

Klaus Schmaranz

Dinopolis – A Massively Distributable
Componentware System

Habilitation Thesis

June 28, 2002

Abstract

Two of today's most used buzz-words in the context of software development are the terms *Componentware* and *Distributed Object-System*. The combination of both ideas is then called a *Distributed Component-System*, meaning a componentware approach where the components are distributed across the network. Today's approaches fulfill the application developers' needs only partly. Also, most are more or less cumbersome to use. I want to call such part-solutions like e.g. Corba, Enterprise Java-Beans and others *first generation distributed component systems*. In fact, Corba has a different origin, but for the moment let me consider it to be a first generation componentware system, too.

In this thesis Dinopolis is introduced, the first system world-wide which can be considered to be a *second generation distributed component system*. In chapter 2 a short overview of two possible use-case scenarios is given as a motivation for the features that Dinopolis implements. The first of the scenarios describes the MTP (=Medical Telematics Platform) project which implements a platform for arbitrarily distributed virtual medical patient records. The second scenario that is sketched deals with the features of modern electronic libraries.

The key for the Dinopolis platform architecture is a clean, modular and comprehensive technical definition of what is considered an object or a component respectively. This definition is outlined in chapter 3 and forms the basis for the overall system architecture which is described in chapter 4. The main part of this thesis (chapters 5–13) is dedicated to the detailed descriptions of all identified modules, their functionalities and the special algorithms that were developed to make Dinopolis work in practice.

Table of Contents

1. Introduction	4
2. Use Case Scenarios	8
2.1 The MTP Scenario	8
2.2 An Electronic Library Scenario	15
2.3 Other Scenarios	17
3. The Dinopolis Object Definition	20
4. Overall Dinopolis System Architecture	25
5. The Kernel Access Module	34
5.1 Mechanisms Defined for Functional Modules	35
5.2 Mechanisms Defined for Working With Objects	42
6. The Object Management Module	46
6.1 Creating a New Object	46
6.1.1 Creating a Simple Object	47
6.1.2 Creating a Compound Object	48
6.2 Storing an Object's Persistent Data	52
6.3 Loading an Object	54
6.4 Deleting an Object	56
6.5 Moving an Object Between Virtual Systems	58
7. The Address Management Module	60
7.1 <i>GUHs</i> and Addresses in Dinopolis	61
7.2 The Design of DOLSA in Detail	63
7.2.1 The Static Case	64

7.2.2	Moving Objects Across Servers	65
7.2.3	Taking Birthplace Servers Offline	67
7.3	Structure of GUHs	70
7.4	The <i>DOLSA</i> base-algorithm	71
7.4.1	The View of the Distributed Lookup-Service	71
	Adding an <i>OLS</i> to the Distributed Service	72
	Changing an <i>OLS</i> 's Hostname	72
	Moving an Entry to a Different <i>OLS</i>	72
	Dealing With an <i>OLS</i> Going Offline	73
	Splitting up an <i>OLS</i>	73
7.4.2	The View of the Requestor	74
7.5	Distributing the <i>SLSvc</i>	77
7.5.1	Organizational Structure of the Distributed <i>SLSvc</i>	77
7.5.2	Adding and Removing <i>SLSs</i>	78
7.5.3	Propagation of Information between <i>MSLSs</i>	80
7.5.4	Propagation of Resolve Requests	81
7.5.5	The <i>SLS</i> caching strategy	84
7.5.6	Reorganizing the <i>SLS</i> Distribution Hierarchy	84
7.5.7	Robustness in Case of Network Failures	86
7.5.8	Additional Administrative Aspects of the <i>SLSvc</i>	87
7.6	Further Investigations	91
7.6.1	Guaranteeing Consistency During Move-Operations	91
	Case 1: One <i>OLS</i> , One Dinopolis Instance	93
	Case 2: One <i>OLS</i> , Multiple Dinopolis instances	93
	Case 3: Multiple <i>OLSs</i> , Multiple Dinopolis instances	94
7.6.2	Catching up on Fast-Moving Objects	95
7.6.3	Merging Servers	97
7.7	The Address Management Module in Respect to <i>DOLSA</i>	97
8.	The Interrelation Management Module	99
8.1	The Logical Concept of Interrelations in Dinopolis	100
8.2	The Technical Concept of Interrelations in Dinopolis	101
8.3	Loading and Storing Interrelations	103

8.4	Creating and Deleting Interrelations	105
8.5	Endpoints of Interrelations	108
8.5.1	Making the Principle Work in Practice.....	109
8.5.2	Access Control Aspects	110
8.6	Interrelations Attached to Parts of Objects	111
8.7	Object Deletion Aspects	114
8.8	The <i>Dynamic Type</i> Mechanism	115
9.	The Virtual System Management Module	118
10.	The External System Management Module	122
10.1	Embedders	124
11.	Dinopolis Network Distribution Management	127
12.	Dinopolis Kernel Module Management	132
13.	The Security Kernel Module.....	135
14.	Conclusion	137
A.	The Hook Design Pattern	139

Acknowledgements

Many persons directly or indirectly influenced this thesis. On the one hand there are the scientists and students that were directly involved in discussions, prototype phases as well as design and implementation of the final system. On the other hand there are at least as many people who had a much more personal role in the whole process by supporting me morally or by “just” being important parts of my life. Several people have multiple roles in the process, because I am in the lucky situation that some colleagues of mine are actually also good friends of mine. It is impossible to give an order of importance of all of the involved persons, because without the help of only one of them this thesis would for sure be different or not existent at all. It is also close to impossible to write down what all the people have done for me, because this would fill a book of its own. For this reason I just want to mention the main roles that the people played in the whole process, well knowing that every single one of them would deserve much more mentioning in different respects. Therefore, when reading the following, please keep in mind that all who are mentioned are equally important and that all of them did much more than what is shortly outlined here.

Hermann Maurer as the head of the IICM supported development of the Dinopolis system throughout the years and he pushed me more than once to finally write down this thesis. He was also always very understanding if the progress of the project seemed to slow down during many iteration steps that resulted in very important internal changes but were not all too obvious from the outside. I especially want to thank Hermann for being more than just “the boss”. Rather, he always had an open ear for everyday’s troubles and has been a good and understanding friend throughout many years of my presence at the IICM.

The Dinopolis team at the IICM is the best combination of brilliant heads and interesting personal characters that is imaginable. No matter whether work was

fun or whether there were hard times (and there were many!), the whole team always did much more than what could be expected from a team. Some of the team members are also very good friends of mine and I'm very grateful that I was lucky enough that our paths met in the past and we have been going parts of them together. As is easily imaginable there has been a considerable member fluctuation throughout the years, nevertheless the key persons have been the same all the time. In the following I just want to mention the core of the current team in alphabetical order, without further describing the roles of the single members, because otherwise the acknowledgements would become way too long. Many thanks go to Karl Bluemlinger, Christof Dallermassl, Dieter Freismuth, Heimo Haub and Philipp Zambelli for the brilliant work that they have done throughout the years and for their cooperation.

Besides thanking the Dinopolis team at the IICM, I also want to thank the MTP team at the German Aerospace Center (DLR). They play the same positive role in the whole process as the team in Graz, work-wise as well as friendship-wise. Again I had good luck to meet such people and to get the chance to also have very good personal friends in some of them. As was the case in Graz, also in Cologne-Porz there was much fluctuation team-wise. Therefore I again just want to mention the current core team in alphabetical order. My special personal thanks to Erwin Bartels, Dittmar Padeken, Dirk Schwartzmann and Michael Wirtz.

As I already wrote above, a thesis is much more than just scientific work. Many very personal aspects play important roles, be it the chance to talk to somebody when there are clouds in the sky rather than sunshine in the life, be it the chance to share good moments or be it just the chance to know that somebody is there for you. I am in the lucky situation to have several really good friends – more and better friends than one could usually expect to have in life. Also some of the members of the Dinopolis team and of the MTP team belong to this category. However, this paragraph is dedicated to the friends that are “just” friends in a sense that they do not have a direct connection to my work. Friendship has no order of importance either, if only one of them were not my friend, my life would be different. Therefore, I again want to mention them in alphabetical order. My thanks and gratefulness for all the times that I could spend with them go to Manuela Burger, Gernot Höbenreich, Renate Pistrich, Monika Tragner and Daniela Wyss.

Very special thanks go also to my parents who did much more for me than what would be considered “normal” for parents. Throughout my whole life I always had their full support, morally, financially and in many other respects for whatever I wanted to do. As usual, not everything that children do is really fully understandable for their parents. Nevertheless, they gave me every chance to do what I thought was good for me. They have always been a firm backing for me throughout my whole life.

1. Introduction

This thesis represents the first comprehensive overall description of the Dinopolis distributed component middleware framework. Starting in 1997 a team of researchers at the IICM has been developing an open-source system that implements a massively distributable componentware approach (see [Freismuth et al 1997], [Dallermassl et al 2000] and [Dallermassl et al 2000b]). From 1997 on design and prototype-implementation phases have been going on until in 1999 version 2.0 of a system called *DINO* (=Distributed Interactive Network Objects) was presented at the CeBit in Hannover as the core of the *MTP* (=Medical Telematics Platform) prototype. The MTP project has been developed together with several researchers at the German Aerospace Center (DLR) and it has been the first system worldwide implementing arbitrarily distributed virtual patient records. Due to the strong interest among medical institutions, phase 2 of MTP, the design and implementation of a production release of the system, started end of 1999. Since then, several researchers at the German Aerospace Center (DLR) and at the IICM have been working closely together on Dinopolis (see [Schmaranz 2002a]) which has its origins in DINO 2.0 and will be the core of the production release of MTP.

From the very beginning several important cornerstones strongly influenced the design of Dinopolis:

- Dinopolis is designed to be completely platform- and technology-independent. The current implementation of Dinopolis is written in Java but other implementations, e.g. in C++, are possible and will be done in the near future.
- Dinopolis is a slim, extensible middleware system rather than a huge monolith. Due to its embedder concept it is able to integrate and combine arbitrary existing systems, such as databases, Web servers, distributed object systems, communication and collaboration systems, etc.

- The core of Dinopolis is a highly sophisticated, distributed, robust component model. Objects and components can reside anywhere on the network and in arbitrary embedded systems. Due to its middleware design Dinopolis takes over object and component integration and management.
- Dinopolis implements a highly sophisticated addressing mechanism that provides globally unique handles which are robust against object and component movement (see [Schmaranz 2002b]).
- Dinopolis implements a highly sophisticated object and component interrelation mechanism that provides freely typed n:m relations between arbitrary objects and components or parts of them (see [Schmaranz 2002c]).
- To keep the kernel of Dinopolis as slim as possible the concept of kernel modules that are loaded on demand is supported.
- Dinopolis supports a highly sophisticated, completely adaptable, role-based security concept.

These cornerstones are a result of several lessons learned from experience with today's heterogeneous distributed systems as well as from the development of Hyper-G (later: Hyperwave, see [Andrews et al 1995]). The main problems with today's heterogeneous and massively distributed information spaces are found in several different areas. First to mention is the way objects are addressed: in principle all addressing mechanisms that are in use today share the same problem, namely that they are not robust against object movement. The few implementations that deal with this problem (see e.g. [Voyager]) do not scale well in respect to the number of objects, users and especially to the number of consecutive move operations.

The second important problem is found in the way *Objects* and *Components* are defined (or in fact *not clearly defined*) in today's systems. In case of the Web an object can be anything. The only existing definition is that a mime-type provides some information. However, this is only a definition of what is called *Content* in Dinopolis. A real *Object* or *Component* represents more than just content! In distributed object systems like *Corba* (see [OMG]) it is defined how to execute methods of an object. Anyhow, this is only a definition of a subset of what is called *Operations* and *Services* in Dinopolis.

The third and not less important problem is found in the way *Interrelations* are treated in today's systems. In case of the Web there exist hyperlinks that repre-

sent 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 and are hardcoded and embedded in HTML documents. Other document formats support other kinds of hyperlinks, especially the XLink recommendation (see [XLink]) 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.

From the beginning of the first design phase in 1997 on it was clear that the key to developing a universal system is the definition of a clean and comprehensive as well as extensible object model. The definition of an *Object* has to reflect a clear separation of *Content*, *Meta-Data*, *Relations*, *Operations* and *Services*. As will be seen in chapter 2 this clear separation reflects a superset of all possible passive and active data models. All further features of Dinopolis are a result of this *Object* definition. Stable and robust addressing is supported for arbitrary embedded systems, even if they do not support robust addressing at all (like e.g. Web Servers). Details of the addressing mechanism will be discussed in chapter 7. Arbitrary n:m interrelations can be defined between arbitrary chunks of data in embedded systems, even if the systems are not even aware of interrelations at all (e.g. links between audio and video-streams or multi-destination hyperlinks in HTML). A detailed description of the interrelation mechanism can be found in chapter 8.

Besides the definition of this object model the second important decision during the development of Dinopolis was to implement it as a middleware layer. It does not make any sense to invent a brand-new system that overcomes all identified problems of today's existing software if this means that everything up to now has to be discarded and everybody has to switch to the "better" software. This approach has always led to religious wars between the defenders of the existing software and the inventors of a new system. Additionally, experience shows that the development cycle goes on and that the new system will soon be replaced by something "much better". In order not to develop "just another system" which overcomes identified problems but causes others and therefore leads to religious discussions, Dinopolis follows a different path: *Dinopolis is a middleware layer on top of existing systems that utilizes and combines their features by means of a highly sophisticated embedding mechanism.* Details of the embedding mechanism will be discussed in chapter 9.

Before going into the depth of the architecture and before discussing some special algorithms implemented in Dinopolis, a few use cases will be sketched in chapter 2 for explanatory purposes. The use cases described are representatives from the developers' point of view of the system, rather than from the end-users' point of view. During the whole design phase the main goal has been to support application programmers in writing end-user applications. Therefore the perspective of the end-users provided the context for the scenarios and the developers' needs were extracted from those contexts.

2. Use Case Scenarios

In this chapter two scenarios will be discussed briefly as representatives for a large number of thinkable applications. One of them is MTP, the Medical Telematics Platform that supports arbitrarily distributed, virtual electronic patient records. The other scenario sketches the needs from the point of view of a modern virtual electronic library.

As has already been mentioned, the following use case scenarios are both described from the application programmers' point of view. Both scenarios have one in common: several tries have been undertaken to implement more or less perfect systems and still there is no system available that comes even close to the requirements below. One of the reasons why this happened is the lack of a platform that provides application programmers with the abstraction level that is necessary to deal with the real problem and with high-level data structuring rather than with low-level data management issues. Dinopolis is designed to be such a platform which takes the burden of dealing with different databases and other storage devices from the application programmers by the concept of embedding them. Also the problem with different data formats that support different features is alleviated, because treatment of data is encapsulated within Dinopolis. All data that application programmers have to deal with is fully compliant to the Dinopolis object model and therefore treatment of data is unified and independent from special data formats.

2.1 The MTP Scenario

The main goal of MTP is to provide arbitrarily distributed, virtual, medical electronic patient records that consist of arbitrary combinations of multimedia and hypermedia objects. The parts of the patient record of one single patient are already distributed across many different systems on the network. In order not to become too theoretical let us consider a typical emergency scenario:

- An elderly woman from Cologne is on vacation in Austria and goes skiing in Bad Aussee. Unfortunately, this woman hits a tree and loses consciousness. The mountain rescue service brings her down to the valley station.
- At the valley station an emergency doctor and an ambulance car are waiting. From a first quick diagnosis the doctor suspects a possible skull fracture. Although the woman is still unconscious, the doctor decides that her condition is stable enough for a transport to the next hospital in Bad Aussee. The ambulance car brings her there.
- In the hospital an x-ray examination is undertaken.
- The x-ray examination proves the doctor's diagnosis and shows a very difficult fracture that cannot be treated in Bad Aussee because a CT would be needed but is not available. Therefore a helicopter flies the woman to the hospital in Graz.
- In Graz a CT-examination of the woman's skull is performed to be able to make a detailed diagnosis needed for a surgery.
- After the surgery the woman needs to stay in the hospital in Graz for two more weeks before a transport to Cologne is riskless.
- After two weeks the woman is brought to the hospital in Cologne. The doctor in Cologne decides that outpatient treatment is riskless and does not keep the woman in the hospital.
- After several sessions the woman is referred to her family doctor for post treatment of the still remaining permanent headache.

Let us consider some internals of this scenario to find out about the role of the Dinopolis platform in a distributed, virtual patient record environment:

- As an assumption let us say that the woman's identity can be found out immediately (e.g. she has a driver's license in her pocket). In this case her patient-record has to be located. This works only if the system supports direct access to a user-record without having to perform a distributed search (in the worst case world-wide!). It can easily be seen that an appropriate distributed lookup mechanism has to be provided to be able to fulfill this requirement.

If the identity cannot be found out immediately a special temporary "John Doe" user record has to be set up. This user record has to be merged with the patient record later when the identity becomes known.

- The reports of all doctors as well as the x-ray data and the CT data have to be written and inserted into the system. Due to e.g. the German and Austrian laws the responsibility to store and archive all the data is up to the appropriate institutions. It is not allowed (and usually not even possible) to store the data e.g. in the hospital in Cologne where the woman may be registered with her user record.

As a result all the data for one single patient is arbitrarily distributed across many different institutions. However, this still means that one virtual, distributed patient record has to exist which combines all those different chunks of data.

- The patient record is not only distributed across the network. The different data-chunks are also stored in completely different systems. For example the x-ray image could be stored in an Oracle database, whereas the CT image resides on a DB-2 server. The doctor's report includes both images, one from the Oracle Server in Bad Aussee, the other from the DB-2 database in Graz together with the text of the medical result. This report is then finally located in a specialized hospital information system in Cologne.
- Storing all data distributed across the network and being able to combine them to a virtual record is one of the difficult tasks. Nonetheless, there is even a much more important task to be performed: the patients' data is extremely sensitive and has to be protected from unauthorized access. According to the law the patient has the right to define who to allow access to the data. Therefore a distributed authentication and authorization mechanism with the possibility to pass on time-constrained rights has to be provided.
- Not only unauthorized access has to be prohibited, parts of the patient record that have the status of "medical documents" must not be changed at a later stage. To prohibit (or at least detect) such changes, a virtual, distributed medical document has to be signed electronically.
- There are cases where institutions are closed down. In this case all the data that the institutions store has to be moved to different institutions or even split up across several institutions. Such move operations must not break the consistency of the patient record.
- Each person in the system can occur in arbitrarily many different roles, for example a doctor can be a patient too. In case of a self-diagnosis a person can even appear

in two roles at the same time. The roles are dynamic in several respects. On the one hand a doctor can become a patient at any time. On the other hand the possible roles that a person can take over change over time. Doctors are not doctors from birth on.

- Not only can persons occur in different roles, they can also have different responsibilities in one and the same role. For example a surgeon can be the responsible doctor for a surgery or just one of the assistants. This also implies different rights in the system.

Lots of different medical scenarios can be found and have already been discussed in the MTP context. However, this would be far beyond the scope of this thesis. The scenario above gives enough insight into the underlying processes to be able to define which features a general platform, in this case Dinopolis, has to provide to enable application developers to design and implement a virtual, distributed electronic patient record:

Objects: All patient records consist of arbitrarily large sets of objects. No matter whether these objects represent textual information, x-ray images, CT images or other data, the operations on objects are always the same:

- No matter in which institution on the network objects are stored, the access mechanism always has to be the same. Therefore Dinopolis has to provide completely network-transparent access mechanisms.
- Also the access mechanism for objects has always to be the same, no matter in which external embedded system objects are stored. Therefore Dinopolis has to provide completely embedding-transparent access mechanisms.
- If objects are moved from one location to another, institution-wise or embedder-wise, they must not change their virtual address. Therefore Dinopolis has to provide a mechanism that supports globally unique object handles that are robust against object movement.
- To be able to compose a single virtual electronic patient record from many objects an appropriate inclusion mechanism has to be provided. Composition is best implemented using arbitrarily typed interrelations that are robust against object movement too. With this mechanism interrelations can represent inclu-

sions, hyperlinks, navigational aids or any other relationship that developers want to implement. The interpretation of an interrelation depends on its type.

- It has to be possible to attach arbitrarily typed descriptive and/or administrative meta-data to objects. Such meta-data could be for example the content-type (e.g. an x-ray image stored in TIFF format), the person who created this image, the creation date and the x-ray machine, etc.
- The content that is encapsulated by an object has to be accessible in a uniform way. However, it is not desirable to just provide streaming access and leave the rest up to the application programmers. There has to be a specialized content handler mechanism that supports uniform high-level treatment.
- Objects cannot only be used to encapsulate passive data. It is also possible that an object e.g. represents a wrapper for an external device, such as an x-ray machine. In this case the operations that the wrapped object provides have to be provided in a uniform way via so-called *Operations*. With the introduction of these operations the object becomes de facto a *Component* in the sense of (distributed) componentware.
- Sometimes it is not enough that operations are provided. Just imagine that an x-ray machine somehow has to be controlled by a person on a computer via a GUI. In this case the application programmers very often cannot know exactly how the machine works and which user interface they shall provide. This is usually up to the developers of the special machine. In this case, objects can provide so-called *Services*. These services are somehow comparable to operations because they transparently pass on some functionality of the underlying system to the platform. The difference is that services are defined to be GUI objects which developers can directly pass on to the users of their applications.
- The way data (i.e. content, meta-data, interrelations, etc.) is combined to an object has to allow electronic signatures. This means that dynamic aspects such as interrelations pointing to a new location after an object has moved have to be hidden behind the scenes. If something would change just for technical reasons although the information behind it does not change this would break the electronic signature mechanism.

Globally unique, robust handles: As has already been mentioned above, one of the key features of Dinopolis has to be the support of globally unique, robust object handles. No matter from which computer an object is accessed and no matter if and how often an object is moved, the handle always has to stay the same to ensure consistency of the record.

Transparent robust replication: For obvious performance reasons in a massively distributed system a mechanism has to be provided that allows transparent replication of objects. A simple cache is not enough because of the highly sensitive nature of medical data. The replication mechanism has to guarantee that the obtained objects are in sync with the original.

Arbitrarily typed robust interrelations: To allow composition of objects to form higher level abstractions, arbitrarily typed interrelations are necessary. These cannot be used only for composition but also for navigation and relation modelling. Due to the possibility to freely define types, interpretation and appropriate treatment of interrelations are up to the developers.

Arbitrarily distributed alias management: Globally unique, robust object handles are a very technical feature and due to administrative purposes they must never be defined or changed internally by application developers. Therefore it is clear that it cannot be guaranteed that such handles are easy to remember. For this reason Dinopolis has to provide a mechanism that allows context specific definition of appropriate easy-to-remember aliases. Here the term *context specific* means that several applications can share the same distributed information space (e.g. MTP and an electronic library as described below). Nevertheless it must not happen that aliases from one context block the creation of aliases in another context because of name-clashes. With an appropriate mechanism the aliases in different contexts do not influence each other unless explicitly desired. For example persons want the same alias in two contexts and therefore a higher-level alias can be defined. It could also be that objects are relevant for more than one context.

Security features: This is the most sensitive point in the whole design of Dinopolis. In the MTP scenario described here there is need for special user definition, authentication, access control, role management, time constraint management and signature management functionality. Other scenarios have partially completely

different requirements. Therefore the real requirement here is that Dinopolis has to support a generic security mechanism for access control. Specific implementations of this generic mechanism are hooked-up into the system to perform the appropriate tasks for the given scenario. This behaviour has to be in principle the same as is implemented in Java's security manager concept but much more granular and configurable.

Transactions: The distributed patient record consists of arbitrary many objects that are distributed across the network and composed by the use of appropriate interrelations. Therefore, high-level operations on the patient record usually involve several objects. In order to guarantee consistency of such operations rather than putting this burden on the application developers, appropriate transaction mechanisms have to be provided by Dinopolis.

Versioning: Once a medical document is electronically signed it is not allowed to change anything at a later stage. Nevertheless, there exist reasons to append new information and add new interrelations (e.g. to new medical research results concerning special diseases, etc.). In this case a new version of a medical document has to be instantiated that exists side by side with the old version. Therefore Dinopolis has to support version control for documents.

Queries: In some cases it is necessary to provide queries that are of either limited or unlimited scope in terms of network distribution. For example doctors could not be sure how to interpret the clinical picture of a patient and look for similar cases to get support for their diagnoses. Therefore distributed queries at least at meta-data level have to be possible. Nonetheless, as was already mentioned, medical data is extremely sensitive and it has to be made sure that queries do not unveil personal information. Therefore the requirement for being able to query data is closely coupled with the requirement to support anonymization of data.

Anonymization: Dinopolis has to support a mechanism that allows anonymization of data according to application specific rules. For example the rules could require to hide patient relevant meta-data and certain objects of the patient record which would allow guessing the patients' identity. In fact anonymization of data is strongly dependent on the implemented data and security model. For this reason anonymization is also a part of the specific security kernel module.

2.2 An Electronic Library Scenario

Electronic libraries are nowadays still understood to be just well searchable storages of electronic literature. I fully agree that the ability to define sophisticated queries is extremely important, but this is only one part of the truth. From my point of view a modern electronic library has to be more than just a storage device. It has to be a fully integrated, interactive electronic publishing solution (see also [Schmaranz 1996]). The term *publishing* does not only mean to make books, journals and articles available for reading. It also means to be able to attach opinions to existing material. Additionally, annotated cross-references between different published material, inserted by experts, are a very important feature. The possibility to place cross-references in a library also makes it possible to alert the readers to new results, long after an article has been published.

The publishing process itself has to be supported by allowing electronic refereeing. The referees' comments are represented by typed interrelations. If allowed by the policy of the publishers, authors have the possibility to watch the progress of refereeing and write their opinions about the referees' comments. This is again done in the form of typed interrelations. The same interrelation mechanism can be used for discussions about published material. Readers can attach their opinions to published material. For this kind of interaction between writers, referees and readers a sophisticated user and role management module is needed.

Extremely important for the acceptance of electronic libraries is the ability to place stable and robust citations in the system. However, just keeping citations stable internally is not the whole story. Citations also need to be kept stable from the system-external point of view. It must not happen that due to restructuring operations an external citation becomes invalid as is the case with today's common use of URLs. As is the case with medical institutions also electronic libraries can be closed down. In this case the content of the closed library is taken over by one or more other libraries and the whole material including all sorts of interrelations is split up across several locations. Nevertheless the original handles for the material must not change, otherwise all citations are broken.

Speaking of citations and interrelations one more aspect comes into play: interrelations are not limited to one server or institution! From my holistic point of view there has to be one huge, arbitrarily distributed electronic library. This means

that interrelations can reside anywhere in the distributed library and can interconnect material that resides anywhere else. With this possibility and the appropriate security and access-control mechanisms it is also possible to build a private, personally structured sub-library by using interrelations. The same feature also allows to provide collections of material for groups of users by making the interrelations visible to groups of persons. Using interrelations for navigation also allows to provide arbitrarily many parallel categorization schemas, public as well as private ones. Depending on the readers' areas of knowledge and depending on their skills in the appropriate areas, the right categorization schema alleviates finding of interesting material enormously.

Very important for electronic libraries is the concept of having time-constrained user-roles. Considering a typical refereeing process there are always one or more authors and one or more referees. Beside that, one or more users perform the administrative tasks for this single process. However, an author in one process can be a referee in another process and take over administrative actions in a third process. Therefore one person occurs in different roles in the system depending on the context. The time constraints are also different from process to process. For example, referees only carry this function until an article has been accepted or rejected. Once the material is published they must not act as referees for it any more. With today's standard user and group access mechanisms, modelling of time-constrained roles is a rather crude affair. The desired mechanism for this kind of role management is to support user-roles and to put users in context with a certain process by means of interrelations. With this mechanisms rules like *this user is a referee for this paper* are definable and manageable straightforward.

From the point of view of publishing companies one aspect is an absolute killer-criterion: utilization of existing systems! More or less all publishing companies already operate some sort of electronic library. This can be just an internal publication database, one or more Web servers or any other thinkable combination of databases and front ends. Therefore one of the strongest requirements for a system to be successful is the support of embedding existing systems and utilizing their features. As has already been mentioned in chapter 1 it is unthinkable to require that existing systems are discarded and replaced by something brand-new. A smooth transition path from existing systems to a new holistic publishing solution has to be

provided that relies on integration and combination of the existing databases to a huge, virtual, distributed library.

A whole book could be written about the features of a comprehensive electronic library and all processes involved in it. However, this is not the topic of this thesis. Therefore I will leave it at this short sketch as an outline for the features that a general platform like Dinopolis has to provide to enable developers to implement such a system. Identifying the technical requirements, it is easy to see that in principle the same considerations apply that were already worked out in the MTP scenario in section 2.1. The only difference is that anonymization does not play a strong role for electronic libraries.

2.3 Other Scenarios

During the design of Dinopolis many other scenarios have been considered, among them distance-education and Web-based-training systems, different electronic knowledge management approaches, communication and collaboration software, company portals and many more. It can easily be imagined that a discussion of all areas of application of Dinopolis that were considered would be way beyond the scope of this thesis.

Therefore just the conclusion from all research that was done in this area is presented here:

- All systems that were investigated added to the whole picture of Dinopolis and each considered system strengthened the clean and common definition of the Dinopolis data, structure, module and control flow model that will be presented in this thesis.
- One of the crucial points of the design of the Dinopolis architecture is a clean internal *hook* and *observer* model to be able to configure the system's responsible modules during runtime.
- The desired robustness against object movement is only achievable with a very special lookup service architecture called *DOLSA* (see also [Schmaranz 2002b]). One of the most important aspects in respect to the highly dynamic nature of an arbitrarily distributed system is scalability.

- Interrelation management has to be completely decoupled from data and document formats. Nevertheless it has to be taken into account that many different data formats support some sort of hyperlink, relation and interrelation mechanisms. These mechanisms have to be embedded into the Dinopolis world as mappings to the internal interrelation management facilities. Again scalability deserves special attention.
- As is the case with interrelations, similar requirements apply to management of meta-data: management of meta-data has to be completely decoupled from data and document formats. If data formats support meta-data management features these have to be seamlessly integrated into the system as mappings to the internal structures.
- Embedding of external systems is one of the most important features to ensure acceptance of Dinopolis.
- It has to be possible to transparently combine several embedded systems to one virtual system. The result is then a system with a superset of the features of the single embedded systems. For example if a company already operates a simple Web server with product descriptions, a database with technical data and a separate database with project management data for developed products it has to be possible to provide one virtual system that interconnects all the data and provides an overall view. All low-level actions that are necessary to build such a virtual combination have to be provided by Dinopolis. Application programmers only have to define the kind of combination and have to add the high-level interpretation logics. The term *high-level interpretation* stands for an abstraction level where only the kind of information matters (e.g. *this is management data*) rather than the storage systems or the way data is stored in them.
- It is impossible to implement special embedders for all kinds of existing systems. Therefore the concept of external system embedding has to be developed in a way that makes it easy for developers to add new embedders for special external systems. Furthermore it is not desirable to define a subset of functionality that all embeddable systems usually support. Embedding has to be done in a way that all functionality that the embedded systems provide can be mapped internally.
- Network, persistence, external protocol and schema, location, relation and replication aspects of a massively distributed system have to be fully transparent.

Application programmers do not have to care where objects reside, locally or remotely, how they are stored, etc. They just work with objects and the rest is managed by Dinopolis and therefore hidden from the developers. This is the only way to enable application programmers to concentrate on the problem-specific abstraction level rather than to have to consider all different kinds of low-level data and control flow aspects.

- It is impossible to foresee all areas of application of a universal, distributed object and component system such as Dinopolis. Therefore the architecture has to reflect this fact by defining a very slim system kernel and a clean concept of kernel modules grouped around this base system. It is not enough to only define a common module management interface, because also some very important parts of the system, such as security features, have to be implemented as kernel modules. Therefore some additional well-defined interfaces and concepts as well as a clean system bootstrapping mechanism have to be implemented.

Before discussing the final Dinopolis system architecture the following chapter describes the object definition that is valid inside Dinopolis. This definition is the key to all further considerations concerning the resulting architecture.

3. The Dinopolis Object Definition

The term *object* as well as the term *component* have been used as buzz-words in different contexts for a long time. Therefore there exist many different and even contradictory definitions. This is also one of the reasons for discussing the concept of a *Dinopolis Object* before coming to the overall Dinopolis system architecture. The following points apply in the Dinopolis context:

- A *Dinopolis Object* is an addressable entity in the system.
- Addressing a *Dinopolis Object* is always done via globally unique handles. It is guaranteed that one handle always refers to one and the same object and vice versa. It can never happen that a *Dinopolis Object* is accidentally replaced by a different one. The Dinopolis handle mechanism guarantees internal consistency (for further details see chapter 7).
- Dinopolis is technically spoken a *distributed component system* (see also [Schmaranz 2002a]). This fact is reflected in the nature of a *Dinopolis Object* which is a component in the sense of componentware rather than just a low-level data-encapsulating entity. For this reason the terms *Object* and *Component* are used interchangeably throughout this thesis unless certain circumstances force the usage of one of the two terms to emphasize a special meaning.
- A *Dinopolis Object* can itself be a compound made up of several *Dinopolis Objects*. Different models of composing compounds apply. All of them are corresponding to the appropriate mechanisms known from OO programming paradigms, e.g. derivation, inclusion, etc. This allows modelling of arbitrary component-type hierarchies.

Please note that compounds in Dinopolis are used to reflect types rather than to reflect e.g. navigational structures. All navigational structuring is done through the use of interrelations (see also chapter 8).

- A *Dinopolis Object* encapsulates content. The term *content* stands for everything that can be considered (passive) data in a broad sense, e.g. a document, streaming data or whatever else could be (persistent) state information.
- A *Dinopolis Object* can hold arbitrary meta-information attached to it. Meta-data can be of arbitrary nature, e.g. descriptive (like *content-type author, creation date*, etc.), administrative (like *access-rules*), application dependent (like *display-hints*), etc. For this reason meta-data of a *Dinopolis Object* is represented as a tree-structured container of hierarchical keys with values of arbitrary type, accessible through the keys. *Arbitrary type* does not mean that it is up to the application to interpret the type correctly. Meta-information is strongly typed, but the types are application-defineable and not limited to a certain hardcoded set of possible types. This means that *Dinopolis Objects* themselves can be used to represent certain parts of the meta-information.
- Arbitrary n:m interrelations can be attached to *Dinopolis Objects* to refer to the whole object or only to parts of it. With the possibility to use *Dinopolis Objects* as meta-data it is therefore also possible to attach interrelations to meta-information or parts of it. This is sometimes important, e.g. when interlinking the *author* attribute of an object with an entry in an address database.
- What makes a *Dinopolis Object* a component in the sense of componentware is the possibility to provide arbitrary functionality to the outside world. Two different kinds of functionality were identified: *Operations* and *Services* (see below). As is the case in OO programming languages the provided functionality of a *Dinopolis Object* are access controlled so that component encapsulation cannot be broken if properly applied. Compared to OO programming languages access control is much more sophisticated in Dinopolis and not limited to e.g. **public**, **private** and **protected**. This would not be feasible for a dynamic, arbitrarily distributed component system, because distinction on an object-type level is for sure not sufficient. For this reason access control in Dinopolis incorporates the whole security mechanism (see also chapter 13) to be able to define rules on a *user, group* and *role* level. A definition of time-constraints is also possible.
- As has been mentioned above, one kind of functionality that a *Dinopolis Object* can provide are the so-called *Operations*. These are fully comparable to meth-

ods in OO programming languages: they can be called with a predefined set of parameters, perform a task and return a result.

- The second kind of functionality that a *Dinopolis Object* can provide are the so-called *Services*. *Services* are comparable to methods with the difference that they provide a user-interface that an application can request and pass on to the end-user. The reason for the necessity of a service-mechanism is easy to find: sometimes it can happen that too much knowledge about the internals of a component is necessary to call the correct *Services* with the correct parameters. Just imagine a *Dinopolis Object* that has its origin in a very special database which supports very special user access rights. If applications would want to provide a GUI element that allows users to change access rights for objects in this database, then in-depth knowledge about access management and rule syntax of this database would be necessary. With this knowledge an appropriate GUI element could be implemented by hand. This is against the philosophy of Dinopolis because the necessity to deal with this low-level knowledge distracts developers from their abstraction level. Further the concept of embedding and virtualizing systems would be broken by the necessity for special treatment of some systems. For this reason such embedded system immanent features are provided through services that are written by the developers of the appropriate embedders or special object-types.
- Considering the composition of *Dinopolis Objects* it is easy to recognize that one of the most important design issues is the definition of a standard, uniform interface to provide access to content, meta-data, interrelations, methods and services. For example, a part of this uniform interface is the possibility to find out about certain capabilities, e.g. whether a *Dinopolis Object* supports versioning.

According to the definition above, the structure of a simple *Dinopolis Object* (as opposed to a *compound* object) looks like shown in figure 3.1. Additionally to the object itself also possible relationships between objects through the use of n:m interrelations is sketched.

Combination of several *Dinopolis Objects* to a composite object, as has been discussed above, is also (internally) modelled by the use of interrelations of a special type. Due to the implementation of composite modelling by special types of interrelations several advantages are gained in comparison to the standard derivation mechanisms of OO programming languages:

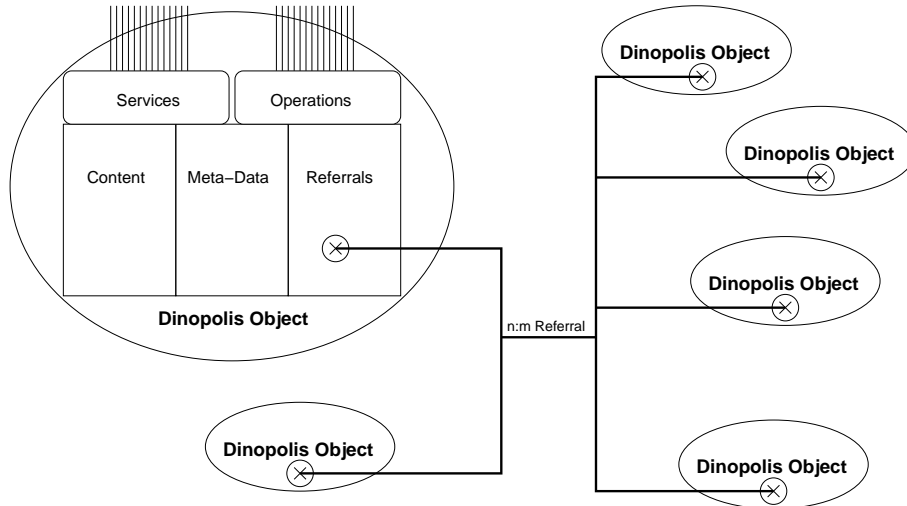


Figure 3.1: A simple *Dinopolis Object*

- The semantic difference between different types of subclassing is made explicit by defining the appropriate type. Therefore it is possible to distinguish between e.g. specialization and extension through subclassing. Arbitrary different semantic types of subclassing can be defined and the rules are part of the appropriate interrelations, rather than predefined at a programming language level (see also chapter 8).
- Not only subclassing is available, also mechanisms like inclusion, delegation and others are easily modellable. This also allows very special object composition models that only make sense for a very limited range of applications.

The exact technical description of the way in which composites are built by the use of interrelations will be given in detail in chapter 8. However, it is necessary to jump ahead a little to be able to sketch a compound object as can be seen in figure 3.2.

The sketch in figure 3.2 is deliberately not drawn following a standardized notation like e.g. the notation for UML class diagrams. The reason for this is an intention to show the technical nature of the single parts involved in the composition rather than the semantic model. The three objects on top of the sketch represent arbitrary simple or composite objects that act as the base “classes” for the composite. The 1:3 interrelation drawn from the composite to the base objects is a typed interrelation that represents inheritance. The bubble *Inheritance Rules* holds the rules that are

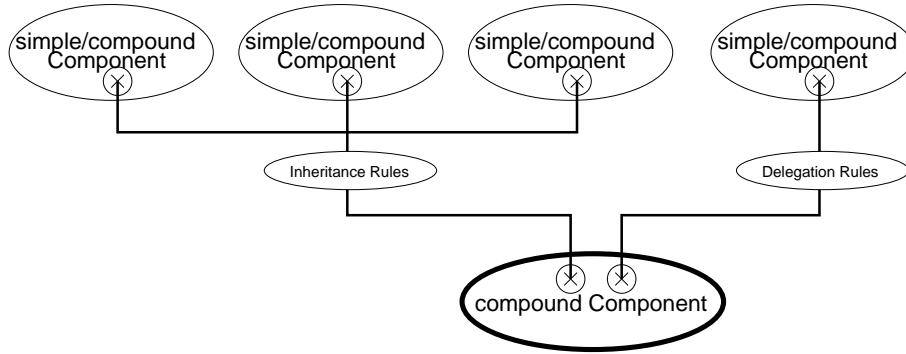


Figure 3.2: A compound *Dinopolis Object*

valid for the special kind of subclassing that applies. The reason why the inheritance rules are drawn in a bubble is the necessary jump ahead to chapter 8, where one of the results will be that interrelations are internally represented as *Dinopolis Objects* themselves. The *Dinopolis Object* that stands behind the interrelation is responsible for holding the appropriate rules and evaluating them on demand (e.g. when a method is called on the derived composite).

The composite in the sketch is not only derived from the three objects on top of the sketch but also utilizes a delegation mechanism to provide certain functionality. The mechanism is the same as has been described for subclassing above: a special interrelation with the appropriate delegation rules interconnects the composite with the destination object of the delegation. This destination object can itself again be either a simple or a composite object.

4. Overall Dinopolis System Architecture

A very accurate data and control flow analysis of all use-cases that were investigated, together with the desired definition of a *Dinopolis Object* taken into account, finally resulted in a very modular overall system architecture that is presented in this chapter. The main goal that had to be achieved was to obtain a very slim base system with maximum flexibility and extensibility. Further, the resulting architecture had to be as independent as possible from existing software and programming languages. However, being independent from existing software does not mean that absolutely everything has to be developed from scratch. It just means that the architecture is modular enough and encapsulation is clean enough that existing technologies can be used but the system is not dependent on special packages. For example the network management module of the final architecture can very well use different existing high or low-level protocols as its transport layer (e.g. Java RMI). Nevertheless it is not dependent on them. To double-check the best possible independence from programming languages, high-order design tests were made with implementation studies in C++ and in Java as today's most widespread representatives of OO programming languages. The whole system is designed to utilize special features like dynamic classloading by name during runtime if available. Nevertheless, it does not rely on the presence of such mechanisms, because they do for example not really exist in C++.

The overall system architecture which was the result of a very long and intense design-phase and involved several scientists and researchers at the IICM and at the German Aerospace Center (DLR) can be seen in figure 4.1.

The boxes represent the single modules and the arrows represent the main data and control flow paths between the modules. The term *main path* is used for paths that involve *direct* calls to the interface of the appropriate module. As will be explained later there also exist *hooked* calls. These are implemented using the *Hook*

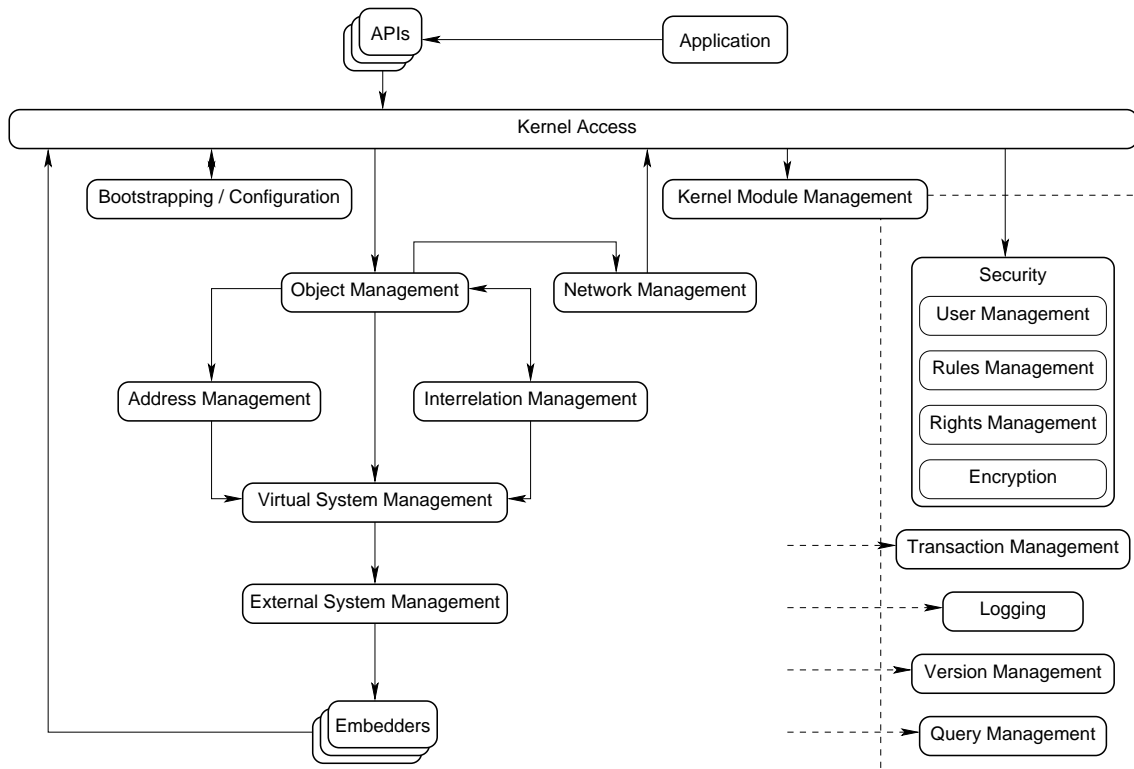


Figure 4.1: Overall Dinopolis system architecture

design pattern (see also appendix A) and therefore do not result in direct dependencies. There were four main reasons to introduce hooks in the Dinopolis system architecture:

1. Hooks drastically reduce dependencies and add to the overall modularity of the system.
2. Hooks decouple the knowledge *when* to trigger an action from the knowledge *who* performs the action (and also *how* to perform it). This greatly enhances the flexibility and configurability of the system.
3. Hooks are connected during runtime and therefore make a clean bootstrapping process possible that follows the logical instantiation and setup order of the system rather than being dictated by technical constraints of reference-chains.
4. The possibility to connect parts of the system during runtime makes it possible to reconfigure the whole system without shutting it down, if desired (and allowed by the security module).

Hooks (i.e. actions that are provided) as well as *hook-ups* (i.e. triggers that need to be connected to certain hooks) are also a part of the interfaces of the single modules.

Besides hooks there also exist many parts in the interfaces of the single modules that are represented by *Observers* (see also [Gamma et al. 1998]). These have also been introduced to reduce direct dependencies and to extend customizability and extensibility of the system.

Before discussing the tasks of the single modules let us have a quick look at some important overall concepts that are reflected in figure 4.1 and that are necessary to gain a deeper understanding of the system:

- In principle the system design distinguishes between three different groups of developers utilizing Dinopolis:
 1. High-level application developers.
 2. API developers.
 3. Kernel-module and embedder developers.
- High-level application developers write end-user applications based on the functionality of Dinopolis that is accessible through the so-called *APIs*. An *API* in this context is a problem-based view to the system that hides certain technicalities. Speaking in terms of software technology an *API* is a *facade* (see also [Gamma et al. 1998]) or it can also be seen as a *view* in terms of the *model-view-controller* paradigm. In fact it was called *View* in the early phases of architectural design, but experience showed that this name caused confusion because it was often misunderstood to be a graphical view of the system (which is not true). The only actions that high-level application developers can request are to choose between one of arbitrarily many provided APIs and to utilize the functionality that they provide. It is possible to work with several different APIs at once (they are synchronized).
- API developers are responsible for providing appropriate problem-based APIs that high-level application developers utilize (e.g. a medical-document based view of the system). The only possibility for an API to communicate with the system is to utilize certain *Proxies* (see also chapter 5 and [Gamma et al. 1998]) that are provided through the *Kernel Access Module*. Every single call through these *Proxies* is fully security-checked by the *Kernel Access Module*. There is no way

to access modules inside the kernel of Dinopolis directly, surpassing the system inherent security mechanisms.

- Kernel-module and embedder developers can be considered system programmers. They provide specialized kernel modules for certain tasks (e.g. a billing module) and embedders for external systems that have to be supported. The important difference between these developers and the high-end application and API developers is that the security-barrier of Dinopolis is located in the *Kernel Access* module. Parts of the system lying in the so-called *kernel area* are only security-checked once when they are loaded, rather than security-checking every single action that they perform (see below).
- The dashed area at the right of figure 4.1 represents the area where kernel modules are located. The *Kernel Module Management* module is responsible for loading them. To prohibit viruses and trojan horses the kernel modules are checked once whether they can be trusted. After they have been loaded they are considered to be trustworthy and therefore well-behaved. This is necessary because otherwise severe problems would occur (e.g. the security module checking itself during its checks resulting in an endless loop, etc.).
- The *Security* module is located in the kernel module space of the system to allow adaption of security measures according to the needs of certain applications. Being a kernel module it is exchangeable, nevertheless its existence is required by the system. This is the reason why there is a solid arrow drawn from *Kernel Access* to *Security*. There can exist an arbitrary number of optional kernel modules with arbitrary functionality. For demonstration purposes four examples of thinkable kernel modules are acting as representatives in the drawing. The dashed arrows indicate that they are accessible by whatever part of the system which is aware of their existence (usually *Proxies* will exist to pass on their functionality to the appropriate *APIs*).
- Embedders are hybrid modules in respect to the clear distinction between kernel and user space which are separated by the *Kernel Access* module. The passive part of embedders (i.e. the part that fulfills requests from inside the kernel and drives external systems) is considered part of the kernel. If an embedder has an active part at all (e.g. when embedding an active system that can trigger events) this active part is considered to lie above the *Kernel Access* module. All actions

triggered by embedders are fully security-checked to prevent attacks through embedded systems. This is the reason why there is an arrow from the *Embedders* to the *Kernel Access* module. As is the case with kernel modules, embedders are checked once when they are loaded. The difference between embedders and kernel modules is that embedders are not considered to be well-behaved. Therefore they are encapsulated in a kernel space of their own and do not have direct access to the functionality of any of the modules in the kernel.

- The *Network Management* module that is responsible for all distribution aspects of Dinopolis is also a hybrid module but is treated slightly differently than embedders. This module is a part of the core of the kernel and therefore not encapsulated in a separate kernel space. Nevertheless, the *Network Management* module is implemented in a way that incoming action requests are not directly executed but routed through the *Kernel Access* module to force a security check. This is the reason for the arrow that points from *Network Management* upwards to *Kernel Access*. In fact the *incoming* part of the *Network Management* module is considered to lie above *Kernel Access*, whereas the *outgoing* part is considered to lie below it. For simplicity reasons the module is not split up into two parts, otherwise the drawing could become confusing.
- The *Bootstrapping / Configuration* module is responsible for bringing up the whole system including all parts of the kernel. The first action during bootstrapping the system is to instantiate the *Kernel Access*. After that several protocols apply for loading the other modules. This is symbolized by the double arrow between the two modules. All other arrows from this module to the rest of the system were omitted because they do not add any information and would only make the drawing confusing.

Before coming to an in-depth discussion of the design details of the single modules and their interfaces as well as to some special algorithms, a short overview about their area of responsibility is given in the following:

Kernel Access: As has already been discussed, the *Kernel Access* module is the security barrier to the kernel. All system calls have to go through this layer. In order not to run into the problem that all system calls that could ever be possible (including calls to completely unknown kernel modules!) have to be

defined in the *Kernel Access* module, a special proxy architecture was chosen for this task. Inside the kernel so-called *functional modules* exist. These can either encapsulate some of the “real” modules discussed here or they can also be facades that combine the functionality of some modules to hide internal details. Every instance of a functional module that is *published* to the outside is encapsulated by the *Kernel Access* module and a *Proxy* is passed on as a representative. Calls to the *Proxy* then end in the *Kernel Access* module which in turn triggers a check by the *Security* module. If the call is allowed by the *Security* module it is finally passed on to the according instance of the *functional module* for execution. The result of an operation (if any) is propagated back up to the caller. To be able to perform the appropriate security checks, each *functional module* is registered with the *Security* module before it is published. Having a *functional module* registered and checked before publication is also the task of the *Kernel Access* module. This mechanism allows that the *Security* module can prohibit publication if it either does not trust the appropriate *functional module* or (depending on the policy) if it does not know it. For further details of this mechanism please read chapter 5.

However, this mechanism alone would only work for single-user applications. To allow multi-user operation, all applications have to have their own execution context. Providing the appropriate contexts is also the responsibility of the *Kernel Access* module and is done by means of context-specific APIs. This is also one of the reasons why applications are not allowed to communicate with the *Kernel Access* module directly but have to request an API instance first.

Object Management: The *Object Management* module is responsible for two closely coupled areas of responsibility in objects’ lifecycles:

1. It has the responsibility to perform all internal object structure operations when creating, loading, storing, deleting or rearranging objects. As *internal object structure operations* all operations are seen that influence content, meta-data, attached interrelations (only attachment of interrelations, not the interrelations themselves!), methods and services. This includes collecting the persistent data for each of the parts and also building the final object according to all compound and subclassing aspects.

2. It has the responsibility for all internal operations that are related to moving objects around either between different virtual systems or across the arbitrarily distributed Dinopolis instances. Depending on the source and destination location of the objects' persistence, internal restructuring of objects can be necessary during movement operations.

Address Management: The *Address Management* module is responsible for all aspects and operations related to handling globally unique, robust object handles. In fact *Address Management* consists of two parts:

1. One part of *Address Management*, namely *Local Address Management*, resides in the kernel of the system and is responsible for the local aspects of physical-address handling. *Local Address Management* also provides the stub to *Global Handle Management*, which is the second part of *Address Management*.
2. *Global Handle Management* represents an implementation of the *DOLSA* algorithm which was especially developed for Dinopolis. This part of *Address Management* is in fact not residing in the kernel itself but is run as a separate network service. Inside the kernel only the stub to it exists, as described above. The reasons to run *DOLSA* as a service of its own were first, because *DOLSA* is so universal that it can also be used by other systems than Dinopolis and second, because of administrative as well as availability reasons. Further it is absolutely desirable to clearly separate a lookup-service from the instances that utilize it.

Interrelation Management: The *Interrelation Management* module is responsible for handling all different kinds of interrelations. The reason why there is a double-arrow line between *Interrelation Management* and *Object Management* is the fact that interrelations themselves are represented by full-featured *Dinopolis Objects* and therefore persistence operations on interrelations are performed by the *Object Management* module.

Virtual System Management: This module is responsible for managing virtual combinations of several external, embedded systems that appear as one single system internally. Arbitrary combinations of any desired complexity are thinkable. Just to present one example, such a combination could be a simple filesystem together with a relational database. The *content* of each *Dinopolis Object* that has its origin in this virtual combination could then reside in the filesystem, whereas its

meta-data is stored (and searchable) in the database. The virtual combination aspect is hidden behind the scenes and from within Dinopolis every object looks the same. This aspect is also called *persistence combination transparency*.

External System Management: This module is responsible for all aspects of managing embedders for external systems. Existing embedder implementations are registered with this module so that they can be instantiated on demand when embedding external systems. Instantiation and appropriate setup of embedders for each concrete case is performed here. As has already been mentioned embedders are also security-checked on instantiation, nevertheless they have a special kernel space of their own which does not allow direct access to the rest of the kernel. Providing this space and encapsulating it is also one of the tasks that the *External System Management* module has to perform.

Embedders: Embedders can be seen to be special drivers that have to fulfill at least a minimum set of functionality necessary for an external system to be embedded. Every access to embedded systems is performed through special embedders. The minimum functionality that is dictated by the interface is only to be able to read data when an address (not handle!) is passed. Handle-to-address mapping and vice versa is already performed by the system. If embedded systems allow write operations there is also a special interface for it. In order not to be forced to define a certain set of operations for every thinkable system that could be embedded, the design of embedders also relies on *Operations, Services, Hooks* and *observable* functionality as well as events. With this concept embedders can pass on the full functionality of an external system to Dinopolis. It is also possible to embed active systems like e.g. chat-servers because triggers from the outside are supported by Dinopolis.

Network Management: The *Network Management* module is responsible for all network transparency aspects of *Dinopolis Objects*. Inside the system all *Dinopolis Objects* are treated the same, no matter whether they reside on the local machine or somewhere else in the virtually distributed world. All aspects of network transparency are therefore managed by this module.

Kernel Module Management: This module is responsible for fetching, instantiating and removing kernel modules. This responsibility also includes that the *Security* module is informed and asked appropriately for each module to be instantiated

or removed. Also the order of module loading and the responsibility for necessary hooks and hook-ups between modules and the kernel are managed by it.

Security: Although the *Security* module is a kernel module, its existence is required. This allows implementations of different complexity to be used for different scenarios. It is even possible to implement an empty *Security* module for very special applications in already secure network regions (although this is not recommended). To prohibit *Security* module implementations that are in fact trojan horses, the bootstrapping mechanism first instantiates a trusted but minimal security module that is able to check whether an implementation of a *Security* module can be trusted. If yes, the desired module replaces the minimal one.

5. The Kernel Access Module

The *Kernel Access* module defines the borderline between the kernel space and the user space of Dinopolis. The criterion for this distinction is the presence of security checks. Every single request coming from the user space that lies above the *Kernel Access* module (see figure 4.1) is security checked before being passed on to the kernel space that lies below the *Kernel Access* module. All interactions that take place between single modules inside the kernel are not checked any more.

To achieve this behaviour, a mechanism had to be found that allows the *Kernel Access* module to intercept all calls from the user space in order to be able to check them. Since there are no interception mechanisms for direct method calls defined in today's OO programming languages and also because such mechanisms would result in unwanted side-effects if they did exist, the design of choice was an implementation that is based on the use of *Proxies*.

However, the use of *Proxies* is not the whole story. For multi-user applications the application context is important to be able to distinguish between different concurrent users. Therefore the *Kernel Access* module also manages execution contexts for *API* module instances.

An additional essential aspect is the distinction between two semantically different kinds of requests. These use in principle the same protection approach, nevertheless they reside on different abstraction levels and are therefore treated separately:

Kernel functionality requests: In terms of Dinopolis the kernel *manages* objects.

Therefore *kernel functionality requests* are those which have to do with management aspects, e.g. *give me this object, create an object, list objects, etc.*

Object functionality requests: Objects support different kinds of access to their content, meta-data and relations. Further, objects also support *Operations* and *Services*. It can happen that objects only encapsulate functionality (e.g. a chatroom object). Nevertheless this functionality differs semantically from core

kernel functionality. Access to the state of objects as well as calling *Operations* and *Services* on them also needs to be security checked.

In order to avoid mixing up the two semantically different kinds of requests they are treated strictly separately from now on. The base mechanisms of the two are similar, however, different constraints apply. Kernel functionality calls are provided by so-called *functional modules* and object functionality requests are provided by *Dinopolis Objects* (short: objects).

5.1 Mechanisms Defined for Functional Modules

The principle of *functional modules* and the appropriate basic mechanisms that control access to them can be sketched as follows:

- In the system's kernel space there exist arbitrarily many so-called *functional modules*. A *functional module* can either represent a *Bridge* (see [Gamma et al. 1998]) providing a front-end for one of the “real” modules described in this thesis or it can be an arbitrary *Facade* (see [Gamma et al. 1998]) that combines technically separated modules to a logical entity. The *functional modules* form the level of abstraction that has to be provided for the API developers. *Functional modules* do not only exist for core parts of the kernel. They can also be provided by kernel modules to pass on their functionality (otherwise kernel modules would not make too much sense anyway).

If special *Bridges* or *Facades* are needed (and implemented) inside the kernel they are not named *functional modules* in order not to lose the clear definition.

- *Functional modules* are not used directly inside the kernel. They are exclusively conceived as an abstraction layer to be passed on to the user space. Passing them on to the user space is called *publishing*. In this process the *functional module* is registered with the *Kernel Access* module. The *Kernel Access* module in turn lets the *Security* module check the published *functional module*. This is necessary to prevent trojan horses from opening trapdoors that provide unauthorized access to the kernel of the system.
- If a *functional module* passes the security check (i.e. the specific implementation is accepted by the *Security* module) it is officially registered as part of the published interface of the system kernel.

- Before applications are allowed to communicate with the Dinopolis system at all they have to register with the *Kernel Access* module to trigger creation of a specific execution context. The execution context is a container of hooks and references to specific parts that are relevant for certain modules. Such specific parts can be e.g. security relevant (user, group and role of a specific application). Also other parts of the kernel can add parts to the execution context, e.g. the transaction kernel module. To enable all parts of the kernel to add desired parts to the execution context, it is possible to register with the *Kernel Access* module to signal this fact. On creation of new execution contexts the registered modules of the kernel are asked to add their part to the context.

Let us consider the nature of the execution context on the example of the security relevant part of it: a specific *Security* module requires a specific security context. One implementation may e.g. just support user and group management, whereas another implementation also supports user roles. One implementation may support authentication by a simple username / password dialog, another implementation may require the use of a smartcard.

This implies that the security part of the execution context contains state information as well as necessary methods to change this context (e.g. authentication). Special functionality of the specific parts of the execution context is also provided in the form of *functional modules* and treated exactly like the other functional modules in the kernel: access is granted through *Proxies*. It is strongly recommended that such functional modules also provide *Services* (as do *Dinopolis Objects*) to enable applications to present appropriate GUIs to the users (e.g. a username / password dialog) without needing an in-depth knowledge about the currently active *Security* module.

- *API* modules (residing in the user space) provide problem specific views for applications. Whatever calls *APIs* provide, they are always performed in the application specific execution context.

In order not to allow *APIs* to open trapdoors to the kernel, all *API* modules have to be registered with the *Kernel Access* module and are security checked accordingly. Only if the *Security* module accepts them, they are published and accessible for applications.

- Applications are not allowed to communicate with the *Kernel Access* module directly with the exception of requesting an execution context and requesting *APIs*. Therefore access of applications to the Dinopolis system is regulated through the *APIs* that are accepted by the *Security* module. This behaviour was introduced to be able to provide limits to applications. For example, in the medical context applications are only allowed to see a medical view of the whole information space rather than a low-level object view which would reveal the internal structure of medical documents. This is achieved by only allowing a very special “medical” *API* and by not accepting others.
- When an application requests an *API*, one more security check is performed because not every application is allowed to see every “allowed” *API*. If access is granted the *Kernel Access* module passes on an instance of the requested *API*. Depending on whether an *API* is designed to hold context specific information, instances of *APIs* can be shared or not. To reflect this situation and let the *APIs* decide on sharing of instances, a *Prototype* based implementation (see [Gamma et al. 1998]) was chosen.
- An *API* is not allowed to access kernel functionality directly either, because each single call has to be security checked. For this reason *APIs* request access to the *functional modules* that they need. Such requests are again security checked and, if granted, the *Kernel Access* module passes on instances of the appropriate *Proxies* to the requesting *API*.
- An application requesting an action calls the appropriate methods of an *API* that it has obtained. Each of the method calls of the *API* is translated into one or more functional system calls of one or more of the *Proxies*. Each system call of a *Proxy* ends in the *Kernel Access* module. The *Kernel Access* module lets the *Security* module check the call. If the call passes the check the *Kernel Access* module propagates it to the appropriate instance of the *functional module* that stands behind the proxy. This finally triggers the “real” action. The result of this action is propagated back up to the application.

It is guaranteed that not more than one instance of a *Proxy* exists for each *functional module*. This forces a context-free implementation of *Proxies* and allows reference-equality comparison of *Proxies*. If two *Proxy* references are equal it is guaranteed that also the *functional modules* behind them are equal.

However, depending on the *functional module* it can happen that context-specific state information needs to be stored (e.g. some sort of history, etc.). In order not to prevent the straightforward implementation of such behaviour (by using multiple module instances), the algorithm for the *Kernel Access* module to obtain a *Proxy* involves the use of *Prototypes* that can internally control the number of instances: if a *functional module* is context-free the existing *Proxy* instance is passed on, otherwise creation of a new instance of the *functional module* is triggered and a new *Proxy* that refers to this one is instantiated and passed on.

In order not to become too theoretical a possible runtime situation is sketched in figure 5.1.

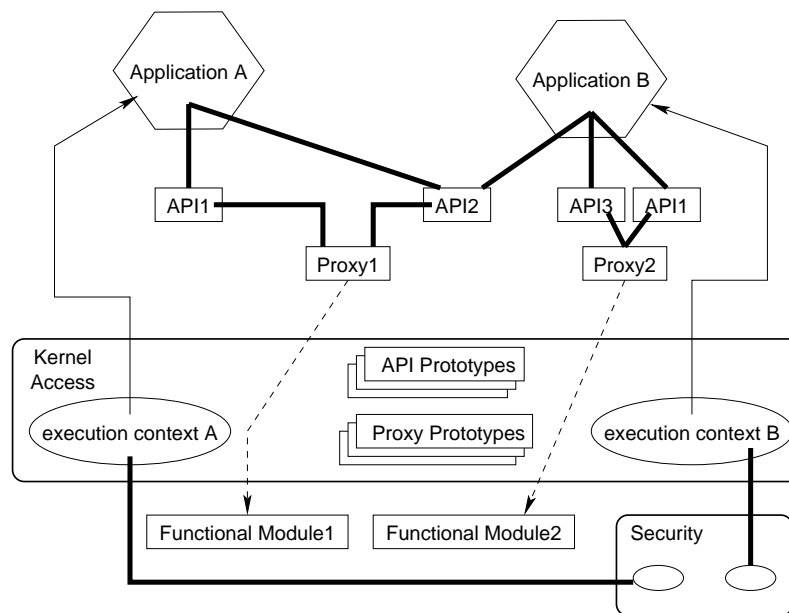


Figure 5.1: One possible runtime situation

First of all, applications have to register with the *Kernel Access* module to obtain their execution contexts. These are kept inside the *Kernel Access* module for security reasons. When a context is requested the *Kernel Access* module first sets up its administrative part. Then it contacts the *Security* module. This one in turn creates the security specific part of the context and adds a reference to it in the *execution context* (please note that the *Security* module could also want to reject an application at all for whatever reason. In this case it can just create an internal “do not allow

anything” context). The reference from the general part to the security specific part of the context is represented by the thick lines in figure 5.1. The fact that each application belongs to exactly one context is represented by the thin arrows from the contexts to the applications.

After registration an application needs one or more *APIs* to obtain access to the system. For each of the available (=allowed) *APIs* the *Kernel Access* module holds a *Prototype*. On request the *Security* module is asked whether the *API* may be given to the application. If yes, an instance of it is requested from the *Prototype* and passed on to the application. The fact that the *Prototype* decides whether or not to create a new instance of an *API* is reflected in figure 5.1: *API2* is shared among applications whereas *API1* is not. The thick lines from the applications to the *APIs* represent references.

In order not to provoke misunderstandings regarding to the existence of different *APIs*: there is no rule whether different *APIs* have to have disjunct functionality or whether they just provide different views of the same functionality. This is up to the designers of *APIs*. Anyway, access to the same objects through different *APIs* is synchronized by the kernel itself and therefore the kernel guarantees consistency of different views of the same object space.

To communicate with the Dinopolis system the *APIs* have to request access to the desired *functional modules*. The *Kernel Access* module checks whether such requests are performed by *APIs*, because direct requests by applications to obtain access to *functional modules* are not allowed for safety reasons.

When an *API* requests access to a *functional module* the *Kernel Access* module lets the *Security* module check permission and if it is granted an instance of a *Proxy* for the desired *functional module* is requested. As is the case with *APIs*, also *Proxies* for functional modules are managed by appropriate *Prototypes*. The *Proxy* prototypes notify the appropriate *functional modules* to give them the chance to create new instances if they are not context-free. If a new instance of a *functional module* is created this results in the creation of a new instance of a *Proxy*. Otherwise an existing instance is shared. In figure 5.1 it can be seen that sharing of *Proxy* instances can also happen between applications. This is not a security problem since all requests are treated according to the execution context of the application.

One case is not explicitly shown in figure 5.1: creation of an additional instance of a *functional module* when a new *Proxy* instance is requested. However, this case is trivial and therefore it does not make sense to discuss it here in detail.

The algorithm for propagation of a request (direct or hooked) from an application down to the kernel and the propagation of the return value back up again according to this architecture is the following:

1. An application calls a method of the appropriate *API*.
2. The *API* calls one or more methods of the responsible *Proxy* (or *Proxies*). This call takes place in the applications' execution context.
3. Each call is propagated from the *Proxy* to the *Kernel Access* module.
4. The *Kernel Access* module contacts the *Security* module and asks to check permission of the call in respect to the execution context.
5. If permission is granted the call is propagated to the appropriate *functional module*.
6. The *functional module* performs its tasks in the kernel and returns the result.
7. The result is passed on to the *Kernel Access* module.
8. The *Kernel Access* module triggers a security check of the result. For example, a call could have been the request for a listing of *Dinopolis Objects*. Several of the returned objects could be invisible according to their access rights. Therefore they have to be sorted out in this step.
9. The part of the result that passed the security check is propagated back to the requesting *Proxy*. If the result contains *Dinopolis Objects* the appropriate *Object Proxies* are returned (see section 5.2). The requesting *Proxy* returns the result to the *API* which in turn returns it to the requesting application.

One more situation can occur, since Dinopolis is not a plain passive system: kernel modules can trigger events and those events have to be propagated to interested applications. Events in Dinopolis can be of arbitrary type, but each event belongs to one of the following *event classes* that define its base semantics:

Hook execution: This event class represents hooked calls triggered by parts of the kernel and executed in the user space.

Observer notification: This event class represents notifications triggered by observable parts of the kernel that are propagated to interested observers in the user space.

System event: This event class represents special system events that applications have to be aware of (e.g. system shutdown).

The difference between the three classes of events is that hook execution and observer notification are directed events that are propagated only to especially registered *Proxies* or *APIs*. System events do not need special registration. Rather they are delivered to all *Proxies* and *APIs* that could need this information. Mostly, system events are broadcast in user space. The algorithm defining the propagation path of events is the following:

1. The event is triggered inside the kernel space and propagated to the *Kernel Access* module either through *functional modules* (in case of hook execution or observer notification) or directly (in case of system events). Please note that also *Dinopolis Object* related events exist. These will be discussed in section 5.2.
2. The *Kernel Access* module triggers a security check of the event and the given parameters. If necessary, parts of the parameters are sorted out or the call is prohibited completely. *Dinopolis Objects* given as parameters are not directly passed on but encapsulated by *Object Proxies* (see section 5.2).
3. If the event passes the security check it is propagated to the desired destination(s).
4. In case of events that result in a return value (e.g. most hook execution events), the return value is propagated back down to the trigger of the event. Also the return values are security checked to prevent trojan horses which could otherwise be smuggled in this way. If the return value contains *Object Proxies* they are resolved and the objects behind them are propagated.

The event model specified here is not only designed for communication between the kernel and the user space. It can also be used for communication between different application instances. For example a chat application can be implemented this way. All events between the different instances of the chat application are routed through the kernel, no matter if the instances reside on the same machine or if a chat across the network is desired. The network aspects of the chat application

are transparently handled by Dinopolis so that the applications do not have to care about them in detail. All events that are sent between the instances of the chat application are security checked.

5.2 Mechanisms Defined for Working With Objects

As is the case with access to kernel functionality, also access to the data and functionality of *Dinopolis Objects* has to be security checked. To achieve this, *Dinopolis Objects* are never passed on to the user space, but they are encapsulated by *Object Proxies*. In principle the request mechanisms are the same ones as those already known from *functional modules*:

1. Applications send their requests to the *Object Proxies*.
2. *Object Proxies* propagate the requests to the *Kernel Access* module.
3. The *Kernel Access* module requests a security check of the request.
4. If permission is granted the *Kernel Access* module passes the call to the “real” object.
5. The object carries out the request and returns the appropriate result to the *Kernel Access* module.
6. The result is security checked and partially sorted out if necessary.
7. If the result contains *Dinopolis Objects* or parts of them (e.g. the content), the *Kernel Access* module requests the appropriate *Object Proxies* and returns them.

To get a clear idea of all mechanisms that are related to the concept of *Object Proxies* the structure of an *Object Proxy* is shown in figure 5.2. The dashed lines in the sketch represent security checked calls, the solid lines represent direct method calls.

Protecting a *Dinopolis Object* by providing an *Object Proxy* also means that no internals of the object may be directly passed on to applications. Therefore also the internals of an object are protected by using *Proxies* as can be seen in figure 5.2: there exist internal *Proxies* for content, meta-data and attached interrelations.

The box labeled *Access Methods* represents the methods that a *Dinopolis Object* provides through its base interface. These methods are drawn explicitly for the *Object Proxy* to show the special call logic:

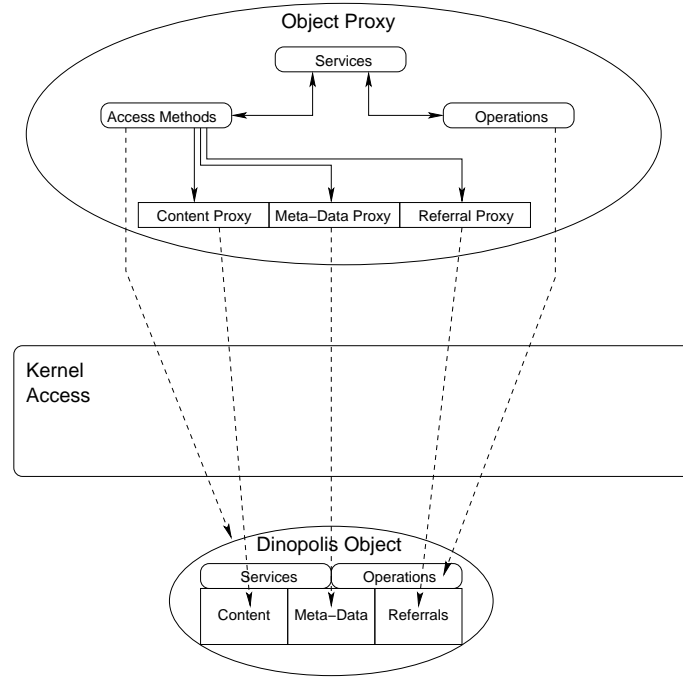


Figure 5.2: Object Proxies in Dinopolis

- Calls that refer to content, meta-data and interrelations are directly executed on the appropriate internal proxies. For example a call to change parts of the meta-data is executed on the meta-data proxy which in turn triggers a security checked call on the “real” meta-data of the *Dinopolis Object* that resides in the kernel space.
- Calls that do not refer to parts of the *Dinopolis Object* for which proxies exist (e.g. *give me your GUH*) are performed as security checked calls on the original object.

Operations and services have a special role in the model as can be seen in figure 5.2: for both there exist no internal proxies. The reason for this lies in their special behaviour and the resulting design:

- As has been discussed in chapter 3 operations represent an object’s explicit dynamic method invocation mechanism (during runtime, as opposed to static method invocation mechanisms managed at compile time). This mechanism is designed to work with *method objects*. They are comparable, but not exactly the same as what the *Command* design pattern (see [Gamma et al. 1998]) describes. Method

objects are instantiated, “sent” to the destination object and executed on it. The logic of *sending* method objects to their destination makes it possible anyway to perform security checks on them, because the *Kernel Access* module has a chance for an interception of the call. Therefore *Proxies* for operations are not necessary.

- Services represent high-level functionality of *Dinopolis Objects* that is GUI-related. Due to their nature all services are some sort of objects. The way how services (i.e. service-objects) are instantiated and initialized in *Dinopolis* is the following:
 1. On request an instance of a service is created through a *Factory*. This instance is not yet related to any instance of a *Dinopolis Object* at all.
 2. The newly created service instance is initialized with the *Object Proxy* for which it was requested. After this initialization step the service is fully functional.

As can easily be seen a service never has direct contact with the original *Dinopolis Object* that resides in the kernel space. It is just related to the appropriate *Object Proxy* and therefore it can work with all access methods and operations that are provided by it. *Object Proxies* implement exactly the same interface as the *Dinopolis Objects* that they encapsulate. Therefore no special care about (programming level) classes has to be taken when implementing services. They can be implemented as if they work directly with the original *Dinopolis Object* that they have been written for. The only limitation is that they must not access variables of the object. However, rules for clean OO design forbid direct access to variables anyway and dictate the use of access methods. Therefore this is not a restriction at all. From this definition of services it can easily be seen that no security relevant actions can be performed by the service that would not be encapsulated by the *Proxy's* protection mechanisms anyway. For this reason the definition of special *Proxies* for services is obsolete.

Now that the whole mechanism is known, it is time for a short consideration about instances of *Dinopolis Objects* and *Object Proxies*.

Dinopolis Objects are the system-internal representatives of whatever data or objects exist in embedded systems. Due to the definition of virtual systems (see chapter 9) one *Dinopolis Object* can be composed of arbitrarily many different chunks of data. For each *Dinopolis Object* a globally unique and robust handle (GUH) exists and objects are instantiated due to a load request with such a GUH. For several reasons (reference equality, synchronization, etc.) it is required that only

one instance of one and the same *Dinopolis Object* exists during runtime. Several load requests with the same GUH are guaranteed to result in the same instance.

The same is true for *Object Proxies*: it is guaranteed that only one *Proxy* exists for one and the same *Dinopolis Object*. All discussions about what would happen if single instances were not required, but that proper synchronization and equality-checking mechanisms have to exist ended with the same conclusion: not requiring single instances is dangerous, because the slightest problems in different object implementations would cause severe danger in respect to the system's integrity.

One more critical point becomes apparent when considering all *Proxy* protection mechanisms described here: performance! A naive implementation of the *Proxies* would for sure cause severe runtime and memory management troubles. For this reason all *Proxies* have to be implemented to support lazy object creation.

6. The Object Management Module

The *Object Management* module is involved in all operations that have to deal with the life-cycles of *Dinopolis Objects*. This includes creating and deleting as well as loading, storing and rearranging objects. The *Object Management* module is always *directly* or via callbacks *indirectly* responsible for the *Dinopolis Objects*' internal structure. The term *internal structure* refers to content, meta-data, attached inter-relations, methods and services. This also includes all compound and subclassing aspects that have already been discussed briefly in chapter 4. In order to get a clear idea of the responsibilities let us consider the operations that involve the *Object Management* module one by one.

6.1 Creating a New Object

In principle creating *Dinopolis Objects* follows the same pattern as creating objects in OO programming languages: a request to create an object of a given type (=class) is sent to the *Object Management* module and if the request can be carried out successfully, the newly created object in its initial state is returned. However, one more property of an object has to be known to the *Object Management* module to enable it to create an object: the desired location of its persistent data in form of a handle that represents a virtual system (local or remote). This does not mean that applications are forced to provide a location. There exists a configurable default mechanism too that allows to choose a location. Nevertheless, sometimes the definition of a certain location is desired and in this case applications have the chance to choose one.

In order not to get lost in details let us for the moment ignore the fact that the location plays an important role in respect to the internals of a *Dinopolis Object*. The more important part here is the logical appearance of a *Dinopolis Object* as it can be seen by application developers.

Due to the system-inherent component-based approach the logical type of a *Dinopolis Object* that applications get to see is in fact defined through two mechanisms that operate on different levels:

1. A class definition on the programming language level. Let us call this one the object's *static class*.
2. A type definition during runtime through dynamic subclassing and compound mechanisms in Dinopolis. Let us call this one the object's *dynamic type*.

6.1.1 Creating a Simple Object

The class definition mechanisms in OO programming languages need no further explanation, *dynamic types* do. Therefore let us first consider an example for a simple *Dinopolis Object* (as opposed to a *compound* object) that (at least theoretically) does not involve the dynamic aspects. On request to create a simple *Dinopolis Object* the tasks that the *Object Management* module has to perform are the following:

1. A new object of the appropriate *static class* is requested through a call to a *Factory* (see [Gamma et al. 1998]). The newly created object is initialized to its empty default state.
2. If the application requested default creation (i.e. without parameters) continue with step 4, otherwise continue with step 3.
3. Besides special functionality that is defined by the object's particular *static class* it is guaranteed that every object implements the base *Dinopolis Object* interface. This standardized interface allows the *Object Management* module to gain access to the objects' internal structure via well-defined methods. If creation with certain parameters for content, meta-data, etc. was requested, the object's internal state is set accordingly through calls to these methods.

Please note that changing the internal state of an object at a later stage by an application is security checked and executing the necessary methods may not be permitted then. The same is the case with creation calls. It is up to the *Security* module, which kinds of calls are allowed and which parameters may be set. However, once a call for creation reaches the *Object Management* module it is already checked and therefore the requested actions are performed without further interaction with the *Security* module.

4. After object initialization is completed the *Address Management* module is asked for a globally unique handle (short: GUH) for the newly created object (see chapter 7). Please note that this GUH does not allow immediate access to the newly created object for anybody else than the requesting application. An application explicitly has to declare an object publicly accessible to allow access (according to the security rules for this object). The reason is that quite often applications have to perform several tasks to finish object creation according to their needs. This behaviour was chosen to guarantee quasi-atomic object creation without the danger that other applications could access an object that is not yet fully functional in the creating application's sense.
5. Eventual system immanent relations are attached to the newly created object (e.g. parent-children relations in subdirectories of a filesystem that could be the storage device of an object).
6. The newly created *Dinopolis Object* is returned to the requestor.

6.1.2 Creating a Compound Object

The way to create dynamically typed compound objects (or in fact compound *components*) is one of the very special features of Dinopolis. With *dynamic typing* the external behaviour of an object can be completely decoupled from its internal structure without breaking its clean base interface. To clarify the motivation for introducing *dynamic types* let us consider the following scenarios:

Split storage of persistent object data: One of the main concepts of Dinopolis is the embedding of external systems. As will be discussed in detail in chapter 9 the persistent data of one object need not reside on one single embedded system, but can be split up across multiple systems. If objects are moved from one virtual system with one specific distribution of embedded systems to another, the way to store them can change. As an example an object could have its origin in a virtual system that just embeds one filesystem. Then it is moved to a different virtual system that embeds a filesystem for storing the content and a relational database for storing the meta-data to make it searchable. To the application the moved object still looks the same. Internally it is restructured, according to the new virtual system setup.

Dynamic types are the cleanest way to model these internals because the programming language defined class of an object must not be changed at all (otherwise the application would have to be rewritten!). Please note that this aspect of dynamic storage types was omitted in the description of creating a simple object in section 6.1.1. However, this aspect is also valid for simple objects and is therefore taken into account in the final algorithm.

Separation of data and behaviour: An object has two main properties, the data that it encapsulates (content and meta-data) as well as its logical behaviour (operations and services). Interrelations can be members of both properties, depending on their use. Parts of the logical behaviour are object-inherent and therefore have to be modelled in the *static class*. Other parts are object-interpretation and context dependent. As an example a simple XML document that is encapsulated in a *Dinopolis object* is statically a general document. Dynamically it can represent a medical document with the appropriate special functionality. This aspect would then be modelled using *dynamic types*. In this case the *dynamic type* adds some special operations and services (maybe also interrelations) that are inherent to a medical document rather than to a general XML document.

As can easily be seen the use of *dynamic types* helps to clearly separate between general, class inherent refinement steps and interpretation dependent refinement steps that are not part of the nature of general datatypes.

Besides the fact that one object can be subject to different high-level interpretations, also the opposite can happen: several different general *static classes* can represent the same logical entity. For example a general XML document and a general HTML document can both be medical documents with the same behaviour. Medical applications (and users) do not have to care about the internal representation.

One more aspect comes into play, considering different document formats: for example HTML defines **META** tags and the meta-information that is provided inside an HTML document also has to be available through the objects' standard mechanism that provides access to meta-data. By using *dynamic types* the methods to access meta-data can be overridden to also include extracted meta-information from HTML documents. Therefore, depending on the format

of the content of an object the methodology can change dynamically, whereas the general type remains untouched.

The same mechanism can also be applied the other way round to enhance features of existing document formats or even to combine several chunks of content to form one virtual content format with additional features. For example it would be possible to combine a series of slides with an audio stream and a synchronization mechanism to construct an automatic slide-show. In this case treatment of the different chunks of data and synchronization would be part of the *dynamic type* of an object.

Implementation of context specific behaviour: Separation of the low-level representation and the high-level interpretation can also be a rather dynamic affair. Depending on the context and depending on a special state an object can change its high-level *dynamic type* during its lifetime.

Let us again consider the dependency of objects on the specific virtual systems in which they are stored. If objects are moved from one virtual system to another not only the internal representation changes. It can also happen that their functionality is affected. Located in one virtual system that only embeds a filesystem, an object does for example not support special search facilities at all. If this object is moved to a different virtual system that embeds a database and a fulltext search engine, special search facilities become available.

A more interesting feature considering context specific behaviour is the ability to change the *dynamic type* during runtime or, even more importantly, to add a special *dynamic subtype* that reflects additional behaviour. As an example let us consider the case of objects that become version controlled at a certain point in time. At the beginning there is just an object of a certain *static class* and *dynamic type*. Later it is decided that this object needs to be version controlled. Therefore dynamically the type “version controlled” is added which takes over responsibility for managing the different versions and for providing the appropriate operations and special services.

Another area of application can be found in the medical context: a *medical document* is defined to be a collection of objects that can be electronically signed. Therefore treatment of electronic signatures as well as special treatment of signed documents (i.e. disallow editing after it was signed) is part of the nature of a

medical document. However, it is not known beforehand, which objects in the system are combined to form a *medical document*. With the ability to add the *dynamic type* at any point in time this behaviour can be modelled.

As can easily be seen from the above scenarios dynamic types add enormously to the data modelling and abstraction possibilities. There exist many cases where implementation of special behaviour statically on a programming language level would make applications much more complicated than necessary. Besides, there exist other cases that could not be cleanly modelled at all on a programming language level. Just consider reconfiguration of virtual systems with the resulting changes in the internal storage structure of objects. From the applications' point of view the objects still remain the same, from the system internal point of view they are not. Statically changing the objects' classes on a programming language level is close to impossible, because it can happen that the same classes are used for different virtual systems and only one of them is reconfigured. This would mean that part of the object space changes classes whereas the other part does not. This finally would result in having to rewrite applications that deal with different virtual systems to reflect this distinction.

One more important aspect of *dynamic types* has to be mentioned: changes of the *dynamic type* are observable by parts of the system that are interested in this information. Therefore reactions are possible. Just imagine a browser application that uses different icons for different logical object types. A change of the *dynamic type* of an object can also mean a change of the logical object type. In this case a different icon has to be shown. Having the possibility to observe type changes this task is straightforward.

After this excursion to motivate the use of *dynamic types* let us consider how the algorithm for object creation changes to reflect this fact and how compound objects are therefore created. First of all it has to be mentioned that the difference between a simple and a compound object is only reflected in its *dynamic type* rather than being reflected in its *static class*. The reason for this definition is that the *static class* defines the general nature (e.g. "this is a document") of an object whereas the *dynamic type* defines its runtime treatment. Applications do not care if a document is internally represented as a simple or compound object, as long as it is a document with the appropriate interface. This distinction through the *dynamic type* therefore

helps to rewrite the algorithm so that it is the same for simple and for compound objects. In fact the only change to the algorithm presented in section 6.1.1 is that between step 1 and step 2 an additional step is inserted:

- 1a. According to the requested type and location the appropriate *dynamic type* description of the object is loaded, the *dynamic type* is instantiated and then attached to the newly created object. Details on the mechanism of loading and attaching a *dynamic type* to an object are discussed in chapter 8.

From the above considerations it can easily be seen that technically there is no difference between simple and compound *Dinopolis Objects*. All objects just have a *static class* and a *dynamic type*, no matter whether they are compound objects or not. For this reason there is no necessity for an explicit distinction between the two in the Dinopolis architecture.

6.2 Storing an Object's Persistent Data

As has already been mentioned, the current persistence location is known for every *Dinopolis Object* instance. The location is defined to be a specific virtual system (it is possible to move objects around, but obviously there is always a current location). It is possible that an object is split up internally into several parts that are stored in different embedded systems, but this behaviour is out of scope of the *Object Management* module and will be discussed in chapter 9. From the point of view of the *Object Management* only the object and component characteristics of a *Dinopolis Object* are of interest. For this reason storing an object's persistent data is straightforward:

- The *Object Management* module requests resolving of the object's GUH to obtain local address information (see chapter 7). If the resolve request results in an address that is not managed by the local Dinopolis instance but by a remote system somewhere on the network the request is passed on to the *Network Management* module and the local *Object Management* module's task is finished. Via the network the remote system is contacted and the *Object Management* module in the remote system stores the object as described here.

- The *Object Management* module passes the object together with the local address information on to the *Virtual System Management*.
- The *Virtual System Management* module finds out from the local address information, in which virtual system the object resides and chooses the appropriate instance for storing it.
- According to the object's *dynamic storage type* content and meta-data is stored in the appropriate embedded systems.
- As will be discussed in detail in chapter 8, interrelations are only attached to an object, rather than being part of an object. Interrelations are objects of their own that technically do not belong to the persistent state of a *Dinopolis Object*. Therefore no extra information about interrelations is stored in this step. If interrelations are added, removed or changed, this is a separate operation.

The request to store a *Dinopolis Object* need not necessarily be triggered by an application. Internally all objects are subject for garbage collection if they are no longer referenced by any part of the system (kernel or applications). Part of the *dynamic type* mechanism in Dinopolis is the existence of dynamic constructors and destructors (see also chapter 8). On garbage collection of an object the appropriate dynamic destructor is called. Depending on the implementation of the destructor and on special hooks the system's behaviour in respect to object destruction can be influenced. Two different opinions exist, how a componentware system shall behave:

1. If an object's persistent state has changed compared to the last stored version of it, a store request has to be triggered internally before discarding it. If applications explicitly do not want changes to be stored they can request to revert the object to its original state before dropping the reference to it. With this behaviour components can be interconnected arbitrarily to model whatever kind of information, knowledge or also applications. Components are always treated as if they were "alive" and in memory. Therefore from a logical point of view there are no explicit load and store operations. Rather the whole system has a state and this state has to be preserved automatically as if it were residing in non-erasable memory. Considerations about transient and persistent data are only of technical nature and must not influence the logic of a componentware system. For this reason all operations that have to do with objects' persistence

have to be encapsulated by the system and hidden behind the scenes unless somebody really requires to influence the behaviour explicitly (which sometimes makes sense too).

2. Store operations have to be requested explicitly. If an object was not explicitly stored it is discarded without saving the changes. This behaviour is favoured by the representatives of the filesystem type of applications that strongly distinguish between transient and persistent state. As an example current document editors do not store any changes before users tell them to do so.

From my point of view both opinions are legitimate and depending on the kind of application one or the other will be the better choice. With the existence of individual dynamic destructors the behaviour can be influenced on a *dynamic type* level (e.g. all “documents” are discarded whereas “desktop objects” keep their state). Sometimes it is not enough to control the behaviour on a type level. Rather individual treatment of single instances is required (e.g. if this object was changed by application X changes are discarded, if it is changed by application Y they are not). To reflect this behaviour individual destructor hooks exist for *Dinopolis Objects*.

6.3 Loading an Object

Loading a *Dinopolis Object* is always requested with a given GUH. The *Object Management* module has to perform the following steps:

1. The *Object Management* module looks up its internal tables, whether an instance of the desired object already resides in memory. If yes, this instance is returned and the task is finished.
2. If no instance of the desired object resides in memory a request is sent to the *Address Management* module to resolve the given GUH. Two different kinds of results are possible:
 - a) The persistent data of the desired object is in the area of influence of the local Dinopolis instance. In this case continue with step 3.
 - b) The persistent data of the desired object lies in a different Dinopolis instance somewhere on the network. In this case continue with step 7.

3. Since the persistent data of the desired object is in the area of influence of the local Dinopolis instance the result of GUH resolving is passed on to the *Virtual System Management* module. This module finds out in which virtual system the data resides and requests a skeleton for the desired object. This skeleton is an empty *Dinopolis Object* instance with the correct *static class*, appropriate information about its *dynamic type* and internal storage information that can be handled correctly by the given *dynamic storage type* to build the object. This skeleton is returned to the *Object Management* module.
4. The *Object Management* module instantiates the *dynamic type* objects according to the information in the skeleton and applies the necessary type-set operations on the skeleton. This results in an empty instance of a *Dinopolis Object* that is already correctly *dynamically typed*.
5. The fully typed but empty skeleton is initialized with its persistent data by calling the constructor of the *dynamic storage type*. This constructor is able to evaluate the internal storage information of the skeleton that was attached by the virtual system in step 3 and fills the skeleton.
6. Local loading continues with step 10.
7. Since the persistent data of the desired object is in the area of influence of a remote Dinopolis instance somewhere on the net the *Network Management* module is contacted and remote loading is triggered.
8. The *Network Management* module passes on the call to the remote Dinopolis instance, where local loading is performed as described here. The result is passed back across the network in form of a *Remote Proxy* (see chapter 11 for details).
9. The *Remote Proxy* is returned to the *Object Management* module.
10. The finished instance of the desired object (or *Remote Proxy* respectively) is registered in the internal tables of the *Object Management* module for later lookup (see step 1) and returned to the requestor.

As can easily be seen the single instance behaviour of *Dinopolis Objects* as required in chapter 5 is fully encapsulated by the *Object Management* module.

6.4 Deleting an Object

In respect to robustness and consistency of an arbitrarily distributed huge information space object deletion is one of the most dangerous operations. However, there exist enough cases that require the existence of an object deletion operation. Two crucial aspects exist that are related to object deletion:

1. Security, which is already covered by the comprehensive concept of the *Security* module in Dinopolis and will not be described here in detail.
2. Safety, which will be discussed in the following.

The important points that are related to the safety aspect of the system are:

- If an object is deleted which is still referenced by parts of the system or by an application this must not result in inconsistent calls or even system crashes.
- If an object is deleted this must not result in inconsistent interrelations that point to nowhere.

Let us first consider the aspect of deletion requests for objects that are still referenced by parts of the system (e.g. by applications). There is one situation that must not happen, namely inconsistent object references. This would either result in severe exceptions or even in a system crash when accessed. Theoretically three possibilities exist to make deletion a robust operation in respect to remaining references:

1. Forbid object deletion if there are still references to the object.
2. Use a garbage collection mechanism that “officially” deletes the object in a sense that it is no longer requestable from the outside. Internally the object is kept (and hidden) until the last reference to it has been given up.
3. Replace the deleted object by a valid “no longer existing” object. This at least fulfills the minimum robustness requirements to stay in a well-defined state, although the state may not be what a requestor wanted.

All three possibilities have their legitimacy in respect to different requirements, therefore the implementation of the deletion mechanism in Dinopolis is based on the use of *dynamic types* and *Hooks* to be able to choose the desired behaviour. As is the case with object loading (see section 6.3) this mechanism allows the definition of the

behaviour for objects that are to be deleted on a type level as well as on an individual instance level. The following steps are performed by the *Object Management* module when deletion of an object is requested:

1. The *Object Management* module triggers execution of the so-called *deletor* of the *dynamic storage type* (as opposed to the destructor, which only cares about objects in memory).
2. The deletor can request information about current references to the object and decide how to proceed. Different possibilities exist:
 - The deletor can refuse object deletion completely. In this case an exception is generated by the *Object Management* module and the operation is finished.
 - The deletor can delay object deletion until no more references to the object exist. This behaviour is supported by reference count observing and object hiding. Both mechanisms will be explained below.
 - The deletor can delete the object and request that an appropriate “no longer existing” object replaces the original.

For system consistency reasons it is guaranteed in either case that no unresolvable reference to an object is left.

3. Depending on the action of the deletor the *Object Management* module performs the remaining cleanup operations for the memory-resident parts of the deleted object.

Reference count observing for objects strongly relies on the fact that single instances are required in chapter 5. Otherwise synchronization of the counters would be very difficult to achieve. In principle the mechanism is straightforward: *Dinopolis Objects* are *observable* (see [Gamma et al. 1998]) in respect to the number of references and also in respect to specific reference holders. The term *specific reference holders* in this context means that it is possible to find out who (e.g. an application, a kernel module, etc.) holds a reference to an object, either directly or via a proxy. This is important to give deletors the possibility to distinguish between different cases. For example a rule implemented in the deletor could be that replacement by a “no longer existing” object is ok if only applications hold references but a delayed deleting has to be performed if also kernel modules hold references.

The possibility of object hiding is important for delayed deletion. If it is requested that an object has to be hidden, this means that references to it are still active but requests for new references to an object are treated as if the object was already deleted. This behaviour is comparable to the behaviour of unlinking files from a Unix filesystem: the file is officially deleted and no longer visible in listings. Nonetheless applications that already have open filedescriptors still have access to the file. The real process of deletion is delayed until all filedescriptors are closed. Deletors can request object hiding if they want to implement this behaviour, but it is also possible to leave the object visible if desired. In both cases delayed deletion of objects can be implemented through reference count observing.

6.5 Moving an Object Between Virtual Systems

Before discussing what happens when moving an object one very important fact has to be stressed again:

Moving an object is an operation that affects only the physical location of its persistent data. It does not (!!!) affect its robust, globally unique handle at all!

Everything that is described in this section is only related to the objects' persistent location and has nothing to do with accessibility from within applications. For details on the implementation of this behaviour please read chapter 7.

The algorithm for moving objects is the following:

1. The *Object Management* module contacts the *Address Management* module to obtain information about the destination virtual system of the move operation. If this system is controlled by the local Dinopolis instance the algorithm continues with step 4. Otherwise it continues with step 2.
2. The *Network Management* module is informed that moving an object to a remote location was requested. This request, together with the appropriate *GUH*, is sent to the remote system.
3. The remote system requests a *Remote Proxy* of the source object. With this *Remote Proxy* the rest of the move operation is carried out exactly like a local move because the *Remote Proxy* represents a fully transparent view to the object.

4. The *Object Management* module contacts the *Virtual System Management* module, presents the source object and asks for a skeleton that is able to hold the given object.
5. The *Virtual System Management* module finds out the appropriate virtual system and propagates the request for a skeleton to it. The skeleton that is created has the correct *static class* and *dynamic type*. It is returned to the *Object Management* module.
6. The skeleton is asked to create an internal representation of the given object as it would be represented at its new location.
7. The new internal representation is made persistent and the *Address Management* module is informed accordingly to perform the necessary steps to reflect the new location of the object (see chapter 7 for details).
8. After all necessary internal tasks of the *Address Management* module have been successfully performed the *Object Management* module is informed. It can then finally trigger removing of the old data from the source virtual system (locally or remotely).

Depending on the destination system parts of the dynamic type of an object will change due to a move operation. Especially the *dynamic storage type* will be affected. However, changes in the *dynamic type* are observable and parts of the system that need this information are able to react accordingly.

7. The Address Management Module

As has already been discussed, the existence of globally unique and robust object handles is the key to a successful design of a distributed component system. According to the requirements the features of *GUHs* (=globally unique handles) in Dinopolis are the following:

- One *GUH* always refers to one and the same object, no matter whether and how often an object changes its physical location. This also includes the situation where the original location no longer exists and one or more different systems have taken over its responsibility.
- A *GUH* that is used for one object must never be used for any other object, even if the original object is deleted. Otherwise it could happen that an object is unintentionally replaced by a different one.
- *GUHs* can be stored anywhere, e.g. even on the users' harddisks outside the Dinopolis system when bookmarking objects. Therefore update-operations on *GUHs* by means of push-technologies are unthinkable.
- Closely and loosely synchronized replication (see also [Schmaranz 2002a]) of objects have to be supported by the *GUH*-resolving mechanism.

In principle there are two main families of algorithms that are robust against object-movement:

1. Forwarding algorithms that rely on traces which are left whenever objects are moved from one location to another.
2. Algorithms based on lookup-service strategies that rely on updates of the lookup-service whenever objects are moved.

Considering the two approaches it becomes clear that in fact forwarding algorithms do not provide real object handles. Rather they rely on physical addressing with an add-on that avoids broken addresses (at least for some time). In principle

the nature of these algorithms would already be a contradiction to the requirement of having *handles* rather than physical addresses. Besides this fact also other drawbacks of forwarding algorithms become apparent. One of the big problems is that they do not scale at all for frequent object movements. Further, servers that go offline would break the consistency of any traces that objects left on them. Finally, the requirement for transparent component replication (see also section 2.1) cannot be fulfilled with forwarding anyhow, because the address is strongly coupled to the physical location rather than being an identifier that can be mapped accordingly. For these reasons the algorithm of choice for implementing the concept of *GUHs* has to be a lookup-service based algorithm.

Very early in the design phase of Dinopolis it became apparent that today's lookup-service implementations do not fulfill all requirements for *GUHs* either:

- Naive implementations based on one central lookup-server or on some statically distributed servers do not scale considering a huge number of existing *GUHs* and a large number of resolve-requests.
- Defining *GUHs* with an internal hierarchical structure like hostnames in DNS (see [Mockapetris and Dunlap 1988]) is not realistic too, as will be pointed out in section 7.2.3.

As a result of these considerations a new algorithm named *DOLSA* (=Distributed Lookup Service Algorithm, see also [Schmaranz 2002b]) was developed for Dinopolis. For reasons of robustness, stability, scalability and availability as well as for some administrative reasons the *DOLSA* algorithm is a combination of the advantages of both algorithm families mentioned above. *DOLSA* implements a self-organizing (but influencable!), arbitrarily distributed lookup-service structure with robust caching and robust short-term forwarding for scalability reasons.

Before presenting the very details of *DOLSA* an overview of the addressing schema of *Dinopolis Objects* is given in the following section.

7.1 *GUHs* and Addresses in Dinopolis

Due to the concept of Dinopolis to embed external systems, much care had to be taken to not make the concept of *GUHs* incompatible with existing external systems.

Even more than just theoretical compatibility is requested, considering that it is not only desired to embed just an Oracle server or a filesystem. The requirement is that existing systems including the data stored in them have to be embedded. This means that it is not possible to dictate a certain schema, because many different addressing schemas already exist in practice. The only way to make embedding of existing systems work was to find a concept that allows arbitrary mappings from existing, non-uniform addressing structures used in external systems to uniform *GUHs* used inside Dinopolis. The overall structure of the lookup architecture used in Dinopolis is sketched in figure 7.1.

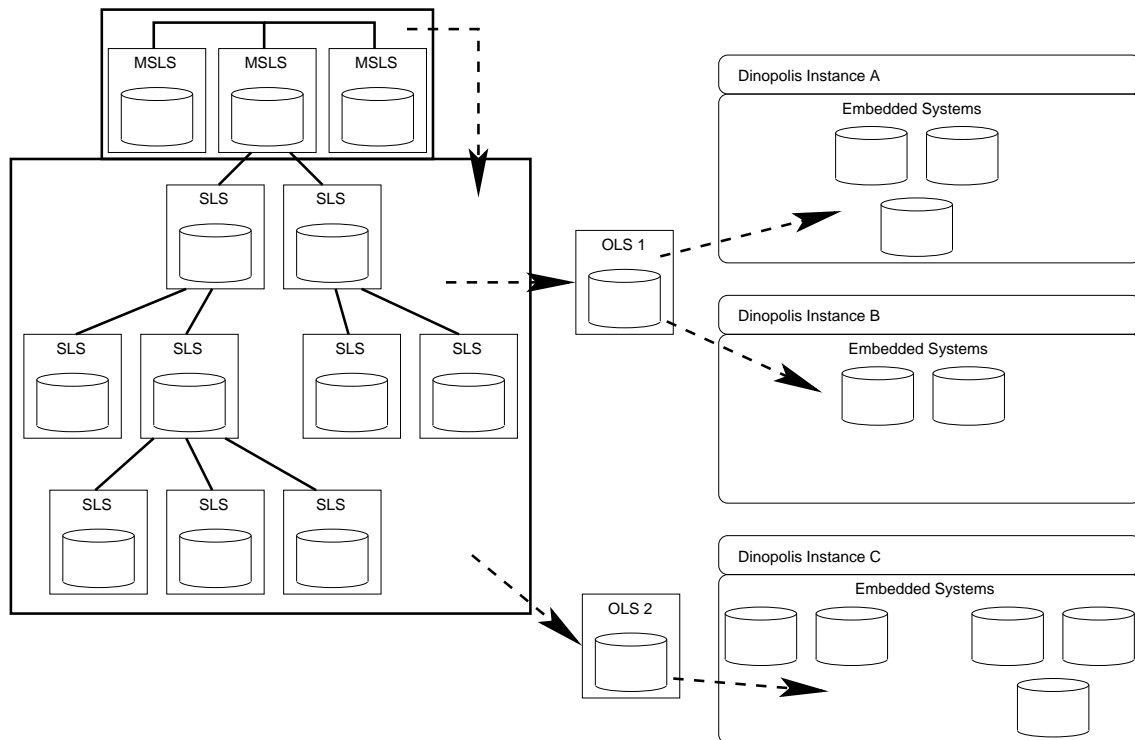


Figure 7.1: The structure of the Dinopolis lookup architecture

The whole lookup architecture involves three distinct layers:

Object lookup servers: There can be an arbitrary number of independent object lookup servers (short: *OLS*). They are distributed across the net and each of them is responsible for the mapping of *GUHs* to local addresses for one or more Dinopolis instances.

Server lookup servers: There can be an arbitrary number of server lookup servers (short: *SLS*) that are distributed across the net. The task of the *SLSs* is to keep track of all existing *OLSs*. A detailed discussion about the necessity of this extra lookup-service layer is delayed until section 7.2.3.

Master server lookup servers: There is a small number of master server lookup servers (short: *MSLS*) that are responsible to keep track of all existing *SLSs*. An exact description of the tasks of the *MSLSs* will be presented in section 7.5.1.

The responsibilities described above are represented by the dashed arrows in figure 7.1. The lines between the *MSLSs* and the *SLSs* represent the communication paths of the single services (i.e. the propagation paths for requests). As can be seen the *MSLSs* all communicate with each other. Additionally they form the root of the tree of hierarchically organized *SLSs*. As will be discussed later, this hierarchy is neither fixed nor reflected in the *GUH*, as this would make a clean algorithm for a scalable distributed lookup-service impossible.

As can easily be seen in figure 7.1 the lookup-service is not residing inside Dinopolis instances. Rather it is a distributed service of its own for robustness and availability reasons. The *Address Management* module represents the Dinopolis system-internal stub to access this lookup-service. Nevertheless, the exact definition of *GUHs* is based on the *DOLSA* algorithm and therefore *DOLSA* is a system-inherent part of the Dinopolis framework. Further, the *Address Management* module relies on the existence of such an algorithm. For this reason it is described here in this context.

To make things clearer, why exactly this architecture was chosen and to be able to identify the responsibilities of the modules in Dinopolis in respect to this architecture let us start at the beginning with the design of *DOLSA*.

7.2 The Design of DOLSA in Detail

Before coming to the dynamic aspects of the distributed lookup-service it is necessary to define exactly what a *GUH* could look like. For this reason, let us as a first step just consider the static case, where many distributed lookup servers exist, but neither objects nor servers can move.

7.2.1 The Static Case

The simplest definition of a *GUH* for this case would be to have consecutive numbers, one for each object. The problem with this definition is the aspect of number-distribution across arbitrarily many *OLSs*. It is necessary to assign certain parts of the whole number-space to certain *OLSs*, otherwise it would be necessary to broadcast resolve-requests to all *OLSs*. For large numbers of objects and/or requests broadcasting does not scale at all.

This problem can easily be overcome by splitting the number that stands behind a *GUH* into two parts. One part is the globally unique ID of the responsible *OLS*, the second part is the locally unique ID of the appropriate object in the *OLS*. The combination of the two parts always results in a globally unique ID for an object. It can easily be seen that there is no necessity to have numerical IDs, the only real requirement is uniqueness of each part. Therefore the structure of a *GUH* that works for the static case looks as follows:

Object Lookup Server ID (short *OLS-ID*): This can be either the IP-address or the hostname of the *OLS*. For obvious reasons it is a better choice to take the hostname.

Local Object ID (short *LOID*): This is the local ID of an object as managed by each *OLS*.

Resolving a *GUH* is therefore performed in two steps:

1. The *OLS-ID* of the *GUH*, respectively the hostname, is resolved.
2. A request to resolve the *LOID* is sent to the appropriate *OLS*.

The result of such a resolve-operation is the physical address of the desired object.

The definition of a *GUH* as a combination of hostname and *LOID* is in principle a re-invention of URLs. Its only advantage is that the *OLS* allows restructuring of the storage space without breaking the addressing mechanism, because addresses of objects are reflected in the *OLS* entries. Exactly this mechanism was (and is still) the strength of Hyperwave (see [Andrews et al 1995]) compared to standard Web-Servers.

Nonetheless, several severe drawbacks of this simple definition can be found when investigating the different dynamic cases of object management in distributed sys-

tems. In the following aspects of system-immanent dynamics and resulting shortcomings are outlined. Modifications of the *GUH*-structure and the resolving algorithm to get those problems under control are presented accordingly.

7.2.2 Moving Objects Across Servers

The first dynamic case to be considered is the occurrence of object movement (see also section 6.5). Moving objects around within one single server is trivial and will therefore not be considered here. In this situation a simple update of an *OLS*-entry is enough.

More interesting is what happens when objects are moved across server boundaries. In fact, the term *across server boundaries* is not really exact, because it was already stated that one *OLS* can also be responsible for more than one server. Therefore the interesting aspect for the lookup algorithm is object movement across *OLS* responsibility boundaries. This is exactly what will be investigated in the following.

In section 7.2.1 a *GUH* was defined as consisting of an *OLS-ID* and an *LOID*. When an object is moved across the responsibility boundaries of an *OLS*, the *OLS-ID* part of the *GUH* changes. After having moved the object to the area of responsibility of a different *OLS*, the “old” *OLS* can no longer resolve the *GUH*.

It was already stated at the beginning of this chapter that a pure forwarding algorithm does not scale for multiple object movements, because of the arbitrary length of forwarding chains. Besides the length problem there exist other problems too, for example circularity of forwarding chains, just to mention one of them.

Nonetheless a slight redefinition of the internal structure of a *GUH* allows the implementation of a chain free forwarding algorithm for the “slow” case of object movement. In this context “slow” means that enough time elapses between move operations of one single object to finalize one complete run of the algorithm proposed below. The “fast” case, i.e. the case where objects move faster from one *OLS*’s area of responsibility to another than the algorithm can manage to finalize one complete run, will be considered later in section 7.6.2. For the moment it is enough to know that this situation can be kept under control without any influence on the scalability or robustness of the algorithm and without considerable influence on its performance.

The structure of a *GUH* is redefined, so that it does not just contain one pair of $[OLS - ID, LOID]$. Rather a *GUH* contains two such pairs:

Birthplace Handle (short *BPH*): The *BPH* is a pair of $[OLS - ID, LOID]$ that represents the handle that an object has got when it was created. This handle is the one that accompanies an object during its whole lifetime and never (!) changes, no matter where an object resides at the moment and no matter how many move, edit or other operations were already performed on it. For each object the *BPH* is carved in stone and lasts forever. The reasons for this extremely strong requirement will be discussed in section 7.2.3.

Current Handle (short *CH*): The *CH* is also a pair of $[OLS - ID, LOID]$ that represents an object's current residence. This handle is updated in the *GUH* when an object movement to a different *OLS*'s area of responsibility is detected.

Further it is defined that the birthplace *OLS* always keeps track of the current location of objects that were "born" in its area of responsibility. Whenever an object is moved, the birthplace *OLS* is informed and registers the *CH* of the object at its new location. At the moment let us assume that handle consistency during ongoing move operations is guaranteed. Further investigations how this is done can be found later in section 7.6.1.

From the point of view of a requestor that wants to resolve a *GUH*, the algorithm is the following:

Case 1: Only *BPH* exists, *CH* is empty: In this case the birthplace *OLS* is contacted. Depending on its answer the requestor reacts as follows:

Address of object: In this case the object is still residing on its birthplace \Rightarrow resolving finished.

New *CH*: The object has moved to a new location \Rightarrow update the *CH* of the *GUH* and contact the new *OLS*.

Note: the case that an object is "fast" and again changed its location before the new *OLS* can be contacted is dealt with later in section 7.6.2.

Case 2: *BPH* and *CH* contain values: In this case the current *OLS* is contacted. Depending on its answer the requestor reacts as follows:

Address of object: In this case the object is still residing on its current location \Rightarrow resolving finished.

Unknown handle: The object has moved to a new location and is therefore no longer known to this *OLS*. \Rightarrow contact the birthplace *OLS* and proceed as in case 1 above.

It can easily be seen that in principle the *BPH* would be enough to make the algorithm work because the birthplace always keeps track of the new location of the object. However, the *CH* is absolutely necessary for scalability reasons. Just imagine the case where one *OLS* becomes overloaded and therefore is split up into two *OLSs* to reduce network traffic. Without the *CH* being part of a *GUH*, splitting would not be any help at all because the original *OLS* is contacted anyway.

7.2.3 Taking Birthplace Servers Offline

With some slight modifications for the sake of consistency during move operations and to be able to deal with fast moving objects the algorithm described in section 7.2.2 is robust against object movement. However, the algorithm implies that the birthplace *OLS* has to exist “forever”. This is an unrealistic assumption!

The algorithm has to be modified to be able to react in the case that a birthplace *OLS* is taken offline. In this case one or several other *OLSs* have to be able to take over the original birthplace *OLS*’s responsibility of object tracing.

The simplest case is that the content of an *OLS* is moved to a different, empty (in the sense of *OLS*) machine. This case is trivial, if the possibility was planned before: A simple update of the DNS entry for the server (e.g. `lookup.my.domain`) is enough.

However, although no real technical problems exist when planning beforehand, other problems may arise. It was already mentioned in the medical scenario that whole institutions can go offline completely. If institutions that go offline operate whole domains of their own, including the required DNS-servers, the whole domains have to be moved. Therefore the institutions that take over the data have to operate additional domains, which can administratively or politically be impossible.

These problems can be overcome by introducing one more indirection in form of a so-called *Server Lookup Service* (short *SLSvc*). Please note that *SLSvc* stands for the whole distributed service that consists of arbitrarily many *SLSs*. The reason to consider this service as a black-box at the moment, rather than considering single

SLSs is found in its specific distribution aspects. These will be described in detail in section 7.5.

- The *OLS-ID* part of handles is no longer represented through the hostname of the lookup-service. Rather it is an arbitrary, unique ID managed by an *SLSvc* which is mapped to the actual hostname. The ID itself does not contain any administrative structures like e.g. the resolve order, as would be the case with DNS. Whenever a lookup server is moved, an update of the ID-to-hostname mapping is performed in the *SLSvc*. With this indirection *GUHs* become completely independent from hostnames.
- For availability and scalability reasons the *SLSvc* itself is distributed across the network in a semi self-organizing manner (see also section 7.5). To avoid getting side-tracked here let us just assume that a scalable and reliable, distributed, self-organizing *SLSvc* is realistic.

Besides the more or less administrative reasons to replace hostnames by IDs of a different kind, a very important technical reason exists which necessarily requires that an *SLSvc* has to exist, rather than relying on hostnames: without a special *SLSvc* it would algorithmically be quite impossible to split up the content of one source *OLS* which goes offline across several destinations. Several further reasons for the necessity for the existence of a special *SLSvc* will be discussed later in section 7.5.

Let us have a closer look at the scenario where one *OLS* is going to be split up across several machines. At the moment it is assumed that the content of one source server is split up across several destinations that did not have any lookup entries before. The aspect of merging entries from different servers is considered later in section 7.6.3. Splitting up the content of one *OLS* across several machines principally results in the situation that the *OLS-ID*-to-hostname mapping is no longer unique. Two possibilities exist to deal with the non-uniqueness of an *OLS-ID*:

1. Non-uniqueness has to be avoided. This can be done in 2 different ways:
 - a) The *LOID* space of the source server is algorithmically split up into several chunks of predefined size. In this case no control exists, *which LOIDs* are moved to which servers. It can only be controlled *how many LOIDs* are moved to the single destinations. Due to the algorithmic nature of the split

operation, the *SLSvc* is able to calculate which destination manages which part of the *LOID* space of the source. Therefore the *SLSvc* can always return a unique hostname for a given pair of $[OLS - ID, LOID]$.

However, this solution is not realistic because usually one wants to have full control over the new location of data. Therefore the algorithmic solution can be sorted out from further considerations.

- b) If algorithmic uniqueness is not feasible, the only chance to avoid non-uniqueness is to pass on all information about every single moved object to the *SLSvc*. The *SLSvc* has to store the whole mapping *LOID* by *LOID* for the whole *OLS* to be able to return a unique hostname for a given pair of $[LSID, LOID]$.

It can easily be seen that this is technically impossible because in the worst case all *LOIDs* for all existing objects are finally duplicated in the *SLSvc*. Besides the fact that this would not scale at all, it would also simply break the distinction of responsibilities between *SLSvc* and *OLS*.

2. Non-uniqueness is allowed for *OLS-IDs*. This is not a contradiction to the concept of globally *unique* handles when requiring that only one *OLS* can be responsible for one particular *LOID*. In this case *non-uniqueness* is in fact reduced to *uncertainty*. On request the *SLSvc* can return more than one hostname, but only one of them represents the responsible *OLS*.

As can easily be seen, the *uncertainty* approach is the only practicable solution when requiring the possibility to split up *OLSs*. However, much care has to be taken: a naive implementation lacks scalability, according to the number of destination servers that a source is split up into. It can happen that for each request a considerable number of *OLSs* has to be asked, until a *GUH* can be resolved. This situation becomes even worse for split chains (i.e. one *OLS* split up into several *OLSs*, some of them split up again, etc.).

One possibility of solving the scalability problem would be that the requestor updates the *BPH* of the *GUH* after a change in the birthplace responsibility has been detected. However, it was already defined that the *BPH* must never change! Besides, allowing changes of the *BPH* would just shift the problem rather than solve it. The *SLSvc* does not know how many instances of *GUHs* for one object exist out there on the network. Therefore it is not known at which point in time all *GUHs*

have been resolved and updated. As a consequence a trace of all split operations and split chains has to be stored in the *SLSvc* forever to guarantee correct resolving. This does not scale either.

Even worse is another consequence of allowing changes to *BPHs*: several different *BPHs* may exist at one point in time for one and the same object, depending on the last try to resolve it. This means that it is not possible to find out whether two *GUHs* refer to the same object unless they are both resolved. This is a completely unacceptable situation, because comparison does not scale and caching as well as replication become very complicated or even impossible.

Last, but not least, changes to *BPHs* make electronic signature of interrelations impossible. A change of a *BPH* would immediately invalidate the signature of an interrelation.

The solution to this dilemma is therefore a further redefinition of the internal structure of *GUHs* with an appropriate adaption of the resolving algorithm. A *GUH* does now not only contain the *BPH* and the *CH*, it also contains a third handle, the so-called *Moved Birthplace Handle* (short *MBPH*). This is also the final definition of the *GUH* as it is used in Dinopolis.

In the following the final structure of *GUHs* and the *DOLSA* base-algorithm are summarized as a reference for further investigations and fine-tuning. All considerations from now on refer to the definitions given here.

7.3 Structure of GUHs

- A *GUH* always consists of three handles:
 1. *BPH* (=birthplace handle)
 2. *MBPH* (=moved birthplace handle)
 3. *CH* (=current handle)
- Each of the three handles consists itself of two IDs:
 1. *OLS-ID* (=object lookup-service ID)
 2. *LOID* (=local object ID)

The following definitions apply to the single handles of a *GUH*:

- The *BPH* always contains a non-empty entry referring to the birthplace of an object.
- The *BPH* is defined when an object is created and it must never change during the whole lifetime of an object.
- It is guaranteed that the *BPH* can always be resolved.
- The *BPH* and only the *BPH* is taken when comparing two handles for equality. If two *GUHs* have the same *BPHs*, it is guaranteed that they refer to the same object. If they do not have the same *BPHs*, it is guaranteed that they refer to different objects.
- The *MBPH* only contains a non-empty entry if the birthplace *OLS* of an object was taken offline. Otherwise the *MBPH* is empty.
- It is not guaranteed that the *MBPH* can always be resolved. Resolving is not possible if the moved-birthplace *OLS* that took over the role of the birthplace *OLS* was taken offline. In this case it is guaranteed that the *SLSvc* can resolve the *BPH* and returns at least one *OLS-ID* of possible moved-birthplace *OLSs* according to the uncertainty approach introduced in section 7.2.3.
- The *CH* only contains a non-empty entry if the object was moved across *OLS* responsibility boundaries. Otherwise the *CH* is empty.
- It is not guaranteed that the *CH* can always be resolved. Resolving is not possible if the object has moved on. In this case the *MBPH* is taken for resolving, if it is not empty. If the *MBPH* is empty or not resolvable (see above), the *BPH* has to be taken for resolving.

7.4 The *DOLSA* base-algorithm

There are two different views of *DOLSA* that make up the whole algorithm:

1. The view of the distributed lookup-service as a whole that has to react on different dynamic aspects to guarantee robust *GUH* resolving.
2. The view of the requestor that holds a *GUH* in hands and wants it to be resolved.

7.4.1 The View of the Distributed Lookup-Service

The core functionality of *DOLSA* can be found in the organization of the distributed lookup-service:

Until the distribution aspects of the *SLSvc* are discussed in section 7.5 let us assume that the starting point for the following description is an existing but empty *SLSvc*. It is further assumed that one host of the *SLSvc* is known and that the *SLSvc* is contacted through this known host.

Adding an *OLS* to the Distributed Service. If a new *OLS* is going to be added to the distributed service, the following steps have to be performed:

- The new *OLS* has to contact the *SLSvc* to register its hostname.
- The *SLSvc* calculates a unique *OLS-ID*, registers the appropriate *OLS-ID*-to-hostname mapping and returns the *OLS-ID*.

Changing an *OLS*'s Hostname. If an *OLS* changes its hostname, it contacts the *SLSvc* and sends an update request. The *SLSvc* updates the registered *OLS-ID*-to-hostname mapping. The *OLS-ID* remains untouched.

Moving an Entry to a Different *OLS*. Moving objects around inside the area of responsibility of an *OLS* is trivial. For this case just an *OLS* update-entry request is necessary. Therefore this situation will not be dealt with in detail here. The more interesting situation occurs when an object is moved from the area of responsibility of one *OLS* to another *OLS*. In this case the following actions are necessary:

1. Triggered by the source *OLS* the destination *OLS* calculates an *LOID* that the entry will obtain when moved. This *LOID* is passed on to the source *OLS*.
2. The source *OLS* marks the entry to be moved and sets an appropriate temporary forwarder (see section 7.6.1 and section 7.6.2 for details).
3. As soon as the entry in the destination *OLS* is active, the destination *OLS* informs the birthplace (or moved-birthplace respectively) that the entry was moved. The birthplace *OLS* (or moved-birthplace *OLS* respectively) updates its appropriate birthplace entry to reflect the changes.
4. As soon as the birthplace updated its entry, the destination *OLS* notifies the source *OLS* which in turn can drop the temporary forwarder.

In case that the source *OLS* is the birthplace of the object, a shortcut of the algorithm without network traffic in step 4 can be implemented.

Dealing With an *OLS* Going Offline. If an *OLS* goes offline and its lookup tables therefore have to be moved to one or more existing *OLSs* the following actions have to take place:

1. The source *OLS* that will go offline contacts the *SLSvc* and reports the start of the move operation as well as all destination *OLSs* that will take over its responsibilities.
2. The tables are moved to the appropriate destination *OLSs*.
3. The source *OLS* reports the end of the move operation to the *SLSvc* and can then go offline.

If an *SLSvc* is contacted to resolve an *OLS* that is just in the middle of performing such a move it returns the original *OLS's* hostname together with the destination *OLSs's* hostnames, *OLS-IDs* and a flag that alerts the requestor that a move operation is just taking place.

If an *SLSvc* is contacted to resolve an *OLS* after moving of the whole content has been finished, only the destination *OLS's* hostnames and *OLS-IDs* are returned.

The responsibility of each of the destination *OLSs* depends on the type of the entry that they receive:

Standard entry: If a standard entry (i.e. not a birthplace entry, see below) is moved to the destination, this is treated as already described.

Birthplace entry: If a birthplace entry is moved to the destination (i.e. the corresponding object had its birthplace in the area of responsibility of the source *OLS*) the *BPH* of the entry is stored in the destination *OLS's* birthplace mapping table. Further, a standard move operation, as already described, takes place.

In case an *OLS* is already the moved birthplace for one or more other *OLSs* the same algorithm applies. So-called *moved-birthplace entries* are treated the same as birthplace entries. For moved-birthplace *OLSs* going offline the *SLSvc* has the responsibility to compact the resulting move chains to a one level uncertainty indication.

Splitting up an *OLS*. If for whatever reason (e.g. massive request overload) an *OLS* has to be split up without going offline itself, this is considered the same case

as moving many entries from one *OLS* to another. Therefore no *MBPH* comes into play, because the birthplace server is still online.

This behaviour is the reason why the *GUH* contains the *CH*, otherwise the load on the original *OLS* could not be reduced by splitting it up. In this situation network traffic could not be got under control any more and the algorithm would not scale. By introducing the *CH* the original *OLS* is only contacted once per *GUH* and then the traffic calms down automatically.

7.4.2 The View of the Requestor

From the point of view of the requestor two different instances exist in principle which can be contacted for resolving: *OLS* and *SLSvc*. Because distribution of the *SLSvc* has not been discussed up to now let us assume for the moment that the *SLSvc* is a server with a known hostname. In fact it is not that simple, but to sketch the base algorithm it is all that has to be known. Further details will be considered in section 7.5.

In principle resolving a *GUH* is a two-step process:

1. For a given *GUH* ask the *SLSvc* for the hostname of the *OLS* which knows the *GUH*-to-address mapping. In most cases the answer will contain the hostname of exactly one *OLS*.
2. Contact the obtained *OLSs* (seldomly more than one) and ask for the resolving of the *GUH*.

With all dynamic aspects that have been discussed up to now taken into account the following tasks have to be performed when resolving a *GUH* (caching in the requestor is not considered here):

1. The Requestor contacts the *SLSvc* and passes the *GUH* on to it.
2. The *SLSvc* decides on the content of the *GUH*, which *OLS-ID* is relevant for resolving:

CH is non-empty: In this case the *SLSvc* looks up its internal tables, whether the *OLS-ID* in the *CH* points to an *OLS* that is registered as being online. If yes, the *SLSvc* goes on with step 3. Otherwise the decision process goes on with the next step described below.

MBPH is non-empty: In this case the *SLSvc* looks up its internal tables, whether the *OLS-ID* in the *MBPH* points to an *OLS* that is registered as being online. If yes, the *SLSvc* goes on with step 4. Otherwise the decision process goes on with the next step described below.

BPH resolving: If the *GUH* is a valid handle at all, the *BPH* contains all information necessary for resolving. The *SLSvc* looks up its internal tables which *OLS-ID* the *BPH* refers to. Three cases are possible:

- a) The birthplace *OLS* is online and therefore the *OLS-ID* can directly be resolved to a hostname. In this case the *SLSvc* goes on with step 5.
 - b) The birthplace *OLS* is no longer online and its content is now handled by one or more moved-birthplace *OLS*s. In this case the *SLSvc* goes on with step 6.
 - c) The birthplace *OLS* is just in progress of moving its content to one or more moved-birthplace *OLS*s and will then go offline. In this case the *SLSvc* goes on with step 7.
3. The hostname of the current *OLS* is returned to the requestor together with a flag that *CH* resolving has resulted in the given hostname.

The following tasks have to be performed then:

- a) The requestor contacts the given *OLS* and passes on the *GUH* with a resolve request for the *CH*.
 - b) If the *OLS* can resolve the *CH*, the mapped address is returned and resolving is finished.
 - c) If the *OLS* cannot resolve the *CH* (=entry was moved to a different *OLS*), it returns an appropriate error. The requestor invalidates the *CH* part in the *GUH* and continues with step 1.
4. The hostname of the moved-birthplace *OLS* is returned to the requestor together with a flag that *MBPH* resolving has resulted in the given hostname.

The following tasks have to be performed then:

- a) The requestor contacts the given *OLS* and passes on the *GUH* with a resolve request for the *MBPH*.
- b) If the *OLS* is still the home of the entry it returns a *CH* (the *CH* is usually different from the *MBPH* after a birthplace move!) together with the mapped

address. The requestor updates the *CH* part in the *GUH* and resolving is finished.

- c) If the requested entry had already moved on to a different *OLS*, only a *CH* is returned. In this case the requestor updates the *CH* part in the *GUH* and continues with step 1.

5. The hostname of the birthplace *OLS* is returned to the requestor together with a flag that *BPH* resolving has resulted in the given hostname.

The following tasks have to be performed then:

- a) The requestor contacts the given *OLS* and passes on the *GUH* with a resolve request for the *BPH*.
- b) If the *OLS* is still the home of the entry it returns the mapped address and resolving is finished.
- c) If the requested entry already moved on to a different *OLS*, only a *CH* is returned. In this case the requestor updates the *CH* part in the *GUH* and continues with step 1.

6. The list of *MBPHs* together with the corresponding hostnames which could be the moved-birthplace *OLSs* of the entry is returned to the requestor. A flag is set that there is uncertainty about the current *OLS* which manages the given *GUH*.

The requestor follows in principle the same procedure that was described in step 4 with one difference: according to the uncertainty approach described in section 7.2.3 the moved-birthplace *OLSs* are contacted one by one, until the handle can be resolved. The *MBPH* part of the *GUH* is updated with the *MBPH* of the *OLS* which is finally able to resolve the request.

7. The hostname of the birthplace *OLS* is returned to the requestor together with a list of *MBPHs* and corresponding hostnames which could be the moved-birthplace *OLSs* of the entry. An according flag is set to signal that moving entries are resolved.

In this case the requestor performs in principle the same procedure that was described in step 6, with the only difference that the birthplace *OLS* (i.e. the original host) is contacted first and asked for resolving. The uncertainty approach only has to be taken for entries that had already moved on before.

7.5 Distributing the *SLSvc*

Up to now it was assumed that the *SLSvc* “exists and is distributed”. However, clever distribution of the *SLSvc* is one of the most crucial points in order to make *DOLSA* useful in practice. Therefore this section is dedicated to an in-depth discussion of the appropriate considerations.

7.5.1 Organizational Structure of the Distributed *SLSvc*

The main goals that influence the organizational structure of the distributed *SLSvc* are:

- The *SLSvc* has to be robust against network failures.
- Traffic balancing has to be easy to manage by influencing the distribution of the *SLSvc*.
- Adding single servers to or removing single servers from the distributed *SLSvc* has to be easy and must not break the *SLSvc*'s consistency. This means that
 1. Distribution can change automatically.
 2. It has to be possible to define constraints for automatic distribution changes because of administrative reasons.

Both of these aspects together are in the following referred to as *semi-self-organizing distribution*.

Development of the distributed *SLSvc* according to these requirements resulted in the structure that was already sketched in section 7.1:

- There exists a small number of *MSLSs*.
- The *MSLSs* all have the same redundant information: they store the *OLS-ID-to-hostname* mappings of all *OLSs* that exist in the *SLSvc*.
- There exists an arbitrary number of *SLSs*.
- Communication-wise the *SLSs* are organized strictly hierarchically. Following the *semi-self-organizing* approach, the distribution hierarchy is adopted automatically according to robustness and network traffic aspects as well as according to administrative constraints.
- The *SLSs* are never the originators of any information (i.e. *OLS-ID-to-hostname* mappings), they are plain cache servers.

- Propagation of resolving information always strictly follows the communication hierarchy.
- Requestors (i.e. *Dinopolis Network Management* modules that want to resolve a *GUH*) send their requests to a convenient *SLS* and must not contact an *MSLS* themselves.
- Which *SLSs* may be contacted by which requestors also follows the *semi-self-organizing* approach with appropriate constraints.

The existence of several totally redundant *MSLSs* comes from the requirement to avoid single points of failure. Strategically well placed *MSLSs* (in terms of network infrastructure) result in a very high possibility that at least one of them is always reachable. What exactly happens in case of network failures and how the *SLS* hierarchy is temporarily reorganized then, will be investigated in section 7.5.7. The distribution hierarchy can also perform self-reorganizing steps in case of temporary or permanent overloads of single *SLSs* or whole subtrees as will be pointed out in section 7.5.6.

7.5.2 Adding and Removing *SLSs*

As a starting point for building up the whole distribution hierarchy let us assume that exactly one *MSLS* exists and that its hostname is known. No *SLSs* at all are online and therefore the existing *MSLS* is empty (i.e. stores no mappings at all).

If an *SLS* that has never been online before is added to the distribution hierarchy the following steps have to be performed:

1. The new *SLS* contacts the *MSLS* and asks for a unique *SLS-ID*. This ID is in principle the same as the *OLS-ID*, just for *SLSs*. Please note that also all *MSLSs* have *SLS-IDs* because they are in fact just very special *SLSs* with additional responsibilities.
2. After having obtained its *SLS-ID* the *SLS* has two possibilities to find its place in the hierarchy:
 - a) The *SLS* already knows its desired parent beforehand (e.g. the administrator already called up another administrator and asked for permission). Please note that this way of defining a parent only works with an explicit permission, since *SLSs* are neither forced to accept arbitrary *SLSs* nor arbitrary numbers

of *SLSs* as children. It is up to the *SLSs*' configuration, which children they accept according to an administrator definable ruleset.

- b) The *SLS* does not know its desired parent beforehand. In this case it can ask the *MSLS* for a list of *SLSs* that would accept it as a child. When asking for a parent a set of administrative data like network region, expected load, etc. is sent to the *MSLS*. The *MSLS* then tries to guess the best choice for a parent according to its internal management information (i.e. accept/reject policies of *SLS*-groups, network neighbourhood data, etc.)

3. From the list of possible parents that the new *SLS* now knows, the possible parent *SLSs* are contacted in arbitrary order and asked whether they are willing and able to accept the new *SLS* as a child.

If either all possible parents are unacceptable or no *SLS* from the obtained list is willing or able to accept the new *SLS* as a child, the decision process can be repeated by starting over at step 2. In this case the list of failed tries is provided for the *MSLS* to influence the new proposals accordingly.

4. At the end of one or more rounds of decision finding, the new *SLS* knows a set of acceptable parents. From this set one parent is chosen and informed accordingly. With this step the new *SLS*'s place in the communication hierarchy is fixed.
5. To avoid situations where contacted maybe-parent *SLSs* first signal willingness to accept the new *SLS* and then reject it after the *SLS* finally decided to be their child, each willingness message has an expiry date attached. Within the given time boundaries it is guaranteed that the promised place in the hierarchy cannot be refused. After the expiry date it is still possible, but not guaranteed that the new *SLS* will be accepted as a child.

The reason for this behaviour can be found in the fact that *SLSs* can be configured to accept a maximum number of children. This is only manageable if no "guarantee forever" is given.

6. After the new *SLS* fixed its place in the hierarchy, it sends the appropriate information together with administrative data about its accept/reject policy to the *MSLS* for future decisions. No *SLS* is forced to accept children, therefore it is possible without any problems to operate private *SLS* part-hierarchies. As has already been discussed, an *SLS* is never the originator of mapping-data,

therefore no information gets lost for the rest of the hierarchy, if part of it is just for private use.

In case of the above example with the single *MSLS* and the first *SLS* wanting to join the hierarchy the following happens: the only parent proposal from the *MSLS* will be the *MSLS* itself. The new *SLS* will therefore take the *MSLS* as a parent, the *MSLS* will accept it and the *SLS* sends its accept/reject policy.

When removing an *SLS* from the hierarchy the following actions have to be carried out:

1. If the *SLS* to be removed has no children, it just informs the parent and the *MSLS* that they should remove it from their internal tables. After that the *SLS* is no longer part of the hierarchy and can go offline. As is the case with *OLS-IDs*, also the *SLS-ID* will never be reused again. If for any reason the *SLS* is brought online again at a later stage it has to be added as a completely new and empty *SLS*.
2. If the *SLS* to be removed has children, it first notifies its parent and passes on the list of children to it.
3. The parent decides whether and how many of the children it is willing to take over from the child that will go offline.
4. All children that the parent is willing to take over are informed accordingly. If some children do not want to be taken over they can trigger an appropriate restructuring operation as will be described in section 7.5.6.
5. Additionally the parent *SLS* informs the *MSLS* about all children that it cannot take over. The *MSLS* then triggers the appropriate restructuring operations as will be described in section 7.5.6.
6. After all necessary restructuring has taken place, the *SLS* that wants to be removed is no longer part of the hierarchy and can go offline.

7.5.3 Propagation of Information between *MSLSs*

For availability and load balancing reasons the scenario with a single *MSLS* as the root of one single tree as discussed above is not the final solution. In figure 7.1 redundancy by using several synchronized *MSLSs* is already included. Achieving fully redundant, synchronized *MSLSs* is straightforward:

Just replace the term *MSLS* in the above algorithm by the term *any of the known MSLSs*. The only constraint is that an operation that started with one distinct *MSLS* has to be carried out till the end with the same *MSLS*. For example, it is not allowed to perform step 1 with one *MSLS* and then switch to a different *MSLS* for step 2.

If there are availability problems with one *MSLS* in the middle of a run of the algorithm (e.g. the *MSLS* becomes unavailable because there is a network breakdown), then execution of the algorithm is canceled and the whole algorithm is re-run from the very beginning with a different *MSLS*.

Synchronization of the *MSLSs* is straightforward too:

- All operations (e.g. adding an *SLS*) are considered local in respect to one *MSLS* as long as not all steps of the appropriate operation are carried out.
- When an operation is finished, all other *MSLSs* are informed.
- If one or more *MSLSs* are temporary unreachable, periodical retries are carried out in the background until the updates can finally be sent.
- If a request is sent from an *SLS* to an *MSLS* which it cannot resolve because some information is missing, it is assumed that a necessary update did not arrive yet. In this case a broadcast message is sent to the other *MSLSs* to find out which of them can resolve the request. In case of a broadcast all *MSLSs* have to send either a positive or a negative answer. This is necessary, because *MSLSs* that are temporarily unavailable have to be detected. Three situations can occur then:
 1. One or more of the *MSLSs* that received the broadcast can resolve the request. In this case the appropriate updates are sent immediately to the broadcasting *MSLS* and the request can be resolved.
 2. No *MSLS* can resolve the request, but all *MSLSs* are available. In this case an appropriate error alert is sent to the requestor.
 3. No *MSLS* can resolve the request, but at least one *MSLS* is temporarily unavailable. In this case an uncertainty error-alert is sent to the requestor to signal that the request can possibly be resolved later.

7.5.4 Propagation of Resolve Requests

The originators of resolve requests are applications that contact an *SLS* which is convenient for them. The contacted *SLS* can be somewhere in the hierarchy. How-

ever, in most cases the contacted *SLS* will be either one of the leaves of the tree or at least far down the hierarchy. For administrative reasons main nodes of the distributed tree will usually not accept resolve requests by Dinopolis instances. They will only accept requests sent by other *SLSs* (mostly only by their children).

The following algorithm applies when resolving a standard request (as opposed to a *force-authoritative-answer request* which will be described afterwards):

1. The *SLS* that is contacted looks up its internal tables (i.e. the cache).
2. If the *SLS* can resolve the request it sends an *unauthoritative answer* (short: *UA*) to the requestor. The term *UA* has the same meaning as in DNS: the answer comes from the cache, so there is no absolute guarantee that it is still correct.
3. If the *SLS* cannot resolve the request itself, it contacts its parent *SLS*.
4. The parent *SLS* starts with step 1.
5. Following the steps, the request is propagated up the hierarchy until it can either be resolved from the cache of an *SLS* or it finally reaches one of the *MSLSs*. If the request can be resolved by the *MSLS*, an *authoritative answer* (short: *AA*) is generated. If it cannot be resolved, a broadcast as described in section 7.5.3 is sent. The answer then depends on the result of this operation, as described above.
6. No matter whether an *AA*, a *UA* or an error alert is the final result of the resolve request, the answer is propagated back down the hierarchy. Every *SLS* on its way stores the answer (even in case of an error) in its internal cache.

If a requestor obtained an *NA* and this answer proves wrong because something has changed in the mean time, a *force-authoritative-answer request* can be sent. This kind of request is also propagated up the hierarchy, but none of the *SLSs* on the way up tries to resolve it. Therefore the request is guaranteed to reach one of the *MSLSs*. Resolving by the *MSLS* is undertaken as usual and the answer is propagated back down the hierarchy again. All *SLSs* on the way of the answer update their internal entries.

However, even an *AA* can be wrong, if there were changes exactly during the small period of time that was needed for the answer to be propagated to the initial requestor. Therefore the final resolving algorithm is slightly changed to:

1. The initial requestor contacts an *SLS* for resolving.
2. The resolve request is propagated up the hierarchy as described above.
3. The answer is propagated back down the hierarchy to the initial requestor and cached as described above.
4. If the answer proves right, i.e. the initial requestor is able to contact the desired *OLS*, the algorithm is finished.
5. If the answer proves wrong, it depends on the kind of answer, which steps have to be performed next:

The answer was an *NA*:

- a) The initial requestor sends a *force-authoritative-answer request*.
- b) If the obtained *AA* proves right the algorithm is finished.
- c) If the obtained *AA* proves wrong, the initial requestor continues with the steps below.

The answer was an *AA*:

- a) The initial requestor sends a *force-authoritative-answer request*. In this request the initial requestor also passes on a list of all *AAs* that it obtained up to now and that proved wrong.
- b) When this additional request reaches the *MSLS*, it is resolved and the result is compared to the list of wrong answers that was provided with this request.
- c) If the new answer is contained in the given list, two cases have to be distinguished:

- i. If the new answer is equal to the last element of the list, this means that the desired *OLS* is not available at the moment for whatever reason.
- ii. If the new answer is contained in the list, but is not the last element of the list, this means that there can be a “hopping-server” problem.

In both cases the *MSLS* updates its internal structures accordingly and sends an unavailable-error alert. This alert is propagated down the hierarchy and cached as usual. In section 7.5.8 *dead* and *hopping* server problems are investigated in further detail.

- d) If the new answer is not yet contained in the list of wrong answers, this means that the desired *OLS* has changed its hostname in the meantime.

Therefore the *MSLS* sends a new *AA* which is propagated down the hierarchy and cached as usual.

6. If the initial requestor obtains an unavailable-error alert, resolving has failed and the algorithm is finished.
7. If the initial requestor obtains an *AA* which can be resolved, the algorithm is finished.
8. If the initial requestor obtains an *AA* which cannot be resolved it starts over at step 5.

7.5.5 The *SLS* caching strategy

In principle *DOLSA* does not require a special caching strategy for the algorithm to work properly. A standard least-recently-used (short LRU) algorithm with a configurable maximum of entries to be cached suits the algorithm's needs perfectly.

To leave room for future enhancements all answers that are generated by either one of the *SLSs* or by one of the *MSLSs* have an extendable *additional info* field attached. This field can transport information like the *time-to-live* in the cache that *SLSs* can use for internal management purposes.

Research is going on to optimize network traffic by modifications of the caching strategy. Intermediate results show that the proposed use of an LRU approach with a self-adapting maximum number of entries is the best of the investigated solutions.

7.5.6 Reorganizing the *SLS* Distribution Hierarchy

There are several reasons for reorganizing the *SLS* distribution hierarchy:

- Network traffic in the distribution hierarchy is very unbalanced and some *SLSs* are overloaded.
- Due to network speed and bandwidth reasons some paths show poor propagation characteristics.
- An inner node of the hierarchy goes offline. Therefore the corresponding subtree has to be re-integrated.
- Administrative reasons dictate restructuring.

The algorithm that an *SLS* has to perform to obtain a new parent is the same as described in section 7.5.2, with the only exception that step 1 does not have to be performed, because an active *SLS* already has an *SLS-ID*.

Changing the distribution structure is therefore straightforward, once it is known how to reorganize the hierarchy. However, aside from forced reorganization due to administrative reasons or because an *SLS* goes offline, restructuring is a matter of response-time optimization.

To be able to implement algorithms that perform response-time optimization, statistical data of the nodes in the hierarchy has to be known. For this reason every *SLS* always keeps track of the following statistical data:

Number of received requests: One entry per child *SLS* and an additional entry for applications using this *SLS* for resolving are stored. Each entry contains the following statistical data:

- Average, minimum and maximum number of requests within the last minute, hour, day, week and month.
- Average, minimum and maximum percentage of requests that are resolved from the internal cache. The percentage is again stored for all of the above periods of time.

Request turnaround time: One entry per child *SLS* and an additional entry for applications are stored. Each entry contains the following statistical data:

- Average, minimum and maximum turnaround time for network connections within the last minute, hour, day, week and month.
- Average, minimum and maximum resolve time for requests within the above periods.

To be able to collect more detailed information about the resolve characteristics in the distribution hierarchy, special flags can be set for each request:

Trace upwards: If this flag is set, each *SLS* in the request-upward propagation path adds a pair of [*SLS-ID*, *timestamp*] as a trace to the request. The whole trace of the request is attached to the answer after the request has been resolved.

Trace downwards: If this flag is set, each *SLS* in the answer-downward propagation path adds a pair of [*SLS-ID*, *timestamp*] as a trace to the answer.

The trace-flags need not be set by the initial requestor, but can be set anywhere in the distribution path. Tracing is then only performed for parts of the path. Because request tracing can reveal sensitive information and because it results in higher network loads as well as in higher CPU loads a tracing policy can be defined for each single *SLS*. This tracing policy contains constraints, regarding who can request tracing as well as how often tracing can be requested for a given period of time.

7.5.7 Robustness in Case of Network Failures

Restructuring the hierarchy as described above need not always be permanent. In case of network failures that make parts of the distribution hierarchy unreachable, temporary restructuring for the period of failure is possible to ensure robustness of the service.

In principle the algorithm for temporary restructuring is the same as for permanent restructuring, with two differences:

1. Temporary restructuring is considered an emergency and therefore the policy of single *SLSs* to accept or reject children is different.
2. Temporary restructuring is not registered in the *MSLSs*. Every *SLS* that can serve as an emergency-fallback registers with one of the *MSLSs*. The emergency-fallback list and its updates are propagated down the hierarchy. *SLSs* can decide whether to store this information or not. In the usual case big *SLSs* further up in the hierarchy will store the list and small ones way down the hierarchy will eventually decide not to store it. The *SLSs* that do not store the list then rely on the fact that very probably at least one *MSLS* is reachable in case of a network failure.

If an *SLS* detects that its parent is not reachable any more, the following algorithm is performed:

1. If the *SLS* stored the emergency fallback list the algorithm continues with step 3.
2. If the *SLS* did not store information about emergency fallback *SLSs* it sends a request for the emergency list to one of the *MSLSs*. Usually at least one of the *MSLSs* should be reachable.

- If no *MSLS* is reachable the algorithm is finished. The *SLS* is temporarily only able to serve requests from the internal cache and has to send temporary-error alerts for requests that would have to be propagated up the hierarchy.
3. The *SLS* contacts one emergency fallback *SLS* from the list. In order not to run into the problem that the first *SLS* in the list is contacted by every *SLS* looking for a temporary parent, the *SLS* to be contacted is chosen randomly.
 4. The contacted emergency fallback *SLS* answers with a grant to resolve requests for a limited period of time. After the grant has expired a request for prolongation of the grant has to be sent or a different emergency fallback *SLS* has to be contacted.
 5. The emergency fallback *SLS* takes actions to prevent cheating. The cornerstones of cheating prevention are the following:
 - The emergency fallback *SLS* tries to contact the supposedly unreachable parent to ask for information.
 - The emergency fallback *SLS* tries to contact the *MSLS* to ask for information.
 - The emergency fallback *SLS* tries to contact other known *SLSs* to ask for information.

7.5.8 Additional Administrative Aspects of the *SLSvc*

Two more administrative aspects exist that are important for the *SLSvc* to work properly and to fulfill the robustness requirements:

Dead server detection: *SLSs* as well as *OLSs* could be taken offline by force. In this case the servers do not have the chance to perform the necessary steps as defined by the remove algorithm. Such servers have to be detected and distinguished from servers which are temporarily unreachable.

Hopping server detection and prevention: By error (or by eventual bad intention) *SLSs* can change their parents rapidly and cause a lot of internal administrative overhead due to massive restructuring operations. The same is true for *OLSs* that change their hostnames very rapidly. Both cases are subsummarized under the term *hopping servers*. Such servers have to be detected and the *SLSvc* has to protect itself from them for reliability reasons.

Dead server detection deals with two different cases:

1. A server was taken offline by force and was never brought back up again.
2. A whole part of the network is unreachable “forever” and therefore servers in this part of the network are de-facto offline by force. This can e.g. happen if a company installs a firewall with very restrictive rules.

The influence on the distributed lookup service is the same in both cases, therefore no distinction is made in this discussion. A server, no matter whether it is an *OLS*, an *SLS* or a *MSLS* is considered dead if it was not reachable at all for a long period of time. The timeout in this case has to be high enough, so that typical downtimes of servers for system maintenance do not lead to the case that a server is considered dead.

DOLSA distinguishes between two different timeouts:

Server down timeout: This is the timeout for a server to be considered temporarily down. Typical values for this timeout are between 1 minute and 2 hours.

Server dead timeout: This is the timeout for a server that is already considered down until it is considered dead. Typical values for this timeout are between 2 weeks and 1 year.

All timeouts are managed by the *MSLS*. The *server down* timeout clock is started when a server is detected as being unreachable for the first time. The timeout clock is stopped and reset when the server is reachable again. The timeout algorithm works as follows:

1. It is the responsibility of *SLSs* to send appropriate unreachable-alerts to the *MSLS*. If an *SLS* detects that either an *SLS* or an *OLS* is unreachable it informs the *MSLS* accordingly. Dinopolis instances contact their default *SLS* if they detect an unreachable server. The *SLS* has to check this alert for correctness by trying to contact the possibly unreachable machine. Only if this attempt fails the information is propagated to the *MSLS*.
2. If an *SLS* sent an unreachable-alert it can be declared responsible for this server by the *MSLS*. In this case the *SLS* has to try to contact the unreachable server periodically (with automatically prolonged periods for each try). As soon as the machine can be contacted again, the *MSLS* is informed accordingly.
3. Responsibility of *SLSs* for unreachable servers is managed by the *MSLS* in the following way:

- a) The first *SLS* that sends an unreachable-alert is automatically declared responsible.
 - b) If many *SLS*s send unreachable-alerts for the same server, the *MSLS* distributes the responsibilities with respect to “geographical” aspects.
 - c) Depending on the dynamics of unreachable-alerts from different regions of the network, the *MSLS* can contact former responsible *SLS*s and withdraw their responsibility.
 - d) If one of the responsible *SLS*s reports that the formerly unreachable server is up again, the *MSLS* informs all responsible parties accordingly.
4. If the responsible *SLS*s do not report reachability of a server within the *server down* timeout period, the *MSLS* itself tries to reach it once. If it is still unreachable the server is considered down and the *server dead* timeout clock is started.
 5. When the *server dead* timeout clock runs, the *MSLS* withdraws the responsibility for watching the unreachable server from nearly all *SLS*s and informs the remaining few that server-death detection is now in progress. The appropriate *SLS*s correct their contact periods accordingly (e.g. only one try per day, depending on the configuration).
 6. If the server is reachable again before the *server dead* timeout expires the *MSLS* is informed accordingly and in turn propagates the information to all responsible *SLS*s.
 7. If the *server dead* timeout expires without a report of reachability, the *MSLS* itself tries to contact the unreachable server once. If it still proves to be unreachable, it is considered dead. All responsible *SLS*s are informed accordingly that they can stop watching it and the ID of the server (either *SLS-ID* or *OLS-ID*) is invalidated forever.

Hopping server detection is easier to perform: each server (*SLS* or *OLS*) has to report a change of its address anyway in order not to become unreachable. Also changes in the *SLS* hierarchy have to be reported, otherwise *SLS*s are excluded from the hierarchy. Therefore it is enough to store a history of timestamps over a predefined period of time, typically in the range of 1 day to 1 month.

According to predefined policies the following rules apply for frequent hostname changes:

1. Servers changing their hostname too often (e.g. more than once per week over one observation period) are marked *hopping dangerous*. If rapid hostname changes go on they are excluded from the network by marking them *hopping* in the *MSSL*.
2. Information about hopping servers is propagated down the hierarchy to all *SLSs*. The *SLSs* have to remove all entries that refer to the hopping server from their cache and replace them with hopping-error alerts.
3. The *MSSL* nevertheless goes on observing the hopping candidate for a predefined period of time. If action calms down within reasonable time the *hopping* flag in the *MSSL* is removed and the server is therefore reachable again. If the server remains hopping it is finally declared dead and no longer observed.

The strong rules for hostname changes are necessary, since changing a hostname is not under control of the *SLSvc*. Changes of the hierarchy of the *SLSvc* are very well under control and therefore the algorithm is different for *SLSs* that are hopping around in the hierarchy:

- There exists a predefined minimal period of time, how long a child has to have the same parent to be considered *well-behaved*. Typical values for this period are between 4 hours and 1 week. The *well-behaved* rules only apply to permanent changes, not to emergencies and resulting temporary restructuring (see section 7.5.7).
- If an *SLS* changes its parents quicker than what would be considered *well-behaved* the following rules apply:
 1. A preconfigured number of quick hops, (e.g. 3) is considered a borderline case and no actions are taken.
 2. When exceeding the preconfigured number of quick hops, the following hops are delayed by force with growing delays per additional hop (e.g. a minimum-delay of 1/8 of the *well-behaved period*, doubled each time up to the maximum defined by the *well-behaved period*).

As an example let us consider the *well-behaved period* to be 128 minutes with a borderline case of 2 hops, a minimal delay of 1/8 of the *well-behaved period* and delay doubling per further additional hop. In this case a delay of 16 minutes applies for the third hop, 32 minutes for the fourth, etc. until the maximum delay of 128 minutes is reached.

3. Once the hopping server has slowed down again for e.g. twice the *well-behaved period*, no more forced delays apply and the counters are reset. If hopping starts again, the whole algorithm is executed from the beginning.

7.6 Further Investigations

Now that the distribution aspects of both *OLSs* and *SLSs* and the resulting algorithms are known, some final investigations about the cooperation between these two blocks of the lookup-service are necessary.

Objects moving on very fast between the areas of responsibility of *OLSs* can become a problem, because race-conditions can occur in the resolving steps. Such situations will be dealt with in section 7.6.1 and section 7.6.2.

One additional aspect of taking birthplace-servers offline (see section 7.2.3) will be described in section 7.6.3, because it is more or less just an implementation detail rather than part of the algorithm.

7.6.1 Guaranteeing Consistency During Move-Operations

Lookup-services in combination with move operations are always prone to race-conditions. These situations lead to the problem that object access can fail, although the object still exists. Two unavoidable delays during object movement are candidates for race-conditions:

Resolve-to-access delay: A requestor obtained an object address for a given *GUH*.

Exactly in the small period of time between resolving the *GUH* and access to the object, this object could move on. Therefore the address would no longer be valid and object access would fail. This case can be brought under control easily with the following algorithm:

1. When object access with an address that was the result of a lookup operation fails, the *OLS* is contacted again. Please note that the case of all systems being online is considered here. If access fails because the destination system is unreachable this is a different story.
2. If the *OLS* still returns the same address for the given handle, the *OLS's* entry is no longer valid and the *OLS* is informed accordingly to trigger removal (or unavailable-flagging) of this entry.

3. If the *OLS* returns a different address for the given handle, a move operation occurred. In this case access has to be tried again with the new address. If access fails again, resolving is started over with step 1. The case of fast moving objects where continuous race-conditions occur is discussed in section 7.6.2.

Lookup-update delay: If an object has to be moved there is always a short period of time during which *OLS* and Dinopolis instance are out of sync. If the *OLS* update takes place *before* the object is actually moved, a requestor could obtain a new address too early and access can fail. A naive solution would be to change the order of operations so that the *OLS* update takes place *after* the object has actually moved. Failures could then be treated as described for *resolve-to-access delay* above.

However, this naive solution just shifts the race-condition, because the requestor could be faster than the update operation. Then the second resolve request returns the same result as the first and therefore the handle would be deleted, although in principle everything is ok.

One prevention of this race-condition is to make move operations atomic by means of locking mechanisms. However, considering that objects can also be moved between areas of responsibility of different *OLSs*, locking is not desirable, because it can lead to difficult-to-solve problems in case of network failures.

Different situations and therefore different algorithms to overcome problems with race-conditions due to *lookup-update delays* have to be distinguished:

1. Moving an object inside one Dinopolis instance.
2. Moving an object between Dinopolis instances that are under the same *OLS's* responsibility.
3. Moving an object between Dinopolis instances that are under different *OLSs'* responsibilities.
4. The case that an object is moved inside one Dinopolis instance for which more than one *OLS* is responsible is not considered here for two reasons:
 - a) A configuration with several *OLSs* for one Dinopolis instance does not make much sense at all, because it just results in administrative overhead without providing any advantages.

- b) If, for whatever reason, such a configuration is really desired, then the parts of the Dinopolis instance that are under the responsibilities of different *OLSs* are considered to be different *virtual* Dinopolis instances with one *OLS* being responsible for each of them from the point of view of the algorithm.

Case 1: One *OLS*, One Dinopolis Instance. In case that a move operation takes place within one Dinopolis instance with one *OLS* being responsible for it, the algorithm is as follows:

1. A forwarder is instantiated locally within the Dinopolis instance under the old address of the object.
2. Inside the Dinopolis instance an atomic move operation of the object to its new address takes place. In this atomic operation the forwarder is changed accordingly to point to the new address. As long as the forwarder exists, the object is therefore accessible under two different addresses.
3. The *OLS* entry is updated.
4. The local forwarder is removed.

Case 2: One *OLS*, Multiple Dinopolis instances. When moving an object between two Dinopolis instances that are both inside the area of responsibility of one *OLS*, a move operation is performed in the following way:

1. The object is copied to the desired location in its destination Dinopolis instance. The original resides where it is and is still fully functional.
2. When the object is finally fully functional in its destination Dinopolis instance, the source Dinopolis instance is informed accordingly.
3. The source Dinopolis instance triggers an update of the *OLS* entry to make it point to the new location.
4. After the update has taken place the original object is deleted from the source Dinopolis instance.

If network problems occur that prevent the move operation from being completed, a rollback is performed by the single parties and the operation is repeated from the beginning.

Case 3: Multiple *OLS*s, Multiple Dinopolis instances. When moving an object between two Dinopolis instances that are in different *OLS*s' areas of influence, the algorithm for a move operation changes to:

1. The object is copied to the desired location in its destination Dinopolis instance. The original resides where it is and is still fully functional.
2. When the object is finally fully functional in its destination Dinopolis instance, the destination *OLS* is updated with the new entry.
3. During updating, the destination *OLS* contacts the source *OLS* and reports the new entry.
4. Upon being contacted, the source *OLS* replaces the original entry with a temporary forwarder.
5. After the above *OLS* update-chain is finished, the destination Dinopolis instance informs the source Dinopolis instance accordingly. The source Dinopolis instance deletes the original object.
6. If the source *OLS* was the object's birthplace or moved-birthplace, the forwarder is kept. Otherwise the source *OLS* contacts the birthplace *OLS* or moved-birthplace *OLS* accordingly to report the changes (i.e. it sends the *CH*). Once this update has taken place, the source *OLS*'s entry is finally deleted.

If network problems occur between the source and destination *OLS* or between source and destination Dinopolis instance, a rollback is performed and the operation is repeated from the beginning. Network problems between the source *OLS* and the birthplace *OLS* do not influence the operation, because this final step is carried out in the background.

However, when carrying out birthplace *OLS* updates in the background, one important point has to be kept in mind: if network problems prevent the source *OLS* from contacting the birthplace *OLS* immediately, this can lead to a further race-condition:

The source *OLS* keeps the temporary forwarder until the update has finally taken place. In the mean time it tries periodically to reach the birthplace *OLS*. Thus it can happen that the object moves on again, before the birthplace *OLS* could be updated. If this is the case, then update requests could reach the birthplace in wrong order and the outdated update could overwrite the new one.

One possible solution for this problem could be the use of timestamps for update requests, because then the birthplace *OLS* can distinguish between actual and outdated update requests. However, experience shows that enormous deviations between the system times of computers in a network are very usual. Implementing time-synchronization algorithms between the single *OLSs* is complicated and in fact unnecessary, considering one important fact: it is not interesting to know exactly *when* a move operation happened. The only important fact is the *order* in which the operations were performed.

Therefore the following steps are added to the above algorithm for the step of birthplace *OLS* updates to be able to track an object temporarily to ensure consistency:

- For a moved object the destination *OLS* stores a temporary trace of the object as additional information in its entry.
- Updates of the birthplace *OLS* always contain the new *CH* together with the eventually remaining (see below) temporary trace.
- The birthplace *OLS* always informs the destination *OLS* when an update finally succeeded. Upon receipt of this notification the destination *OLS* can delete the temporary trace because it is no longer needed.
- The birthplace *OLS* keeps eventual temporary traces until all source *OLSs* that are contained in them have reported the changes. In this case the changes are simply discarded, because they are recognized as outdated.
- After all missing update-notifications have finally reached the birthplace *OLS*, it can delete the temporary trace information.

The advantage of this algorithm is that forwarders are collected successively until a final stable state is reached. Nevertheless, updating the birthplace *OLS* can take place in the background and does not slow down object movement. A synchronized version of updating the birthplace *OLS* is unthinkable, because a temporarily unreachable birthplace would prevent all move operations completely for this period of time for objects that have been born there.

7.6.2 Catching up on Fast-Moving Objects

The algorithm to overcome *resolve-to-access delays* described in section 7.6.1 has one weak point: If objects are moving on very often within a very short period

of time it can happen that resolving is always one step behind. One can say that usually it is slower to move objects than to resolve their *GUHs*. One can also take into account that the move algorithms described above collect chains of forwarders. Therefore it is in fact extremely unusual that resolving cannot catch up, even on very fast moving objects.

However, it is never good practice to rely on such probabilities, because reality proves that even the smallest probability is not equal to zero. Therefore *DOLSA* also takes this aspect into account. The algorithm to ensure that resolving can catch up on fast-moving objects is in principle the same as the algorithm for catching up on *SLSs* hopping between parents as described in section 7.5.8. Object movement is slowed down by force after too many move operations in too short a time. Considering that e.g. mobile agents that are hopping around rather quickly shall also be possible, the parameters for *well-behavedness* have to be chosen accordingly. Taking a maximum of e.g. 9 hops per 3 minutes for *well-behaved* objects is a good starting point depending on the desired applications.

Although slowing down objects guarantees that requestors can catch up on them, there are also cases, where object movement shall be prohibited for single objects as soon as possible. Such a case would e.g. be an emergency-alert for a travelling agent. For this reason a special stop-object-broadcast exists. This broadcast has to reach all *OLSs* quickly, which is easy, if for each *OLS* one *SLS* is responsible. One *SLS* can also be responsible for several *OLSs* because *responsible* in this context only means that if an *OLS* has to be contacted, it is known who does it.

Due to the hierarchical organization of the *SLSvc*, propagation of a broadcast is a massively parallel operation that reaches all *OLSs* in just a few steps. When an *OLS* receives the broadcast to stop a certain object (defined by its *GUH*) it has a look if this object is currently in its area of responsibility. If yes, it is flagged *frozen* to prevent further move operations. Additionally, an answer is sent to the originator of the broadcast to report where the object resides at the moment. The answer can either be sent directly or, if network problems occur with the direct connection, routed back on the track of the broadcast.

7.6.3 Merging Servers

One necessary implementation detail of *OLSs* was not discussed in depth in section 7.2.3: it is not possible that the *LOIDs* from the birthplace *OLS* are simply reused on the moved-birthplace *OLS* if the moved-birthplace *OLS* already contains entries. In this case uniqueness could not be guaranteed (e.g. assume that both servers simply implement a counter starting with ID 0). Further, it is up to the single implementations of *OLSs*, how *LOIDs* are managed and which schema they follow.

For this reason *OLSs* have to implement a namespace mechanism to be able to store mappings taken over from other birthplace *OLSs* or moved-birthplace *OLSs*.

7.7 The Address Management Module in Respect to *DOLSA*

After this very common description of the *DOLSA* algorithm that represents the basis for Dinopolis address management it is finally time to clarify what the tasks of the *Address Management* module inside a Dinopolis instance are exactly. In chapter 4 it was mentioned that the *Address Management* module is responsible for *local address management* as well as for *global handle management*. In respect to *global handle management* it represents the Dinopolis internal stub that is used to communicate with the lookup-service. According to the description of the *DOLSA* based lookup service above the *Address Management* module plays the role of the requestor in the *GUH* resolving process.

Concerning *local address management* the role of the module is also quite simple: it is up to the implementation of embedders whether they support direct (physical) addressing for external systems or whether they want to utilize the features of a lookup-service too. In case that embedders implement direct addressing, the *Address Management* module plays no role at all. In case that embedders want to use a lookup-service for their internal address management purposes the *Address Management* module provides the appropriate calls to support private handle spaces in the *OLS*. In analogy to *GUHs* the resulting private handles are called *LUHs* (=locally unique handles). This feature can be very useful in order to become robust

in respect to embedded system internal restructuring operations that have to be hidden from Dinopolis but are not subject of *GUH* management.

8. The Interrelation Management Module

Besides robust treatment of object handles another part of the Dinopolis design is unique and not found in any other of today's existing systems: the use of common interrelations between arbitrary objects or parts of them, no matter whether they support interrelations or not, for modelling any kind of semantic relations.

One of the most important necessities in any system and as such the starting point for the design of the Dinopolis interrelation management mechanism is the clean and complete separation of navigation from addressing. It has been common knowledge among computer scientists for a long time that addressing and navigation are semantically two completely different mechanisms. However, looking at the real world it can be seen that these semantically different mechanisms are in most cases mixed up completely which causes lots of problems. Just considering whatever kind of filesystem in today's operating systems the whole mess becomes apparent:

- A filesystem supports logical structuring of files through subdirectories. However, the logical structure is reflected in the path that is used to address a file. The result is that logical restructuring breaks the addressing mechanism.
- Most filesystems support one or the other kind of symbolic links. These reside somewhere in the structure of the filesystem and internally store a unidirectional (!) pointer to some other point in this structure. This means that every move operation which is applied to the symbolic link itself breaks the addressing mechanism as described above. It also means that every move operation which is applied to the destination of the symbolic link additionally breaks the logical structuring mechanism because then the symbolic link points to nowhere.

The example of the filesystem was deliberately taken because most of today's hypermedia systems, object systems and component systems rely on the capabilities of a filesystem in one way or the other. Their internal structure is mapped to the filesystem and the filesystem's logical structure is reflected in the way the systems

present their content to the outside. Some systems rely on the use of databases which usually alleviates the problem drastically (e.g. Hyperwave). Nevertheless these systems usually also have several shortcomings because of the way how *general* semantic modelling using interrelations is supported (or actually unsupported).

8.1 The Logical Concept of Interrelations in Dinopolis

The approach taken in Dinopolis is a strict separation of *addressing* which is influenced by the technicalities of the system as was discussed in chapter 7 and *interrelation management* which is influenced by logical and semantical requirements. All navigational issues are modelled by the use of arbitrarily typed interrelations that are attached to *Dinopolis Objects*. This is also true for embedded systems that mix up addressing and interrelations as is the case with filesystems. Internally this mixture is broken up by the embedder and a cleanly separated representation according to the *Dinopolis Object* model is presented.

Interrelations are not only designed for modelling arbitrary hierarchical or non-hierarchical navigation paths through the object space. The design behind interrelations represents a holistic view of possible logical relatedness of objects. Up to now two applications of interrelations have already been discussed: modelling of compound objects and dynamic types. Many other applications are thinkable, such as hyperlinks, object inclusion, high-level document composition, chat connections, etc., just to mention some obvious candidates. Whenever objects are related in one way or the other, the Dinopolis interrelation mechanism comes into play. The cornerstones of interrelations can be subsummarized as follows:

- For the sake of generality $n : m$ interrelations have to be provided. In many cases only $1 : 1$ or $1 : n$ interrelations will be needed, nevertheless there exist situations, where the general $n : m$ case applies, e.g. for interconnecting two version controlled objects. It is true that $n : m$ interrelations can be emulated by using many $1 : 1$ interrelations. Nevertheless, this would cause avoidable overhead and therefore interrelations in Dinopolis are designed to represent the $n : m$ case.
- Interrelations can be used to interconnect arbitrary *Dinopolis Objects* without any limitations concerning their locations on the net.

- Interrelations can also be used to interconnect arbitrary *parts* of *Dinopolis Objects* to each other. For example a citation from within the content of one *Dinopolis Object* (that represents a document) to parts of the content of a different *Dinopolis Object* (that also represents a document) is possible.
- Interrelations cannot only interconnect objects with objects, they can also be attached to interrelations. Referring to a interrelation allows for example to model a link in a document that itself points to a hyperlink between other documents.
- Interrelations are of arbitrary logical type and can be treated type-dependently. For example, there is a semantic difference between a hyperlink, an inline image and a dynamic type. Therefore the logical type has to be reflected in the interrelations.
- Arbitrary meta-information can be attached to interrelations and to their single endpoints.
- Interrelations have to be robust against object movement. If an object changes its physical location this must not break the interrelation mechanism and result in interrelations pointing to nowhere.
- Logically spoken, interrelations can be directed or directionless. Technically spoken, interrelations have to be implemented in a way that internally it is possible to reach all endpoints of a interrelation if only one of them is known.

8.2 The Technical Concept of Interrelations in Dinopolis

The property which is most difficult to control for interrelations in today's systems is robustness against object space restructuring. This is not true for Dinopolis due to the concept of *GUHs*. As has been discussed, interrelations are always attached to objects. Therefore the endpoints of interrelations can be represented by *GUHs* and with this decision robustness of interrelations is guaranteed.

The requirement that interrelations can be used to interconnect arbitrary objects anywhere on the net is also easy to meet by making interrelations objects of their own. Consequently interrelations in Dinopolis are represented by *Dinopolis Objects* themselves. With this decision the requirements to be able to attach arbitrary meta-data and to define interrelations that are attached to other interrelations are implicitly fulfilled: *Dinopolis Objects* can hold meta-data, therefore interrelation

objects can hold meta-data. Interrelations are always attached to *Dinopolis Objects*, therefore they are attachable to interrelation objects. Being *Dinopolis Objects*, interrelations in Dinopolis gain an additional degree of freedom, allowing usage-scenarios that cannot be modelled with any other of today's systems:

- Interrelations are addressable entities, therefore they can be explicitly stored anywhere on the network. This makes it possible to construct independent, structured collections of interrelations, e.g. a personal electronic library. Arbitrary many different classification systems and navigation paths through the object space are manageable by just utilizing the appropriate interrelations as structure objects.
- Interrelations are individually security-checked and therefore access rights for interrelations are controllable completely independently from the objects that they refer to.
- Utilizing the possibility to have $n : m$ interrelations, context specific hyperlinks can be modelled easily. For example a hyperlink has several different formats of a document as its destinations. According to the requestor one of the types can be chosen automatically, depending on different factors, e.g. which types the application supports.
- Arbitrary passive and active behaviour can be modelled by the use of specific *static classes* and *dynamic types* (see chapter 6). Considering security issues one more time this means that access is not only controllable independently from the endpoints, it also means that it is possible to implement arbitrary mechanisms, e.g. to support inheritance of access rights from the endpoints. For example in one scenario it could be desired that interrelations are shown, even if the object(s) on the other end of those interrelations are not accessible with the current user access rights. When trying to follow such an interrelation the system would require re-identification to be able to access the endpoint. In a different scenario it could be desired that interrelations pointing to unaccessible objects are made invisible so that not even the information about the existence of such an interrelation is revealed. This situation happens quite often in medical applications, because data is sensitive enough that even the knowledge about its existence would reveal too much information.
- *Dynamic types* themselves are attached to objects by using interrelations.

In the following the tasks of the *Interrelation management* module are described for standard operations related to interrelations. Further, the special mechanisms that are related to the handling of *dynamic types* are outlined, because these mechanisms are needed internally by the Dinopolis system.

8.3 Loading and Storing Interrelations

Describing the process of loading and storing interrelations is deliberately placed before coming to creation and deletion of interrelations. The reason is the existence of the so-called *implicit* interrelations that will be identified in this section and that have much influence on the other processes.

In principle loading interrelations is exactly the same as loading any other kind of *Dinopolis Objects* (see section 6.3). However, one special case exists that deserves some extra consideration:

In section 8.1 it was mentioned that the implicit navigational structure of a filesystem is internally mapped to interrelations. This implies that interrelations are not *explicitly* stored somewhere and loaded from there. Rather they are created from filesystem-immanent information. Due to the *dynamic storage type* that is supported in Dinopolis this situation is easy to manage. In section 6.3 it was defined that the *dynamic storage type* is responsible for loading the internals of the object. Due to this behaviour it is easy for embedder developers to provide the appropriate *dynamic type* that can evaluate the external systems' implicit information and convert it into an internal interrelation object.

Storing interrelations is also exactly the same process as storing any other kind of *Dinopolis Objects* (see section 6.2). For the case of implicit interrelations again some additional discussion is necessary. Two obvious cases can happen:

- There were no changes to the interrelation and therefore no store operation is necessary. This case is trivial. Nevertheless this case is mentioned here, because implicit interrelations sometimes can behave quite differently from explicit ones and it can happen that no explicit store operation at all is necessary or supported.
- If there were changes to the interrelation that would make a store operation necessary a very important question arises: *what exactly does a change of an implicit*

interrelation mean in the context of external systems' implicit structuring mechanisms? Four different cases can be identified:

1. Changing an interrelation triggers an external restructuring operation immediately. For example changing a parent-child interrelation that is filesystem-immanent would cause an immediate move of the child. This behaviour has to be reflected by the *dynamic type* of the interrelation and therefore the necessary operations in the external system take place immediately when the change of the interrelation is requested. Therefore an explicit store operation is not needed in this case and results in *no operation*.
2. Changing an interrelation does not trigger an immediate external restructuring operation. For example, changing the destination of a symbolic link in the filesystem need not necessarily be carried out immediately. In this case the *dynamic type* supports a standard store operation which performs the external changes on request. This behaviour is in fact the same as with explicit interrelations.
3. Depending on the requested change an unsupported restructuring operation could be requested. For example changing a filesystem immanent parent-child interrelation could lead to a circular relationship. This operation is therefore not allowed and it is the responsibility of the *dynamic type* to prohibit it and raise the appropriate exception. In this case no explicit store operation is necessary either, because changes were prohibited.
4. Sometimes it is desired behaviour to reflect changes on implicit interrelations by making them explicit. Depending on the virtual system configuration (see chapter 9) arbitrary mixtures of implicit and explicit interrelations inside one virtual system are possible. Considering the example of a virtual system which consists of two filesystems on one Unix machine it can happen that a parent-child relationship is changed in such a way that the consequence is to move a file from one filesystem to the other. In order not to lose the logical parent-child relationship the *dynamic type* could decide to make the relationship explicit by storing it as a symbolic link.

It can easily be seen that *dynamic types* are the key to be able to implement mappings of implicit interrelations. Without them a clean mapping to “real” interrelations inside the system would be a very difficult task that would break the clean system architecture. Working with interrelations an application always has

to be aware that no assumptions are valid about *when* the appropriate operation is triggered that makes applied changes permanent. For this reason *dynamic types* support requests for information about their behaviour.

One more interesting case has to be considered when speaking of implicit interrelations. Inside Dinopolis they are *Dinopolis Objects*, therefore they can be accessed via a *GUH*. Nevertheless these objects are usually not really *created* in the sense of an explicit call to create a new *Dinopolis Object*, they just “appear”. In order to guarantee consistency of the *GUH* mechanism the following rules apply for implicit interrelations:

- Although they are *Dinopolis Objects* no *GUH* is assigned to them if they just appear and disappear again without anybody noticing them explicitly. This means that they are sort of temporary objects as long as nobody accesses them directly. This behaviour is no contradiction to globally unique robust handles, because as long as nobody knows that they exist at all, nobody will request them.
- When such an implicit interrelation is asked for its *GUH* the first time, it is assumed that access to this interrelation can happen through a resolve request at a later stage. In this case the *Address Management* module is contacted and the interrelation is registered with a fixed *GUH* from this moment on.

The reason, why *GUH* registration for implicit interrelations is delayed until the first direct request occurs is scalability. Registering each implicit interrelation immediately as soon as it appears for the first time would result in an unnecessary high number of registration requests and also in a high number of *GUHs* that will never be used again.

8.4 Creating and Deleting Interrelations

Interrelations are *Dinopolis Objects*. Therefore creation is a straightforward process as described in chapter 6. At least this is true for interrelations that are explicit. Implicit interrelations as discussed above usually cannot be created directly, because they are a result of other operations. For example creation of a *Dinopolis Object* which has its persistence location in a filesystem will also result in the creation of an implicit parent-child relationship. However, implicit interrelations are not

a problem of the creation process at all, because it is easy to handle that they can just “appear”. Requests to create implicit interrelations are handled by the appropriate virtual system configuration and treated accordingly (mostly resulting in an exception). Therefore even this is not a problem and no further considerations are necessary.

For this reason let us come back to creation of explicit interrelations: what makes a *Dinopolis Object* an interrelation is its *static class* as well as its *dynamic type*. Special interrelation types with arbitrary context-dependent functionality are modelled using these concepts. Creating an interrelation does not immediately bind it to certain endpoints. This has to be done explicitly as described in section 8.5. The reason for this behaviour is easy to see: having $n : m$ interrelations means that endpoints can be attached and detached during their lifetime on demand. Although it does not make much sense at a first glance it is possible to have $n : 0$, $0 : m$ or even $0 : 0$ interrelations in Dinopolis. In practice such degenerated interrelations have some significance, considering the following example of an electronic library built with Dinopolis:

Let us assume that a huge number of documents has to be inserted into the library. Let us further assume that all documents are arbitrarily interlinked. Either the source or the destination document (in respect to the hyperlinks) is inserted into the system first. For applications it is much easier to handle the case that a relation with only one of the desired endpoints can be created and the second endpoint can be attached to it at a later stage, namely when the other document is inserted. Otherwise insertion would have to be done in a special way by first inserting all the documents and then in a second run inserting all the hyperlinks. As a result the application would be much more difficult to implement and also some additional (unnecessary) requirements about the data interchange format for lots of documents would apply.

As a different example let us consider the simple case of an application supporting bookmarks. The straightforward way to implement them is to store interrelations that just have one endpoint. Due to the concept that Dinopolis interrelations support arbitrary meta-data nothing more is necessary for storing and also for visualizing those bookmarks. Without allowing that degenerated interrelations exist, senseless objects would have to be created that represent the second endpoint.

As a consequence interrelation object management and interrelation endpoint management are strictly separated in the Dinopolis design and therefore also treated separately in this thesis.

More interesting than creation of interrelations is the process of deleting them, because some safety and security aspects come into play that have much influence on this process. The principle of interrelation deletion is easily sketched as a two step process:

1. Detach all interrelation endpoints from the objects that they refer to.
2. Delete the interrelation object itself.

However, from the point of view of system consistency and also for security reasons some aspects of deletion have to be considered:

- If a security check results in the case that at least one endpoint of an interrelation must not be detached from the referenced object then the interrelation must not be deleted at all. Otherwise the operation would result in an endpoint that hangs in a vacuum. It was not discussed up to now how endpoints are exactly implemented, but it can easily be seen that an endpoint without an interrelation that it belongs to cannot be considered a consistent state. Therefore this situation must not occur.
- If the interrelation is an implicit interrelation with its origin somewhere in an external system, its removal can have interesting and also unwanted consequences. Just consider a parent-child relationship in a filesystem. Deleting this relationship in fact means deleting the whole subtree underneath it in the filesystem!

Especially in the case of implicit interrelations no general rules can be given how an algorithm has to react to ensure that no unwanted effects occur. Also the case that at least one endpoint cannot be detached is situation dependent: it could be desired that in such a case just the interrelation itself with the single endpoint remains in the system. On the other hand it could also be desired that the whole interrelation with all endpoints remains untouched. The only solution that allows configurable treatment of such cases is again the use of *dynamic types* that implement the desired behaviour on an application-context basis.

8.5 Endpoints of Interrelations

As has been pointed out, the persistence of interrelations is in principle independent from the objects that they interconnect. Schematically, interrelations are following the principle sketched in figure 8.1.

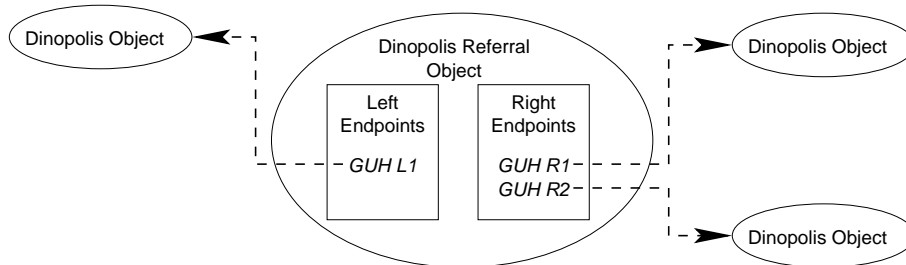


Figure 8.1: Schematic view of Dinopolis interrelations

The situation in figure 8.1 shows an interrelation that interconnects one object with two other objects. What exactly the semantic meaning of the endpoints is, whether the left side is the source and the right side the destination of a hyperlink or whether the left side object is a document that includes one or the other object on the right side alternatively, etc. is up to the interpretation of the interrelation's type. No matter which interpretation is desired, the principle of interrelations is always the same:

- An interrelation is a *Dinopolis Object*.
- The left and right side endpoints of an interrelation can be accessed via operations that the interrelation supports.
- All endpoints are stored (and requestable) as *GUHs*. Therefore robustness of interrelations is guaranteed.

Although this principle fulfills all requirements concerning flexibility and robustness and is therefore perfectly suited for Dinopolis, there is one aspect that needs further investigation: performance! When having an interrelation in hand everything works perfectly. However, what happens when requesting all interrelations that are attached to a *Dinopolis Object* as has been required in chapter 3? Just implementing the principle that is sketched in figure 8.1 would mean that a request for

interrelations results in a massively distributed search operation, because interrelations can be stored anywhere on the net. This is for sure not the desired behaviour, because it would make interrelations unusable.

8.5.1 Making the Principle Work in Practice

The first idea that comes to mind when considering the performance problem of the above principle is to revert the internal (!) direction of interrelations, by storing all *GUHs* of all attached interrelations inside the *Dinopolis Object* that represents one of their endpoints. Unfortunately, this is not the solution either, because it just shifts the problem. With this mechanism it is easy to access all interrelations that are attached to one object, but resolving each interrelation to obtain the objects at its other end then again results in massively distributed search operations.

As can easily be seen, the only solution to the problem is internal bidirectional storage. Every object holds all the *GUHs* for all interrelations attached to it and every interrelation holds all the *GUHs* of both, the left and the right hand side endpoints. At a first glance this seems to be a satisfactory solution for the problem, but in reality it is not. One special, but very important situation is neglected here: it can happen that a *Dinopolis Object* resides on a read-only system (e.g. on a CD-ROM). Prohibiting to create interrelations that point to objects on read-only systems is an unacceptable restriction. Fortunately, *Dinopolis* implements virtual systems that for example also allow to combine read-only systems with others that allow writing. This approach finally solves the problem satisfactorily, at least in the case of interrelations that need to be *attached* to objects residing in read-only systems.

If interrelations have to be *detached* from objects that are already part of the objects' persistent data on e.g. a CD-ROM, one more problem arises: it is not possible to detach an entry from the object on the read-only storage. Therefore the *dynamic storage type* for the virtual read-only / read-write combination of embedded systems has to be implemented in a way that it also allows to write "deleted" markers on the read-write part of the system, for interrelations that are no longer attached to an object. All interrelations that are marked "deleted" are then no longer included in the objects' internal representations. This principle which is not only useful for dealing with relations but also has many other applications, is called *shadow storage*

principle. Compared to other systems it is quite easy to implement it in Dinopolis by the use of the appropriate *dynamic storage types*.

8.5.2 Access Control Aspects

Read-only systems and shadow storage are one facet of object access, the other facet is security. In many of today's systems there is no distinction between read-only systems that technically do not allow writing and security mechanisms that intentionally forbid writing. The technical restrictions can easily be overcome in Dinopolis as has been discussed. Now let us have a closer look at the security mechanisms that are supported for interrelations inside Dinopolis, because they also differ from standard security mechanisms of other systems in many respects.

Usually interrelations are not explicitly protected by security mechanisms. The interrelations themselves are visible as long as the object that they are attached to is visible. Requesting the opposite endpoints of interrelations usually works as long as the objects on the opposite side may be read. In Dinopolis much more control is provided:

- Access rights can be attached to the endpoints of interrelations as well as to the interrelations themselves.
- The protection mechanisms are arbitrarily fine-grained so that it is e.g. possible to define explicit rules for attaching or detaching endpoints that have nothing to do with write access to the object at the endpoint. For example it is possible to define that an object must not be edited and no existing interrelation endpoints may be detached but new interrelations may be set. Also the other extreme, namely that an object may be edited but no interrelation endpoints may be changed at all, is possible. Arbitrary combinations are possible.
- Fine-grained access mechanisms do not only apply for the endpoints of interrelations, but also to the interrelation objects themselves. The order of interrelations can be security-protected according to their type (e.g. 1 : 1 is forced). It is also possible to control both the left and right-hand side of interrelations independently from each other. For example one could allow to attach new right-hand endpoints but no left-hand endpoints. Detaching of endpoints may be forbidden completely. This can be very useful when modelling hyperlinks that point to alternative documents of arbitrary different formats which are chosen on a context basis. In this

case the order of interrelations is limited to $1 : m$, meaning that the single left-hand endpoint represents the source of a hyperlink and all right-hand endpoints represent the different alternatives. Further it means that the source endpoint must never change, whereas new alternatives can be added (but not removed). Again arbitrary combinations are possible.

Many reasons exist that require fine-grained security mechanisms for interrelations, one of them is very Dinopolis specific: without such mechanisms the implementation of *dynamic types* would be very prone to inconsistencies! *Dynamic types* are based on the use of appropriately typed interrelations, because this is the cleanest way to attach them to objects. Changing anything, either the endpoints or the interrelations themselves or some of the meta-data of interrelations would cause a change of the *dynamic type*. This can be desired, if it is well-controlled. However, if it is not controlled well enough it can cause severe problems, because the nature of objects can change in very unwanted ways or at unwanted times.

8.6 Interrelations Attached to Parts of Objects

Up to now it has been discussed that endpoints of interrelations are *Dinopolis Objects*. There exist cases, where the endpoints of interrelations should not be the whole objects but only parts of them. For example let us consider the case of a link in HTML. There it is possible (and desirable) to let the link point inside the document. Modelling this situation with *Dinopolis Objects* and interrelations, the result would be that the source HTML document would be the content of one object, the destination document the content of another object and the link an interrelation that points from one part of one object's content to one part of the other object's content.

By simply storing *GUHs* as endpoints of relations this scenario can only be got under control if each content of the objects representing the HTML documents would be made up of single *Dinopolis Objects*. The composition of objects would have to be fine-grained enough so that every single letter in the documents would be represented by an appropriate *Dinopolis Object*. As can easily be imagined, this is not realistic, because it does not scale at all. Although everything *can* be represented by *Dinopolis Objects* in the system, it is not possible to require that everything *must*

be represented that way. *Dinopolis Objects* shall be used to represent the logical and semantical building blocks of the information space. They shall not be misused to put technical burdens on the developers. Depending on the desired application and depending on the content type it can make sense to represent content in the form of many structured *Dinopolis Objects*. If it does not, nobody should have to do it. Considering a different example, e.g. letting hyperlinks point inside video-streams it can easily be seen that the plain object model would fail completely.

Several more situations exist, where troubles arise with the mechanisms that have been discussed up to now. A very important one was already mentioned in chapter 3 as a requirement for *Dinopolis Objects*: it has to be possible that interrelations interconnect meta-data. For obvious reasons it is not desirable either that all chunks of meta-data have to be *Dinopolis Objects* themselves. This would drastically influence scalability and performance of the system.

These problems can be overcome by introducing the so-called *Object Subsets* with the following properties:

- An object subset is represented by an object together with arbitrary additional information that defines the subset.
- Directly requesting an object subset is done by requesting resolving of the *GUH* with the given subset information.
- Object subset extraction is applicable to every *Dinopolis Object* as long as it supports the appropriate subset request type.
- To provide a clear mechanism for subset extraction the subset information is typed by means of an arbitrary *static class*. The only restriction for this class is that it has to implement a common “subset” interface.
- The result of subset extraction can be an object of arbitrary type. It can also be a *Dinopolis Object*, but it need not be one.

The algorithm for subset extraction works as follows:

1. Subset extraction is called directly on the *Dinopolis Object*. This call includes an instance of the object subset information class with the appropriate information set.

2. The *dynamic type* of the object is responsible for either performing extraction itself or for finding an appropriate handler which is able to perform extraction. If neither is possible, an exception is raised and the algorithm is finished.
3. The result of subset extraction is an arbitrary object which is passed back to the requestor.

If the resulting subset is not a *Dinopolis Object*, no special treatment is necessary, because arbitrary objects are not under control of the special Dinopolis lookup-service.

It is possible to store subset descriptors by storing the *GUH* of the containing object with the additional subset request information. However, if the result of subset extraction is not a *Dinopolis Object*, no guarantees about robustness of this request information can be given.

If the resulting subset is a *Dinopolis Object* itself, the situation is different. In principle the same mechanism applies that is already known from treatment of immanent interrelations, discussed in section 8.3: as long as no *GUH* is requested from the subset, the object is treated as a temporary object and therefore not passed on to the lookup-service. Once the *GUH* is requested, the lookup-service obtains control over this object to ensure robustness. Due to this behaviour it is possible to store a *GUH* directly for the subset. This *GUH* is robust, as opposed to storing a *GUH* for the enclosing object together with non-robust subset information.

When implementing subset extractors the following aspects have to be kept in mind:

- It is good practice to return subsets as *Dinopolis Objects*, because then robust *GUHs* can be applied to them.
- Nevertheless, the decision whether or not to return *Dinopolis Objects* is not only a matter of personal taste and good practice. It also means that the subset extractor is responsible for ensuring robustness. Ensuring robustness means that the extractor is either internally able to keep track of changes that would affect the subset (e.g. content editing operations) or that it can notify the lookup-service appropriately as described for the *DOLSA* algorithm (see chapter 7) if changes apply. If the implemented subset extractor is not able to ensure consistency it must not return a *Dinopolis Object*!

Back to the main topic of this section – interrelations that point to subsets of *Dinopolis Objects* store the appropriate subset information to reflect this. If the result of the subset is not a *Dinopolis Object* they store the *GUH* together with the non-robust subset information. If the result is a *Dinopolis Object* they store both, the *GUH* of the main object and the *GUH* of the resulting subset. The reason for this decision has an administrative background: if for whatever reason the subset is no longer available (e.g. the paragraph that the interrelation pointed to was deleted in the content) it has to be possible at least to request the enclosing object. This is not possible with the *GUH* of the subset, rather the *GUH* for the enclosing object is needed. The two *GUHs* are independent from each other from the point of view of the lookup-service.

8.7 Object Deletion Aspects

Several interrelation attachment- and deletion-related aspects were already mentioned in section 8.5.1 and section 8.5.2, one more situation still deserves some consideration: what happens to interrelations, if an object is deleted that they are attached to?

As can easily be seen, the case of interrelations residing in a read-only storage is easy to get under control using the shadow storage principle. Therefore this case will not be dealt with in detail here. The more important problems are security related. The following situations can be identified:

- At least one interrelation that is attached to an object does not allow write access to its according endpoint information.
- At least one interrelation that is attached to an object requires that this attachment has to remain in existence, because otherwise the interrelation would become invalid according to its type. For example an interrelation could represent a 1 : 1 hyperlink that points to an object. If the object is deleted, the interrelation would lose one of its endpoints, which would make it senseless.

The case that an interrelation does not allow write access to its according endpoint information can in principle only result in the situation that this interrelation locks the object down and prohibits deletion. However, also a different behaviour

could be desired, namely that the endpoint cannot be detached explicitly, but the whole interrelation could be deleted completely if this situation occurs. It is up to the *dynamic type* of the interrelation, which exact behaviour it implements. The possibility that interrelations are used to lock objects down and prohibit deletion can be very useful. For example in the MTP scenario (see section 2.1) signed medical documents have to be protected from deletion of parts of them. However, much care has to be taken when defining access rights to interrelations to not provoke unwanted locking situations.

More or less the same strategy can also be applied to the second case above: either the interrelation prohibits object deletion by force and locks it down or it is itself deleted completely. In this case also a fallback solution can be implemented by the *dynamic type*: as long as the interrelation is invalid in respect to its semantic meaning, it is not resolvable according to its intended use, but it still exists. This means that resolve requests to obtain the other endpoint are not satisfied, whereas requests to obtain the interrelation itself are. With this behaviour to support temporary unavailable functionality it is possible to attach a different endpoint to such an interrelation to make it work again. Deleting it would mean that it is totally lost. For this reason this behaviour should usually be the implementation of choice. Especially when heavily restructuring the object space, such an implementation of the *dynamic type* can ease the job of the application that manages restructuring enormously. Actions like deleting one object and inserting some new objects with hyperlinks redirected to them is easy to implement if the interrelations are not per definition deleted when they become temporarily invalid. Otherwise much care would have to be taken to ensure the correct order of restructuring operations. Sometimes it would even be impossible to find a correct order, if several documents have cross-relations.

8.8 The *Dynamic Type* Mechanism

Although very powerful, the implementation of the *dynamic type* mechanism is rather straightforward. From OO programming languages the principles of subclassing, overloading, overriding and delegation are well known. The only problem that had to be solved was the possibility to attach desired behaviour during runtime.

If it would be possible to redirect operation calls to the correct runtime instance of an object *dynamic typing* is easy to implement. The desired mechanism is very similar to the mechanism of `virtual` methods in C++.

The first step to be able to change operation calls during runtime is a *Hook*-pattern-based implementation of operations (see also appendix A). If subclassing or object composition aspects dictate changes of the call structure of an object, the calls are simply re-hooked to the appropriate operation instances that implement the desired behaviour. This is exactly how *Dinopolis Objects* support *dynamic types* from a technical point of view. Due to the special implementation which uses hooks it does not matter whether *dynamic types* are attached during object creation or changed at any other point in time. In order not to be misunderstood: it does not matter *technically*. This does not mean that the types for objects can change in an uncontrolled way and at uncontrolled times, because this would cause severe inconsistencies.

To obtain a clean *dynamic type* mechanism *Dinopolis* supports the composition of dynamic classes that represent the type. Each class has a unique name and supports a set of operations. Each class can also be derived from arbitrary many superclasses. Overloading and overriding of operations are deliberately implemented in a way that declarations for both mechanisms have to be given explicitly. This means that a dynamic class requests explicitly that e.g. `operation A` should override `operation X` of its superclass. Preference was given to this behaviour rather than implementing the well-known “*operations with the same name and the same parameters override...*” principle, because this allows much more fine grained control. Dynamic classes are themselves *Dinopolis Objects* to be able to load them during runtime, either locally or across the net. The only speciality of these objects is that they always have the *dynamic type* `DynamicClass` and cannot be of any other dynamic type. Without this rule there would be the danger of deadlocks, endless recursions and ambiguities.

Subclassing hierarchies of dynamic classes are one part of the mechanism. The other part of it is the way, how these dynamic classes are attached to an object to define (part of) its *dynamic type*. Dynamic classes are *Prototypes* (see [Gamma et al. 1998]) and when they are attached to *Dinopolis Objects* an instance is requested. Depending on whether a dynamic class needs instance-specific data

it can create a new instance on request or just provide the same instance on every request. The instance is first asked for the special dynamic methods, as there are *constructors*, the *destructor* and the *deletor*. These special methods are hooked up by the *Object Management* module directly. The reason for direct hooks here is that during construction, destruction and deletion it cannot be guaranteed that the object is already (or still) “fully existing” and therefore it is not guaranteed that the interrelation mechanism is fully functional.

After construction of the primitive skeleton of an object the *Object Management* module attaches all *dynamic types* that are involved to the created object by using interrelations. During this process the rest of the dynamic methods that is not considered *special* is hooked up through interrelations. The reason for this is, because the special rules, *how* the subclassing mechanisms are applied are part of the kind of subclassing (e.g. *extension*, *narrowing down*, *delegation*, etc.). After this process is finished, the resulting final skeleton is fully configured and functional, because all method calls are hooked up according to the definition of the single *dynamic type* parts. Now final initialization through the appropriate *dynamic storage type* is triggered, resulting in the requested *Dinopolis Object*.

Changing parts of the *dynamic type* of an object during runtime is possible if it passes the appropriate security checks. In this case simple re-hooking of the type-specific parts is undertaken.

9. The Virtual System Management Module

One of the most useful concepts in Dinopolis that is not found in any other of today's systems is the possibility to define virtual systems by combining arbitrary embedded external systems to one virtual space transparently. The idea behind virtual systems is best explained with a small example. Therefore let us consider one of the typical scenarios from the MTP context (see also section 2.1).

A hospital operates a digital x-ray machine and the data that it produces is stored in a special purpose Dicom database. This Dicom database is able to handle some image related meta-information but the functionality to build several different classification structures for navigational purposes on top of the stored data is missing. A desired classification structure could for example be an organization of the data by clinical pictures. Further it would also be important to store some meta-information that is not directly related to images. Rather the meta-information has an administrative background like the ward, where an x-ray image was produced, the personnel which was involved when taking the image, etc.

The desired solution would be to operate the existing Dicom database side by side with e.g. a standard relational database that is able to store the desired common data. Whatever the Dicom database is not able to handle because of its special purpose design is stored in the standard database. From within Dinopolis the combination of the two is invisible, because the data is represented in a *Dinopolis Object* which encapsulates the storage structure transparently. In figure 9.1 a sketch of this scenario is shown.

The interrelations between the different *Dinopolis Objects* in figure 9.1 represent one navigational structure by classification. Each *Dinopolis Object* appears as if it were residing in one single system. This logical view is shown for the object in the middle of figure 9.1. The dashed arrows that have their origin in the *Dinopolis Object* on the left represent the internal object structure that is encapsulated: The

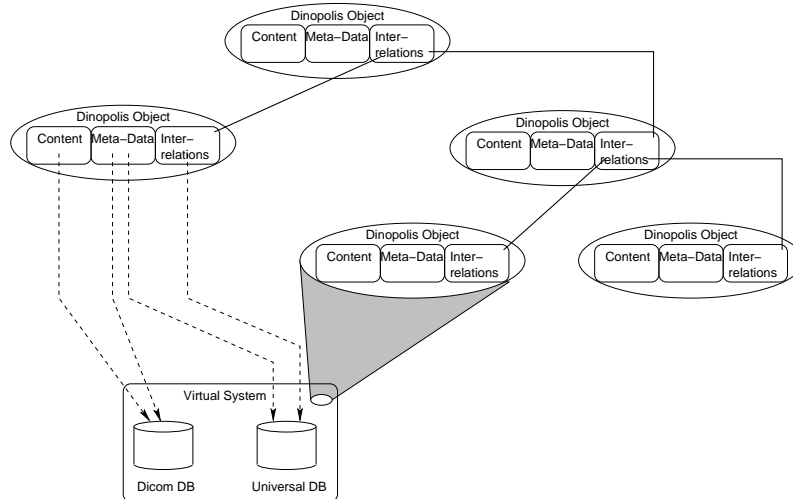


Figure 9.1: A possible virtual system scenario

content (i.e. the x-ray image) resides in the Dicom database. The image related part of the meta-data also originates in the Dicom database, whereas the administrative part of the meta-data is stored in the universal database. The interrelations that are used to build the logical navigational structure are completely stored in the universal database. From the applications' point of view the separation of the data is not recognizable. All that applications get to see are *Dinopolis Objects* with their well-known properties.

The task of the *Virtual System Management* module is to manage arbitrary combinations of embedded external systems. It is not responsible for the process embedding the external systems. This is the task of the *External System Management* module. Therefore only the combination aspects are considered in this chapter. Also degenerated combinations (i.e. only one external system alone) are treated as virtual systems. The reason is that reconfigurations of virtual systems are possible at any time. Therefore it can happen that a virtual system consisting of one single external part is later reconfigured by adding one or more other external systems. In this case it is essential that the view inside Dinopolis to this virtual system does not change.

Speaking of virtual systems some addressing aspects have to be taken into account, considering that the *GUH* mechanism applies to the *Dinopolis Object* that is seen from within the system:

- The *Virtual System Management* module already works with addresses rather than with *GUHs* (these are resolved by the *Address Management* module, see chapter 7). If desired, a private space in the lookup-service can be utilized via the *Address Management* module for internal reasons.
- Since a *Dinopolis Object* can have its persistent data distributed across several embedded external systems the *Address Management* module provides the support for *GUHs* that are resolved to address-maps (i.e. arrays of addresses that allow named access to the single parts) rather than only to single addresses. This feature can be utilized by the *Virtual System Management* module to identify the addressed virtual system and to load the different chunks of an object according to its configuration.
- The *Virtual System Management* module is responsible for ensuring consistency of the object space when a virtual system configuration is changed. For example if a virtual system initially only utilizes an embedded filesystem and is later re-configured by adding a database that stores the meta-data of objects, the *GUHs* of the according objects must not change. It has to be enough to update the according entry in the lookup service to reflect the changes internally. From the outside nothing has to be recognizable (with the exception of added functionality of objects).

By utilizing the addressing and *dynamic type* facilities of Dinopolis the behaviour of virtual system configurations is easily modellable:

- After *GUH* resolving an address map is passed on to the *Virtual System Management* module. From this map the part that addresses the specific virtual system is taken to identify it.
- The *Virtual System Management* module asks the identified virtual system for the *dynamic storage type* that it uses to manage persistent data.
- The *dynamic type* object knows how to interpret the remaining parts of the address map and is therefore able to initialize an object correctly.

Restructuring virtual systems means that the distribution of the single parts of *Dinopolis Objects* across embedded external systems changes. This means that the *dynamic storage type* as well as the address map that results from *GUH* resolving

changes. However, robustness of the system is guaranteed because *GUHs* themselves do not change at all.

For availability reasons it is strongly recommended that special implementations of virtual systems support temporary existence of parallel old and new configurations. The reason for this recommendation becomes clear when considering systems that store huge amounts of data. Internally changing the persistence of the data can therefore result in massive dataflow between external embedded systems. Supporting temporary parallel existence of two configurations would mean that it is not necessary to take the whole system offline until restructuring has been finished. With the exception of very special cases of configurations this behaviour is quite easy to achieve:

- When an object has been converted from its old to its new representation an update of the lookup-entry is requested anyway. Part of the address map that is stored in the lookup-service can hold information whether this is an old or a new entry by referring to the appropriate *dynamic storage type*.
- When an object is requested from the virtual system it can easily find out how it has to be created by analyzing the information of the address map. Therefore it can react by providing either the old or the new *dynamic storage type*.
- As soon as the whole internal reconfiguration process is finished the old *dynamic storage type* is obsolete.

Although the proposed mechanism could also be applied permanently, it is not recommended to misuse it this way. Operating two different configurations in parallel has to be done by providing two distinct virtual systems for semantic and also for consistency reasons.

10. The External System Management Module

The *External System Management* module is responsible for all tasks that have to do with the embedding of external systems. Embedders are instantiated through this module (after according security checks) and configurations of embedded systems are registered so that consistency of the distributed object space of Dinopolis can be ensured.

Embedding external systems for the first time is requested explicitly. Once an external system has been embedded the *External System Management* module registers this system internally (utilizing the lookup-service), so that it is possible to re-embed it properly at a later stage if desired. This behaviour is necessary because system shutdowns must not have an influence on the object space. If it is desired that a system is not embedded any longer, an explicit “unembed” request removes it from the configuration. The *External System Management* module is responsible for performing all tasks that are necessary to ensure consistency of the object space after an unembed operation.

As has already been mentioned in chapter 4, embedders are logically residing in the kernel space of Dinopolis. However, external systems are not considered trusted in respect to the strong security requirements of Dinopolis and this also applies for embedders. For this reason embedders reside in a special, encapsulated kernel space of their own which does not allow unchecked access to the rest of the kernel. The *External System Management* module is responsible for the administration of this encapsulated kernel space.

According to the philosophy of Dinopolis, arbitrary external systems can be embedded. Embedders are comparable to drivers for external devices in operating systems and they can therefore be written from arbitrary developers that want to provide support for their special systems. The embedder developers are responsible

for registering their custom embedders with a *Factory* (see [Gamma et al. 1998]) from which they are loaded by the *External System Management* module.

When embedding an external system is requested explicitly, meaning that the external system is embedded for the first time, the *External System Management* module has to perform the following tasks:

1. The appropriate embedder is requested from the *Factory* and the *Security* module is asked to check its integrity.
2. If the embedder passes the security check a request to connect to the external system with the configuration data provided by the request is sent to it.
3. After having connected to the external system, the embedder is registered with the *Kernel Access* module and obtains a special embedder execution context. This is necessary because embedders are not considered trusted and the special execution context protects the system.
4. The *External System Management* module finally registers the embedder with the lookup-service to be able to re-instantiate it on request at a later stage (see below). The behaviour of re-instantiating embedders on demand is comparable to *automounting* in Unix.

Depending on the external system and the embedder, some special tasks can be necessary that require execution of arbitrary protocols. For example an external system could require user identification. To be able to provide arbitrary functionality that could be needed, the *External System Management* module does not only register internal embedding information but also registers a *GUH* for the embedder to allow applications to request a *Dinopolis Object* which encapsulates it. This allows that embedders can pass on arbitrary supported functionality to *Dinopolis*. Nevertheless, it is not possible to open trapdoors because of the strong *Proxy*-based security mechanism (see chapter 5) that applies to *Dinopolis Objects*. The *External System Access* module is responsible for interpreting the “physical” address of the embedder object correctly (see also chapter 9 for a description of address maps).

If, for whatever reason (e.g. system shutdown), an external system is temporarily not embedded, but a request for data residing on it is sent, the *External System Management* module is responsible for re-embedding it to satisfy the request. The process of re-embedding is straightforward for the *External System Management* module:

1. The lookup-service is contacted to obtain the configuration data.
2. The tasks for embedding are performed as described above.

After having re-embedded the external system, the request can be satisfied. The process of re-embedding is fully transparent and therefore not noticeable for requestors.

If an embedded system has to be unembedded explicitly for whatever reason, the *External System Management* module has to perform the following tasks:

1. It has to take care that all *GUHs* that it registered explicitly for the embedder as well as for special data that resides in the embedded system are unregistered. Please note that the term *registered explicitly for the embedder* refers to all *GUHs* that were not registered through the *Virtual System Management* module. For those *GUHs* the *Virtual System Management* module itself is responsible.

In order not to cause inconsistencies, the process of unregistering *GUHs* is the same as the process of deleting objects (see section 6.4 and chapter 8), with the exception that the persistent data is not explicitly deleted from the external system (it is not really desirable to erase a harddisk when unembedding a filesystem!).

2. After unregistering the *GUHs* the instance of the embedder is removed from the system in a way so that it will never be re-embedded again automatically.

Please note that unembedding of external systems is always triggered by the *Virtual System Management* module, because embedded systems are always part of a virtual system configuration. This is the reason, why it was stressed that embedders only have to unregister their own *GUHs* and the rest is taken over by the *Virtual System Management* module. Unembedding can be requested for two different reasons: reconfiguration of virtual systems or complete elimination of them. Therefore the *Virtual System Management* module has to have the chance to act accordingly.

10.1 Embedders

Because the concept in Dinopolis is that arbitrary external systems can easily be embedded and combined, the requirements that apply to embedders have to be easy

to meet. For this reason only the minimal functionality that the system needs (load data) to work with embedders must be implemented, additional functionality (save data, create object, move / copy data, access-rights related methods, etc.) can be provided via *Hooks*. Functionality of external systems that is not even part of the kernel's core calls can be provided via *operations* and *services*. This means that every embedder can provide the full functionality that the external system supports, but no embedder is forced to emulate functionality that is unsupported.

In the minimal case an embedder only supports a request to load data when given an address. The address itself is not dictated by Dinopolis. Rather it is the native address supported by the external system. All address handling and mapping issues that apply within Dinopolis are managed by the *Address Management* module (see chapter 7). External systems that are read-only (e.g. an x-ray device) are supported, because not even the store operation is mandatory.

The usual case for embedders will be that certain special functionality is inherent to the external systems and therefore it makes sense to map it to embedder-inherent functionality. For example let us consider an embedder for a Unix filesystem. Besides many other things the filesystem supports loading and saving data as well as a move operation. Further also access-checks as well as functionality to change access rights are supported. Certain meta-data like the size of files is also a native part of a filesystem. Writing an embedder that maps this functionality to the internal data and control-flow structures of Dinopolis would mean that this embedder supports the following:

- The minimal required standard functionality to provide access to a file when requested using a standard file-path.
- *Hooks* for *save*, *move* and *create* operations. Again the addressing mechanism for these operations is a standard file-path.
- A *Hook* for access checks that can be utilized by the *Security* module when dealing with data originating in the embedded filesystem.
- *Hooks* and *Operations* to work with meta-data (e.g. provide read-only access to the filesize).
- *Operations* to change access rights.
- *Services* to provide GUI elements for the access-right operations.

The embedder has to register all its special functionality (*Hooks*, *Operations* and *Services*) when the request from the *External System Management* module arrives that it has to connect to the external system.

11. Dinopolis Network Distribution Management

The *Network Management* module is the responsible part when dealing with remote *Dinopolis Objects*. As has been stated in chapter 4 the *Network Management* module consists in principle of two logically separated parts:

1. The outgoing part that logically resides in the kernel space. This part is responsible for sending requests.
2. The incoming part that for security reasons logically resides in the user space. This part receives requests and triggers their execution.

In section 6.3 the principles of dealing with remote objects via *Remote Proxies* were already described. In figure 11.1 the involved parties are shown.

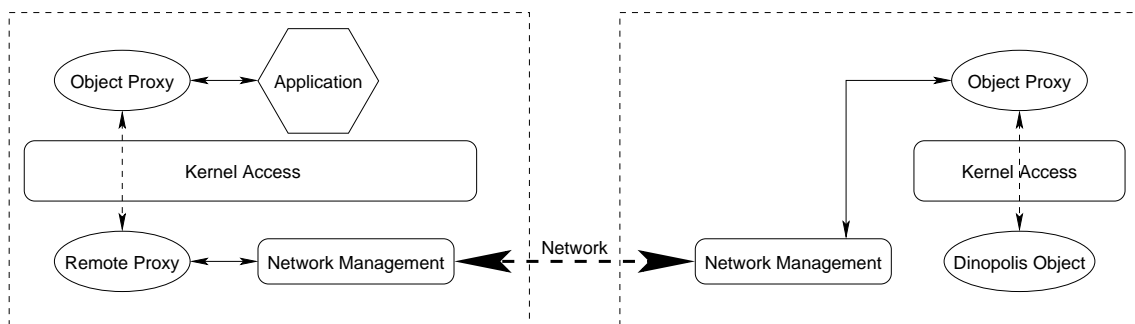


Figure 11.1: The principle of *Remote Proxies* in Dinopolis

The two dashed boxes in figure 11.1 represent single Dinopolis instances that reside on different machines. The dashed arrows stand for security-checked calls, the solid arrows show direct requests. An application, as shown in the left Dinopolis instance, does not recognize the difference between a remote and a local *Dinopolis Object*. In both cases it deals with an *Object Proxy* as described in chapter 5. In case of a remote object the ways of a request and of the according response are the following:

1. The application calls a method on the *Object Proxy*.
2. The object proxy propagates the call through the *Kernel Access* module, where a security check is triggered.
3. If the call passes the security check it is propagated to the *Remote Proxy*.
4. The *Remote Proxy* contacts the *Network Management* module and requests to send the call to the remote instance. This means that on the side of the requestor the outgoing part of the *Network Management* module is concerned with the request.
5. In the remote Dinopolis instance the incoming part of the *Network Management* module receives the request.
6. According to the identifier that is sent together with the request, the network management module is able to locate the *Object Proxy* of the remote object.
7. The desired method of the *Object Proxy* is called.
8. The object proxy propagates the call through the *Kernel Access* module, where a security check is triggered.
9. If the call passes the security check it is propagated to the *Dinopolis Object* and executed there.
10. The result of the call is propagated back through the *Kernel Access* module, security checked and parts of it are sorted out if necessary (see chapter 5).
11. The (part of the) result that passed the security check is handed over to the *Network Management* module. Please note that still its incoming part is involved on the remote side, because the terms *incoming* and *outgoing* are related to the way of a request rather than to network send and receive operations.
12. When the result arrives in the instance of the requestor it is propagated to the *Remote Proxy* that triggered the network action.
13. The *Remote Proxy* returns the result to the *Object Proxy* through the *Kernel Access* module, where it is security checked and sorted out accordingly.
14. Finally the *Object Proxy* returns the result to the application.

As can be seen, remote calls are security checked twice, once on the requestor's side and once in the remote instance. This is necessary, because the security policy could involve the rule that only certain users may access remote objects at all. It can also be defined that only certain remote objects may be accessed or that only certain methods may be called on them. For obvious reasons the remote instance

also performs security checks as a protection against intruders and also because its security policy must not be changed through remote calls.

The *Network Management* module is designed in a way that it does not require the implementation of a special low-level protocol. Rather it can utilize arbitrary existing protocols like e.g. RMI as a transport layer. The high-level logic of the *Network Management* module does not even require that either a connection-oriented or a connectionless transport protocol is used. Very important for scalability reasons is that the *Network Management* module performs call multiplexing internally. This means that not more than one connection between Dinopolis instances is *required*, no matter how many remote objects are involved (if absolutely desired, e.g. for performance reasons, more than one connection is *allowed*). Besides multiplexing, the *Network Management* module also performs request-response decoupling. This means that it does not happen that the connection is blocked for other calls in the time between sending a request and receiving the response. This behaviour is especially important for multithreaded applications that work with remote objects. Blocking calls would represent a severe performance bottleneck in this case.

Two different kinds of requests are supported by the *Network Management* module: *synchronous* and *asynchronous* requests. The call logic of a synchronous request is exactly the one described above. The call logic of an asynchronous request is to send an event to the remote object and return immediately without waiting for correct event delivery.

The asynchronous requests were introduced for scalability reasons: it can happen that many remote proxies in many different Dinopolis instances exist somewhere on the network. *Dinopolis Objects* are observable and therefore also remote observing is supported. Depending on the objects' internal logic and depending on the actual observers it can be decided that it is not important at all, whether a remote notification arrives or not. In this case asynchronous requests are used to deliver the notifications because this protects the *Dinopolis Object* from network related delays.

To understand the observer-related performance issues and to get a clear picture of all relationships between the different parties that are involved in a standard distributed case, a typical situation is sketched in figure 11.2.

The solid lines in figure 11.2 represent direct calls, the thin dashed lines represent security-checked calls and the thick dashed lines represent logical network

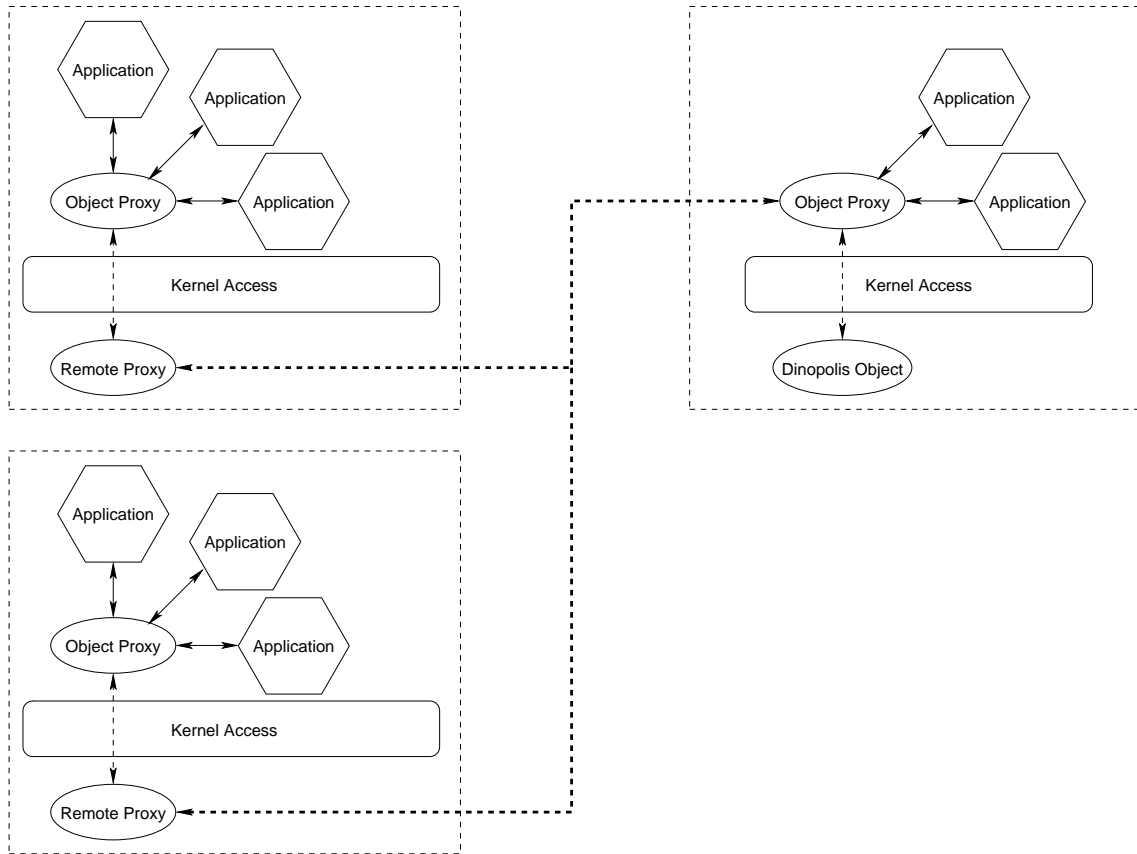


Figure 11.2: Involved parties in a standard distributed situation

connections. As can be seen, only one instance of each *Dinopolis Object* exists (see chapter 5). The *Object Proxy* in the instance where the *Dinopolis Object* resides is also a single instance in respect to the *Dinopolis Object* which it encapsulates. Several local applications can deal with the local *Object Proxy*. Applications in other Dinopolis instances deal with *Object Proxies* that encapsulate *Remote Proxies*. Also the *Remote Proxy* is a single instance in respect to each Dinopolis instance. Each application is able to observe an object, no matter if it is a local or a remote object. Notifications triggered by a *Dinopolis Object* are propagated as follows:

1. The *Dinopolis Object* sends the notification to the local *Object Proxy*.
2. The local *Object Proxy* acts as the first multiplier in the propagation path: it sends the notification to all local applications that are registered as observers. Additionally it sends the notification to all *Remote Proxies* (via the *Network Management* module).

3. Each *Remote Proxy* sends the notification to its according *Object Proxy*.
4. Each of the *Object Proxies* acts as a second multiplier in the propagation path: the notification is sent to all local applications that are registered as observers.

All object-related requests (e.g. loading, saving, etc.) are treated by the *Network Management* module such that they are just sent to the remote instance for execution. The incoming part of the *Network Management* module on the other side de-facto acts like a standard application which is local to that instance. For this reason it does not make much sense to describe all those processes in detail, because they are the same ones as these already discussed in chapter 6. The only difference is that the calls are sent across the network first and therefore decoupled from the local Dinopolis instance.

12. Dinopolis Kernel Module Management

One of the principles of Dinopolis is that the base system is slim and special functionality is added via so-called kernel-modules. This means that functionality which is not utilized does not require resources. For example it is not necessary that transaction management is supported in an instance where only applications run that do not need transactions at all. The *Kernel Module Management* module is responsible for loading and unloading such kernel modules as well as for providing a special space in the kernel together with appropriate kernel access entry points for modules.

The term *kernel access entry points* above stands for a collection of requests to and from the kernel. All those requests in both directions are implemented as *Hooks* or *Hook-Ups* respectively which kernel modules can utilize. To be able to pass provided functionality on to the user space, kernel modules can publish appropriate *functional modules* (see chapter 5). The module-environment related functionality of the *Kernel Module Management* module is therefore the following:

- Provide a list of hooks that the kernel as well as other kernel modules provide. Via these hooks every module that is loaded is able to communicate with the kernel environment and to request actions that are necessary for its tasks.
- Support requests through which loaded kernel modules are able to publish the hooks that they provide for other modules.
- Provide requests through which kernel modules can publish their *functional modules*.
- Support automatic hook connection facilities. This means that e.g. a loaded module could support transactions, if a transaction module exists, but can also work properly without them. Therefore the module registers transaction related *Hooks* and *Hook-Ups*. As soon as a transaction module is loaded, these *Hooks* and *Hook-Ups* are connected automatically.

All processes that deal with loading kernel modules, interconnecting them with the rest of the kernel and publishing *Hooks*, *Hook-Ups* and *functional modules* are security checked appropriately. This means that the *Security Module* itself is for sure the first module that has to be loaded.

Loading of kernel modules can require a certain order because of module dependencies. For this reason the *Kernel Module Management* module provides facilities to define the load order. Two mechanisms apply that influence the load order:

1. Automatic ordering by analyzing published dependencies for each module.
2. Specific ordering and special bootstrapping in case of circular dependencies or in case of explicit demand of the modules.

Automatic ordering due to dependencies is straightforward as long as no circular and therefore contradictory dependencies exist. Specific ordering is provided via special *Observers* and appropriate *load hooks* which allow the modules to influence the bootstrapping process. If e.g. circular dependencies exist the dependent modules can define a certain load order. For example let us consider the case of two modules that are dependent on each other. In this case the following steps are performed:

1. According to the load order that is requested by the modules the first of the two is loaded.
2. After having been loaded all hook-up and publication steps of the module which are not related to the second module (from which it is dependent) can take place.
3. When the common steps are finished, the loaded module registers with the appropriate observable parts of the *Kernel Module Management* module. Therefore it is able to trace the process of loading and can wait until the second module is loaded to finish initialization.
4. If the second module is loaded all its connection steps are performed and the *Observers* of the first module are informed accordingly that initialization can be finished.
5. On notification the first module takes all remaining steps to finish initialization by connecting to the second module.

It can easily be seen that arbitrary complex bootstrapping patterns can be performed due to the use of *Observers* that allow to trace the load mechanism. Therefore arbitrary dependencies can be resolved properly. Loaded modules that definitely

need others to be able to fulfill their tasks can also explicitly request loading of other modules if they are not registered in the default configuration of the system. After the whole bootstrapping process has been finished all modules are informed accordingly. If dependencies are not satisfied at this point for any reason (e.g. the *Security* module refused one of the kernel modules, but one of the loaded modules is dependent on its existence), the appropriate exceptions are thrown.

It is not only possible to load modules during the bootstrap and initialization process. Kernel modules can be loaded on demand during runtime. The process then is the following (omitting explicitly mentioning the single security checks, because they always apply anyway):

1. All existing modules are notified that a new module is going to be loaded.
2. The requested module is loaded and initialized.
3. All other existing modules are informed accordingly that the new module is loaded. This enables them to adjust their own *Hooks*, *Hook-Ups* and published parts accordingly if this is desired.
4. After the whole initialization process has been finished all modules are informed accordingly.
5. If any of the modules reports a problem with the changed configuration that is not solvable, the system is reverted to the old configuration that existed before the load request.

13. The Security Kernel Module

Depending on different application scenarios arbitrary security policies are thinkable. For this reason the decision was to implement security features in form of a kernel module which is loaded during the system bootstrapping process. In order not to design a potential security leak that allows trojan horses to be loaded as *Security* kernel modules a special algorithm applies:

1. A system-immanent, hardcoded, minimal security module exists that has the only functionality to check a *Security* kernel module candidate.
2. The *Security* module is the first kernel module to be loaded.
3. The system-immanent *Security* module checks whether the desired *Security* module is acceptable.
4. On acceptance the minimal module is replaced by the “real” *Security* module and the bootstrapping process goes on.
5. If the desired *Security* module is not accepted the bootstrapping process is stopped completely in order not to bring up an insecure Dinopolis instance.

It is not possible to run a Dinopolis instance based on the system-immanent, minimal *Security* module. If no “real” *Security* module is configured, bootstrapping is stopped. However, this does not mean that every Dinopolis instance must operate a highly sophisticated security policy. It is also possible to load an empty *Security* module that allows everything. The reason for this design decision is to prevent an unintentional start of an insecure Dinopolis instance as a result of having forgotten to configure the *Security* policy. If an empty *Security* module is explicitly declared acceptable for the system-immanent *Security* module and also installed explicitly it is considered intentional that no security policy applies and therefore it is allowed.

The tasks that the *Security* kernel module has to perform are the following:

- On request it has to check kernel modules and *functional modules* for acceptability. It is not defined by the Dinopolis design, *how exactly* these checks are performed.

Usually such checks will be based on the use of electronic signatures, checksums and certificates.

- On request it has to create and register (or also unregister) internal information which is needed in connection with the execution context. This internal execution context information is based on its policy in respect to users, groups, roles and other security-related issues. No special requirements apply, because it is up to the implementation of the *Security* module, which policy it supports.
- On request it has to check request execution on a call basis. Each request that is propagated through the *Kernel Access* module has to pass a security check that is dependent on the applications' execution context. The request check also includes a check of the parameters sent with the request.
- To be able to perform checks on a call basis at all, possible calls are registered with the *Security* module during bootstrapping and on system configuration changes.
- On request it has to check the return values that are the result of executed calls and sort them out properly.
- On request it has to perform encryption and electronic signature related tasks. Because there exists a wide variety of different methodologies for these tasks which all have different applications as well as different requirements (e.g. use of smart-cards, etc.), only common check hooks are provided by the *Kernel Access* module. All special functionality has to be passed on to the system by registering the appropriate *functional modules*.
- Also a request to exchange the *Security* module during runtime is supported. In this case the current *Security* module has to perform the appropriate checks and if these are passed it is exchanged with the new *Security* module.

As can be seen, the design of the *Security* module concept in Dinopolis is made as general as possible to make the implementation of arbitrary security concepts easy. For example, the MTP *Security* module (see section 2.1) implements a very sophisticated user and group as well as time-based role management concept. Electronic signature and encryption facilities are implemented on the basis of a general smartcard concept that allows the use of different kinds of crypto and non-crypto smartcards. However, an exact description of the MTP security concept would be way beyond the scope of this thesis.

14. Conclusion

The object-oriented software development paradigm has been the paradigm of choice for a long time. Properly applied (and only then!) it allows easy mastering of huge software packages by proper structuring and encapsulation. A well designed distributed component system provides all the benefits of OO programming languages and goes some steps further:

- Modular composition and code reuse are supported during runtime rather than only on a programming language level.
- The use of components is not limited to passive data. In fact everything can be a component, even existing applications which can seamlessly be integrated into the internal component model.
- Network aspects are transparently integrated into the system rather than putting the burden of distributed systems on the application developers.

During the whole design phase of Dinopolis the motto was: *“a good distributed component system is one that the application developers don’t notice”*. Using components that are distributed across the network has to be as easy as using standard local objects in OO programming languages. For the developers of components as well as for the developers that utilize them, all runtime-composition aspects and all distribution aspects have to be fully transparent.

Dinopolis implements a highly sophisticated data model which completely separates addressing from navigation, globally unique and robust handles from physical addresses as well as content-related aspects from component-related aspects. Objects in OO programming languages rely on the use of *static classes*. Consequently components in a distributed component system rely on *dynamic types*. This makes it possible to utilize all OO modelling aspects with distributed components as well.

Dinopolis allows to integrate arbitrary existing systems due to the concept of embedding. Embedding makes it possible to combine different existing systems so

that each of them adds value to the others. Components in this scenario need not have their origins in one embedded system, they can also be composed of parts that reside separated from each other in different systems. Internally the developers will never notice the difference.

Robustness, security, reliability and scalability are the most crucial aspects when developing distributed applications. The most critical requirements in this respect came from the MTP project, because medical data is extremely sensitive information and because distribution aspects play the most important role there. Dinopolis as the basis for the MTP system is potentially designed to be able to manage life-long patient records for every human being on earth. What this means in terms of distribution, number of existing systems, structure of the object space as well as reliability and security is easily imaginable. Not to forget that many data-management solutions have already been in use in the medical area for a long time which must not be discarded but embedded, otherwise MTP would never be accepted. For these reasons the design phase for Dinopolis was extremely long, resulting in a very detailed architecture with thorough documentation. Much effort has gone into uncountable refinement and partial redesign steps as well as into the development of new algorithms like e.g. *DOLSA*, just to mention one of them.

As always in the area of software development new needs arise as soon as one version of a system is available. To be able to react to new demands properly, the design of the Dinopolis kernel is extremely modular to support adding of new functionality as well as exchanging existing functionality on demand.

The result of the long design phase is that Dinopolis is the first second-generation distributed component system according to the definitions given in [Schmaranz 2002a]. In fact it is even more than that: Dinopolis implements a completely new and clean component development paradigm which cannot be found in any of today's systems.

A. The Hook Design Pattern

The *Hook* design pattern is in fact a very simple structural pattern which allows to decouple the knowledge of *when* an action is triggered from the knowledge of *how* to perform the action. It is only mentioned here, because it cannot be found in literature in this form. Dinopolis relies heavily on the use of *Hooks* for modularity and configurability reasons.

There are uncountable examples to motivate the use of the *Hook* pattern. For explanatory purposes let us consider a modular approach to design the menubar in an application with a graphical user interface:

Users clicking on a menupoint, e.g. *Print* trigger an action. Inside the menubar module it is known that printing is desired but it is not known, how to do it.

Depending on the printer, arbitrarily many different functional modules (=drivers) exist that know how to send data to it. It depends on the special printer, which of the drivers is used for printing.

The *Hook* design pattern now defines the following solution that allows a combination of the two participants that are involved in the action:

- The participant that knows *how* to perform an action (in the above example the printer driver) represents a *Hook* by implementing an abstract `Hook` interface. This interface defines an `action` trigger.
- The participant that knows *when* to perform an action (in the above example the menupoint) represents a *Hook-Up* by implementing an abstract `Hookup` interface. This interface defines a `setHook` method which allows to connect the *Hook-Up* with the appropriate *Hook* during runtime.
- The third participant is not really an integral part of the pattern, but for reasons of completeness it is mentioned here: there has to be a module that knows about the desired connections of *Hook-Ups* with *Hooks*. In case of the example above, some module has to know which printer shall be used and that the menupoint

Print is the trigger for the action. Let us call this participant the *Connector*. During bootstrapping or at any other point during runtime the *Connector* sets up the appropriate interconnections by calling `setHook` on the appropriate *Hook-Up* with the desired *Hook*.

As can easily be seen the *Hook* pattern completely decouples the involved parties and it also allows arbitrary reconfigurations of the behaviour during runtime. Just the connections between *Hook-Ups* and *Hooks* have to be updated to change the behaviour.

At a first glance *Hooks* look very similar to the *Observer* design pattern (see [Gamma et al. 1998]). However, there are very important semantical and behavioural differences between the two:

- *Observables* do not require that *Observers* are attached to them at all in order to work properly. *Hook-Ups* need at least one *Hook*, because this is the functional participant.
- *Observers* get notifications that allow to react on changes. *Hooks* have to perform actions. This means that semantically *Observers* are sort of event listeners, while *Hooks* are methods.

References

- [Andrews et al 1995] Andrews K., Kappe F., Maurer H., Schmaranz K.: *On Second Generation Hypermedia Systems*, Proceedings ED-MEDIA 95, Graz (1995), 75–80.
- [Aly et al 1998] Aly F., Bethke K., Bartels E., Novotny J., Padeken D., Schmaranz K., Schwartmann D., Wilke D., Wirtz M.: *Medical Intranets for Telemedicine Services: Concepts and Solutions*, Proceedings G7 Meeting “The Impact of Telemedicine on Health Care Management”, Regensburg (1998), available online at <http://www.uni-regensburg.de/Fakultaeten/Medizin/Uch/g7/program/mon.htm>.
- [Baggio et al. 2000] Baggio A., Ballintijn G., van Steen M.: *Mechanisms for Effective Caching in the Globe Location Service*, Proceedings ACM SIGOPS, Kolding, Denmark (2000), 55–60.
- [Baggio et al. 2001] Baggio A., Ballintijn B., van Steen M., Tanenbaum A. S.: *Efficient Tracking of Mobile Objects in Globe*, The Computer Journal, Vol. 44/5 (2001), 340–353.
- [Bakker 1998a] Bakker A., van Steen M., Tanenbaum A. S.: *Replicated Invocation in Wide-Area Systems*, Proceedings ACM SIGOPS, Sintra, Portugal (1998), available online at <http://www.cs.vu.nl/pub/papers/globe/sigops.98.pdf>.
- [Bakker and van Steen 1998b] Bakker A., van Steen M.: *Replicated Invocation in Wide-Area Systems: A Possible Solution*, Proceedings of the Fourth Annual ASCI Conf., Lommel, Belgium (1998), available online at <http://www.cs.vu.nl/pub/papers/globe/asci.98.1.pdf>.
- [Bakker et al. 1999] Bakker A., van Steen M., Tanenbaum A. S.: *From Remote Objects to Physically Distributed Objects*, Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems, Cape Town, South Africa (1999), 47–52.
- [Ballintijn et al. 2000] Ballintijn G., van Steen M., Tanenbaum A. S.: *Scalable Naming in Global Middleware.*, Proceedings PDCS-2000, Las Vegas (2000), 624–631.
- [Ballintijn et al. 2001] Ballintijn G., van Steen M., Tanenbaum A. S.: *Scalable User-Friendly Resource Names*, IEEE Internet Computing, Vol. 5/5 (2001) 20–27.
- [Berners-Lee et al 1994] Berners-Lee T., Masinger L., McCahill M.: *RFC 1738: Uniform Resource Locators (URL)*, available online at <ftp://ftp.internic.net/rfc/rfc1738.txt>
- [Dallermassl et al 2000] Dallermassl C., Haub H., Maurer H., Schmaranz K., Zambelli P.: *Dinopolis - A Leading Edge Application Framework for the Internet and Intranets*, Proceedings WebNet 2000, San Antonio, TX (2000), 111–116.
- [Dallermassl et al 2000b] Dallermassl C. Haub H., Krottmaier H., Schmaranz K., Zambelli P.: *Using Highly Sophisticated Middleware for Building Arbitrarily Distributed Teaching Environments*, Proceedings ICCE/ICCAI 2000: Learning Societies In The New Millennium: Care ativity, Caring & Commitments, Taipei (2000), 1439–1442.
- [Dallermassl et al 2000c] Dallermassl C. Haub H., Krottmaier H., Schmaranz K., Zambelli P.: *Adaptive Learning Environments*, Proceedings ICCE/ICCAI 2000: Learning Societies In The New Millennium: Care ativity, Caring & Commitments, Taipei (2000), 1443–1446.
- [Dicom] *The Dicom Home Page*, electronically available at <http://medical.nema.org>.
- [EJB] *Enterprise Java Beans Technology*, electronically available at <http://java.sun.com/products/ejb>.

- [Freismuth et al 1997] Freismuth D., Helic D., Meszaros G., Schmaranz K., Zwantschko B.: *DINO - Distributed Interactive Network Objects - The Java Approach*, Proceedings Ed-Media '97, Calgary (1997), available online at http://www.iicm.edu/liberation/iicm_papers/edmed97/dino.html.
- [Homburg et al. 1995] Homburg P., van Doorn L., van Steen M., Tanenbaum A. S., de Jonge W.: *An Object Model for Flexible Distributed Systems*, Proceedings of the First Annual ASCI Conference, Heijen, Netherlands (1995), 69–78.
- [Homburg et al. 1996a] Homburg P., van Steen M., Tanenbaum A. S.: *Communication in GLOBE: An Object-Based Worldwide Operating System*, Proceedings of the Fifth International Workshop on Object Orientation in Operating Systems, Seattle, WA (1996), 43–47.
- [Homburg et al. 1996b] Homburg P., van Steen M., Tanenbaum A. S.: *An Architecture for a Wide Area Distributed System*, Proceedings ACM SIGOPS, Connemara, Ireland (1996) 75–82.
- [Homburg et al. 1996c] Homburg P., van Steen M., Tanenbaum A. S.: *Distributed Shared Objects as a Communication Paradigm*: Proceedings of the Second Annual ASCI Conference, Lommel, Belgium (1996), 132–137.
- [Kuz et al. 1998] Kuz I., Kermarrec A. M., van Steen M., Sips H.J.: *Replicated Web Objects: Design and Implementation*, Proceedings of the Fourth Annual ASCI Conf., Lommel, Belgium (1998), available online at <http://www.cs.vu.nl/pub/papers/globe/asci.98.2.pdf>.
- [Kuz et al. 2000] Kuz I., Verkaik P., van Steen M., Sips H. J.: *A Distributed-Object Infrastructure for Corporate Websites* Proceedings DOA'2000, Antwerp (2000), 165–176.
- [Kuz et al. 2002] Kuz I., van Steen M., Sips H. J.: *The Globe Infrastructure Directory Service*, Computer Communications Vol. 25/9 (2002), 835–845.
- [Leiwo et al. 1999] Leiwo J., Haenle C., Homburg P., Gamage C., Tanenbaum A. S.: *A Security design for a wide-area distributed system*, Proceedings ICISC'99, Seoul, South Korea (1999), in LNCS 1787, 236–256.
- [Mockapetris and Dunlap 1988] Mockapetris P., Dunlap K. J.: *Development of the domain name system*, Proceedings ACM SIGCOMM 1988, Stanford, CA (1988), 123–133.
- [Gamma et al. 1998] Gamma E., Helm R., Jonson R., Vlissides J.: *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison Wesley (1998).
- [OMG] *The Object Management Group's Home page*, electronically available at <http://www.omg.org>.
- [RMI] *Java Remote Method Invocation*, available online at <http://java.sun.com/j2se/1.4/docs/guide/rmi>.
- [Schmaranz 1996] Schmaranz K.: *Professional Electronic Publishing in Hyper-G - The Next Generation Publishing Solution on the Web*, Schmaranz K., Proceedings WebNet '96 and J.UCS Vol.2, No.9, 650–658 (1996).
- [Schmaranz 2002a] Schmaranz K.: *On Second Generation Distributed Component Systems*, J.UCS Vol.8, No.1, 97–116 (2002).
- [Schmaranz 2002b] Schmaranz K.: *DOLSA - A Robust Algorithm for Massively Distributed, Dynamic Object-Lookup Services*, submitted to J.UCS (2002).
- [Schmaranz 2002c] Schmaranz K.: *Robust Interrelation Management in Massively Distributed Component Systems*, submitted to J.UCS (2002).
- [Shen et al. 2000] Shen E., Majumdar S., Abdul-Fatah, I.: *High Performance Adaptive Middleware for CORBA-Based Systems*, Proceedings ACM PODC'00, Portland, Oregon (2000), 199–207.
- [Stoica et al. 2001] Stoica I., Morris R., Karger D., Kaashoek F. M., Balakrishnan H.: *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*, ACM SIGCOMM'01, San Diego, CA (2001).
- [Terry 1984] Terry D. B.: *An analysis of naming conventions for distributed computer systems*, Proceedings ACM SIGCOMM 1984, Montreal (1984), 218–224.
- [Tzagarakis et al. 2000] Tzagarakis M., Karousos N., Christodoulakis D.: *Naming as a Fundamental Concept of Open Hypermedia Systems*, Proceedings Hypertext 2000, San Antonio, TX (2000), 103–112

- [van Steen et al. 1995] van Steen M., Homburg P., van Doorn L., Tanenbaum A. S., de Jonge W.: *Towards Object-based Wide Area Distributed Systems*, Proceedings of the Fourth Int'l Workshop on Object Orientation in Operating Systems, IEEE, Lund, Sweden (1995), 224–227.
- [van Steen et al. 1996a] van Steen M., Hauck F. J., Tanenbaum A. S.: *A Model for Worldwide Tracking of Distributed Objects*, Proceedings TINA '96 Conference, Heidelberg, Germany (1996), 203–212.
- [van Steen et al. 1996b] van Steen M., Hauck F. J., Tanenbaum A. S.: *A Scalable Location Service for Distributed Objects*: Proceedings of the Second Annual ASCI Conference, Lommel, Belgium (1996), 180–185.
- [van Steen et al. 1998a] van Steen M., Hauck F.J., Ballintijn G., Tanenbaum A. S.: *Algorithmic Design of the Globe Wide-Area Location Service*, The Computer Journal 41/5 (1998), 297–310.
- [van Steen et al. 1998b] van Steen M., Tanenbaum A. S., Kuz I., Sips H. J.: *A Scalable Middleware Solution for Advanced Wide-Area Web Services*, Proceedings Middleware '98, The Lake District, UK (1998), available online at <http://www.cs.vu.nl/pub/papers/globe/middleware.98.pdf>.
- [van Steen and Ballintijn 2002] van Steen M., Ballintijn G.: *Achieving Scalability in Hierarchical Location Services*, Proceedings CompSac'2002, Oxford UK (2002), available online at <http://www.cs.vu.nl/pub/papers/globe/compsac.02.pdf>.
- [Voyager] *ObjectSpace's Home page*, available online at <http://www.objectspace.com>.
- [XLink] W3C: *XML Linking Language (XLink) Version 1.0*, online at <http://www.w3.org/TR/xlink>
- [Znati, Molka 1992] Znati T. B., Molka J.: *A Simulation Based Analysis of Naming Schemes for Distributed Systems*, Proceedings of the 25th annual Symposium on Simulation 1992, Orlando, FL (1992), 42–51.