between explicit and implicit interrelations. Furthermore all requirements proposed so far are valid for implicit interrelations with a few exceptions:

- Implicit interrelations are generated by embedded systems. Therefore they are not objects respectively components in the sense of componentware. From this it follows that it is not possible to have explicit interrelations to implicit interrelations.
- Meta-data for implicit interrelations is only provided at the time when they are supported by the embedded system.

- Consistency of implicit interrelations may not be guaranteed if it is outside the control of the middleware framework. Consider again symbolic links in a filesystem. These are unidirectional pointers to some place in the filesystem. This means that a move operation on the destination will cause a symbolic link that points to nowhere.

These restrictions are avoided if each implicit interrelation is converted into an explicit one when the external system is embedded. At a first glance this seems to be a satisfying solution for the problem, but in reality it is not. Explicit interrelations have to be addressable and therefore this approach would not scale well, because for every system immanent interrelation a globally unique handle is generated. For example browsing a filesystem and the resulting automatic generation of explicit interrelations may cause waste of globally unique handles and furthermore this would cause performance problems. Nevertheless it has to be possible to convert an implicit interrelation to an explicit one.

3.10 Operations on Interrelations

To be able to work with Interrelations some operations have to be provided by the Distributed Component System:

Create Interrelation: There must be an operation which allows to create a new interrelation between two arbitrary objects.

Delete Interrelation: There must be an operation which allows to delete an existing interrelation. This interrelation can be an explicit interrelation and it can connect arbitrary objects. It has to be assured that all objects attached to the Interrelation are notified about the pending deletion request. If deletion of an interrelation would lead to an inconsistent state, it has to be prohibited or delayed.

Attach an Endpoint to an Interrelation: There must be an operation which allows to attach an object to an interrelation.

Detach an Endpoint from an Interrelation: There must be be an operation which allows to detach an object from an interrelation. It has to be assured that this object attached to the interrelation is notified about the pending detach request. If this detach process would lead to an inconsistent state, it has to be prohibited or delayed.

Resolve an Interrelation: There must be an operation which allows to resolve an interrelation. But a resolve operation is not always trivial and an interrelation can provide various resolve operations depending on its type.

Consider as an example a $1 : 1$ interrelation of type hyperlink. Starting from an object that represents the source of that hyperlink, a resolve operation has to return the endpoint representing the destination object. A more complex example would be an $n : m$ dimensional multilingual interrelation. In this case for a given source object in a specific language the resolve operation should return the corresponding destination object which has the same language. This means that a resolve request from a german document should also return a german document.

A further valid resolve operation would return all endpoints that are attached to this *Interrelation Object*.

The operations presented here are the most basic operations. Other operations dealing with meta-data or providing some functionality to generate back links are also thinkable. Furthermore operations for storing and loading interrelations have to be provided.

Chapter 4

Addressing, Naming and Navigation

Interrelations have to be robust against object movement and consistent in respect to object deletion. These two requirements are proposed in the previous chapter (see Section 3.3). They also are fundamental for a distributed component framework. The key to stability and consistency is a sophisticated addressing mechanism. This addressing mechanism is responsible for handling and resolving addresses of objects. This chapter deals with considerations about addressing, naming, navigation, and structure of objects in distributed component systems. The terms of use are defined and requirements of an addressing mechanism are sketched. Transparency aspects concerning network, persistence, protocol, schema, and location are discussed. Some problems of existing systems are pointed out on the base of the addressing system of the WWW. Then the PURL (Persistent URL) system, the "handle system", and the DOI (Digital Object Identifier) System which is an implementation of the 'handle system" are introduced and compared.

Finally the addressing mechanism of the Dinopolis distributed component framework is presented. It is named DOLSA (Distributed Object Lookup Service Algorithm) and guarantees robustness and stability as well as scalability concerning object movement.

4.1 Definitions

In most of today's systems addressing, naming, and structure are totally mixed up. This is very misleading and therefore the terms of use are defined:

Name is a label to an object. It is used to distinguish between different objects. Furthermore it has to be independent of the physical location of an object.

Address specifies the physical location of an object. An address has to be unique to be able to resolve it. The address changes if an object has moved from one physical location to another physical location.

Structure of the object-space is defined by the relationships between the objects. These interrelations between objects are used for navigation through the object-space.

Resource is defined as *"anything that has identity"* [Berners-Lee et al., 1998]. Examples are documents, images or a collection of other resources. Resources in the WWW are retrieved by identifiers such as URIs.

Identifier is used to refer to a resource(also [Berners-Lee et al., 1998]). An identifier is often a string of characters.

Handle is a unique name for digital objects and other Internet resources. It is defined as an identifier for an object that is independent of its physical location. It must be possible to resolve a handle unambiguously. Therefore a bijective mapping between handles and objects has to exist. This resolve process can be made via a lookup-service.

4.2 Addressing vs. Navigation

A distributed component system has to provide ways to navigate through the object space. Parunak proposes in [Van Dyke Parunak, 1989] several navigational strategies. According to Parunak these are:

Identifier Strategy assigns a unique identifier to each entity of interest. By itself, this strategy degenerates to exhaustive search.

Path Strategy provides a procedural description of the way to the target. This strategy uses the identifier strategy to describe the entities and is only useful if the number of arising nodes is typically much less than the total number of nodes.

Address Strategy the address of the target is provided. It is only needed to resolve this address to get to the desired target.

In addition he identifies two more strategies, the *Direction* and *Distance Strategy* , but these are not relevant for information systems. It should be clear that the address strategy needs robust addresses of object locations to work well. But in today's systems addresses are often not robust. Two reasons can be identified for this circumstance:

- Structure of object-space is part of the address

  Most of today's systems are organized hierarchically. Furthermore this hierarchical structure is represented in the address space. Consider a filesystem where files and directories form a hierarchical structure. The location of a file is identified by the path to it (e.g. `/tmp/foo/bar.txt` on a unix-file system). It should be clear that a change of the structure of the filesystem or a change of the name of one directory would change the address of this object.

- Addressing and naming of objects is totally mixed up. The name of an object is often part of the address. Consider again a filesystem. If the name of a document is changed, then also its filename will be changed. But this results again in a change of the address of this object.

A solution for this problem is a strict separation of addressing, naming and structure. In addition this separation is also a requirement to obtain protocol and schema transparency [Schmaranz, 2002c]. Protocol and schema transparency are besides network, persistency, and location transparency the keys to distribution. Schmaranz defines transparency as a kind of hiding complexity by encapsulating it on different abstraction levels.

Two important design issues can be derived from the requirement for separation. Handles are used to identify objects. These handles are independent of the physical location of the object. Therefore a bijective mapping must exist between the handle and the object to provide a way to resolve a handle unambiguously. As can be seen afterwards it makes sense to use a lookup-service for the resolve operation of this mapping. Furthermore handles should not contain any name information which can be interpreted by humans [Blümlinger and Dallermassl, 2002]. This name information should be added to an object as meta-data. The second important design issue relates to the way navigation is provided. Navigation through the object-space has to be done by the use of interrelations. The main focus of this thesis lies on the development of such an interrelation mechanism. Therefore a description of the detailed design of a highly sophisticated interrelation mechanism can be found in Chapter 6. The use of handles guarantees a robust addressing mechanism. This applies only for the case when the hierarchical structure is changed or the meta-data representing an object's name is altered. But handles do not guarantee stability against object movement. Consider the case when an object is moved from one server to another. Since the handle is independent of the object location, the addressing mechanism would break. It should be clear that the mapping between the handle of an object and the new location of this object has to be updated in some way to provide a stable addressing mechanism. There are two main approaches to solve this problem. The first approach is a forwarding algorithm. Whenever an object is moved to another physical location a "forward" object is placed at the orignal location which refers to the new location. This approach was implemented in a previous version of Dinopolis to provide stable relations. It is described in detail in [Blümlinger, 2000]. But this approach has two main drawbacks. It does not scale well for frequent object movements. It is possible that long "forwarding chains" occur due to frequent object movements. As a result the response time is weak. Furthermore if a server goes offline it would break this "forwarding chain" and a moved object will not be reachable until the server goes online. This may cause inconsistency.

The second approach is a lookup-service algorithm where entries in the lookup-service are updated automatically whenever an object has been moved to another location. A naive implementation will have problems with scalability and will also provide only poor response times. Therefore an algorithm (DOLSA, see Section 4.7) was developed for

Dinopolis. It combines the advantages of both approaches while avoiding their drawbacks.

4.3 Uniform Resource Identifiers

Besides the network and document protocol the WWW defines an addressing system [Berners-Lee et al., 1994]. This addressing system defines a URI (Uniform Resource Identifier). It is a compact string of characters that is used for identifying a resource. As the name suggests identifiers in this addressing system are "uniform". This means that it is possible to refer to objects which are accessed by different network protocols such as FTP (File Transfer Protocol) or HTTP in a uniform way. This addressing system was designed to be able to handle existing network protocols. Another important design aspect was easy extensibility of new network protocol types. The addressing system of the WWW is defined in [Berners-Lee et al., 1998]:

URI (Uniform Resource Identifier) is used for identifying resources. It can be a location, a name, or both a location and a name. The syntax of a URI depends on the used schema. It contains the name of the scheme followed by a colon and then the scheme specific part. The scheme specific part is a string whose interpretation depends on the scheme. Examples for common used URIs are the mailto-scheme for electronic mail addresses or HTTP scheme for HyperText Transfer Protocol services. Some URI schemes support a hierarchical naming system. The hierarchy of the name is denoted by a forward slash ("/") delimiter separating the components in the scheme.

URL (Uniform Resource Locator) identifies a resource via their address instead of the name of this resource. URLs form a subset of an URI naming schema. A URL consist of the following parts: The network protocol (e.g. HTTP or mailto) to access this resource. The host address or domain name where this resource is stored and the path or file name of this resource on the host machine. For example `http://www.dinopolis.org/index.html` is a valid URL (Uniform Resource Locator).

URN (Uniform Resource Name) provides a globally unique and persistent identifier for a resource. URNs form, as well as URLs, a subset of an URI naming schema. A URN (Uniform Resource Name) consists of the string "`urn`" and a namespace identifier followed by a namespace specific string (e.g. `urn:foo:a123,456`). The syntax of URNs is defined in [Moats, 1997].

The most frequent used URIs in the WWW are URLs. But URLs have several drawbacks. First of all it is possible that two ore more URLs refer to the same resource or object. Consider a document stored on different servers or this document can be retrieved by different access protocols such as HTTP or FTP. Therefore it is not possible

to compare two objects by the use of their URLs. A second problem occurs if the domain name of the host changes or the document is moved to another host with a different domain name. In this cases the URL of the resource changes and therefore URLs are not stable against object movement. Furthermore a change of the access protocol also changes the address. Consider a text-file which is first accessed via a FTP and then the protocol is changed to HTTP. The third major problem of the URL addressing mechanism is that the hierarchical file structure is often mapped to the URL. Restructuring of the filesystem then leads to a change of the URL. The conclusion of this considerations is the fact that URLs are neither consistent nor stable. At a first glance it seems that a PURL (Persistent URL) (see Section 4.4 for details) or a URN (Uniform Resource Name) are possible solutions for this problem.

A URN is a persistent label of a resource. Several requirements on URNs are summarized in [Sollins and Masinter, 1994]. It is a name with a global scope. This means that a URN does not imply a location. A URN has to be globally unique and therefore the same URN cannot be assigned to different resources. Furthermore it should not be reused. If an object is deleted the URN of this object should not be assigned to another resource. This guarantees consistency and a permanent lifetime of URNs. Existing name systems (e.g. ISBN (International Standard Book Number)) are supported by the URN schema as well as future extensions of the scheme. Since URNs represent names of locations the corresponding addresses have to be determined if such an URN is requested. Most often the address of a location which is identified by a URN is represented by a URL. Therefore the URN has to be resolved in some way. In other words the URN is translated into the corresponding URL.

Two systems implementing the URN recommendation are introduced in Section 4.5 and in Section 4.6.

4.4 PURL (Persistent URL)

The PURL Service[1] is a HTTP redirections service. It was introduced in [Shafer et al., 1996] as *"a naming and resolution service for general Internet resources"*. It follows the URN concept. Generally spoken a PURL is a Persistent URL which provides a way for assigning a name to an object which stays stable even if the object is moved to a different location. The PURL service relies on indirection. This means that a PURL doesn't point directly to an Internet resource. It points to an HTTP resolution service. This resolution service assigns a URL to the PURL (Persistent URL) an returns this URL to the requestor.

But the PURL service has several drawbacks [Stone, 2000]. First of all it relies on HTTP. On the one hand it needs an HTTP server as a host which is responsible for resolving the PURL. On the other hand a PURL always represents an HTTP URL. A second problem of the PURL service is the fact that it relies on DNS. If the domain

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

name of a resolution server changes, the PURL changes too and therefore it should be clear that in this case a PURL isn't consistent anymore.

Besides these drawbacks there are some other limitations which make the PURL Service unsuitable for a distributed component framework. Names of resources (PURL) are defined by humans. As mentioned above this is not a good solution. Furthermore PURLs are not updated automatically. This task has to be performed by the maintainer of the resolve server.

4.5 The Handle System

According to [Han, 2002], the Handle System[1] is *"a comprehensive system for assigning, managing, and resolving persistent identifiers, known as "handles", for digital objects and other resources on the Internet."* The Handle System was developed by CNRI (Coporation for National Research Initiatives). It is a protocol specification and there exists a reference implementation of it. The Handle System makes it possible for a distributed computer system to store handles of resources and resolve those handles to locate and access the resources. It provides persistent handles over the change of the resource location. Handles in the Handle System can be used as URNs. They are globally unique within the Handle System. Furthermore can a handle refer to multiple instances of a resource

A handle consists of two parts. The prefix represents the naming authority and the suffix represents a local unique name concerning the naming authority. Suffix and prefix are separated by a forward slash. The next section introduces DOI (Digital Object Identifier) System which is an implementation of the handle system.

4.6 DOI (Digital Object Identifier)

According to [DOI, 2004], the DOI (Digital Object Identifier) System[2] is *"a system which provides a mechanism to interoperably identify and exchange intellectual property in the digital environment"*. It is a implementation of the Handle System. A DOI is a persistent identifier and furthermore it is an implementation of URI [Berners-Lee et al., 1998]. It is a standard for online content identification and governed by the International DOI Foundation.

Objects which are identified by a DOI are called entities. A DOI is rather a name of an entity than an address. The use of DOIs is not limited to the WWW. Rather it can be used on every network.

A DOI consists of an alphanumeric string which is assigned to an entity as a label. Furthermore some descriptions (meta-data) can also be assigned to the entity. The DOI

[1] `http://www.handle.net`
[2] `http://doi.org`

has to be resolved in some way to get the corresponding resource where it is located. This resolve process has to ensure persistency and is done by the Handle System (see Section 4.5). The alphanumeric string consists of two components, called prefix and suffix. The DOI prefix also consists of two components and always starts with "10." This distinguishes a DOI from any other Handle System implementation. The second element of the prefix is a string that is assigned to the organization that wants to register the DOI. The DOI suffix identifies the entity uniquely to a given prefix. It is separated by a forward slash from the DOI prefix. A DOI suffix can be any alphanumeric string. This can be any sequential number or an existing identifier (e.g. ISBN).For example **doi:10.1000/186** [1] is a valid DOI.

It should be clear that the combination of prefix and suffix has to be unique otherwise the mechanism would not work correctly. Therefore unique prefixes have to be assigned to organizations. These organizations therefore have to ensure unique suffixes. However, internal checks for uniqueness are performed at the registration process by the DOI System. A unique identifier should not be reused in the DOI system.

4.7 DOLSA (Distributed Object Lookup Service Algorithm)

DOLSA (Distributed Object Lookup Service Algorithm) is the addressing mechanism of the Dinopolis middleware framework. The reason for the development of DOLSA is the lack of existing addressing mechanisms which are consistent and robust against object movement. The few mechanisms that are implemented do not scale well. DOLSA was introduced and described in [Schmaranz, 2002b] and Schmaranz states about the motivation of developing a new addressing mechanism:

> ... it became clear that consistency and robustness of object-addressing are the most crucial features in a massively distributed system.

The goal that had to be achieved at the development process is a common way to provide stable, globally unique identifiers. In DOLSA these identifiers are named handles. To avoid name clashes, the term GUH (Globally Unique Handle) was introduced to name such handles. A GUH always refers to the same object and it is stable against object movement. If an object is deleted, its GUH must not be reused for any object. The location where a GUH is stored, is not restricted. Therefore it is possible to store a GUH anywhere.

Since a handle is an identifier that is independent from the physical location of an object, some resolve process is necessary when requesting an object with a given handle. In Dinopolis this resolve process is done by a distributed lookup-service (DOLSA). This distributed lookup-service consist of three kinds of lookup-servers. The OLSs (Object

[1] This is the DOI of [DOI, 2004] available via `http://dx.doi.org/10.1000/186`.

Lookup Servers) are distributed across the network and are responsible for mapping GUHs to addresses that are internally used to retrieve objects. MSLSs (Master Server Lookup Servers) determine which OLS (Object Lookup Server) is responsible for resolving a GUH. These MSLSs (Master Server Lookup Servers) are synchronized and keep their information redundant. The third kind of lookup-servers are SLSs (Server Lookup Servers). These are only cache server of data managed by MSLSs.

A handle is the combination of an Id of the responsible OLS and some local object Id. Furthermore a GUH consists of three such handles. The BPH (Birthplace Handle) always refers to the birthplace of an object. It is defined when an object is created and it must never change during the lifetime of an object. Furthermore it is guaranteed that the BPH can always be resolved. The BPH is also used to compare objects for equality. The second part of the GUH is the MBPH (Moved Birthplace Handle). It only contains data if the birthplace OLS of an object was taken offline and it is not guaranteed that the MBPH can always be resolved. The third part of a GUH is the CH (Current Handle). It represents an objects current residence. It is only updated if an object is moved across OLS responsibility boundaries.

A more detailed consideration about further aspects of this algorithm such as moving objects across servers and taking lookup-servers offline are out of scope of this thesis. As mentioned above, a detailed description of this algorithm can be found in [Schmaranz, 2002a] and [Schmaranz, 2002b]. Furthermore there are some investigations about guaranteeing consistency during move operations, dealing with “fast-moving” objects, and merging lookup-servers.

Chapter 5

The Dinopolis System Architecture

Dinopolis is a distributed middleware componentware framework. At the beginning of this chapter the key features of Dinopolis are presented. After that the *Dinopolis Object* is introduced. Besides DOLSA (Distributed Object Lookup Service Algorithm) (see Section 4.7) it is one of the main parts of the *Dinopolis Object* system architecture. It encapsulates arbitrary content and meta-data and provides functionality to access this data. Furthermore it is addressable and therefore it is possible to interconnect *Dinopolis Objects* with the interrelation mechanism describe in this thesis. Finally a overview about all main modules of the systems is presented. An overall description of the Dinopolis system architecture is presented by Klaus Schmaranz in [Schmaranz, 2002a].

5.1 Features of Dinopolis

Dinopolis is intended to ease application development. Therefore one of the main aims of Dinopolis is providing a possibility for an easy integration of existing systems. But it should also be possible to develop distributed applications from scratch. Network transparency and a robust addressing mechanism are two further aspects taken into account at the development process of Dinopolis. In addition the possibility to model arbitrary relationships between components and objects should be supported in Dinopolis. Out of these consideration the following cornerstones of Dinopolis are resulting:

- Dinopolis is platform as well as technology-independent. It is implemented in C++, but other implementations are also possible. For example a prototype implementation of the Dynamic Type Mechanism was written in Java.

- Dinopolis is a flexible and extensible middleware system. It has a very slim and modular system kernel. The kernel modules can be loaded on demand. This keeps the kernel as slim as possible.

- Existing systems such as filesystems or databases can easily be integrated in Dinopolis. This task is managed by a flexible embedder concept.

- A robust and consistent addressing mechanism is provided by Dinopolis. Objects are identified by handles and these handles are robust against object movement within the system.
- Dinopolis implements a highly sophisticated, distributed component model. Components are configurable during runtime. This component model and the addressing mechanism allows it to work with objects and components distributed all across the net.
- Relationships between components in Dinopolis are modeled by a highly sophisticated interrelation mechanism. This interrelation mechanism is also used for navigation through the object space. It stays consistent even when an interconnected object is deleted.
- Dinopolis also supports a state of the art security concept, which has been included within the whole development process of Dinopolis.

5.2 Dinopolis Objects

Functionality is provided to applications through distributed objects. These distributed objects are addressable entities. They are identified within the Dinopolis system by handles and they are called *Dinopolis Objects*.

Recapitulating:

- A *Dinopolis Object* is a component in the sense of componentware. Hence it provides functionality to the outside world. There exists a highly sophisticated security mechanism to control access to the *Dinopolis Object*.
- A *Dinopolis Object* is addressable across the network by the use of GUHs. As described in the Section 4.7 these handles always refer to one and the same object. The addressing mechanism in Dinopolis guarantees robustness against object movement and stability against object deletion.
- A *Dinopolis Object* is attachable to interrelations. Thus it is possible to model arbitrary relationships between *Dinopolis Objects*.
- A *Dinopolis Object* is configurable at runtime. This means that it is possible to add or remove functionality without the need to rebuild the framework.

Figure 5.1 shows the internal structure of a *Dinopolis Object*. It shows that a *Dinopolis Object* consists of the following:

Content A *Dinopolis Object* encapsulates content. This content can be accessed by a content data handler. The *Dinopolis Object* can be asked for this handler.

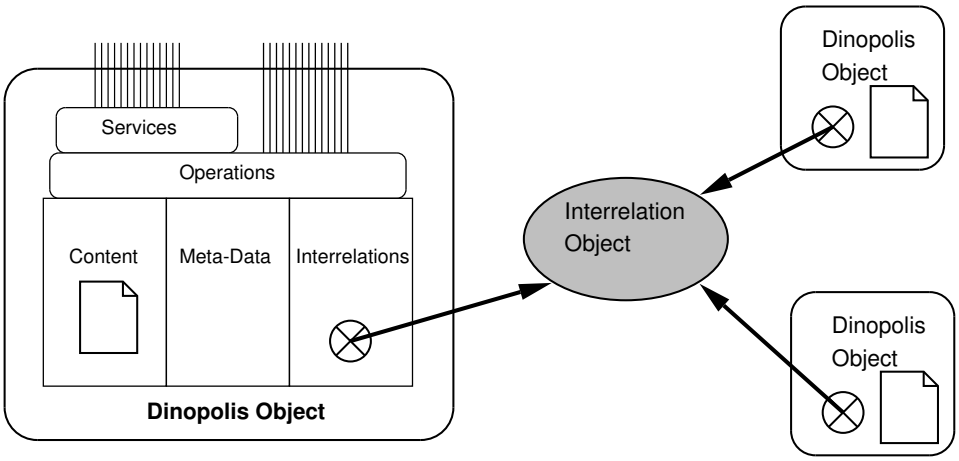

Figure 5.1: A simple *Dinopolis Object*

Meta-Data A *Dinopolis Object* can hold meta-data. This meta-data can be of arbitrary type. It is organized in a tree structured container. Meta-data can be accessed by a meta-data handler. The *Dinopolis Object* provides an operation to request this handler.

Interrelations The interrelations part of a *Dinopolis Object* holds the information about attached *Interrelation Objects*. This information is represented by endpoint objects. The interrelations part can be accessed via a interrelation handler. An operation is provided by the *Dinopolis Object* to request this handler.

Operations provide functionality. They are comparable with methods known from object oriented programming languages. It can distinguished between standard operations which have to be provided by all *Dinopolis Objects* and extended operations. These extended operations heavily depend on the type of the *Dinopolis Object*. Therefore they differ between different *Dinopolis Objects*.

Services also provide functionality. They are comparable with operations, but in addition they provide a user-interface to applications. This mechanism is necessary because sometimes an in-depth knowledge of the internals of an object would be needed to perform a specific task. But this would break the philosophy of Dinopolis and therefore *Services* are provided.

Operations and *Services* provide functionality of *Dinopolis Objects* to applications. As already mentioned, it is possible to add and remove such functionality of a *Dinopolis Object* at runtime. This task is managed by the help of the *Dynamic Type Mechanism*.

The types of a *Dinopolis Object* can be divided into static and a dynamic part. The static part is represented by the *Static Type* of a *Dinopolis Object*. It includes a guaranteed base interface to provide access the internal structure of a *Dinopolis Object*. Every *Dinopolis Object* has to provide this interface. Dynamic parts can be attached or detached from *Dinopolis Objects* at runtime. They are called *Dynamic Types* and provide a way to alter the functionality provided by a *Dinopolis Object* at runtime. All functionality which is provided through a *Dinopolis Object* first has to be declared by a *Declaration* and second it has to be defined by a *Definition*. Compared with object-oriented programming languages *Declarations* declare the interface of an object while *Definitions* provide the implementation of an object. Functionality is added to *Dinopolis Objects* at runtime by attaching a *Definition* to it. This *Definition* provides the implementation. Edmund Haselwanter gives a detailed description of the dynamic type mechanism in his master's thesis [Haselwanter, 2003].

5.3 Design Overview

Dinopolis provides a very modular and slim system architecture. Furthermore kernel modules can be loaded when needed and unloaded when no longer needed. This provides a flexible and extensible system architecture. In the following the main modules of the Dinopolis system are briefly described:

Object Management Module is the central part of the Dinopolis system. It is responsible for all operations which are dealing with the life-cycle. These are operations for creating and deleting as well as loading and storing of *Dinopolis Objects*. It is also responsible for the internal structure of a *Dinopolis Object*.

Address Management Module provides a stable address space for Dinopolis Objects. It is responsible for managing GUHs which address *Dinopolis Objects*. These handles are robust and therefore it is guaranteed that addresses remain stable even when an object is moved to a different location. The address management must be able to resolve any given GUH.

The address management module is divided into the three submodules. The OLS are responsible for local object resolving. The SLS (Server Lookup Server) and the MSLS (Master Server Lookup Server) performs global resolving.

Network Management Module provides transparent access to *Dinopolis Objects* that are located on remote Dinopolis instances. No knowledge about the physical location of an object is needed to work on them properly.

Interrelation Management Module is responsible for handling all different kinds of interrelations between *Dinopolis Objects*. Since the focus of this thesis lies on this Interrelation Management Module a detailed description of it can be found in Chapter 6 and Chapter 7.

External System Management Module manages all tasks that have to do with embedding of external systems. The main tasks are registering and unregistering of external systems as well as embedding and unembedding external systems. It has to be possible to embed arbitrary external systems. This is done by the use of so-called embedders. An embedder is comparable to a driver for external devices in operating systems. At least an embedder has to support a request to load data with a given address. Of course it is possible to extend an embedder with additional functionality (e.g save data, create object ...).

Virtual System Management Module is responsible for managing so-called Virtual Systems. Virtual Systems are combinations of several external, embedded systems. This combination appears as one single system. Consider as an example the combination of a filesystem with a database. The content of a *Dinopolis Object* that has its origin in this virtual system could reside in the filesystem, whereas meta-data is stored in the database.

System Data Storage Module stores and loads the state information of a *Dinopolis Object*. First of all this contains type information about the *Dinopolis Object*. This is necessary to be able to reconstruct the state of a *Dinopolis Object* during a load operation. The system data storage also plays an important role for the interrelation management module. The information about attached interrelations is made persistent within the system data storage. This circumstances are discussed in detail in Section 6.6.

Kernel Access Management Module is the access point for applications to kernel functionality. It is responsible to trigger security checks. These security checks are performed by the security management module.

Security Management Module is a required kernel module. It is responsible for all security aspects in Dinopolis. This security management module is as general as possible. From that it follows that different security policies are possible for different application scenarios.

Module Management Module is responsible for loading and unloading of kernel modules.

Configuration Management Module manages the bootstrapping process of a Dinopolis instance. First of all the kernel access management module is instantiated. All required kernel modules are loaded. After that process other modules are loaded.

Chapter 6

Interrelations in Dinopolis

This chapter describes the technical concept of the Interrelation mechanism in Dinopolis. It is shown how the mechanism works. *Interrelation Objects* are introduced and the characteristics of these objects are shown. Furthermore *Endpoints* are introduced and it is shown what they are good for. The correlation of interrelations to endpoints is pointed out and the possibilities how to organize interrelations' data are sketched. Finally *Implicit Interrelations* are introduced in this chapter.

6.1 The Interrelation Model

Generally spoken, in Dinopolis interrelations are uniquely addressable connections, relations or dependencies between arbitrarily many Dinopolis objects of arbitrary type or parts of them. An interrelation consists of an *Interrelation Object* and several endpoint objects. These endpoint objects represent anything that is connected by an interrelation. As seen in the previous sections, stability against reconstruction of the object space is the most critical and important feature of a highly sophisticated interrelation mechanism. Since the addressing mechanism of Dinopolis (see Section 4.7) implements the concept of GUHs this is not a problem at all. Interrelations and their attached endpoints are always identifiable by GUHs and therefore a robust interrelation mechanism is guaranteed. Technically spoken endpoint objects are attached to both the *Interrelation Object* and the *Dinopolis Objects* that are interconnected by the interrelation. On the one hand it is possible to model $n_1 : n_2 : \ldots : n_m$ dimensional relationships between objects with this interrelation model. One the other hand interrelations in Dinopolis are bidirectional. To be more precise, interrelations are rather multidirectional since interrelations are not only two-dimensional but also $n_1 : n_2 : \ldots : n_m$ dimensional. Multidirectional in this context means that it is possible to reach all endpoints of an interrelation if only one of them is known. In addition another consequence of multidirectional interrelations can be identified. It is possible to model both directed and directionless interrelations.

Figure 6.1 shows an example of three *Dinopolis Objects* interconnected by interrelations. *Dinopolis Object* 1, 2 and 3 encapsulate some documents. The dashed lines indicate logical relationships between *Dinopolis Objects*. Solid lines represent connections between *Interrelation Objects* and *Dinopolis Objects*. The following relationships

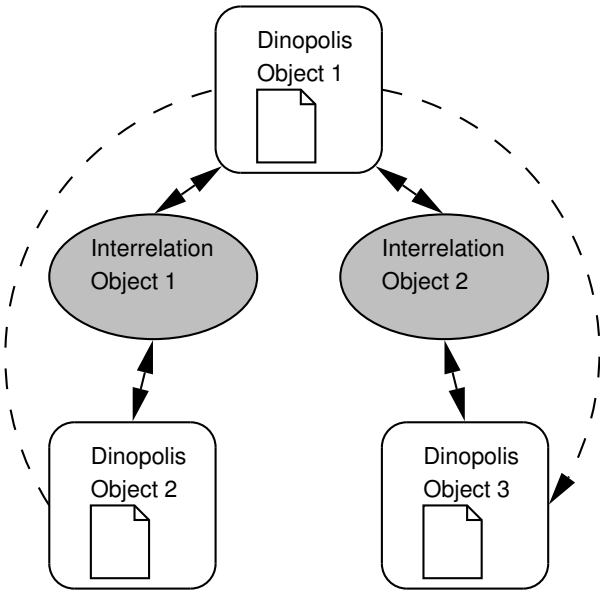

Figure 6.1: Example of Interrelations between Dinopolis Objects

can be identified:

- *Dinopolis Object* 1 relates to *Dinopolis Object* 2. This relationship is directionless and it is modeled by *Interrelation Object* 1.
- In addition *Dinopolis Object* 1 relates to *Dinopolis Object* 2. But this relation is directed. *Dinopolis Object* 1 is the source and *Dinopolis Object* 2 represents the destination of this relationship. It is equivalent to the hyperlink paradigm known from the WWW. This relationship between *Dinopolis Object* 1 and *Dinopolis Object* 2 is modeled by *Interrelation Object* 2.

6.2 The Interrelation Object

Logically spoken an interrelation in Dinopolis is a container of connections. It is represented by an *Interrelation Object*. A connection is modeled by two endpoint objects. Endpoint objects are discussed in detail in the next section (see Section 6.3). Several of the general requirements that are discussed in Chapter 3 lead to the fundamental design decision that *Interrelation Objects* are *Dinopolis Objects* themselves. This decision has several important impacts:

- *Dinopolis Objects* are addressable entities, hence interrelations are addressable

over the network by GUHs. This allows to store interrelations anywhere on the network.

- Every *Dinopolis Object* has some types. Thus the requirement that a way for assigning types to interrelations has to be provided, is satisfied implicitly.
- The functionality of an *Interrelation Object* is easily extensible by the use of the dynamic type mechanism the Dinopolis middleware framework provides for *Dinopolis Objects*. This makes it possible to model arbitrary passive and active behavior of Interrelations.
- *Dinopolis Objects* can hold arbitrary meta-data, thus it is possible to attach meta-data to interrelations. It is also possible to assign meta-data to certain endpoints of an interrelation.
- Interrelations always interconnect *Dinopolis Objects*, therefore it is possible to create interrelations between interrelations.
- Security checks are performed individually for interrelations. Therefore access permissions for interrelations can be controlled independently from the objects that they interconnect.

These circumstances guarantee the highest possible integration of *Interrelation Objects* within the Dinopolis system. Besides holding information about interconnected objects, *Interrelation Objects* also provide functionality to manage these interconnections. Therefore operations for attaching and detaching *Dinopolis Objects* are provided. Furthermore it is possible to resolve an interrelation in a certain way. A detailed discussion about the way an interrelation can be resolved is given in Section 6.7.3. It should also be clear that it has to be possible to make *Interrelation Objects* persistent. For this purpose operations to create, store, load and remove *Interrelation Objects* are provided. As mentioned above, *Interrelation Objects* are *Dinopolis Objects* and therefore these operations are already provided by *Dinopolis Objects*. It is obvious that this circumstance is a further benefit which results of the design decision that *Interrelation Objects* are *Dinopolis Objects* themselves.

6.3 Endpoints of Interrelations

Interrelations interconnect arbitrary objects. In Dinopolis these connections are modeled by *Endpoint Objects*. When in the following *Endpoint Objects* are meant the short name "*Endpoints*" is used. An *Endpoint* represents everything that is interconnected by an interrelation. As seen above this can be for example a *Dinopolis Object*, a part of a *Dinopolis Object* or an *Interrelation Object*. Since interrelations in Dinopolis are multidirectional they have to know about the *Dinopolis Objects* attached to them and vice versa. This information is held by endpoints. It has to be distinguished between:

Interrelation side Endpoint resides only in *Interrelation Objects*. Holds the information about the attached *Dinopolis Object*.

Dinopolis side Endpoint As the name implies a Dinopolis side Endpoint is part of a *Dinopolis Object*. It resides in the interrelations part of the *Dinopolis Object* and it holds information about the attached *Interrelation Object*.

So endpoints are containers, that hold all the information about connections. One *Dinopolis Object* may be attached multiple times to one interrelation. Since these are different connections the interrelation holds different endpoints. Endpoints are managed by *Endpoint Containers*. They provide operations to add and remove endpoints and to retrieve certain endpoints.

Figure 6.2 shows an example of three *Dinopolis Objects* that are interconnected by two interrelations. *Dinopolis Object* 1 with the GUH `0x0001` is interconnected with *Dinopolis Object* 2 (GUH = `0x0002`) by the *Interrelation Object* 1 (GUH = `0x0010`). Furthermore *Dinopolis Object* 1 relates to *Dinopolis Object* 3(GUH = `0x0003`). This relationship between *Dinopolis Object* 1 and *Dinopolis Object* 3 is modeled by *Interrelation Object* 2 (GUH = `0x0020`). To be able to distinguish easier between *Interrelation Objects* and *Dinopolis Objects*, *Interrelation Objects* have a grey background and *Dinopolis Objects* have a white background. Every Object contains an endpoint container which holds the single endpoints. A dashed line indicates a connection between two objects. The endpoints held by the endpoint containers of *Interrelation Object* 1 and *Interrelation Object* 2 are Interrelation Side Endpoints. Hence the endpoints contained in the endpoint containers of *Dinopolis Object* 1, 2 and 3 are Dinopolis Side Endpoints. Every endpoint, both Interrelation Side Endpoints and Dinopolis Side Endpoints, hold at least the following data:

GUH of the corresponding object of the other side of the connection (e.g. the endpoint of the *Dinopolis Object* 2 in figure 6.2) It contains the GUH `0x0010` which identifies the *Interrelation Object* 1. In contrast the second endpoint held by the endpoint container of *Interrelation Object* 1 contains the GUH that identifies *Dinopolis Object* 2 (`0x0002`).

Endpoint ID As a *Dinopolis Object* may be attached several times to the same *Interrelation* for different reasons, each connection needs a unique identifier. This Id is generated by the *Interrelation* and it has to be unique within the interrelation. As the name implies GUHs are globally unique within the object space and resulting form that fact the concatenation of the GUH identifying the interrelation and the endpoint ID is also globally unique. As can be seen in figure 6.2 the *Interrelation Object* 1 holds two endpoints with different endpoint Id's. The endpoint referring to *Dinopolis Object* 1 has the endpoint Id `1` and the endpoint Id referring to *Dinopolis Object* 2 has the endpoint Id `2`. The situation looks a little bit different for *Dinopolis Object* 1. Both endpoints referring to *Interrelation Object* 1 and *Interrelation Object* 2 have the same endpoint Id(`id_=1`). It seems as if the endpoints

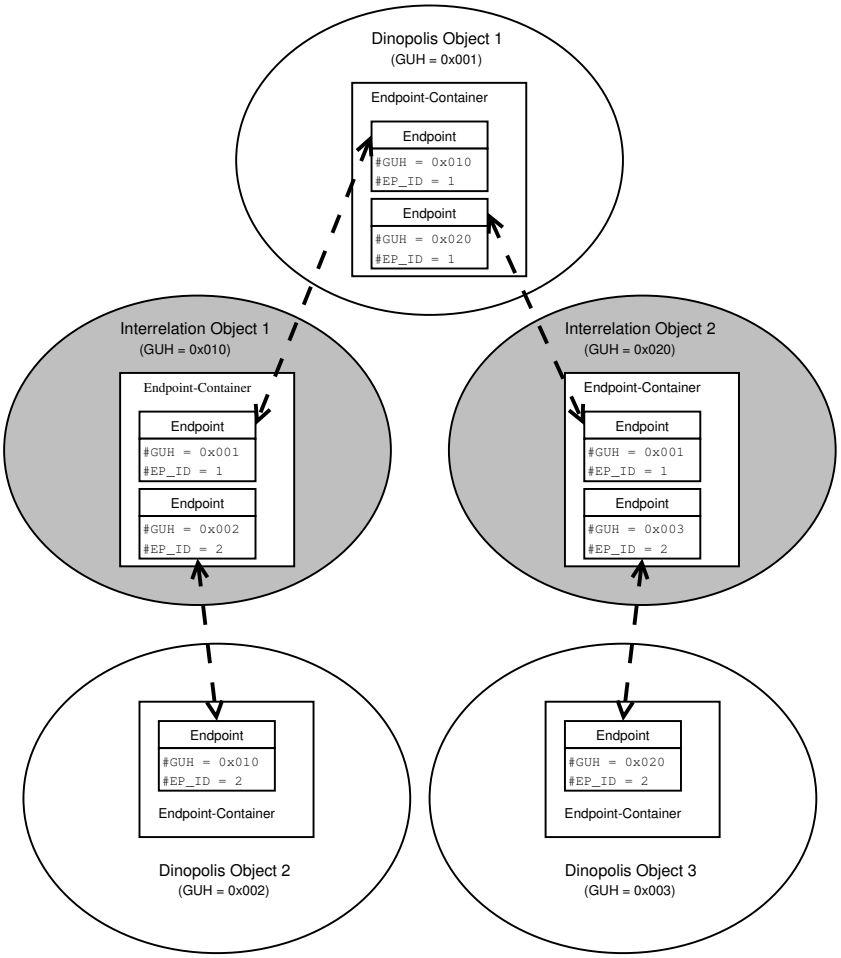

Figure 6.2: Schematic View of Endpoints

id's are ambiguous. But the concatenation of the GUH and the endpoint Id is unique (`0x010:1` vs. `0x020:1`).

There exist cases, where interrelations do not interconnect whole *Dinopolis Objects* but only parts of them. For example, consider the case of an anchor in an HTML document. That anchor may identify a point within the content of the document. To model such a link, a way has to be found to refer into the content of an object. This is done by the use of an internal key to the object-subset. This key heavily depends on the type of the *Dinopolis Object*'s content. Therefore it is not possible to generate a general object-subset mechanism and the content data handler of the *Dinopolis Object* generates this key that identifies some part in the object-subset. Since the content data handler has created the internal object-subset key, it is the only part which can interpret this key

correctly. It has to be clear that therefore referring to the subset of a *Dinopolis Object* is not guaranteed to be stable against object movement. If an interrelation points to an object-subset the internal key is part of the Dinopolis side endpoint. Since it can also be possible to refer to certain *Metadata* of the *Dinopolis Object*, the subset-key can also be managed by the meta-data handler of the *Dinopolis Object*.

6.4 Semantics of Endpoints

To give semantics to an interrelation, it is possible to group endpoints within an interrelation. For this reason so called *Endpoint Classification Sets* are introduced. As known from its mathematical sense, a set is a collection of distinguishable objects. In some cases it might be desirable that the sets are well-ordered. Therefore a "well ordering relation" can optionally be defined for a set . In mathematics various operations on sets are defined. The most important ones are union, intersection and difference. These set-operations are exactly what is needed to implement arbitrary resolve strategies on interrelations. Of course it is possible to have more sophisticated set-operations. But even if most of these operations are rather powerful, their software-implementation generally suffers from a lack of efficiency. The efficiency-problem of most of these operations can be solved by caching. Therefore the implementation of sets in Dinopolis has to be balanced between performance and memory efficiency.

As the name implies the elements of these interrelation-sets are endpoints. All endpoints of a set have something in common. In other words, they all have a certain property. Every endpoint with a certain property is assigned to an *Endpoint Classification Set* that represents this property. Consider as an example a hyperlink known from the WWW. An interrelation of type hyperlink may have two *Endpoint Classification Sets*. One set represents sources and the second set represents destinations of the hyperlink-interrelation. When creating such a hyperlink the *Endpoint* referring to the source document is assigned to the first *Endpoint Classification Set*. The other *Endpoint* is assigned to the second *Endpoint Classification Set* which represents destinations. Of course each *Endpoint* of an interrelation may be assigned to several different *Endpoint Classification Sets*. Again consider as an example a hyperlink. But now it is a multilingual hyperlink. Besides the *Endpoint Classification Sets* for source and destination objects, several sets indicating the language of the endpoint (e.g. german-language) exist.

Consider as a further example figure 6.3. It shows the logical view of an interrelation. Compared with the already presented examples (figure 6.1 and figure 6.2) again three *Dinopolis Objects* are related in some way. But in this example the relationship is modeled by only one *Interrelation Object*. The semantics of this *Interrelation Object* are defined by the three *Endpoint Classification Sets*, Set A, Set B, and Set C. The interrelation side endpoint which represents the connection to *Dinopolis Object* 1 is assigned to all three sets. The endpoint referring to *Dinopolis Object* 1 is only in *Endpoint*

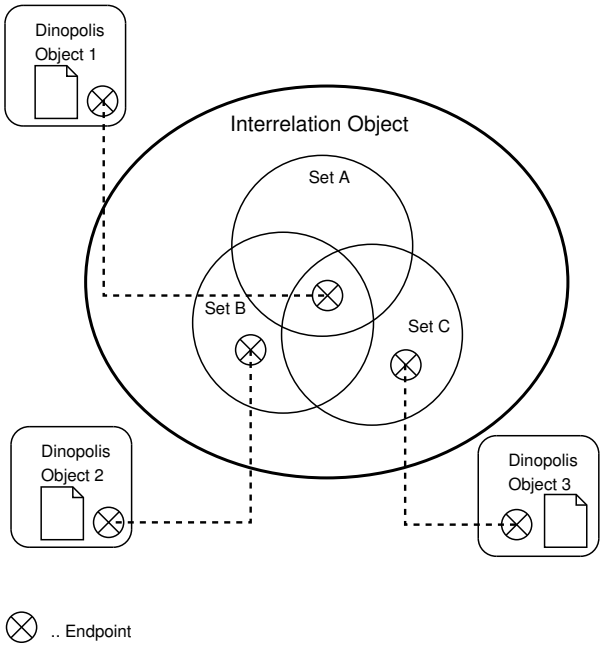

Figure 6.3: The Interrelations logical view

Classification Set A and the remaining endpoint which refers to *Dinopolis Object* 2 is assigned to Set C. It can also be seen that the Dinopolis side endpoints that are referring to the *Interrelation Object* are part of *Dinopolis Object* 1, 2 and 3.

As already indicated a resolve operation is performed by executing some set operations. In this example a thinkable resolve process would be as follows:

1. Starting from *Dinopolis Object* 1 the *Interrelation Object* which is referred by the Dinopolis side endpoint is requested.

2. All interrelation side endpoints are returned which are in the same *Endpoint Classification Set* as the endpoint representing *Dinopolis Object* 1. As result of this request the endpoints pointing to *Dinopolis Object* 2 and 3 are returned.

3. Starting from the result of the last operation another refinement of the resolve result can be achieved by the following strategy. Return all endpoints that are in Set C. This will return the endpoint which points to *Dinopolis Object* 3.

4. The last step to finish the resolve process properly is to get the *Dinopolis Object* which is referred by the endpoint of the last step. In this case this would be *Dinopolis Object* 3.

Several kinds of *Endpoint Classification Sets* can be distinguished:

Explicit Sets are either predefined by the type of an interrelation or they can be created explicitly. Consider again a hyperlink. It has two predefined sets. The first for destination endpoints and the second indicating source endpoints. As already mentioned it is possible to add sets manually to *Interrelation Objects*.

Implicit Sets are supported too. An example for a useful implicit set would be one, that contains all *Endpoints* of the same type. Caching the results of mathematical set-operations could be another reasonable case.

Default-Set Since resolve operations are mainly based on sets, every single *Endpoint* has to be assigned to at least one of those. Therefore each interrelation has to know a default-set, that holds all endpoints which are not assigned to a specific set.

The sets of an *Interrelation Object* are managed by the *Set Container*. It provides all functionality needed to organize sets and provides methods to add and remove sets to/from an interrelation. A method to move endpoints between sets is also provided. This operation has to be atomic on the set container, because each endpoint has to be element of at least one set. Further the above mentioned set-operations are also provided by the set container. These are needed for various resolve operations.

6.5 Meta-data

Arbitrary meta-data can be attached to *Dinopolis Objects*. Since *Interrelation Objects* are *Dinopolis Objects* themselves it should be clear that meta-data can be attached to *Interrelation Objects*. Meta-data in Dinopolis is organized in a tree-structured container. The values of this structure are accessible via hierarchical keys.

Besides the general meta-data about the interrelation itself, it may be useful to hold meta-data about its *Endpoint Classification Sets* and for each *Endpoint* it holds. Therefore it is possible to attach meta-data for both endpoints and *Endpoint Classification Sets*. For example meta-data about the interconnected *Dinopolis Object* may be useful for performance reasons. Caching this information may prevent from unnecessary accesses to the *Dinopolis Object* over the network. For the sake of uniformity this meta-data is organized in the same way as the meta-data of *Dinopolis Objects*, i.e. as a tree-structured container of hierarchical keys with values of arbitrary type.

6.6 Persistence

As seen above information about connections between *Dinopolis Objects* is managed by *Interrelation Objects*. This information are *Endpoints* which are managed by the *End-*

point Container and *Endpoint Classification Sets* which are managed by the *Set Container*. It has already been mentioned that *Interrelation Objects* are *Dinopolis Objects*. Hence, interrelations are addressable by GUHs and it is possible to store interrelations anywhere on the net. There are several ways how this information about interrelation is treated. It depends on this treatment how data is made persistent. Several proposals are possible and described below. Please remember the internal structure of *Dinopolis Objects* (see Section 5.2). It is mandatory to understand the following approaches:

- In the internal structure of *Dinopolis Objects* the encapsulated information is held in the content part. Information about connections held by an *Interrelation Object* can be seen as its content. In the first approach the content part of the *Interrelation Object* is used to hold the interrelation specific data. This can be seen in figure 6.4. The advantage of this approach is that the clear design of *Dinopolis Objects* is not broken. Loading and storing of *Interrelation Objects* is exactly the same as loading any other kind of *Dinopolis Objects*. However two circumstances have to viewed in more detail. It is not trivial to hold additional content within an interrelation. For example such content may be an image which is used for visualization purposes. If in this case content is requested from an *Interrelation Object*

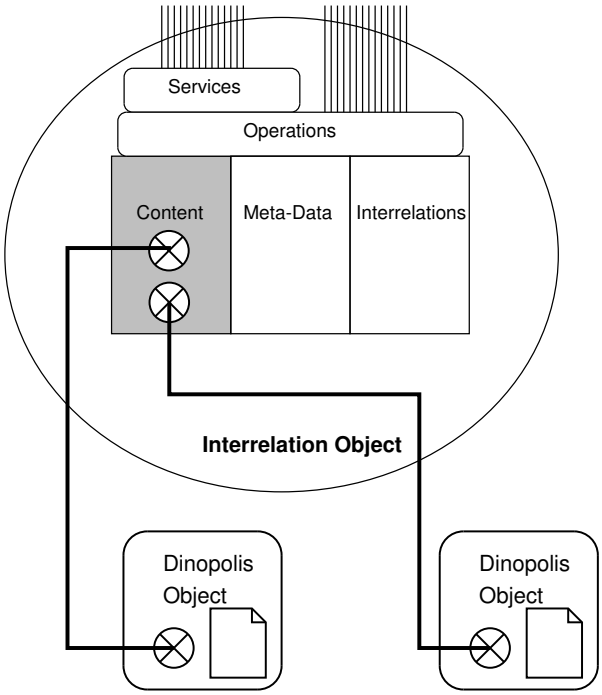

Figure 6.4: Interrelation Data treated as content

by the content handler, it has to be distinguished between interrelation data and such additional content. The restriction to prohibit such additional data may solve this deciding problem. A second aspect has to be considered. A virtual system has to be able to handle the content of *Interrelation Objects*. It is not guaranteed that every virtual system is able to handle the content of *Interrelation Objects*. It should be clear that in such a case it is not possible to store an *Interrelation Object* within this virtual system.

- In a second approach the static type "interrelation" extends the internal structure of a Dinopolis Object by an additional "*Interrelation Object* part" as shown in figure 6.5. This part encapsulates the information about *Endpoint Classification*

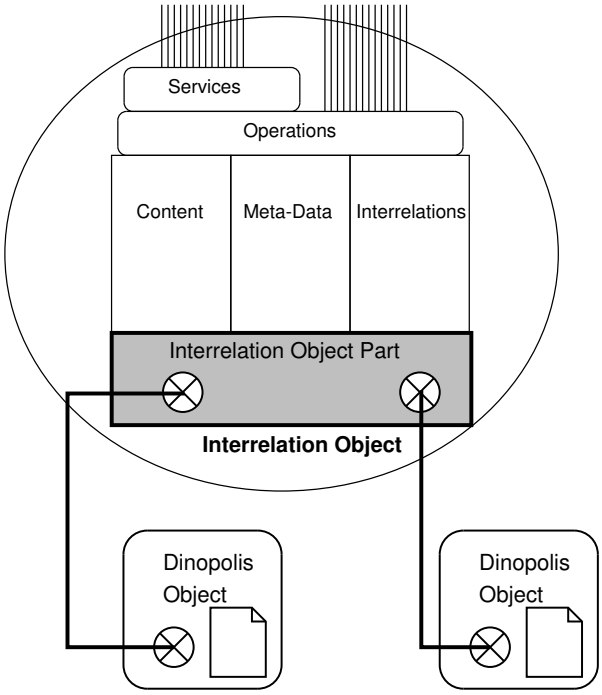

Figure 6.5: Interrelation extended by *Interrelation Object* part

Sets and endpoints of an interrelation. The main disadvantage of this approach is that the clear design of the architecture of a *Dinopolis Object* is broken. A way to store this "*Interrelation Object* part" has to be found. To provide a general solution for this problem the design of *Dinopolis Objects* has to be changed. But this is unacceptable since the slim design of *Dinopolis Objects* will be blown up by this attempt. Another possibility to implement this approach is to use an alternative place where the "*Interrelation Object* part" is stored. One opportunity is to use the system data storage to handle this information.

- In a further thinkable approach the interrelations part of the *Dinopolis Object* is used to manage this task. Information about *Endpoint Classification Sets* and endpoints are held in the interrelations part of the *Interrelation Object*. This case allows it to hold additional content within an *Interrelation Object*.

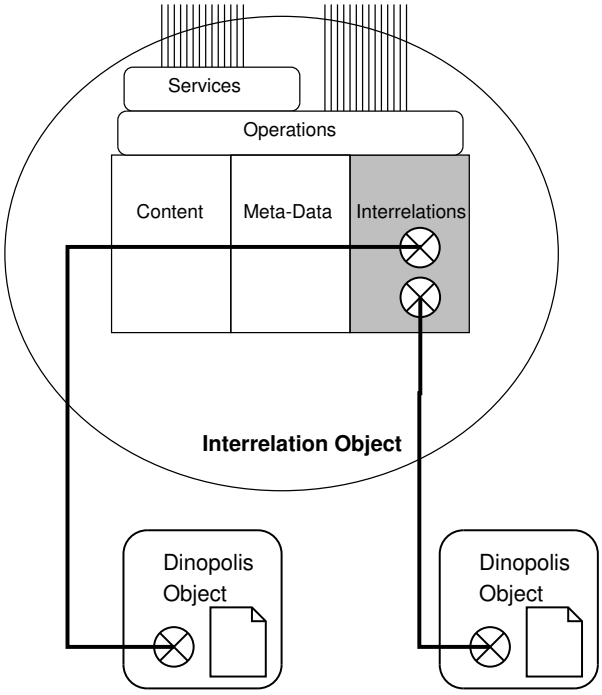

Figure 6.6: Interrelation Data resides in interrelations part

The schematic view can be seen in figure 6.6, but it may look a bit confusing at first glance. The interrelations part of a "non-Interrelation" *Dinopolis Object* only holds information about each interrelation the *Dinopolis Object* is attached to. Therefore a possibility to extend the interrelations part of an *Interrelation Object* has to be provided. This means that the interrelations part of an *Interrelation Object* encapsulates two different Interrelation data entities. On the one hand it encapsulates the *Endpoint Classification Sets* and the endpoints which are identifying attached *Dinopolis Objects*. On the other hand information about connections the *Interrelation Object* is attached to is also encapsulated by the interrelations part since it is possible to attach an *Interrelation Object* to an other *Interrelation Object*.

This approach has several drawbacks. It should be clear that it has to be distinguished which data entity is requested. Further it would not be possible to attach

Dinopolis Objects to this interrelations if the interrelation's persistent data lies in a read-only system. Some aspects about the impacts of read-only systems on interrelations are discussed below.

The first approach, treating *Endpoint Classification Sets* and endpoints as content of an *Interrelation Object* (see figure 6.4), has been chosen for a prototype implementation of the interrelation mechanism of Dinopolis. Since the design of Dinopolis is modular and flexible it is an easy task to change the policy how data is organized within *Interrelation Objects*.

Another aspect comes into play when discussing how to load and store interrelations. Since interrelations are multidirectional *Dinopolis Objects* which are attached to *Interrelation Objects* they also have to hold information about this connection. This information is represented by Dinopolis side endpoints (see figure 6.2). These endpoints are managed by the endpoint container of the *Dinopolis Object*. Logically spoken this container is the interrelation part of the *Dinopolis Object*. At a first glance it looks like a very trivial challenge to make this data persistent. The interrelations part of the *Dinopolis Object* is stored in the system by the virtual system management module. But this straightforward approach is not always practicable:

- The system where the *Dinopolis Object* resides is a read-only system and therefore it is not possible to store information in this system. Consider a CD-ROM as an example for a read only system. It should be clear that it is not possible to attach an *Dinopolis Object* residing in such an system to an interrelation. A store operation of the interrelations part would fail. But this restriction is unacceptable.

- However, even if the *Dinopolis Object* resides in a writable system it could be undesired to store information about attached interrelations in this system. Consider a *Dinopolis Object* encapsulating a document stored on a user's personal computer's hard-disk. This document is a very good and interesting scientific paper and many people would like to refer it by the use of interrelations. For every citation of this article an endpoint has to be stored on the user's private hard-disk. If too many people would like to link to this document, the system could be swamped with the extensive write accesses. It may be possible that users hard-disk run out of disc-space if it is a small system. Therefore the user may permit storing information about interrelations on his personal computer.

6.6.1 System Data Storage

The result of these considerations is the need for a general way for *Dinopolis Objects* to make the information about connections persistent. Therefore and among other things the *System Data Storage* is introduced.

The *System Data Storage* is a kernel module of the Dinopolis system. It is responsible for storing and loading state information of *Dinopolis Objects*. In other words, *Dynamic*

Definitions which are attached to a *Dinopolis Object* may have a state, which has to be stored, when the object is removed from memory. Since these *Definitions* are "generated" by Dinopolis kernel modules, *Virtual Systems* are generally not able to hold this information. Therefore the data is stored in the *System Data Storage*. Since Definitions themselves have to know how their data is stored, they have to provide methods which are called when the store operation of the *Dinopolis Object* is triggered. This behavior can be provided by using the hook pattern. The hook pattern is described in [Schmaranz, 2002a].

Furthermore it is possible to store and load information about attached interrelations within the system data storage. As seen above it is not guaranteed that the *Virtual System* is able to store this data. The same applies for the persistent data: the Object Management Module needs to manage an object's type information (static type, dynamic types, ...).

Data chunks are identified in the *System Data Storage* by the GUH of the *Dinopolis Object* and a qualifier. This qualifier represents a name for the *Dynamic Definition* which state has to be stored. The interrelations part of a *Dinopolis Object* is defined by a *Dynamic Definition*. Therefore a qualifier identifying this definition is used when storing or loading data held in the interrelations part of the *Dinopolis Object*.

6.7 How the interrelation mechanism works

So far this chapter described the model of interrelations as well as the main data-structures used by the mechanism. Now it is time to sketch how the mechanism works in general. Therefore operations to work properly with interrelations are introduced in this section. A use case scenario considering as an example a hyperlink that is modeled with this interrelation mechanism is presented in Section 7.1. It has to be distinguished between life-cycle operations, administrative operations, and resolve operations. These are discussed in detail in the following. For further information about some of these operations please look at Chapter 8. It gives a detailed description of some selected processes and algorithms applying *Dinopolis Objects*.

6.7.1 Life-cycle Operations

The possibilities how interrelation specific data is organized are already presented in this chapter. Now the operations dealing with this data-persistence are discussed. These operations come into play whenever the life-cycle of an object changes. Life-cycle operations are:

- Create
- Delete

- Load
- Store

As seen above it has to be distinguished between:

Interrelation Object If the data of an *Interrelation Object* is treated as content, it is stored in the virtual system where the *Interrelation Object* resides. In this case the virtual system has to be able to handle this data and the life-cycle operations are propagated by the interrelation management module to the object management module. In all other cases when the data is not treated as content alternative ways to make this data persistent are needed. One of these alternatives is the system data storage.

Interrelations part of Dinopolis Object Since it is not always possible to store this information within the virtual system where the *Dinopolis Object* resides, it is handled by the system data storage. However, the object management module is responsible for all life-cycle operations applying *Dinopolis Objects*.

6.7.2 Administrative Operations

Several operations are required to be able to manage interrelations. Therefore the interrelation management module provides operations to administrate *Interrelation Objects*. These operations are as follows:

- Attach *Dinopolis Object* to an *Interrelation Object*

  A new connection between *Interrelation Object* and *Dinopolis Object* is created. Therefore a new endpoint representing the *Dinopolis Object* is added to the *Endpoint Container* In addition the endpoint can be added to several *Endpoint Classification Sets*. This step is optional. If it is not performed the endpoint is automatically assigned to the default set. Further, meta-data describing this endpoint can be added to the *Interrelation Object* in this operation.

  To complete the attach operation an endpoint representing the *Interrelation Object* has to be added to the interrelations part of the interconnected *Dinopolis Object*.

- Detach *Dinopolis Object* from an *Interrelation Object*

  The endpoint representing the connection between the *Interrelation Object* and the given *Dinopolis Object* is removed. The endpoint is also removed from all *Endpoint Classification Set* and all optional meta-data about this endpoint is removed, too.

  It is also required to remove the endpoint representing the *Interrelation Object* from interrelations part of the interconnected *Dinopolis Object*. If this is not possible the detach operation has to be delayed or a rollback has to be performed.

- Create *Endpoint Classification Set*

  Additional *Endpoint Classification Sets* could be created within an *Interrelation Object*.

- Delete *Endpoint Classification Set*

  An *Endpoint Classification Set* can be removed from an *Interrelation Object* at runtime. If it is not possible to remove that set, removal will have to be inhibited. This case may occur if a *Endpoint Classification Set* is part of the *Static Type* of the *Interrelation Object* or if it is essential for a resolve operation.

- Assign an endpoint to an *Endpoint Classification Set*

- Remove an endpoint from *Endpoint Classification Set*

All of these operations are called on *Interrelation Objects* only, but internally some of them have to interact with the interconnected *Dinopolis Object* (e.g. detach *Dinopolis Object*).

6.7.3 Resolve Operations

The *Interrelation Object* provides several operations in the sense of the Dinopolis architecture to get endpoints which represent attached *Dinopolis Objects*. First of all it is possible to request all endpoints which are attached to the interrelation. Secondly operations are provided to *resolve* an interrelation. Resolve in this context means that depending on a given endpoint, (several) endpoints are returned. The behavior of these resolve operations depends on the *Static* and *Dynamic Type* of the *Interrelation Object*. A detailed description about resolve aspects of interrelations is given by Thomas Oberhuber in [Oberhuber, 2004]. Further it is possible to ask a *Dinopolis Object* for its attached interrelations. This functionality is provided by a so called *Interrelation Handler*. This handler is described in detail in Section 7.3.3.

6.8 Implicit Interrelations

Up to now the technical concept of interrelations which are created explicitly by clients was discussed in this chapter. It was seen that these so called *Explicit interrelations* are *Dinopolis Objects* and therefore they are addressable by GUHs. But besides them so-called *Implicit Interrelations* also exist. These implicit interrelations are used to map object relationships of existing embedded systems into the Dinopolis system. Such relationships are for example a parent-child relationship in a filesystem. A symbolic link in a filesystem also represents such a system immanent relationship.

Dinopolis has to provide an opportunity to easily embed existing systems. This task is managed by the *Virtual System* Management Module. This module also has to provide

a way to navigate throw the embedded system. It should not be surprising that implicit interrelations are also used to navigate through the embedded system. Furthermore they are not managed by the interrelation management module but by the *Virtual System* Management Module.

As the name implies, implicit interrelations are generated implicitly and they have there origin in external systems. One other important difference compared with explicit interrelations can be identified. Implicit Interrelations are **no** *Dinopolis Objects*. The result of this fact is that implicit interrelations are not addressable by GUHs. Furthermore this means that is it not possible to work on implicit interrelations directly, as it is done with explicit ones.

Therefore the only way to "resolve" an *Implicit Interrelation* is to request *all* objects, that are attached to it, which is done by a call on the corresponding *Endpoint* directly. Hence the *Endpoints* have to hold all the data that is needed to "resolve" the *Implicit Interrelation*. This information is largely dependent on the unit (e.g. a *Virtual System*), that holds the objects, which are interconnected by an *Implicit Interrelation*. Therefore this unit either creates the *Endpoints* for the *Implicit Interrelations* itself or it provides the information that is necessary to create them to *Dinopolis Objects*. Hence it is guaranteed that an *Endpoint* can retrieve all objects of the *Implicit Interrelation* that it represents from the corresponding embedded system.

Working with Dinopolis Objects, that are interconnected with *Implicit Interrelations* is mostly the same as working with any other kind of *Dinopolis Objects*. The only major difference is that it is not possible to request those *Interrelations* themselves from the according *Endpoints*. It is only possible to retrieve all objects "attached" to an *Implicit Interrelation*.

As already mentioned implicit Interrelations are used to navigate through external systems. Consider as an example a filesystem. Figure 6.7 shows a sample sector of the structure of someone's home directory in an unix filesystem. Three directories (`~/`, `doc` and `src`) and two files (`Thesis.pdf` and `Thesis.tex`) can be seen in this sample. A parent-child dependency can be identified between the home directory (`~/`) and the `doc` as well as the `src` directory. The document `Thesis.pdf` is contained in directory `doc` and the file `Thesis.tex` is contained in the folder `src`. `Interrelations.pdf` which is contained in the directory `~/` represents a symbolic link to the document `Thesis.pdf`. In figure 6.7 this relationship is indicated by the arched dashed arrow.

When navigating through an external system, first one or more entry points are requested from the appropriate *Virtual System*. In the example above, the user's home directory (`~/`) represents one entry point for this external system. An entry point represents a not yet loaded *Dinopolis Object* for the requesting client. It can be asked for its implicit interrelations. This interrelations return a collection of (not yet loaded) *Dinopolis Object*. Therefore it is possible to navigate through the extended system. This means for the example shown in figure 6.7 that a request for implicit interrelations on the entry point would return the interrelations representing the parent-child interrelation too the directories `doc`, `src` and `Interrelations.pdf`. Requesting implicit

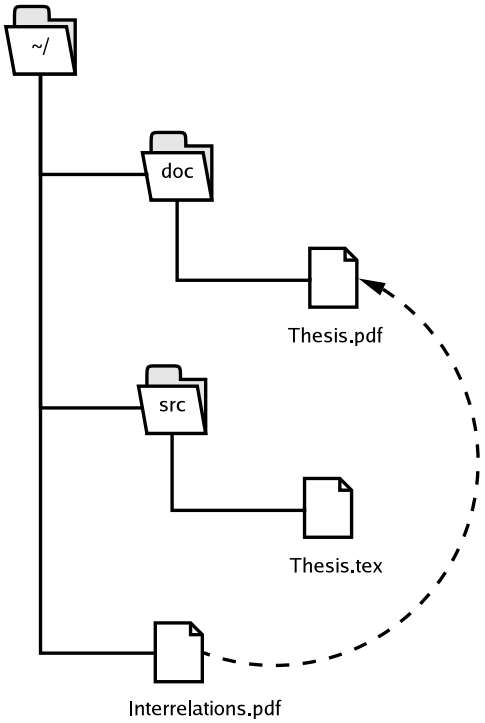

Figure 6.7: Example of a filesystem

interrelations from `Interrelations.pdf` returns a symlink interrelation which points to the document `Thesis.pdf`. It is possible to follow this interrelation. Up to now no *Dinopolis Object* is completely loaded but when `Thesis.pdf` is requested the *Dinopolis Object* that encapsulates this document is completely loaded.

This navigation with implicit interrelations is also used to find a location for creating a new *Dinopolis Object* within the external system. Please note that implicit interrelations are managed by *Virtual System* rather than by the *Interrelation Management* module. Moreover it is possible that implicit interrelations have their origin in the content of a Dinopolis object. As an example consider a anchor in a HTML document. In this case the content handler of the *Dinopolis Object* may extract this link information and provide them through implicit interrelations to the client.

Chapter 7

Design Details

This chapter shows some selected design details of the interrelation mechanism in Dinopolis. It is the result of the evolution of the design process of the interrelation management module. As already mentioned in the introduction of this thesis (Chapter 1) the work on this module was undertaken in teamwork by Thomas Oberhuber and me. We have split the topics of our master's theses. Therefore a very focused enumeration of all design details of the interrelation management module is presented here. Thomas was concentrated on the design of *Interrelation Objects*, especially on the design of *Endpoint Classification Sets* [Oberhuber, 2004]. The inspection of design issues in my thesis deals with the situation at the *Dinopolis Object* side.

First of all in this chapter, a use case scenario is presented. It sketches the usage of an interrelation of type hyperlink known from the traditional WWW. Then some software requirements are presented. This consideration is focused on requirements *Dinopolis Objects* have to fulfill concerning interrelations. Another section in this chapter deals with submodules and classes of the interrelation management module. Again the focus lies on *Dinopolis Objects*. Therefore Endpoints, endpoint containers and the interrelation handler are viewed in detail. Finally this chapter specifies some processes and algorithms needed to work with interrelations properly.

7.1 A Use Case Scenario

This scenario describes the application level use cases of an interrelation of type hyperlink. A hyperlink is a directed relation between two objects (e.g. documents, movies, images etc.) or two sets of objects. Three possible examples of Hyperlinks are considered. A hyperlink with one source and one destination represents a $1 : 1$ hyperlink. It is similar to the link definition known from HTML. When one source relates to several destinations it is a $1 : m$ hyperlink and the general case of several sources are interconnected with several destinations are $n : m$ hyperlinks. All of these circumstances can be modeled with $n : m$ *Interrelation Objects* of static type hyperlink. Such a hyperlink can hold meta-data to provide additional information about itself and its source and destination objects. Figure 7.1 shows a $1 : 1$ hyperlink modeled with an *Interrelation Object*.

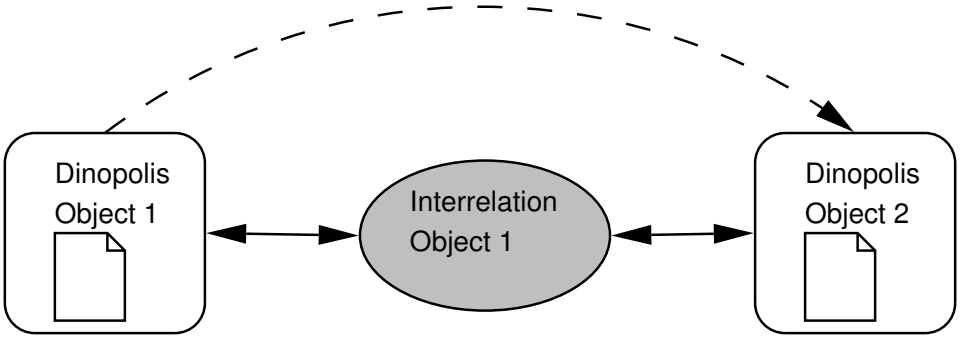

Figure 7.1: Use Case Scenario hyperlink-interrelation

Use Case: Create a new Hyperlink

This use case describes how two or more objects are interconnected by a hyperlink. Several steps must be performed to achieve this:

1. create new empty hyperlink interrelation:
   a) create new empty standard interrelation object.
   This interrelation has the static type *Hyperlink*. From this it follows that it has two *Endpoint Classification Sets*. One for source objects and the other one for destination objects.
   b) create additional *Endpoint Classification Sets* within the interrelation.
   This step is optional and has not to be performed at all times.
2. attach sources to the hyperlink.
   One or several *Dinopolis Objects* are attached to the interrelation. They are also assigned to the source *Endpoint Classification Set*. If additional *Endpoint Classification Sets* exist, endpoints would be assigned to certain of these sets. Finally meta-data for this endpoints is added to the *Interrelation Object*.
3. attach destinations to the *Interrelation Object*.
   One or several *Dinopolis Objects* are attached to the interrelation. They are also assigned to the destination *Endpoint Classification Set*. If addition *Endpoint Classification Sets* exist, endpoints would be assigned to certain of these sets. Finally meta-data for this endpoints is added to the *Interrelation Object*.
4. attach interrelation to the *Dinopolis Objects*
   Endpoints representing the *Interrelation Object* are attached to all *Dinopolis Objects* which were attached to the *Interrelation Object* in the previous two steps.

Use Case: Attach *Dinopolis Object* to a hyperlink

The following steps can be outlined:

1. At the *Interrelation Object*
   An endpoint that represents the *Dinopolis Object* is assigned to the correct *Endpoint Classification Set*. This is either the set which identifies sources or the set which identifies destinations. If additional *Endpoint Classification Sets* exist, endpoints would assigned to certain of these sets. Additionally meta-data for the endpoints can be added optionally.
2. At the *Dinopolis Object*
   An endpoint which represents the *Interrelation Object* is attached to the *Dinopolis Object*.

Use Case: Detach *Dinopolis Object* from a hyperlink

Two steps must be performed to detach a *Dinopolis Object* from the hyperlink:

1. At the *Interrelation Object*:
   The given source or destination endpoint is removed from all *Endpoint Classification Sets* it is assigned too. Then it is removed from the endpoint container. Furthermore available meta-data about this endpoint is removed.
2. At the *Dinopolis Object*:
   The endpoint representing the interrelation is detached from the interrelations part of the *Dinopolis Object*.

Use Case: Delete hyperlink

To delete a hyperlink all attached *Dinopolis Objects* have to be detached and the following steps have to be performed:

1. At the *Dinopolis Object*:
   The endpoint representing the interrelation is detached from the interrelations part of the *Dinopolis Object*.
2. The *Interrelation Object* is removed. This is only possible if all attached *Dinopolis Objects* were detached correctly in the previous step.

Use Case: Resolve Hyperlink

Resolving a hyperlink means that starting from a source the corresponding destination object is returned. In this use case it is assumed that the hyperlink is a $1:1$ hyperlink. This means it exist one source and one destination object. Furthermore it is assumed that both objects are only attached to this hyperlink interrelation. It follows:

1. At the source *Dinopolis Object*

   The endpoint representing the *Interrelation Object* is requested. Then the *Interrelation Object* is fetched.

2. At the *Interrelation Object*

   If the requesting *Dinopolis Object* is a source object the corresponding source endpoint is requested from the interrelation and the destination *Dinopolis Object* is returned.

   If the requesting *Dinopolis Object* is no source object it is not possible to resolve the interrelation properly, and the process terminates.

It should be clear that depending on the *Endpoint Classification Sets* several possibilities to resolve a hyperlink are thinkable. Consider a multilingual document. Starting from the german source document a desired resolve strategy would be to return the corresponding destination document which is written in the same language as the source document. In this case the object which encapsulate the german destination document would be returned. Thomas Oberhuber gives a detailed discussion about resolve aspects of interrelations in [Oberhuber, 2004].

Use Case: Get all interrelated objects

For a given object it may be of general interest to get all objects which are interconnected to it. Thinkable applications therefore are "back-links" or link maps. This use case describes how it is possible to achieve this. Starting from a *Dinopolis Object* that represents the destination of a hyperlink the following steps has to be performed:

1. At the starting *Dinopolis Object*

   It is assumed that this *Dinopolis Object* relates only to one interrelation. Therefore it holds only one endpoint. First of all this endpoint is requested. With this endpoint it is possible to fetch the *Interrelation Object*.

2. At the *Interrelation Object*

   All source endpoints are requested. Now it is possible to get all handles of the corresponding source *Dinopolis Objects*.

3. The interrelated *Dinopolis Objects* are returned

There exist several use cases which are dealing with meta-data. Since a detailed consideration on aspects of meta-data in distributed systems is far beyond the scope of this thesis, this use cases are skipped here.

7.2 Software Requirements

In addition to the general requirements on interrelations which were proposed in Chapter 3 this section sketches some selected software requirements.

The *Interrelation Management* module of the Dinopolis framework has to be extensible and flexible. Future evolution in the sense of exchanging and extending functionality should be possible without the need for basic redesigns. Therefore the general requirements proposed in Chapter 3 are observed during the development process of the interrelation mechanism of Dinopolis. Concerning these requirements, interrelations have to be type-able and should support meta-data. Furthermore they have to be addressable and it should be possible to create interrelations between interrelations. These requirements and the facts that *Dinopolis Objects* are addressable, can hold meta-data, and are extensible by *Dynamic Definitions* lead to the most important software requirement of the interrelation management module:

⇒ **Interrelations have to be *Dinopolis Objects***

By following this software requirement all above mentioned general requirements on interrelations are satisfied implicitly. Furthermore interrelations being *Dinopolis Objects* guarantee the highest possible integration of interrelations within the Dinopolis system.

An *Interrelation Object* holds the information about attached *Dinopolis Objects*. It was shown that interrelations should be multidirectional as well as multidimensional. The following software requirement can be derived:

⇒ **Connections have to be represented by endpoints.**

One endpoint represents the *Interrelation Object*, the second endpoint represents the *Dinopolis Object* of the connection. Since interrelations have to be multidimensional it must be possible to attach several endpoints to one *Interrelation Object*. Furthermore it must be possible to group these endpoints within the *Interrelation Object* to give the semantics to an interrelation. The *Interrelation Object* has to provide operations to request endpoints. Adding and removing an endpoint also has be supported. Further it should be possible to organize the grouping of endpoints within the *Interrelation Object*. To be able to perform this task the *Interrelation Object* has also to provide some operations.

Besides a robust addressing mechanism, a bidirectional interrelation mechanism is the key for robustness against object movement. Therefore interrelations are bidirectional. To achieve this it follows:

⇒ ***Dinopolis Objects* must hold endpoints**

As already mentioned these endpoints held by the *Dinopolis Object* represent the corresponding *Interrelation Objects* of the connections. It has to be assured that *Dinopolis Objects* do not need to know too much about the mechanisms dealing with interrelations.

For modularity reasons, the *Dinopolis Object* themselves should not be bothered with managing their interrelations. It follows:

⇒ ***Dinopolis Objects* have to provide an *Interrelation Handler***

This interrelation handler provides all the functionality needed to work on the *Dinopolis Object* side of the interrelation properly. Therefore it is possible to ask this *Interrelation handler* for information about the interrelations, the *Dinopolis Object* is attached to. In addition an *Interrelation Object* must also provide functionality to request information about attached *Dinopolis Objects*. Furthermore it should be possible to fetch the corresponding objects. As a result of these considerations it can be said:

⇒ **It must be possible to request endpoints**

A last aspect comes into play when discussing software requirements in this thesis. Remember that interrelations also have to be consistent against object deletion. When a *Dinopolis Object* is deleted, the corresponding endpoint of the connection has to be informed in some way to be able to react on the pending object deletion request. This lead us to the following requirement:

⇒ **Deleting a *Dinopolis Object* has to result in detaching it from all *Interrelation Objects* it is attached to**

If it is not possible to detach the *Dinopolis Object* from all *Interrelation Objects* the pending deletion request has to prevented or delayed. Therefore it should be clear that if this requirement is not satisfied, it is not possible to guarantee a consistent interrelation mechanism concerning object deletion.

For a detailed description of all identified software requirements please again have a look at Thomas' thesis [Oberhuber, 2004]. Thomas sketches all identified user requirements in his thesis.

7.3 Submodules and Classes

This section sketches some submodules and classes of the *Interrelation Management* module of the Dinopolis middleware framework. The focus lies on classes and submodules of the interrelations part of *Dinopolis Objects*. These are endpoints, the endpoint container and the interrelation handler. It is mentionable that the structure and the interface of Dinopolis side endpoints do not differ much from *Interrelation* side endpoints. This statement also is valid for endpoint containers managing either Dinopolis side endpoints or *Interrelation* side endpoints.

Please refer to Thomas Oberhuber's master's thesis [Oberhuber, 2004] for other submodules whose detailed description is missing here.

7.3.1 Endpoints

Endpoints are dealing with information about connections between *Interrelation Objects* and *Dinopolis Objects*. They encapsulate this data and provide an interface to manipulate this data. An endpoint holds the following data.

GUH refers to the corresponding object. It is used to provide bidirectional connections. In the case of *Dinopolis Object* side endpoints the corresponding object is an *Interrelation Object*. In the other case of interrelation side endpoints the GUH refers to a *Dinopolis Object*.

Endpoint Identifier consists of the GUH of the *Interrelation Object* and an internal identifier. It is used to identify an endpoint within an *Interrelation Object* uniquely. It is generated by the interrelation and this identifier is unique within the generating interrelation. The concatenation of interrelation GUH and internal identifier is globally unique.

Object Subset Key is used to let an interrelation point to somewhere within a *Dinopolis Object*. As *Interrelations* may refer to some part of *Dinopolis Objects*, an internal key to object-subsets is necessary. As the exact implementation of this key heavily depends on the type of the *Dinopolis Object*'s content. The *Content Data Handler* is responsible for its handling. Since *Endpoints* can also represent connections to certain meta-data of *Dinopolis Objects*, in this case the object subset key is requested from the *meta-data Handler* too.

Note that only *Dinopolis Object* side endpoints would have an object subset key. In this case they are called *Subset Endpoints*.

Sometimes it may be useful to cache descriptive data about the other side of a connection (e.g. the type of the corresponding object), because of performance issues. Therefore this data can be held within the endpoint. This behavior depends on the policy how endpoints are organized internally.

Besides holding information about interrelations an endpoint always provides the following interface:

- Get handle of the attached object

  This method returns the GUH of the corresponding object. If the endpoint is a *Dinopolis Object* side endpoint, the returned GUH identifies the corresponding *Interrelation Object*. In the other case, if the endpoint is an interrelation side endpoint, the GUH of the corresponding *Dinopolis Object* will be returned.

- Get *Endpoint Identifier*

  The *Endpoint Identifier* of this endpoint is returned

There are no set methods provided by an endpoint. The reason for this circumstance is the fact that the GUH and *Endpoint Identifier* are set during the creation process of the endpoint. It is not allowed to change their values during the lifetime of the endpoint.

As mentioned above, in the case that an interrelation refers to some part of a *Dinopolis Object* (e.g. content, meta-data) a *Dinopolis Object* side endpoint has an *Object Subset Key* and is called *Subset Endpoint*. *Subset Endpoints* provide methods to set this subset key as well as to request this subset key.

7.3.2 Endpoint Container

The main task of the *Endpoint Container* is to manage the organization of all endpoints within an *Interrelation Object* or *Dinopolis Object*. Therefore it must be able to hold endpoints and provide methods for accessing them. It is possible to iterate through the container's elements.

It is not a public interface of the *Dinopolis Object*. This means that the methods provided by the Endpoint container can only be called internally. If it is the Endpoint container of an *Interrelation Object*, these methods are only callable by the *Interrelation Object* itself. In the other case, if it is the endpoint container of the interrelations part of a *Dinopolis Object*, these methods are only accessible by the interrelation handler and by the corresponding *Interrelation Objects*. In the following the methods provided by an endpoint container are listed:

- Add an endpoint

  A new endpoint is inserted in the *Endpoint Container*.

- Remove an endpoint

  An endpoint with an given identifier is removed. If this endpoint is not contained in the *Endpoint Container*, an exception will be thrown.

- Get a certain endpoint

  The endpoint with the given Identifier is returned. If this endpoint is not contained in the *Endpoint Container* an exception will be thrown.

- Get all endpoints

  An iterator on the *Endpoint Container* is returned.

- Get the number of endpoints

  The number of all endpoints held by this *Endpoint Container* are returned.

7.3.3 Interrelation Handler

The *Interrelation Handler* provides all the interrelation-related functionality that other modules and applications need from *Dinopolis Objects*. Applications, modules and other *Dinopolis Objects* can request an *Interrelation Handler* from a *Dinopolis Object*. The Interrelation Handler provides some general operations:

- Request an endpoint

  An endpoint with a given identifier is requested from the *Dinopolis Object*.

- Request all endpoints

  All endpoints which are held in the interrelations part of the *Dinopolis Object* are returned.

- Request number of endpoints

  The number of all endpoints held by this *Dinopolis Object* are returned.

Please note that all endpoints held by the interrelations part of a *Dinopolis Object* represent connections to *Interrelation Objects*.

Chapter 8

Processes and Algorithms

This chapter shows some selected interrelation-specific processes. The focus lies on operations which are provided by the *Interrelation Handler* and on two operations that are needed internally to be able to add endpoints to a *Dinopolis Object* and remove endpoints from a *Dinopolis Object*. This chapter is adapted from the corresponding chapter of the interrelation management module design document [Oberhuber and Thalauer, 2004]. Furthermore not all processes are presented here because this would be way beyond the scope of this thesis. In contrast Thomas Oberhuber describes in [Oberhuber, 2004] the following processes in detail:

- Create Interrelation
- Attach *Dinopolis Object* to an Interrelation
- Detach *Dinopolis Object* from Interrelation
- Resolve an Interrelation.

8.1 Processes concerning the Interrelation Handler

All *interrelation*-related functionality provided by a *Dinopolis Object* is delegated to the *Interrelation Handler*. Therefore a *Dinopolis Object* provides a basic operation to request its *Interrelation Handler*. In this section some general operations provided by the *Interrelation Handler* are described:

8.1.1 Get an Endpoint

An endpoint is requested from an *Dinopolis Objects*. This endpoint represents a connection to an *Interrelation Object* the Dinopolis Object is attached to. This process can be outlined as follows:

1. The *Dinopolis Object* is asked for its *Interrelation Handler*.

2. The operation `getEndpoint` is called on the *Interrelation Handler*. The endpoint Identifier is passed as argument to this operation.

3. The internal method `getEndpoint(EndpointID id)` is called on the *Endpoint Container*, which returns the desired endpoint

4. The desired *Endpoint* is returned to the caller.

If the endpoint with the given endpoint identifier is not attached to the *Dinopolis Object*, an exception will be thrown.

8.1.2 Get all Endpoints

A request for all endpoints held in the interrelations part of a *Dinopolis Object* is called on its Interrelation Handler.

1. The *Dinopolis Object* is asked for its *Interrelation Handler*.

2. The operation `getAllEndpoints` is called on the *Interrelation Handler*. This request is forwarded to the endpoint container.

3. An Iterator on all attached endpoints is returned to the caller.

If no endpoints attached to the *Dinopolis Object*, an empty iterator will be returned.

8.1.3 Get number of Endpoints

A request for the number of *Interrelations* that the *Dinopolis Object* is attached to, is called on the *Interrelation Handler* of the *Dinopolis Object*:

1. The *Interrelation Handler* is requested from the *Dinopolis Object*.

2. The internal method `getNumberOfEndpoints()` is called on the *Endpoint Container*, which returns the number of all attached endpoints.

3. The *Interrelation Handler* then returns the number of attached endpoints to the caller.

8.2 Internal functions of Dinopolis Objects

Since *Interrelations* are bidirectional connections in the Dinopolis architecture, a request for attaching a new *Dinopolis Object* needs interaction with the *Object Management Module*. After the new endpoint is attached to the *Interrelation Object*; the *Dinopolis Object* is requested from the *Object Management Module* and a new endpoint which represents the *Interrelation Object* is attached to this requested *Dinopolis Object*. In the other case, if a *Dinopolis Object* has to be detached from an *Interrelation*, the *Dinopolis Object* is requested from the *Object Management Module* and the endpoint which represent the *Interrelation Object* will be detached from the *Dinopolis Object*.

Therefore a *Dinopolis Object* provides two internal operations. One which allows to add an endpoint to the *Dinopolis Object*. And another operation which removes an endpoint from the *Dinopolis Object*. These two special operations are accessible by *Interrelation Objects* only:

8.2.1 Adding an Endpoint to a Dinopolis Object

1. When a *Dinopolis Object* is attached to an *Interrelation Object*, this *Interrelation Object* contacts the *Dinopolis Object* to register the new connection. Therefore the *Dinopolis Object* provides an internal method to add a new *Endpoint* to the interrelations part of the *Dinopolis Object* by passing the GUH of the *Interrelation Object*, the *Endpoint Identifier*, and a optional key to a subset of the *Dinopolis Object*.

2. Request a new empty endpoint from the *Endpoint Container*.

3. Assign the *Endpoint Identifier* to the new *Endpoint*.

   This *Endpoint Identifier* was generated within the *Interrelation Object*. It is unique within the *Interrelation Object*.

4. If a key to a *Dinopolis Object*-subset is given, add it to the *Endpoint*.

5. Register the new *Endpoint* with the *Endpoint Container*.

8.2.2 Removing an Endpoint from a Dinopolis Object

1. When a *Dinopolis Object* is detached from an *Interrelation Object*, this *Interrelation Object* contacts the *Dinopolis Object* to remove the connection. Therefore the *Dinopolis Object* provides a internal operation to remove an *Endpoint* from the interrelations part of a *Dinopolis Object* by passing the *Endpoint Identifier*.

2. Call the remove method on the *Endpoint Container* an pass the *Endpoint Identifier* as argument. If the desired *Endpoint* does not exist, an exception will be thrown.

8.3 Detaching a Dinopolis Object from all Interrelation Objects

A request for deleting a *Dinopolis Object* requires detaching it from all *Interrelation Objects*. The following algorithm describes the way this is done:

1. At the *Dinopolis Object* which should be deleted

   Request all endpoints which are held by the endpoint container of the interrelations part of the *Dinopolis Object*

2. For all endpoint which are returned in the last step.

   Fetch the *Interrelation Object* which is represented by the endpoint. Detach the endpoint which represents the *Dinopolis Object*.

3. The *Dinopolis Object* is detached from all *Interrelation Objects*

   Now it is possible to delete the *Dinopolis Object*. Since object deletion is managed by the object management module it it is beyond the scope of this chapter and therefore a detailed description is skipped here.

Chapter 9

Conclusion and Outlook

9.1 Conclusion

Within a distributed component system it has to be possible to model arbitrary relationships between objects and components and parts of them. In existing distributed systems these relationships are often modeled by the use of hyperlinks. But it was shown in this thesis that these hyperlinks are a rather poor concept to model such relationships. Some important requirements are not fulfilled:

- Interrelations between objects should be robust against object movement and stable against object deletion. In other words, the "dangling link syndrome" has to be prevented. On the one hand a stable addressing mechanism is required to reach this goal. It is further necessary to strictly separate between addressing, structure, and navigation. On the other hand bidirectional connections are also required to achieve robustness and consistency.

- Interrelations should be objects themselves. On the one hand this has the benefit that they are addressable and therefore it is possible to model interrelations between interrelations. On the other hand this guarantees the highest possible integration of interrelations within the distributed system. To be more precise, interrelations which are objects can be of arbitrary (interrelation) type and can hold meta-data.

The interrelation mechanism of Dinopolis satisfies all these requirements . It is therefore the mechanism of choice to model arbitrary relationships between arbitrary objects and components.

9.2 Outlook

In these days a prototype of the Dinopolis middleware framework is implemented. This prototype should show that all considerations about interrelations are correct. On basis

of this prototype a further prototype of the ESRM (Electronic Study Record Manager) application is implemented and all the benefits of the overall Dinopolis system architecture are shown.

Since the system architecture of Dinopolis is as flexible and modular as possible it should be no problem at all, to implement a *stand-alone* version of the interrelation mechanism presented in this thesis. Giving the proof that this assumption is correct may also be a interesting exercise for the future.

References

[Han, 2002] (2002). *Handle System*. Corporation for National Research Initiative. available online `http://hdl.handle.net/4263537/4094`.

[DOI, 2004] (2004). *The DOI handbook*. International DOI Foundation (IDF). available online `http://dx.doi.org/10.1000/186`.

[Andrews et al., 1994] Andrews, K., Kappe, F., Maurer, H., and Schmaranz, K. (1994). On Second Generation Hypermedia Systems. *Journal of Universal Computer Science*, 0(0):127–135. `http://www.jucs.org/jucs_0_0/on_second_generation_hypermedia`.

[Berners-Lee et al., 1994] Berners-Lee, T., Cailliau, R., Luotonen, A., Nielsen, H. F., and Secret, A. (1994). The World-Wide Web. 37(8):76–82.

[Berners-Lee and Connolly, 1995] Berners-Lee, T. and Connolly, D. (1995). RFC 1866: Hypertext Markup Language — 2.0. Status: PROPOSED STANDARD.

[Berners-Lee et al., 1999] Berners-Lee, T., Fielding, R., Frystyk, H., Gettys, J., Leach, P., Masinter, L., and Mogul, J. (1999). RFC 2616: Hypertext Transfer Protocol —HTTP/1.1. Status: PROPOSED STANDARD.

[Berners-Lee et al., 1998] Berners-Lee, T., Fielding, R., and Masinter, L. (1998). RFC 2396: Uniform Resource Identifiers (URI): Generic syntax. Status: DRAFT STANDARD.

[Bieber et al., 1997] Bieber, M., Vitali, F., Ashman, H., Balasubramanian, V., and Oinas-Kukkonen, H. (1997). Fourth generation hypermedia: some missing links for the world wide web. *Int. J. Hum.-Comput. Stud.*, 47(1):31–65.

[Blümlinger, 2000] Blümlinger, K. (2000). Advanced Link Management in Hypermedia Systems. Master's thesis, IICM, Graz University of Technology. available online `http://www.dinopolis.org/documentation/misc/theses/`.

[Blümlinger and Dallermassl, 2002] Blümlinger, K. and Dallermassl, C. (2002). Addressing in Distributed Systems.

[Blümlinger et al., 2003a] Blümlinger, K., Dallermassl, C., Haub, H., and Zambelli, P. (2003a). Controlling Access to Distributed Object Frameworks. In *Lecture Notes in Computer Science, Conference Proceedings of JMLC 2003*. Springer Verlag.

[Blümlinger et al., 2003b] Blümlinger, K., Dallermassl, C., Haub, H., and Zambelli, P. (2003b). Dynamic Typing of Objects. submitted to J.UCS.

[Blümlinger et al., 2003c] Blümlinger, K., Dallermassl, C., Haub, H., and Zambelli, P. (2003c). Object Life-Cycle Management in a Highly Flexible Middleware System. In *Lecture Notes in Computer Science, Conference Proceedings of JMLC 2003*. Springer Verlag.

[Bush, 1945] Bush, V. (1945). As we may think. *The Atlantic Monthly*, 176(1):101–108. available online `http://www.theatlantic.com/unbound/flashbks/computer/bushf.htm`.

[DeRose et al., 2001] DeRose, S., Maler, E., and Orchard, D. (2001). XML Linking Language (XLink) version 1.0 — W3C recommendation 27 june 2001. Technical Report REC-xlink-20010726, World Wide Web Consortium. available online `http://www.w3.org/TR/2001/REC-xlink-20010627/`.

[Dyke and Sollins, 1995] Dyke, J. R. V. and Sollins, K. R. (1995). Linking in a global information architecture. In *World Wide Web Jornal - A Publication of the W3C*, volume 1, page 493. World Wide Web Consortium, O'Reilly. available online `http://www.w3journal.com/1/e.193/paper/193.html`.

[Freismuth et al., 1997] Freismuth, D., Helic, D., Meszaros, G., Schmaranz, K., and Zwantschko, B. (1997). DINO - Distributed Interactive Network Objects The Java Approach. Proceedings Ed-Media '97. available online: `http://www.iicm.edu/liberation/iicm_papers/edmed97/dino.html`.

[Halasz and Schwartz, 1994] Halasz, F. and Schwartz, M. (1994). The dexter hypertext reference model. *Commun. ACM*, 37(2):30–39.

[Haselwanter, 2003] Haselwanter, E. (2003). Aspects of Component Composition in Distributed Frameworks. Master's thesis, IICM, Graz University of Technology.

[Kappe, 1995] Kappe, F. (1995). Maintaining link consistency in distributed hyperwebs. *Proceedings of INET'95, Honolulu, Hawaii*. available online: `http://www.isoc.org/HMP/PAPER/073/abst.html`.

[Krottmaier and Maurer, 2001] Krottmaier, H. and Maurer, H. (2001). Transclusions in the 21st Century. *Journal of Universal Computer Science*, 7(12):1125–1136. `http://www.jucs.org/jucs_7_12/transclusions_in_the_21st`.

[Lennon, 1997] Lennon, J. A. (1997). *Hypermedia Systems and Applications*. Springer.

[Maurer, 1996] Maurer, H., editor (1996). *Hyperwave - The Next Generation Web Solution*. Addison-Wesley. available online `http://www.iicm.edu/hwbook`.

[Moats, 1997] Moats, R. (1997). RFC 2141: URN syntax. Status: PROPOSED STANDARD.

[Nelson, 1965] Nelson, T. H. (1965). Complex information processing: a file structure for the complex, the changing and the indeterminate. In *Proceedings of the 1965 20th national conference*, pages 84–100. ACM Press.

[Nelson, 1992] Nelson, T. H. (1992). *Litarary Machines 93.1*. Mindful Press.

[Nielsen, 1995] Nielsen, J. (1995). *Multimedia & Hypertext : the Internet and beyond*. AP Professional.

[Oberhuber, 2004] Oberhuber, T. (2004). Aspects of Structured Component Spaces in Distributed Systems. Master's thesis, IICM, Graz University of Technology.

[Oberhuber and Thalauer, 2004] Oberhuber, T. and Thalauer, S. (2004). *Interrelation Management Module Design*. MTP/IICM. ProductId: 256:228:1.0.2.

[Schmaranz, 2002a] Schmaranz, K. (2002a). Dinopolis - A Massively Distributeable Componentware System. Habilitation Thesis.

[Schmaranz, 2002b] Schmaranz, K. (2002b). Dolsa - a robust algorithm for massively distributed, dynamic object-lookup services. submitted to J.UCS.

[Schmaranz, 2002c] Schmaranz, K. (2002c). On Second Generation Distributed Component Systems. *Journal of Universal Computer Science*, 8(1):97–116. `http://www.jucs.org/jucs_8_1/on_second_generation_distributed`.

[Shafer et al., 1996] Shafer, K., Weibel, S., Jul, E., and Fausey, J. (1996). Introduction to persistent uniform resource locators. In *Conference Proceedings of the Annual Meeting of the Internet Society INET96*. available online `http://purl.oclc.org/OCLC/PURL/INET96`.

[Sollins and Masinter, 1994] Sollins, K. and Masinter, L. (1994). RFC 1737: Functional requirements for uniform resource names. Status: INFORMATIONAL.

[Stone, 2000] Stone, L. (2000). Competitive evaluation of purls. Webpage. available online `http://web.mit.edu/handle/www/purl-eval.html`.

[Tekelenburg, 2004] Tekelenburg, S. (2004). Navigating the WWW. Webpage. available online `http://www.euronet.nl/~tekelenb/WWW/LINK/index.html`.

[Trigg, 1983] Trigg, R. H. (1983). *A Network-Based Approach to Text Handling for the Online Scientific Community*. PhD thesis, Department of Computer Science, University of Maryland.

[Van Dyke Parunak, 1989] Van Dyke Parunak, H. (1989). Hypermedia topologies and user navigation. In *Proceedings of the second annual ACM conference on Hypertext*, pages 43–50. ACM Press.

Glossary

CamelCase: *"Bicapitalization or camel case, frequently applied to the term itself and written CamelCase, is the capitalization of more than one word within a compound word or multi-word symbolic name. Words written this way call to mind a camel in profile with a head and one or more humps. This is also known as BumpyCase, StudlyCaps, and WikiWord."*
from `http://en.wikipedia.org/wiki/CamelCase`.

CeBIT: *Centrum der Büro- und Informationstechnik* is the world's most important computer world's fair.

CERN : *European Centre for Nuclear Research* is the European Organization for Particle Physics Research. It is situated near Geneva between France and Switzerland and it is the world's biggest particle physics laboratory. The acronym original stood for *Conseil Européen pour la Recherche Nucléaire* and was changed to *Centre Européen pour la Recherche Nucléaire.*

XML: *eXtensible Markup Language* is a subset of SGML. It defines a particular text markup language for interchange of structured data.

List of Acronyms

BPH ..... Birthplace Handle

CH ..... Current Handle

CNRI ..... Coporation for National Research Initiatives

DINO ..... Distributed Interactive Network Objects

DLR ..... German Aerospace Center

DNS ..... Domain Name System

DOI ..... Digital Object Identifier

DOLSA ..... Distributed Object Lookup Service Algorithm

ESRM ..... Electronic Study Record Manager

FTP ..... File Transfer Protocol

GUH ..... Globally Unique Handle

HTML ..... HyperText Markup Language

HTTP ..... HyperText Transfer Protocol

IICM ..... Institute for Information Systems and Computer Media

ISBN ..... International Standard Book Number

MBPH ..... Moved Birthplace Handle

Memex ..... Memory Expander

MSLS ..... Master Server Lookup Server

MSLSs ..... Master Server Lookup Servers

MTP ..... Medical Telematics Platform

OLS ..... Object Lookup Server

OLSs ..... Object Lookup Servers

PURL ..... Persistent URL

SLS ..... Server Lookup Server

SLSs ..... Server Lookup Servers

UID ..... Unique Identifier

URI ..... Uniform Resource Identifier

URL ..... Uniform Resource Locator

URN ..... Uniform Resource Name

W3C ..... World Wide Web Consortium

WWW ..... World Wide Web

Xlink ..... XML Linking Language

Index

A

address 21, 23
  resolve 21
address management module 32
addressing mechanism 30
addressing system 9, 24
anchor 8, 9, 10
attach 19, 47

B

Bush, Vannevar 6

C

C++ 29
CEBIT 10
CERN 8
component 30
configuration management module 33
create interrelation 19

D

Dangling Link Syndrome 14
database 29, 33
definition 32
definitions 6, 21
delete interrelation 19
detach 19, 47
Dexter Model 7
DINO 10, 16
Dinopolis object 30, 55
  content 30
  Delete Operation 56
  dynamic type 32
  interrelations part 31, 47
  meta-data 31
  operations 31
  services 31
  static type 32
Dinopolis system 1, 11, 29
DOI System 26
DOLSA 23, 27
dynamic definition ..... *see* definition
dynamic type mechanism 32

E

embedder 29, 33
endpoint 55, 57
endpoint classification set 39
endpoint container 58
external system management module .. 33

F

filesystem 29, 33
forwarding chain 23

G

Graz University of Technology 9, 10

H

handle 22, 23, 30
  resolve 22
Handle system 26
HTML 8, 12
HTTP 9
Hyper-G ..... *see* Hyperwave
hyperlink 7, 8, 9, 10, 14
hypermedia 6
hypertext 6
Hyperwave 9, 14, 16

I
identifier 22
IICM 1, 9, 10
inclusion ..... *see* transclusion
Interrelation handler 56, 59, 60
Interrelation management module .. 1, 32, 56
Interrelation object 35
Interrelations 1, 34
  characteristics 1
  consistency 14, 21
  endpoint 36
  implicit 18
  multidimensional 16, 34
  multidirectional 15
  operations 19
  robust 21
  semantics 39
  stability 14
  types 12, 34

J
Java 29

K
kernel access management module 33
kernel modules 29, 33

L
life-cycle 46
link ..... *see* hyperlink
lookup-server 27
lookup-service 22, 23

M
Memex 6
meta-data 13, 41
middleware 29
module management module 33
MTP 10

N
name 21, 23
navigation 21, 23
  address strategy 22
  identifier strategy 22
  path strategy 22
  strategies 22
navigation paths 6
Nelson, Ted 6, 7
network management module 32
Nielsen, Jakob 7
node 7

O
Oberhuber, Thomas 1, 5, 56
object management module 32

P
physical location 21, 23, 27, 32
PURL 25

R
requirements 12
resolve 7
resolve interrelation 19, 48
resource 22

S
Schmaranz, Klaus 1, 5, 23, 27, 29
security management module 33
structure 21
  hierarchical 22
system data storage 33, 45

T
transclusion 7
transparency
  protocol 23
  schema 23

U
URI 9, 24
URL 24
URN 24
use case scenario 51

V
virtual system management module ... 33

W
W3C 9, 13
Wiki 16

WWW 8, 12, 24

X

Xanadu 7
XLINK 9
XML 9