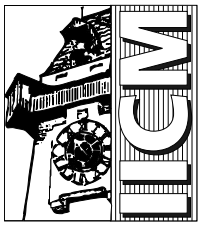
Graz University of Technology

Institute for Information Processing
and Computer Supported New Media



Diplomarbeit aus Telematik
Informationssysteme

Aspects of Integration of Heterogenous Server Systems in Intranets - the Java Approach

Christof Dallermassl

Betreuer
DI. Dr. techn. Klaus Schmaranz
Begutachter
Univ. Prof. Dr. phil. Dr. h. c. Hermann Maurer

Graz, November 1999

Acknowledgements

I would like to thank Prof. Hermann Maurer and Klaus Schmaranz for their guidance and support throughout this project. Special thanks to all the members of IICM for their help.

Thanks to all friends who have helped with fruitful discussion and comments during the evolution of this work. Finally I would like to thank Gudrun and my family for their love, support, and patience during all the years of my studies.

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

Signature of the author:

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

Abstract

In the field of information systems the number of different databases and protocol standards is increasing rapidly. It requires a great deal of energy for applications to handle the specific ways to access these systems.

This thesis presents a concept how to logically unlink access from the system. Thereto a middleware layer is inserted between the client and the server which abstracts the functionality of the underlying system and therefore provides a protocol-independent access.

Within the scope of this work, the requirements for such a middleware are set up by use of an example and compared with the functionality of available products. Different problems that arise when integrating server systems and providing information stored in those server systems are discussed and solutions for these problems are considered.

Zusammenfassung

Im Bereich Informationssysteme steigt die Anzahl der verschiedenen Datenbanken und Protokollstandards rapide an. Für Applikationen, die auf diese Systeme zugreifen wollen, bedeutet es einen immensen Aufwand, die jeweiligen Zugriffsmöglichkeiten zu kennen und zu implementieren.

Diese Diplomarbeit präsentiert ein Konzept, wie die Art des Zugriffes vom System logisch getrennt werden kann. Dazu wird eine Mittelschicht zwischen den "Client" und den "Server" eingezogen, die die Funktionalität des darunterliegendes Systems abstrahiert und so einen protokollunabhängigen Zugriff ermöglicht.

Im Rahmen dieser Arbeit werden Anforderungen an eine solche Mittelschicht anhand eines Beispieles aufgestellt und verfügbare Produkte daraufhin untersucht. Verschiedene Probleme, die im Zusammenhang mit dem Einbinden von Serversystemen oder mit dem Anbieten von Informationen stehen, werden behandelt und Lösungsmöglichkeiten beschrieben.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Today's world of computers consists of many different types of operating systems, different types of database systems, and different types of servers providing information. Users wanting to retrieve information from the server of their choice have to know exactly which client to use or at least which protocol the client has to use. A mediator-system that is able to connect to numerous server systems on one hand and to serve the information stored in those systems to a variety of different clients on the other hand would solve this problem.

This system does not only need to provide the connectivity to clients or servers, but has to be platform-independent to eliminate other limitations.

1.2 Structure of this Work

Chapter 1 *Introduction:* This chapter gives an introduction to middleware technology and presents the problem definition.

Chapter 2 *Requirements for the Dino System:* According to the problem definition, the requirements for a highly integrative system are set up by means of an example. These requirements include the accessibility of the server's content for a variety of clients, the

need for a highly flexible security system, different server systems that the middleware should be able to connect to, and a broad range of features concerning documents located inside the system.

Chapter 3 *Technology Used:* Different technologies that are used by the sketched middleware system or by one of the products compared are outlined and shortly discussed. This includes the communication infrastructure like RMI or CORBA, gives an overview of transaction models in general and of specific implementations, and describes some important Java interfaces that unify access to different server systems. One section in this chapter is commited to the Document Object Model (DOM), one of the key technologies in middleware systems.

Chapter 4 *Product Comparison:* Middleware systems are booming at the time of writing. Some of the products available can be used only with the proprietary database system of the manufacturer, some concentrate on providing the content of a relational database for Web browsers, but some are also extremely flexible. This chapter gives a rough overview of the available systems, outlines their field of application, discusses their technology, illustrates their strength, and exposes their weaknesses.

Chapter 5 *Implementation:* General concepts of the implemented middleware system are presented and a short historical review explains the reasons for some of the design decisions for the current version of the system. A broad range of problems concerning the integration of external server systems and providing their content to a variety of clients is discussed.

Chapter 6 *Conclusion and Outlook:* The achieved goals of the middleware systems are summarized and unsolved problems as well as future work is outlined in this chapter.

1.3 Related Work

A complex discussion about access control in heterogenous server systems can be found in [Hau99].

1.4 Middleware/Three Tier Architecture

Using a client/server architecture is a standard way to access a remote server system.

Webopedia [Web99] defines "client/server architecture" as follows:

> A network architecture in which each computer or process on the network is either a client or a server. Servers are powerful computers or processes dedicated to managing disk drives (file servers), printers (print servers), or network traffic (network servers). Clients are PCs or workstations on which users run applications. Clients rely on servers for resources, such as files, devices, and even processing power.
>
> [...]
>
> Client-server architectures are sometimes called two tier architectures.

Current implementations of the two tier architecture provide limited flexibility in moving (repartitioning) program functionality from one server to another without manually regenerating procedural code [Sad97a].

The three tier software architecture emerged in the 1990s to overcome the limitations of the two tier architecture. The third tier (middle tier server) is located between the user interface (client) and the data management (server) components. This middle tier provides process management where business logic and rules are executed. The three tier architecture is used when an effective distributed client/server design is needed that provides increased performance, flexibility, maintainability, reusability, and scalability compared to two tier, while hiding the complexity of distributed processing from the user [Sad97b].

In figure 1.1 it is shown that the three tier architecture has major advantages, as access to any of the servers can be obtained using any of the APIs[1] or protocols provided on the client side.

Systems using this three tier architecture are often called middleware systems. The definition of the term "middleware" according to [Web99] is the following:

> Software that connects two otherwise separate applications. For example, there

[1] Application Programming Interface

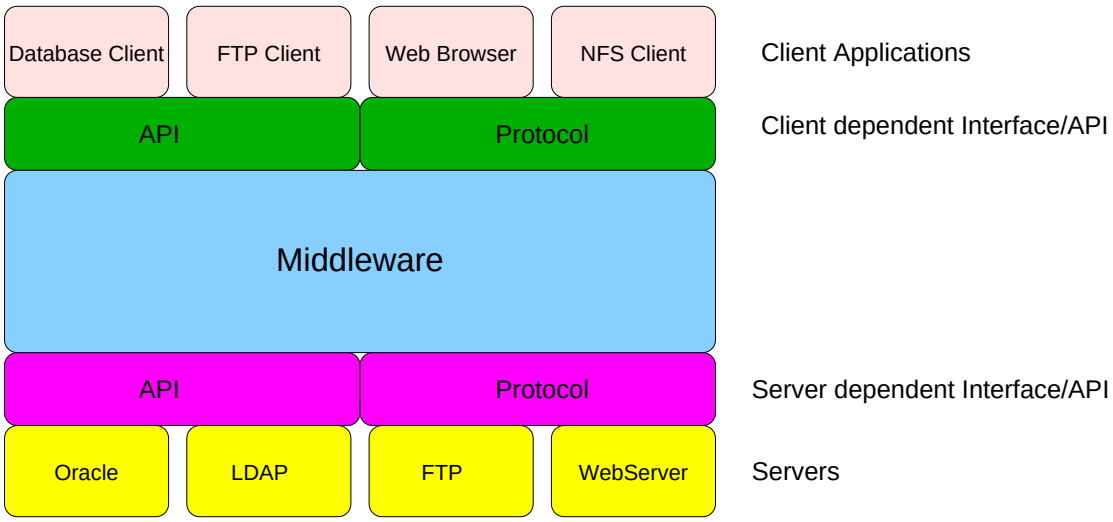


Figure 1.1: Three tier/middleware architecture enables clients to access a wide variety of server systems without the knowledge of the specific protocol/API.

are a number of middleware products that link a database system to a Web server. This allows users to request data from the database using forms displayed on a Web browser, and it enables the Web server to return dynamic Web pages based on the user's requests and profile.

The term middleware is used to describe separate products that serve as the glue between two applications. It is, therefore, distinct from import and export features that may be built into one of the applications. Middleware is sometimes called plumbing because it connects two sides of an application and passes data between them.

According to [Ber99], the most important advantages of the middleware approach are the following:

- The desire to use open services and protocols.
- The wish to redeploy logic at will and unconstrained by infrastructure; this necessitates using open APIs and protocols, which are widely supported across most infrastructure products.

- The need to support cooperating mixed-architecture applications.
- The urge to move network and service infrastructure decisions out of application space, so that system managers can make infrastructure decisions without being hampered by applications that depend on proprietary protocols or features.

1.5 Problem Definition

The problem is to find a technology for integrating heterogenous server systems. Such a technology must be scaleable, platform-independent, and it must allow completely different systems to be embedded on one hand, and on the other hand a lot of different clients shall be able to access the data stored in the embedded systems. The implementation of such a system at the IICM is called Dino, standing for **D**istributed **I**nteractive **N**etwork **O**bjects.

Following the definitions of section 1.4, Dino is the glue between different types and manufacturers of databases (in the widest meaning of this word) or applications on one side and a broad range of clients using any API or protocol on the other side.

To give an idea of a real-world example, a medical document as shown in figure 1.2 is used. The parts of this document are distributed across different databases (sketched in figure 1.3).

The document is composed of a medical report, an image that represents a thermic diagnosis, and a whiteboard for communication with a remote specialist. Including meta-data, the report is stored as an XML document in the (local) filesystem, whereas the thermic image and its meta-data reside in a Hyperwave server. Additionally meta-data concerning the overall content (report and images) are kept in an Oracle database. This database holds all relations and signatures concerning the content as a whole. User authentication uses the X.509 certificates stored in an LDAP server. The whiteboard is an active content and communicates with its counterpart.

The medical staff needs to be able to access the document using their preferred wordprocessor for changes and a Web browser for a quick overview. Figure 1.3 gives an outline of all parts involved in this use case.

In the given medical record, there are different kinds of relations involved:

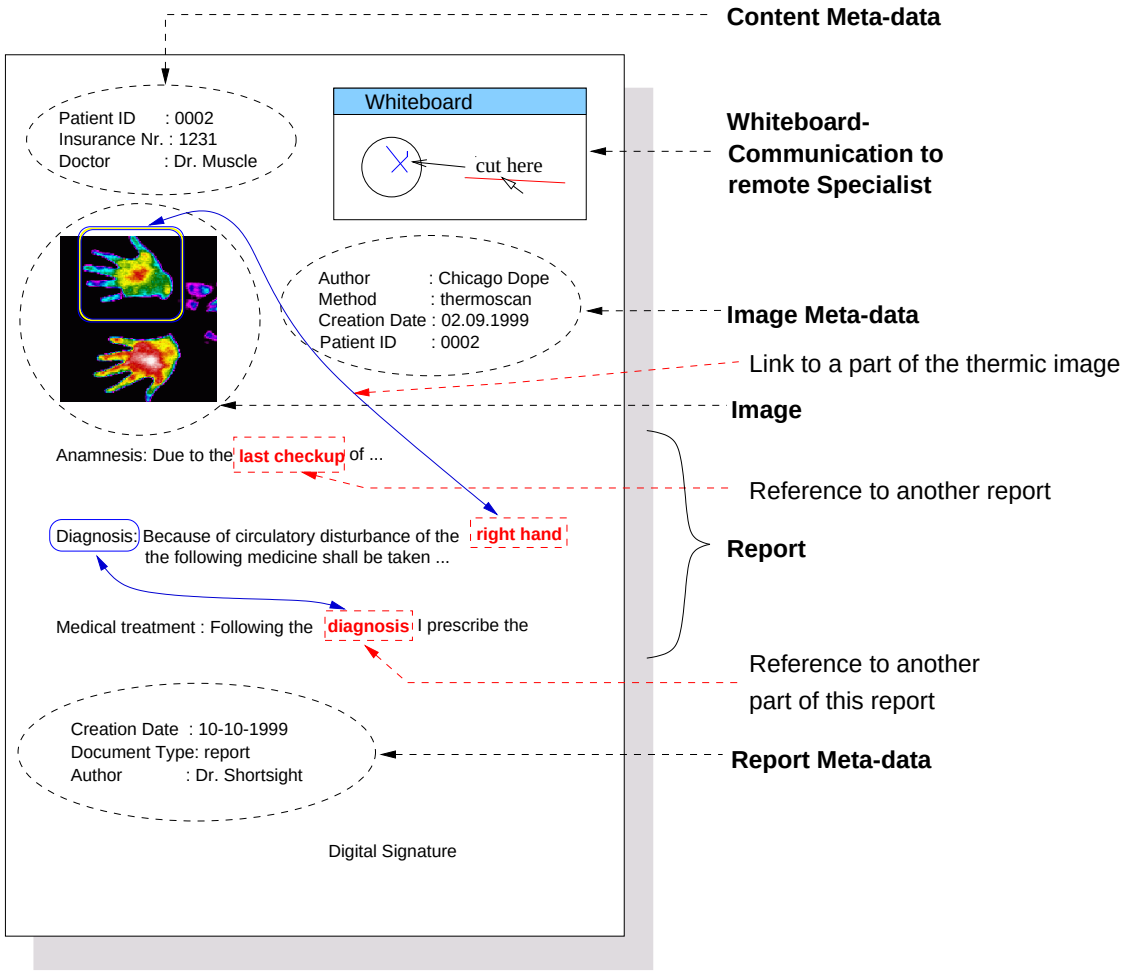


Figure 1.2: A medical document illustrates the requirements of a middleware system.

- Intra-document relations (from one paragraph in the report to another)
- Intra-content relations (a relation from the report to some position inside the image)
- Relations to an external location (to another report), so called inter-document relations.

The collection and analysis of these user requirements results in the software requirements dicussed in chapter 2.

In the following, answers to the questions below will be given:

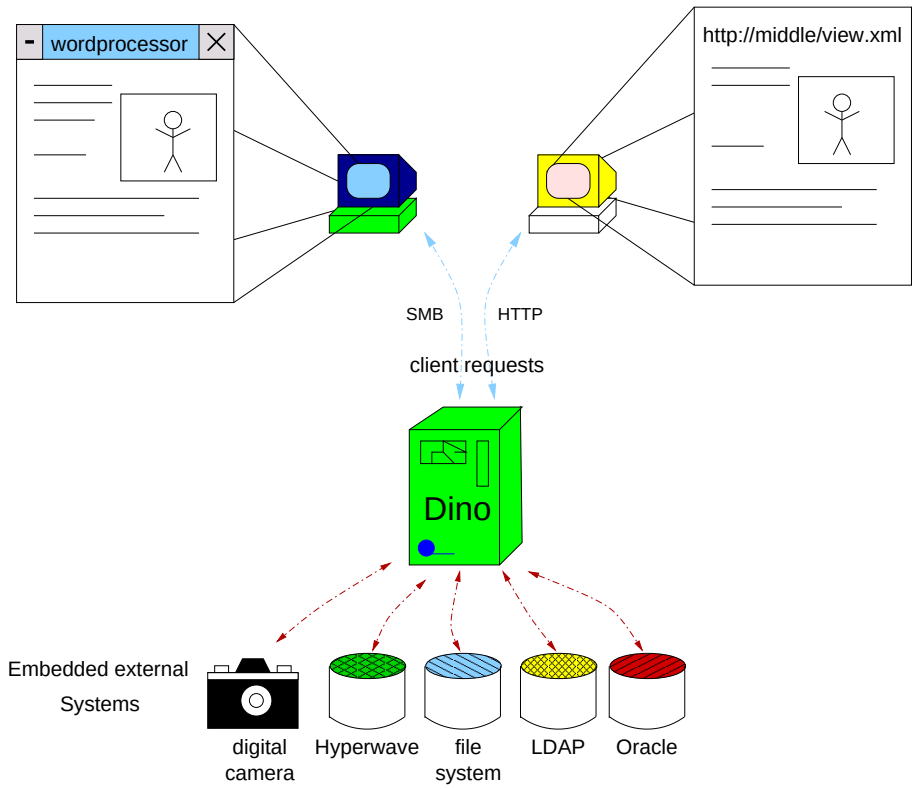


Figure 1.3: An example of a middleware system integrating a variety of server systems and providing the content to different clients.

- Are there middleware systems existing that satisfy the requirements? How do they work?
- Is there one product that satisfies all requirements?

  Is re-invention of a middleware system necessary at all?
- What standards are used by middleware systems?

  There exist standards for communication, content handling, meta-data retrieval. Are they taken into account by middleware systems?
- How platform-independent is the concept of Dino?

  The system should work on different platforms. What is the range of supported platforms and architectures?

- How secure is the system?

  It is not acceptable that all users in the system may access any content without access right checking. Is it possible to integrate other user management systems?

- How universal is the concept?

  The system must be able to integrate new technologies without changes of the concept.

- What problems occured when integrating new systems into Dino.

  Different problems were occuring when integrating specific systems. Do they concern the whole concept or can they be solved using available techniques?

- How can new clients access the content of Dino and which aspects have to be paid attention to?

  Using different ways accessing the objects inside the Dino system requires different protocols, different document formats and different ways to provide the content.

Chapter 2

Requirements for the Dino System

In this chapter the requirements (see also [IIC99b]) of Dino, a middleware system layer that solves the problems defined in section 1.5, are sketched.

Before going into details let us first clarify the terms used.

- **Embedding:** Is the term that is used for the process of integrating an external system into the Dino world.
- **External System:** Is the term that is used for servers providing any type of information in any form that can possibly be embedded in the Dino system.
- **Dino Node:** Is the term that is used for a single addressable object in the Dino world that virtually covers one or more objects of embedded external systems. A Dino Node provides all functionality necessary to work with the embedded objects in the given context. This includes but is not limited to structure and content manipulation, definition of relations between Dino Nodes and special functionality of embedded systems.
- **Relation:** Is the term that is used for an interconnection of two Dino Nodes. Please note that the term Relation in this document always stands for 1:1 Relations. Relations in Dino are typed and special cases of relations, so called *structural* Relations, allow navigation through the Dino space.
- **Content:** Is the term that is used for the combination of data and meta-data of a

Dino Node.

2.1 Platform Independence

Dino has to be as platform independent as possible to ensure global availability. For this reason, it should be implemented in 100% pure Java[1]. Java is widely accepted by different vendors of operating systems, manufacturers of databases, and industry and so the support for accessing external systems is extremely sound. Sun's Java Development Kit (JDK)[2] must be supported, others (as IBM's jikes[3]) are likely to be supported and evaluated.

2.2 External Gateways

The middleware system would be useless (and not a middleware system), if it would not allow arbitrary clients to access the data stored in the embedded systems. External Gateways transform objects provided by the Dino system into whatever fulfills the protocol specification. An HTTP[4] gateway for example provides the objects' contents as byte streams, including information about the MIME[5] type.

In figure 2.1 it is shown that External Gateways are defined as modules set on top of the Dino system, that holds and manages all objects. These modules must be independent of the Dino Kernel, so adding new gateway modules is possible without changing the system.

Gateways may be arbitrarily complex, for example an HTTP gateway can provide automatic user-interface configuration and special functionality to make parts of the Dino system functionality accessible for the outside world.

In addition to External Gateways there are APIs for Java and mappings for remote object systems (CORBA[6], RMI[7]) defined, which enable the use of objects over the network and in the case of CORBA make the object system of Dino usable by other programming

[1] `http://java.sun.com`
[2] `http://java.sun.com/jdk`
[3] `http://www10.software.ibm.com/developerworks/opensource/jikes/`
[4] Hypertext Transfer Protocol
[5] Multipurpose Internet Mail Extensions, see [FB96]
[6] Please see section 3.1.1 for details.
[7] Please see section 3.1.2.

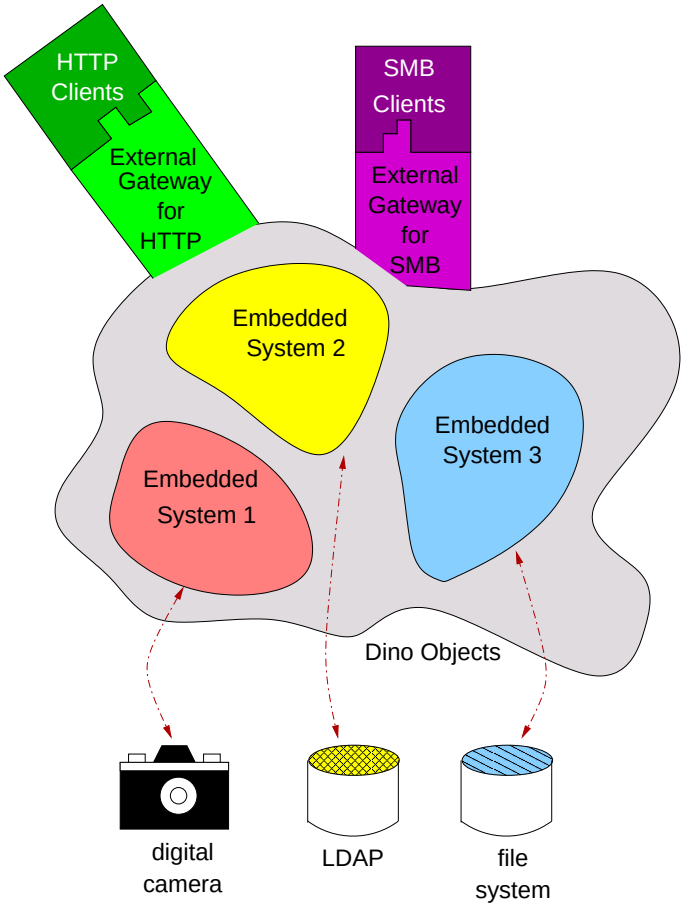


Figure 2.1: External gateways provide Dino objects.

languages like C++.

2.3 Embedded Systems

It must be possible to integrate external systems and to use the data stored there independent of their origin. Special Dino modules are responsible for connecting external systems, requesting data, and wrapping the results into objects administrated by Dino.

It is not specified which types of external systems are to be embedded. As long as there is a network protocol or an API that allows Dino to access, it must be possible to integrate

the system and to provide the data stored in it to any Dino client (using the Dino External Gateways or API).

When using a Dino client (via an External Gateway or via API calls), functionality must not be lost in comparison with a client specialized for this type of network protocol that is connected directly to the embedded system. Additional functionality of an embedded system must be accessible by Dino clients.

On the other hand, accessing an embedded system via Dino must not be disturbingly slower than raw access without Dino. It is not desirable to have to change to a much faster computer just because of embedding a system with Dino, so it has to be possible to run Dino and embed the server on the same computer without a significant performance loss.

2.4 Scalability

The Dino system must be scalable concerning following aspects:

- The number of embedded systems
- The number of objects in the embedded systems and so the number of Dino Nodes
- The number of users working with Dino simultaneously
- The number of Dino systems connected to form a peer-network

It must be possible to connect several Dino systems to a peer-network forming a larger Dino system. In this network, all operations have to be transparent concerning the different Dino systems. This Dino cluster has to support traffic splitting (load balancing). By the use of a Dino cluster, bottlenecks can easily be removed without applying changes to the applications accessing the system.

2.5 Security

The security aspect must not be neglected. The administrator or any authorized user of a Dino system must have the possibility to create user rights to allow/deny the access of

specific users/groups on arbitrary documents. Access rights are defined preferably by use of a rule set to minimize the effort of administration.

The check of rights has to be independent of the way users access the system. It must be assured that no operation is executed without being checked by the Security Manager residing in the Kernel of the Dino system.

Existing user managers of embedded systems must be integratable, so adding new embedded systems must not result in unknown security behaviour!

User/system authentication must be possible using new technologies like smartcards or fingerprint scanners as well as digital signatures and encryption.

For a more detailed specification of security in the Dino system, please see [Hau99].

2.6 Addressability of Objects

All objects residing in a Dino system must be addressable using globally unique handles. This ensures that they can be referenced directly and therefore can be accessed as fast as possible.

It is desirable that Nodes, even if they are moved to a different physical location (e.g. moved from an embedded FTP server into a relational database), keep their unique handle. If external circumstances imply that the handle of the object changes, a mechanism must be provided that helps to keep the handle stable for the outside world.

2.7 Relations

Dino has to support different types of interconnections between objects in the Dino addressing space:

- Relations that can be mapped 1:1 to a physically existing relation, like a parent-child relation from a directory to a file in a file system.
- Logical relations, like a link from an annotation to the document it belongs to. These are not necessarily stored on the same external system as the documents.

Nevertheless this kind of relation can also represent structural links, like the parent-child relation mentioned above.

Using logical relations, it is possible to add functionality to an external system that would otherwise be unable to store arbitrary relations between documents. A Unix filesystem for example is able to create symbolic links, but is not capable of storing other types of relations, like annotations, bookmarks, etc.

Relations have a source, a destination, and a type. They are always bidirectional, so the destination Node knows about relations pointing to it. By using subtypes like `symlink.to` respectively `symlink.from` or `hierarchy.down` it is possible to imply a logical direction of the relation.

Relations are to be kept stable if possible. So moving objects inside the Dino system does not lead to broken relations. This functionality is provided by the system so the user does not need to take care of relations pinned to an object when changing the structure.

2.8 Structure Manipulations

The Dino system has to provide operations to manipulate the structure of the Dino Nodes represented by Relations (move, copy, delete, ...). These methods must be able to change the physical structure of data in the embedded external system (e.g. move a file from one directory to another) as well as the logical virtual structure in the Dino system (change Dino relations).

2.9 Version Control

The administrator/user must be able to choose if objects stored in the Dino system are version controlled, so changes can be undone. This functionality is independent of capabilities of underlying embedded systems.

The functionality must be similar to the concurrent version control system (CVS)[8] which supports checking-in, checking-out, locking, reverting, tagging, and concurrent merging.

[8] `http://www.cyclic.com`

These operations must be applicable for single Nodes as well as for clouds of Nodes.

2.10 Content

Dino must be able to store a wide variety of document types and formats. It has to be extremely flexible and extendable, so the conversion from given document formats to the internal object model or vice versa must be achievable as effortless as possible.

Content must not necessarily be just passive. Active components like a whiteboard that allows collaborative working will be necessary to increase the 'quality of working' with the Dino system. The usage of the whiteboard embedded in the medical report (as shown in the example in figure 1.2) allows communication with other users directly without the need of an additional application and without the loss of information due to the separation of application and data.

Nevertheless a document that contains active components must be convertible to other document formats without loss of information. The functionality of the active part is probably lost, but the information must be provided! When the medical report for example is converted to an HTML document, the active whiteboard-component could be converted to a Java applet. In this case, the functionality of the whiteboard is guaranteed even after conversion. If the new document format does not allow active content, the whiteboard is probably transformed to an image showing the static information of the whiteboard at the time it was converted.

2.11 Distributed Objects

Any documents, or generally objects, in the Dino system must be distributeable. The term distributeable implies that the object does not need to be located as a whole on one server, but parts of it may be situated on different locations. The medical document of figure 1.2 for example is composed of an image, a report, a whiteboard, and the inherent meta-data sets. Each part of these can have its source in any server system as long as these are connected to the Dino system.

This requirement demands from Dino that all objects stored in any of the connected server

systems are uniquely addressable so they can be referenced by the embedding document.

2.12 Object Model

In section 2.10 it has been stated that the object model for objects held in the Dino system must be extremely flexible and extendable. One of the main tasks of Dino is to provide different clients with different formats of the content objects. This asks for a model that is widely accepted by different manufacturers and their products.

Chapter 3

Technology Used

"Capital letters were always the best way of dealing with things you didn't have a good answer to." :-)
Douglas Adams

In this chapter an overview of technology needed for building Java middleware components will be given.

3.1 Communication Infrastructure

At the time of writing, Dino can use either RMI or a special Dino-Dino low-level protocol to communicate with other Dino systems. Support for CORBA is planned for Dino clients using other programming languages than Java. Nevertheless, as most of the middleware systems discussed in chapter 4 offer the possibility to use CORBA, a short overview is given here as well.

3.1.1 CORBA

The Common Object Request Broker Architecture (CORBA) [Obj97] is a standard for designing distributed applications. It was developed by the Object Management Group (OMG)[1] and is an object-oriented approach to the problem of distributed systems. CORBA

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

enables full transparency over a wide range of programming languages and operating systems. The implementation of CORBA is called *Object Request Broker (ORB)*. Compared to the client server structure where client and server have to deal with network connections on their own, CORBA uses a communication middleware. The server object is defined and accessed through an interface. Therefore the connection is completely transparent. This structure is shown in figure 3.1: At the server side the functionality is defined by an interface. Clients access the server through a client-side instance of this interface, called stub [OPR96].

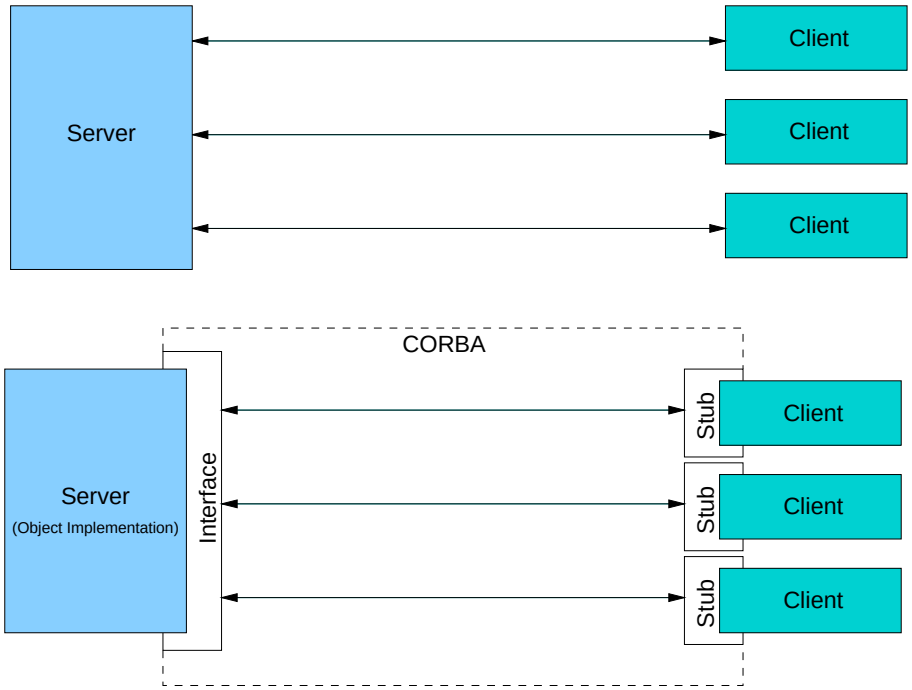


Figure 3.1: Client-server computing (top); Distributed computing with CORBA (bottom) [Bac98]

At the most basic level, CORBA is a standard for distributed objects. CORBA allows an application to request an operation to be performed by a distributed object and for the results of the operation to be returned back to the application making the request. The application communicates with the distributed object that is actually performing the operation. This is basic client/server functionality where a client sends a request to the server and the server responds to the client. Data can be passed from the client to the server and is associated with a particular server operation. The result is then returned to

the client in the form of a response. A good introduction to basic CORBA concepts can be found in [OPR96].

CORBA uses the language independent Interface Definition Language (IDL). An interface consists of a set of named operations and the parameters for these operations. The parameter types are then mapped to a given programming language, like C, C++, Java, Smalltalk, ...

Using CORBA implies the use of an Object Request Broker that is usually a commercial product. There are some free versions available, but they often do not support the full specification.

The CORBA architecture in principle contains the following components:

- **Object Request Broker (ORB)**, which enables objects to transparently send and receive requests in a distributed, heterogeneous environment. This component is the core of the OMG reference model.
- **Object Services**, a collection of services that support functions for using and implementing objects. Such services are considered to be necessary for the construction of any distributed application. Of particular relevance in this context is the Object Transaction Service (OTS), discussed more in detail in section 3.2.1.
- **Common Facilities**, are other useful services that applications may need, but which are not considered to be fundamental such as desktop management and help facilities.

Inter ORB Protocol (IOP)

The *General Inter ORB Protocol (GIOP)* [Obj97] defines the low level communication protocol of Object Request Brokers. This includes

- **Common Data Representation (CDR)** is a transfer syntax mapping OMG IDL data types into a bicanonical low-level representation for "on-the-wire"transfer between ORBs and Inter-ORB bridges (agents).
- **Message Format** concerning object requests, locating object implementations, and managing communication channels.

- The **Transport Assumptions** describe the general assumptions made concerning any network transport layer that may be used to transfer GIOP messages. The specification also describes how connections may be managed, and constraints on GIOP message ordering.

A specific mapping of GIOP which runs directly over TCP connections is called the *Internet Inter ORB Protocol (IIOP)* It adds the following element to the GIOP specification:

- **Internet IOP Message Transport** The IIOP specification describes how agents open TCP connections and use them to transfer GIOP messages.

As a more detailed description of CORBA and IOP is out of scope of this thesis, please see [Lug99] for a good overview of resources available.

3.1.2 Remote Method Invocation (RMI)

Remote Method Invocation (RMI)[2] is an API provided by Sun to create distributed Java applications. It enables a Java application to invoke methods of Java objects on remote hosts. Once a reference is obtained, either by looking up the remote object in the bootstrap naming service provided by RMI or by receiving the reference as an argument or a return value, a Java application can call methods of it just like for a non-remote object. RMI uses object serialization for sending objects across a network.

The low-level protocol for network communication, the RMI Wire Protocol, uses TCP directly or HTTP in the case where firewalls or proxies deny a direct connection. The type of connection used by RMI may be customized via a socket factory, so for example an SSL connection might be realized. Remote classloading for example from a given file- or Web-server is supported using a remote classloader.

Figure 3.2 illustrates how a client accesses a remote object to invoke a method on it. The reference to a remote object is obtained using the RMI Registry, a naming-service that allows to bind objects to a name. Clients may retrieve the object by requesting it using this name.

[2] `http://java.sun.com/products/jdk/rmi/`

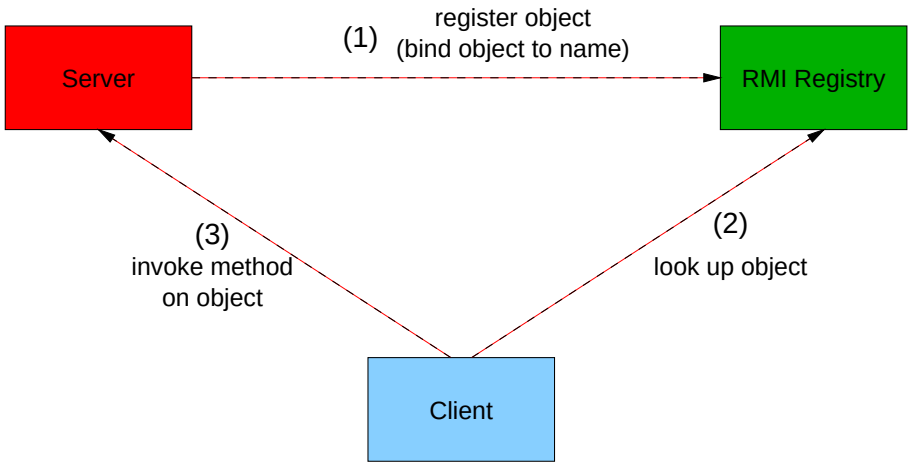


Figure 3.2: Invoking a method on a remote object using RMI. The object has to be registered and bound to a name in the RMI Registry (1). The client requests the name of the Registry and the remote object is returned (2). Then the client invokes a method on this object, which is executed on the Server (3).

RMI over IIOP

Sun just announced on their RMI-Web site[3] that

> Remote Method Invocation (RMI) over Internet Inter-Orb Protocol (IIOP) delivers CORBA compliant distributed computing capabilities to the Java 2 platform and to JDK 1.1. RMI over IIOP was developed by Sun and IBM. The joint work by Sun and IBM to implement Object Management Group (OMG) standards demonstrates the spirit of collaboration that continually moves the Java platform forward.
>
> RMI over IIOP combines the best features of RMI with the best features of CORBA. Like RMI, RMI over IIOP speeds up distributed application development by allowing developers to work completely in the Java programming language. When using RMI over IIOP to produce Java technology-based distributed applications, there is no separate Interface Definition Language (IDL) or mapping to learn.

[3] `http://java.sun.com/rmi`

It seems that CORBA and RMI start to "drift together", so Java developers do not need to decide on using either RMI or CORBA.

As this is a brand new development (October 1999), the author of this thesis has neither tested RMI over IIOP, nor was it ever used in the Dino system. This will be part of the future work in this area.

3.2 Transactions

Transactions are a vital point in middleware systems. They are an important programming paradigm for simplifying the construction of reliable and available applications, especially those that require concurrent access to shared data.

A transaction is a unit of work that has the following characteristics [OMG97]:

- **Atomicity:** The transaction completes successfully (commits) or if it fails (aborts) all of its effects are undone (rolled back).
- **Consistency:** Transactions produce consistent results and preserve application specific invariants.
- **Isolation:** Intermediate states produced while a transaction is executing are not visible to others. Furthermore transactions appear to execute serially, even if they are actually executed concurrently.
- **Durability:** The effects of a committed transaction are never lost (except by a catastrophic failure).

A transaction can be terminated in two ways: committed or rolled back. When a transaction is commited, all changes made within it are made persistent (forced to stable storage, e.g., disk). When a transaction is aborted (rolled back), all changes made are undone.

At the point of writing, no transactional functionality was implemented in the Dino library, nevertheless transactions are an important topic for future development. For this reason, a short overview on transaction services available for Java is given:

3.2.1 Object Transaction Service

The Specification of the Object Management Group (OMG) states in [OMG97], that the *Object Transaction Service* defines interfaces that allow multiple, distributed objects to cooperate to provide atomicity. These interfaces enable the objects to either commit all changes together or to rollback all changes together, even in the presence of failure. No requirements are placed on the objects other than those defined by the `Transaction Service` interfaces.

Transactional behavior can be a quality of service that differs in different implementations.

3.2.2 Java Transaction Service/Java Transaction API

Sun's Java approach to transactions is split into two parts [Mor99] which work in tandem:

- The **Java Transaction Service (JTS)**[4] has two specific roles:
  1. It specifies the implementation of a transaction manager that supports the JTA interface at high-level. This means that JTA implementations will exist inside the transaction manager and will use the `javax.jts` interface to establish a contract with the transaction manager.
  2. It implements the OMG Object Transaction Service (OTS) 1.1 low-level specification. This means that all Java transaction managers use the CORBA IIOP (Internet Inter-ORB Protocol) standard for communication with other JTS- or OTS-based transaction managers.
- The **Java Transaction API (JTA)**[5] consists of three parts:
  1. A high-level application interface (`javax.transaction.UserTransaction`) that allows Java objects to demarcate transaction boundaries. This means that the Java objects will associate the thread they are running in with a current or new transaction context. It is important to note that if the transaction manager implementation does not support nested transactions, the object cannot create a new transaction boundary if it has already established one.

[4] `http://java.sun.com/products/jts/`
[5] `http://java.sun.com/products/jta/`

2. A Java mapping of the X/Open XA transaction protocol (`javax.transaction.xa`), which allows XA-compliant resources to participate in a global transaction being controlled by an external transaction manager. This interface is the contract between the Transaction Manager and the Resource Manager.
3. A high-level interface (`javax.transaction.TransactionManager`) that allows application servers to manage transaction boundaries on behalf of objects they are managing. This interface is very similar to `javax.transaction.UserTransaction`, but adds support for suspending and resuming transactions, which allows objects to associate and disassociate from transaction contexts quickly.

These interfaces only provide application developers with transactional semantics, such as commit or rollback. The real power of transaction occurs within the resource managers. Resource managers are processes that are transaction-aware and can take an action based upon a commit or rollback decision. If they have no idea about how to re-establish the state of the system before the transaction started, the entire use of transactions in the application is for naught. This is the reason why database management systems that support transactional semantics or JTS-enabled application servers are vital to any Java applications using transactions.

3.3 JDBC

JDBC[6] (often thought of as standing for "Java Database Connectivity") is a Java API for executing Structured Query Language (SQL) statements. It consists of a set of classes and interfaces for accessing DataBase Management Systems (DBMS) using Java.

JDBC deals with the following tasks of a database query:

- establish a connection with a database
- send SQL statements
- process the result.

[6] `http://java.sun.com/jdbc`

JDBC deals with the difficulties that arise when using different DBMSs, as the data types used sometimes vary. Another area of difficulties is the lack of SQL conformance. SQL-dialects vary widely in functionality (some do support e.g. stored procedures or outer joins, others don't) or syntax (there exist specialized derivatives of SQL designed for specific DBMSs for document or image queries, for example). To deal with this problem, JDBC allows any query string to be passed through to an underlying DBMS driver. This means that an application is free to use as much SQL functionality as desired, but it runs into the risk of receiving an error on some DBMSs whereas it works on others. The SQL string is not parsed at all by JDBC, this allows the use of special SQL statements that are understandable only by the given DBMS, but also bears the risk of undetected bugs [WFC+99].

3.4 JNDI

Java Naming and Directory Interface[7] provides a unified interface to multiple naming and directory services, like Yellow Pages (now called NIS), LDAP (Lightweight Directory Access Service), NDS (Novell Directory Service), or DNS (Domain Name Service). Directory services provide access to a variety of information about users, machines, networks, services, and applications in the Internet and in Intranets stored in one of the services mentioned above. For example, DNS may be used as the top-level naming facility for different organizations within an enterprise. The organizations themselves may use a directory service such as LDAP, NDS, or NIS to authenticate users and provide resources for them [Sun99b]

3.4.1 Overview of Interfaces

The JNDI API is divided into different interfaces [Sun99b]:

- **The Naming Interface**[8] defines basic operations such as adding a name-to-object binding, looking up the object bound to a specified name, listing the bindings, removing a name-to-object binding, ...

[7] `http://java.sun.com/jndi`

[8] Java class `javax.naming`

- **The Directory Interface**[9] enables access to attributes for objects stored in directories. This includes searches for specific attribute values.
- **The Event Interface**[10] informs about events generated by a naming or directory service, e.g. a rename of a directory entry.
- **The Service Provider Interface (SPI)**[11] provides the means by which different naming/directory service providers can develop and hook up their implementations so that the corresponding services are accessible from applications through JNDI.

3.4.2 JNDI & Dino

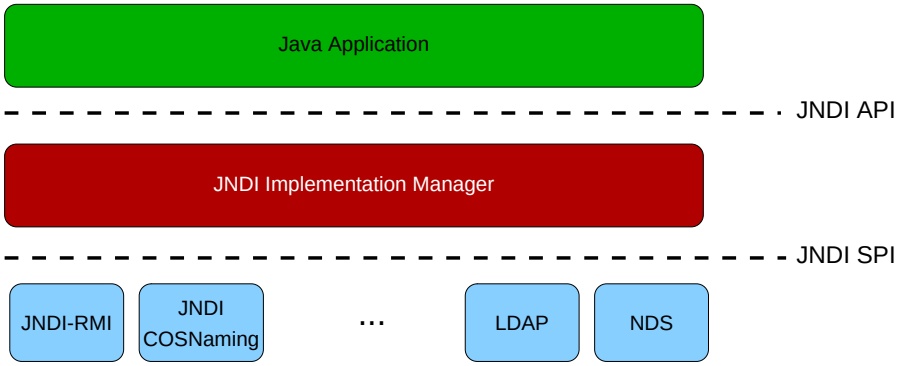


Figure 3.3: JNDI layer model: The integrated services are separated by an abstraction layer from the applications accessing the system.

Figure 3.3 shows the layer model of JNDI. It follows the same concept as Dino does, even though there are some important differences:

- JNDI supports only hierarchical structures of objects, whereas Dino is able to map any structure by using Dino references. This is due to the fact that JNDI defines the structure by using the namespace and all directory services use a hierarchical namespace (e.g. DNS, LDAP, ...).

[9] Java class `javax.naming.directory`
[10] Java class `javax.naming.event`
[11] Java class `javax.naming.spi`

- The objects stored in a directory are of arbitrary type and the type cannot be requested from the service provider. The service provider that provides the local filesystem for example, returns an object of the Java class `java.io.File`. Other service providers may return other types, but most of them do not return objects at all. Dino always returns an object using the Document Object Model (DOM) (see section 3.5) and is therefore providing additional functionality.

JNDI is used by Dino as an external embedded system, and so it enables Dino to connect to all directory services provided by JNDI (LDAP, NIS, NDS, ...).

3.4.3 LDAP

DNS, NIS, and NDS were already playing an important role in the past, but LDAP definitely is getting the most attention at the moment. LDAP was developed as a low cost, PC-based front-end for accessing X.500 directories, as the ISO standard proved too cumbersome and overhead-intensive [How99]. Nowadays, nearly all email programs are able to search in various LDAP directories around the world for email addresses and most of the high-end operating systems (Linux, Windows NT, ...) can use LDAP databases for user authentication.

3.5 Document Object Model (DOM)

The Document Object Model Specification [W3C98] states, that

> the Document Object Model (DOM) is an application programming interface (API) for HTML and XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated.
>
> With the Document Object Model programmers can build documents, navigate their structure, and add, modify, or delete elements and content.

Using DOM, any documents have a logical tree-like structure. As the name of the model shows, DOM follows an object oriented design: documents are modeled using objects.

Shady Grove	Aeolian
Over the River, Charlie	Dorian

```
<TABLE>
  <TBODY>
    <TR>
      <TD>Shady Grove</TD>
      <TD>Aeolian</TD>
   </TR>
   <TR>
     <TD>Over the River, Charlie</TD>
     <TD>Dorian</TD>
  </TR>
  </TBODY>
</TABLE>
```

Figure 3.4: On the left side a table and on the right side its XML code. The tree-like Document Object Model can be anticipated from this code.

This model includes the structure as well as the behaviour of the documents and the objects of which it is composed.

Consider the table shown on the left side of figure 3.4 as an example. The XML code derived from this table (right side of figure 3.4) already gives an impression of the tree-like structure of the Document Object Model of the table.

Figure 3.5 shows the object model of the table. It can be seen clearly, that each component of the table is represented by an object and all objects are interconnected in a tree structure.

Though DOM has a close coherence to HTML and XML, it is not limited to these two document formats, though any document written in HTML or XML can be represented in DOM.

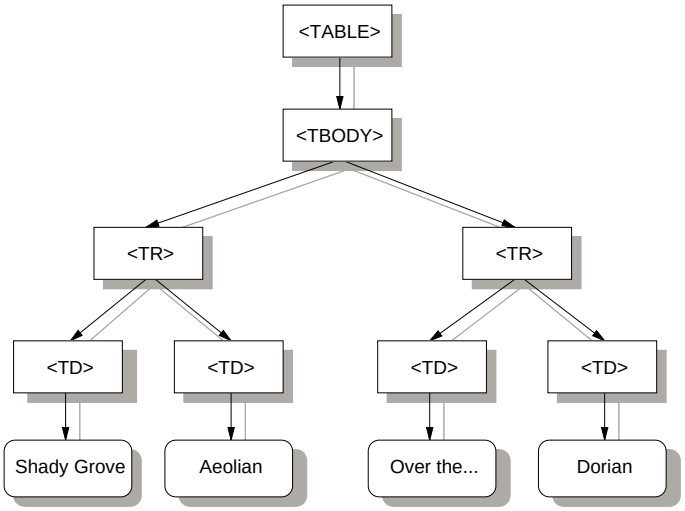


Figure 3.5: DOM representation of the example table. [W3C98]

Chapter 4

Product Comparison

This chapter compares several systems that are able to integrate different server systems. This can be done by

- providing more than one way to access the stored data from a client,
- by integrating more than one system to store the data,
- or by doing both.

It is obvious that the latter is the most interesting one, and so the main focus is directed to those "middleware" systems.

The term "application server" is very often used interchangeably with "middleware" or "three tier". According to [Wri99],

> an application server is a three tier architecture with an application's development and deployment layer between the client and backend or legacy systems. This is why the application server space is also known as middleware. Web servers reside in the middle layer, along with all kinds of applications such as CGI[1] and ASP[2] that interface directly with the Web server. The significance of application servers is that they also interact with the backend directly –

[1] Common Gateway Interface
[2] Active Server Pages

> but the Web server and the client do not. The application server becomes the platform where you embed the intelligent logic of Web-based applications that access enterprise-wide resources.

In general, most application servers or middleware systems tend to start from three different origins:

- Web servers with added functionality (e.g. CGI scripts that perform SQL queries, format the results, and display the information).
- Database servers with added functionality (e.g. a database with an integrated Web server, so clients are able to connect to the database directly for some (provided) SQL queries).
- "Real" middleware systems (usually coming from the world of client/server technologies) that connect diverse kinds of services.

Depending on the intended use, a solution should be chosen from one of the three categories given above. Nevertheless the main focus in this chapter will be directed on the middleware systems mentioned in the last point.

A Web server in combination with CGI or ASP can solve nearly any given problem. However, there are a couple of issues (programming issues, security issues, scalability issues) that make this a non-trivial task. A simpler way, however, might be to use one of the existing application servers that grew out of the database side.

These products are very efficient in manipulating databases, they tend to have more security features, and the business logic doesn't reside on the client as would be the case with embedded scripting models. Database-centric application servers tend to be more robust when it comes to data access and manipulation in Intranet or Extranet environments but may be light on Web integration features.

On the other hand, what if the primary objective is to enhance the Web site by connecting to some server-based applications and retrieving assets stored in a database as needed. Unless visitors should be able to enter or change database records, one probably won't need all the features found in a database-centric product. Here is a case where a Web-centric application server might be a better choice. Web-centric application servers may

have fewer or less sophisticated tools for dealing with database manipulation, but they are often more savvy about HTTP servers, firewalls, Java, DHTML[3], HTML, XML, etc.

In other cases, there might be a need for communication between a database, an email server, a Web server, and a transaction server. Or the custom-built warehouse inventory application must be accessible from a Web site. In this instance, the choice goes towards an application server that was designed specifically to connect diverse applications. [Wri99]

The next sections compare the possibilities of products already available or being available in the near future.

4.1 Oracle

Oracle is one of the largest players in the database business. Although their products Oracle8*i* and Oracle Application Server (OAS) are not among the ones providing the highest flexibility concerning middleware aspects, their products cannot be neglected.

4.1.1 Oracle8*i*

Undoubtedly the latest version of Oracle's database product falls in the category "database servers with added functionality". The aspect that makes this database appear in this comparison is the *Internet File System*(*i*FS) feature of Oracle8*i*. As [Ora98b] or more in detail [Ora99] states, *i*FS provides universal access to any data in the database. While the *i*FS runs within Oracle8*i*, it appears as if it were simply another filesystem volume on the network. Any data or files in *i*FS can be accessed using any of these following protocols.

- SMB - Use this protocol to access the *i*FS through Microsoft Windows and Linux clients as a remote filesystem.
- HTTP - Use this protocol to access the *i*FS with Web browsers and network computers.
- FTP - Use this protocol to see contents of the *i*FS with (command-line) FTP clients. The contents of the *i*FS are displayed as standard FTP directories and use `GET` and

[3]Dynamic HTML

`PUT` commands to move files.

- SMTP, IMAP4, POP3 - Use these email protocols to access the *i*FS through clients like Eudora, Microsoft Outlook, and others.

Additionally Oracle provides APIs for Java, CORBA or PL/SQL[4] to develop applications using the *i*FS.

*i*FS organizes its content in folders. Depending on the protocol used, the folders appear as the protocol expects (e.g. using an email client, the folders appear as a set of email-folders, the objects in the folders appear as emails).

*i*FS folders or objects in them may have additional features:

- **Multiple parents:** While for example an email may appear in the 'Marketing Initiatives' folder, it might also appear in the 'Sales Resources' folder.

  This looks like a great thing at the first glimpse, but experience shows that this feature is more a problem than a helper. It would be better not to map the internal structure of the folders to a graph, but to use relations/links instead. As none of the protocols mentioned above is able to handle a graph (they all work with a tree mapping of the data), unforeseeable things might confuse the users . They might not expect that if they delete a file in one directory, that a file in another directory disappears as well.

- **Advanced searches:** as the *i*FS is situated inside the Oracle8*i* relational database, users can search in the content of *i*FS like in a relational database. This feature also depends on the configuration of the Oracle *inter*Media[5] option.

  This is a nice feature, if Oracle's *inter*Media is installed (as the objects are stored as BLOBs[6], they are not searchable), but even then, the use of special tools is required, as none of the protocols mentioned above does specify any possibilities to search. Offering features that some of the protocols do not support is a general problem of

[4] A procedural language extension to SQL.

[5] Oracle's *inter*media adds support that enables Oracle8*i* to manage multimedia content, including text retrieval capabilities and support for audio and video.

[6] Binary Large OBjectS

middleware systems (and a problem of Dino, too), so it must not be overvalued in this case.

- **Indexing and meta-data:** *i*FS automatically generates meta-data and indexes its content. This meta-data is generated out of external characteristics, like the date created, the date last saved, the file format, etc., or from internal contents of the document, such as title and keywords found within a Microsoft Word file. Using this feature implies the existence of a renderer for the given file format, so this feature is restricted to supported formats.

  Though it is quite time-consuming to parse all documents added or modified in the system, a similar functionality will be added to Dino to facilitate powerful searches.

- **Parsers and renderers:** When placing a document in the *i*FS, the system can decompose (or parse) the document automatically. When someone needs to view the document, the *i*FS can recompose (or render) the document in whatever fashion the application developer directs. Parsing and rendering helps to create custom views of the same document, showing different ranges of information, or perhaps the same information in different file formats.

  Dino uses the Document Object Model (DOM) as an internal data structure to be able to easily render objects in another format.

- **Check in, check out (CICO):** Basic file CICO features are implemented in the *i*FS, which lock documents that have been checked out until they are explicitly checked back in (or an administrator releases the lock).

  This feature implements a very basic version control, but does not fulfill the needs of collaborative work, like merging of concurrent versions or operations on more than one object.

- **Versioning:** When copying a file from the *i*FS to a client or when editing it directly within the *i*FS, a new version of that file can be created. An administrator can also identify one particular version as the official version, hiding all others from "normal" users.

- **Change notification:** When inserting, updating, or deleting a file, the *i*FS can generate an email notifying one or more users of the change.

- **Automatic expiration:** The *i*FS can be configured to purge files from a folder after some time has passed.

The main problem in nearly all of the features listed above is that the functionality is not accessible for standard protocols. For example, there is no way to check in or check out a file using the standard features of the SMB or FTP protocol as they do not support check in or out of files. Using additional features results in the need of specific tools or applications. Though this is a general problem of middleware systems it is mentioned here, because this is a clear drawback in the use of *i*FS.

4.1.2 Oracle Application Server (OAS) 4.0

While Oracle8*i* is a database with additional features, Oracle Application Server is a real middleware not bound to one specific source. As described in [Ora98a], the OAS is situated between any client device that communicates over HTTP or IIOP[7], and any database that supports the ODBC or JDBC gateways. OAS 4.0 supports multiple programming models (Web, CORBA/EJB) and languages (Java, Perl, C, PL/SQL, LiveHTML and Cobol).

Oracle's application server is described more in detail as a representative for all application server with similar functionality. See section 4.5 for a list of applications servers.

An OAS solution has three distinct parts:

- **HTTP listeners** handle all HTTP requests for static pages or CGI calls. If the request is for a dynamic page, it is forwarded to the OAS.
- **OAS (the application server itself)** provides resource management when handling requests for applications deployed as cartridges on the server. It provides a common set of services for managing these applications, including load balancing, automatic recovery of failed processes, security, directory service, and transaction.
- **Application Cartridges**[8] are managed objects or components deployed on OAS.

[7] Please see section 10 for a short description of IIOP.

[8] A *cartridge* is Oracle's historical term for business components, i.e., a package of coded business logic. The terms *component* and *cartridge* are used interchangeably in Oracle's papers.

The OAS architecture allows to be distributed across a cluster of machines and enables different kinds of load balancing (weighted load balancing by applications or dynamic load balancing by system metrics).

Another important feature is how the client interacts with the server. There are three different models:

- **Request-Response Model:** Clients' requests can be sent via HTTP or IIOP. After the result is returned, the connection between the browser and the Web server is dropped, and the context for the interaction between the browser and server is lost. This is called a "stateless" request.
- **Session Model:** This kind of connection is only available to HTTP requests. It is useful in situations where a logical interaction between an end user and an application consists of multiple, interrelated HTTP requests (and responses).
- **Transaction Model:** A transaction is similar to a session in that it usually consists of multiple HTTP/IIOP requests. However, unlike sessions, transactions are used to ensure that interactions involving business-critical information either complete entirely and permanently, or not at all. OAS supports two different transaction models:
  - Declarative transactions must be "declared" at runtime and do not require explicit programming, while
  - programmatic transactions must have been written explicitly into the application's logic using key words that describe when a transactions begins or ends and when it should be committed or rolled back.

Furthermore the Oracle Application Server supports SNMP which enables any applications running on the OAS to be discovered, identified, and monitored by any SNMP based management application.

Another important aspect is security. OAS supports multiple authentication mechanisms including authentication per username/password, domain-based authentication, IP-based authentication, and database authentication where applications are identified using the database user Id and password. Support for X.509 (X.509v3) certificates and storage of

certificates in an LDAP directory were added since version 3.0 of OAS. Secure Sockets Layer v.3 (SSLv3) protects the IIOP traffic from client objects to server objects. This prevents attackers from viewing or tampering with HTTP or IIOP data. For further details see [Ora98a].

A major drawback in the concept of Oracle's Application Server is the lack of different types of connection as it only serves as a middleware layer for relational databases (via ODBC or JDBC). On the client side only Web browsers or clients communicating via IIOP are accepted. For this reason a direct comparison to Dino is useless, as OAS' field of application is quite different to Dino's.

4.2 NetDynamics

Sun distributes the NetDynamics Application Server 5.0 that provides access to databases and other systems via a Web server per HTTP or IOP. The Application Server itself is a multithreaded Java server built on an integrated CORBA infrastructure.

External stores[9] are connected to the Application Server by *Platform Adapter Components (PAC)*, an abstract interface to databases, packaged applications, or legacy software systems. Sun/NetDynamics provides a couple of PACs, like one for Microsoft's Component Object Model (COM), PeopleSoft, and SAP R/3.

Nevertheless is JDBC the major way to connect the NetDynamics Application Server to relational databases. Using this general interface, the server is able to connect to Oracle, Microsoft SQL Server, Sybase, IBM DB2, and Informix. But as JDBC is a very general interface that forwards only the SQL queries as textual strings, it might be that the specialities of the different SQL-dialects cannot be considered when adding new databases to the application server.

On the client/applications side, the only way to access external stores is to use the HTML pages provided by the server or by using a programming language that is supported by the CORBA mapping of NetDynamics (normally Java or C++). Other protocols are not supported. As the fundamental architecture of this server is based on CORBA, it inherits all advantages and disadvantages of this technology. Please see section 3.1.1 for details

[9] *External Store* is the term used by NetDynamics for connected databases.

about CORBA-technology.

The application server can be installed as a distributed server and is then capable of moving pieces of the overall system where they perform best. Dynamic load balancing is performed on three levels: multimachine, multiprocess and multithread.

4.2.1 Components

Among others, the main components in the application server are the following:

- A **Service Manager** handles all services on a single machine, including starting, stopping, and restarting them (in case of failure) and is responsible for configuration of the services.
- The **Security Manager** manages a repository of security access control lists (ACLs) that ensures authorized access to components. Security on the service level consists of authentication, authorization, and optional channel encryption. NetDynamics provides access via SSL (Secure Sockets Layer) and Secure HTTP (HTTPS). For CORBA objects, access can be granted at service, interface, or individual operations level.

  It is not possible to forward security requests to the security manager of the connected database(s), thus existing and tested user rights have to be duplicated for the application server.
- The **Plug-in Server** handles the communication between the application server and the Web server.
- The **Service Locator** represents a naming service for NetDynamics services and enables a client to find the least-loaded service across multiple machines.

NetDynamics provides two optional components:

- The **Gatekeeper**(supplied by Visigenic) provides IIOP access for applets to bypass the sandbox restrictions of an applet in a Web browser. Normally an applet is only allowed to connect to the Web server it was loaded from. As this Web server need

not necessarily be the same as the application server, the Gatekeeper (started on the same machine as the Web server) forwards all requests to the application server. The Gatekeeper acts as an IIOP proxy. The principle of the Gatekeeper is shown in figure 4.1.

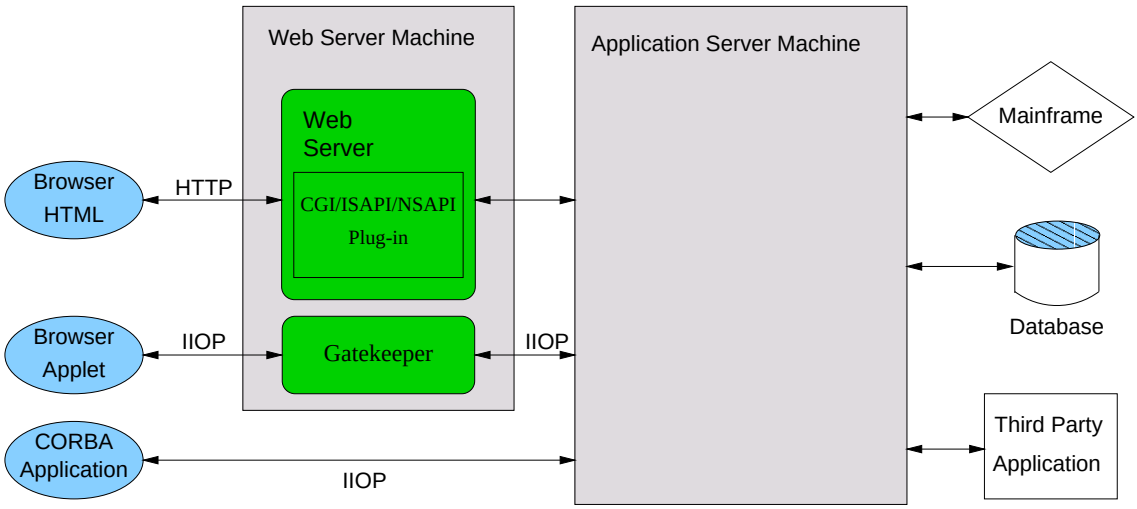


Figure 4.1: The principle of a gatekeeper to bypass the Java sandbox restrictions [Sun99a].

- An **SNMP agent** collects information about the components of the application server and provides them to an SNMP management tool.

4.2.2 Services

A *service* in a NetDynamics server provides a set of operations to applications. It consists of a monitor for external control and a pool of *workers*. A *worker* is an instance of a CORBA object to perform a specific action (e.g. a query in a database). Using statistics about the *workers* a *service* has the possibility to perform load balancing within its pool of *workers* (that is, to direct a client to the least-loaded *worker*).

The standard services are:

- Connection Processor Service (CP): it hosts HTML pages created by the NetDynamics development tool. It executes HTML pages (that means, it executes data

objects and generates HTML pages from that). For all applications that rely on HTML pages, the application code is executed in the CP, while applications written as Java applets are executed on the client side.

- JDBC Service: This is probably the main component in the NetDynamics application server. It is responsible to connect to external relational databases and transmits SQL-queries or stored procedure calls.
- EJB Service: The EJB Service implements an Enterprise Java Bean container. It is used to host reusable server objects.
- Persistence Engine Service (PE): It provides a repository for stateful applications (either HTML or applets). An example of this might be a shopping cart, which must be accessible from all pages of the specific application.
- Global Session Service (GS): A repository for global states (available to all applications).
- Logging Service (LS): A facility for logging debugger information or error messages.
- Debugger Service: Exclusively used by the development tool of NetDynamics it enables remote debugging of HTML pages or applets (AppletViewer Debugger Service).
- Platform Adapter Component Service (PAC): PACs enable seamless integration of custom services in the application server. Custom services might add new functionality, provide bridges to external data sources, or add an additional layer of business logic to a generic data source. So these services work as drivers for additional data sources. NetDynamics supplies a PAC SDK[10] to facilitate development. There are four types of PACs:
  - Procedural PAC: provides an RPC[11]-like interface.
  - Query PAC: provides a database-like interface.
  - Hyper PAC: provides a service intended to be used by other PACs, other applications, or applets.

[10] Software Development Kit
[11] Remote Procedure Call

– Object PAC: provides an interface to distributed or/and remote objects, such as COM components or BusinessBeans.

NetDynamics facilitates the development of new PACs to integrate new types of servers with the PAC SDK. For additional information please read *Using the PAC SDK* or *PAC SDK Reference Guide* both available from NetDynamics.

The simplified structure of a NetDynamics Application Server is shown in figure 4.2.

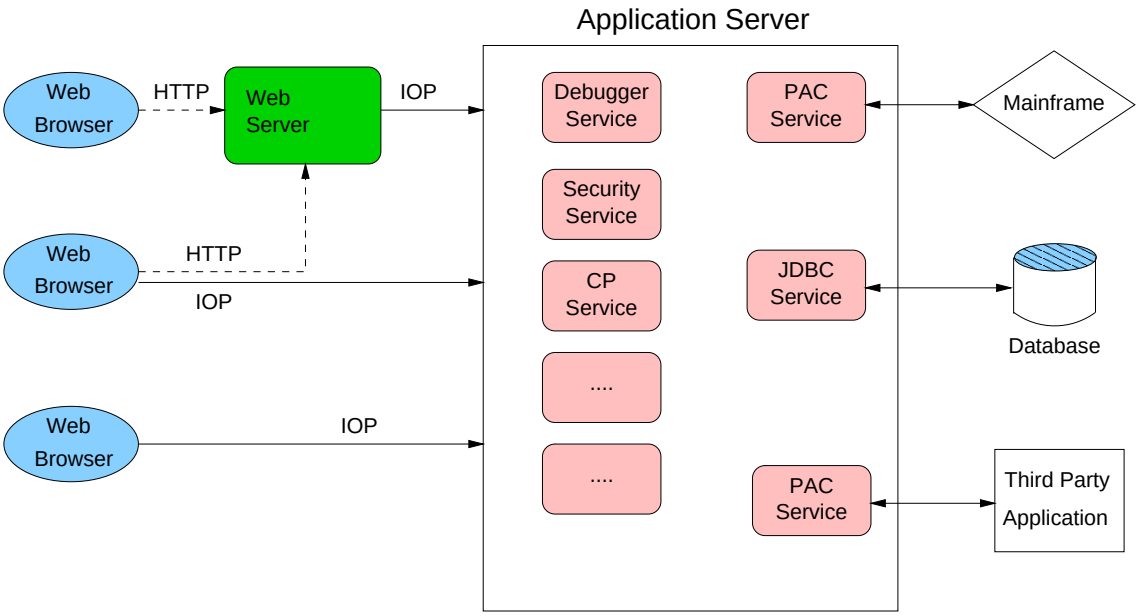


Figure 4.2: The structure of the NetDynamics application server illustrating the connection types and some internal modules. [Sun99a]

All of the information above is taken from [Sun99a].

4.2.3 Development

NetDynamics offers a wizard-based development environment helping to create Java and/or HTML applications. The *Object Framework* [Sun99c] is a Java class library that enables programmers to access the application server's data objects and visual controls.

> The NetDynamics Object Framework (OF) provides the application programming interface (API) used to support, customize, and extend NetDynamics applications. The OF provides classes that encapsulate HTML and Java user interface support, connection management, database access, visual object-to-database binding, state management, reusable business logic, and common utilities. The OF class library is implemented almost entirely in Java, with minimal C++ extensions.

4.2.4 Conclusion

NetDynamics offers a development environment making it possible to easily integrate simple database queries.

A major drawback of the NetDynamics concept is that the clients' requests are more or less directly forwarded to the external stores. Notes like

> **Note:** Not all database vendors support all these types of parameters.

in the Object Framework Overview [Sun99c] give the impression that a general abstraction layer inbetween the client and the external databases is missing. Even if it is possible to integrate different types of external stores, the prime focus of this application server is clearly seen to be JDBC to relational databases.

JDBC might be a good way to connect Java applications to a relational database, but as the SQL dialects vary widely (e.g. from MS-Access to Oracle) and SQL-queries are handed over to JDBC as Strings, the independence from different databases is not really granted unless the common intersection of all SQL dialects is used, which does not really offer a lot. JDBC also has the big disadvantage that the SQL-statements are not checked for correct syntax during compilation of the applications, so bugs are discovered at runtime. This makes it extremely difficult and lengthy to test the applications.

NetDynamics Application Server is a good choice if one wants to make the content of a database accessible to the Web, and the database does not offer the tools to help with this task. Additionally it separates the application's code a little (see above the statement of JDBC) from the database so a change to a different database product is easier, but still not completely painless!

4.3 ObjectSpace Voyager

ObjectSpace's Voyager is more like a distributed object storage facility with an application server put on top of it. According to [Gla99], the Voyager product line consists of Voyager ORB[12] that simultaneously supports CORBA and RMI[13]. The professional version of the ORB integrates JNDI[14], CORBA naming service, COM support, Dynamic XML, persistent replicated directories, transactions, and security features.

4.3.1 Voyager ORB

ObjectSpace's object request broker allows objects to communicate bidirectionally with other CORBA, RMI, or COM programs. It offers a single simple API to access different available naming services, like COS naming, RMI registry, VDIR[15], or JNDI. Additionally, the *universal directory* can be accessed by CORBA and RMI objects and so allows interchangeable objects from the CORBA-world to the RMI-world, both using their native APIs to bind objects. Figure 4.3 shows how objects might choose to use the different interfaces to access either the original service or the *local universal directory*.

Voyager generates all skeletons needed as well as helper classes and stubs at runtime, so creation of distributed systems is simplified a lot. It really seems very easy to create and use a remote object with this technology: When an object is bound to a remote naming service, a proxy object is returned that implements the same interface as the remote object. If the proxy class does not already exist, it is automatically created at runtime. Messages sent to the proxy are automatically forwarded to the remote object and the result is returned to the local client. The major advantage of this way is that neither stub generators were needed, nor were any auxiliary files/classes created (by the programmer!). For a simple example, see [Gla99, page 10].

[12] Object Request Broker
[13] Remote Method Invocation
[14] Java Naming and Directory Interface
[15] Voyager federated directory service

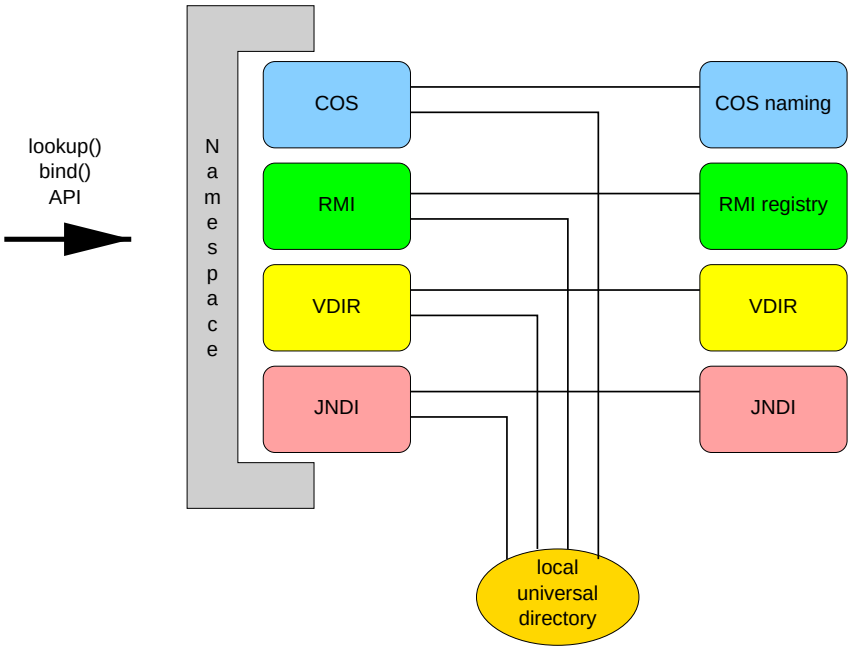


Figure 4.3: Universal naming service of a Voyager ORB [Gla99].

4.3.2 Voyager ORB Professional

The Voyager ORB Professional adds some important features to the basic ORB. Among them are:

- **Dynamic XML:** A tool reads any XML DTD[16] and generates a set of Java interfaces that correspond to the elements of the DTD.
- **Universal Gateway:** Voyager bridges protocols between clients and servers. E.g. a native RMI client is able to send messages to an object in a native CORBA server, even though RMI does not support IIOP.
- **JNDI Integration:** The *universal directory server* is made accessible using the Java Naming and Directory Interface.
- **Persistent Replicated Directory:** All local naming services benefit from this persistence (CORBA, RMI, and the Voyager directory service).

[16]Document Type Definition, information about the structure of a XML document

- **Security:** Network communication is encrypted and authenticated using SSL and applications are able to tunnel their information through firewalls using HTTP or standard SOCKS5 protocol. It also provides a framework for developers to integrate existing enterprise security. See also section 4.3.4 for a critical statement to this point.

- **Transactions:** Voyager Transactions is a CORBA OTS[17]-compliant distributed transaction facility. Using a two-phase commit Voyager allows that multiple resources participate in a transaction across multiple Java Virtual Machines. They support both JDBC resources (one-phase commit) and JTA resources (two-phase commit).

4.3.3 Voyager Application Server

Voyager Application Server offers an Enterprise Java Beans (EJB)[18] environment built on top of the Voyager ORB Professional. Following clients are supported: CORBA, RMI, Microsoft's Distributed Component Object Model (DCOM)[19] (in the near future), and HTTP (Web browsers).

4.3.4 Conclusion

The only weakness of ObjectSpace's Voyager is its authentication - only a reference implementation providing access control with a username/password based on an authentication mechanism that uses a internal userlist, passwordlist, and access permissions is shipped with Voyager. No other implementations are supplied to access e.g. an LDAP server for authentication. Nevertheless this functionality can be added by using the framework provided, but it has to be developed by the customers themselves.

Prafulla S. Deuskar used ObjectSpace Voyager to implement a scheduler agent and as a conclusion he writes in [Deu] that

> Voyager provides powerful features that can support mobile as well as dis-

[17] Please see section 3.2.1 for further details about the Object Transaction Service.

[18] `http://java.sun.com/products/ejb`

[19] `http://www.microsoft.com/com/tech/DCOM.asp`

tributed applications. It has an API that is intuitive and clear. It is also fully compatible with other distributed object models such as CORBA and DCOM.

4.4 ActiveWorks Integration System

ActiveWorks Integration System of ActiveSoftware[20] is less an application server but a real middleware system. Its main focus is to connect different types of servers or applications via so called *Adapters*.

Adapters do not communicate with each other but they send and receive events from the *Information Broker*[21]. They are dedicated to one application or system and understand its API or protocol and the API's correspondence to ActiveWorks events. Like this they provide a bi-directional pathway between the native format and the Information Broker. Adapters may vary in dynamism:

- **Static Adapters:** Behaviour is "hard wired". When an event arrives, the Adapter invokes its application, usually passing data from the event. If the event is a request, the Adapter delivers a new event containing data returned by the application. Changing a static Adapter has to be done by updating the source code, compiling, testing and deploying the changed Adapter.
- **Configurable Adapters:** These are the simplest of the dynamic adapters. The adapter developer defines in a predefined range how the Adapter reacts on given events. It then looks for configuration parameters during initialization that tell it which events to subscribe to, and which fields to use in which API calls.
- **Reconfigurable Adapters:** These are configurable Adapters that update themselves on the fly. So configuration is not done at initialization any more, but can be done during runtime. So it may react on new events or new fields. Nevertheless reconfiguration is done on request from the Information Broker.
- **Programmable Adapters:** In addition to loading configuration data at runtime,

[20] `http://www.activesw.com`

[21] Please see figure 4.4 for an overview.

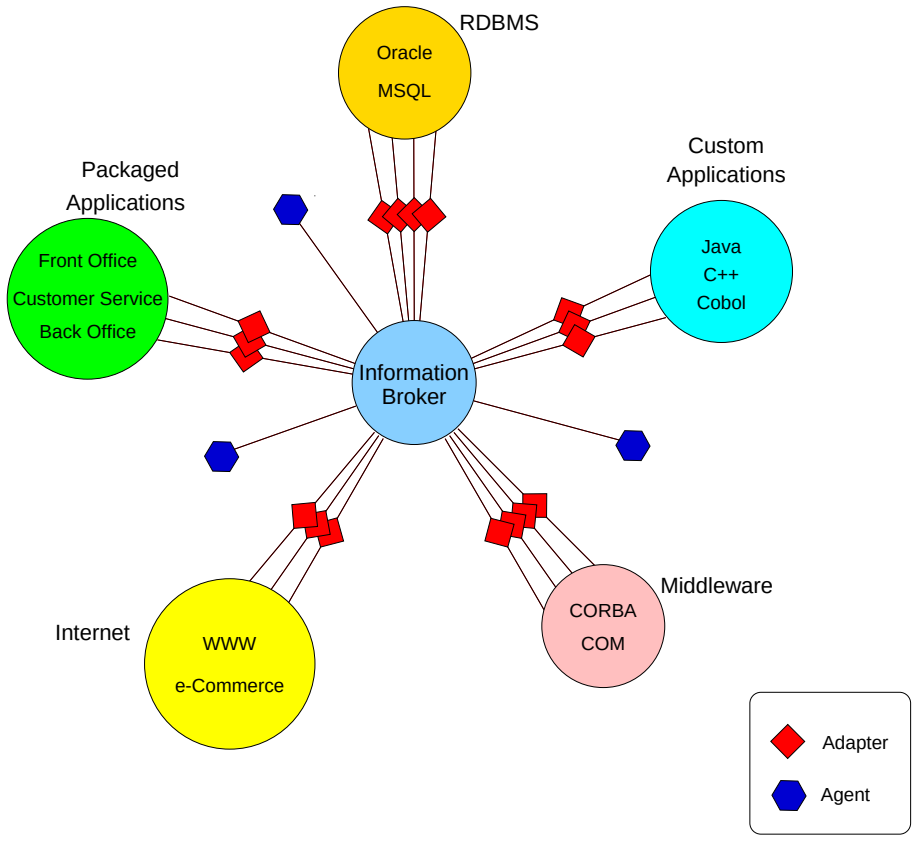


Figure 4.4: ActiveWorks Integration System connects to different systems using Adapters. [Bra99]

a programmable Adapter loads code. The Java language makes programmable adapters feasible on many different platforms.

Another way to extend the system is by using *Agents*. These are stand-alone programs that react to one or more event types. ActiveSoftware provides packaged agents for customization. These are specialized and exist in three different flavors:

- **The Integration Logic Agent** is an extensible Java framework and is the Agent analog of a programmable and reconfigurable Adapter. It maps event types to classes or beans.
- **The Business Rule Agent** is an advanced framework for developing knowledge

based Agents. It is "programmed" by rules, so it is customizable by technically oriented analysts whose expertise is domain knowledge, not conventional computer programming.

- **The Data Transformation Agent** is specialized for converting complex data from one format to another. It might be used to transform a database query to XML.

Additional information or further details can be obtained from the Web site of ActiveSoftware or in [Bra99].

4.4.1 Conclusion

The concept of this product is quite interesting. It connects different applications, so they can interact without the need of a common interface. As these connectors (adapters) are highly customizable, they might serve as a source of documents or data as well as to provide documents to other applications.

A major drawback is that the whole system is based on events. So the focus of the system lies on actions, not on the data achieved by these actions. Objects returned by a request are not directly addressable in the external applications and are identified only through the call they were delivered from.

Nevertheless is the system a highly flexible one and features like the addressability of data might be added through customization of Agents or Adapters.

4.5 Other Application Servers

A good overview for application servers can be found at Serverwatch from internet.com[22].

In the area of Application servers a lot of different companies distribute similar systems having more or less the same functionality. They all provide connections to databases (Oracle, AS/400, ...) via a Web-based interface or CORBA classes and consequently do not really differ from the products above, so they are not discussed more in detail here:

[22] `http://serverwatch.internet.com/reviews/app-reviews.html`

- Netscape Application Server[23]
- Inprise Application Server[24]
- Sybase Enterprise Application Server[25]
- Sonera Application Server[26]
- IBM WebSphere[27]
- BEA Weblogic[28]
- Bluestone Sapphire[29]
- Galileo[30]
- Interop Server[31]
- Silverstream[32]
- Vision Business Logic Server[33]
- Apple's WebObjects[34]

Some other middleware products sounding interesting were found on the Internet, but due to their stage (pre-alpha) or due to other circumstances not a lot of information could be found.

- DataGate[35] supports quite a lot of protocols (among others FTP, NFS...) and is the first one that runs the site's Web server with its own software! Unhappily, there is nothing but a short overview available.

[23] http://www.netscape.com/appserver/v2.1
[24] http://www.borland.com/appserver/
[25] http://www.sybase.com/products/easerver
[26] http://www.sonera.fi/english/solutions/datavoice/application.html
[27] http://www-4.ibm.com/software/webservers
[28] http://www.beasys.com/products/weblogic/server/index.html
[29] http://www.bluestone.com/products/sapphire/
[30] http://www.esemplare.com/galileo.html
[31] http://www.intertop.com/isuite/iserver.html
[32] http://www.silverstream.com
[33] http://www.vision-soft.com/products/bls.htm
[34] http://www.apple.com/webobjects/
[35] http://www.stc.com

- TIB/ActiveEnterprise[36] also has quite a good concept and uses adapters to connect to external systems (mainly to SAP R/3, Oracle, PeopleSoft, ...). There is an adapter SDK available (only for C++) which uses an UML/XML-based meta-data representation.

- webPump[37] is a "Java-based database distribution and management tool designed to enhance data access over the corporate intranet", but seems to be pre-alpha, so no information is available at the Web site!

[36] http://www.tibco.com
[37] http://www.mtools.com

Chapter 5

Implementation

This chapter explains the general concepts of Dino and discusses different aspects of the integration of heterogeneous server systems.

Integrating external systems or providing information for different systems (External Gateways) leads to a variety of problems. Some of them concern only specific protocols, others are general problems found during the development of Dino.

5.1 General Overview of Dino

The design of Dino is layer-oriented and modular: figure 5.1 illustrates that if Dino is cut apart vertically, the layer-model can be seen.

Slicing it up horizontally shows that each layer consists of different modules. Some of them can be added/removed (even during runtime), others are absolutely necessary for operation, but nevertheless replaceable. An embedded system would be an example of a module that can be added or removed during runtime, while the Security Manager could be replaced by another implementation, but not during runtime as it is absolutely necessary for the system.

As the description of the complete internals of Dino [IIC99a] would go beyond the scope of this thesis, only a short overview of the layer model is given here:

Application Layer Applications using the API provided by Dino reside in this layer.

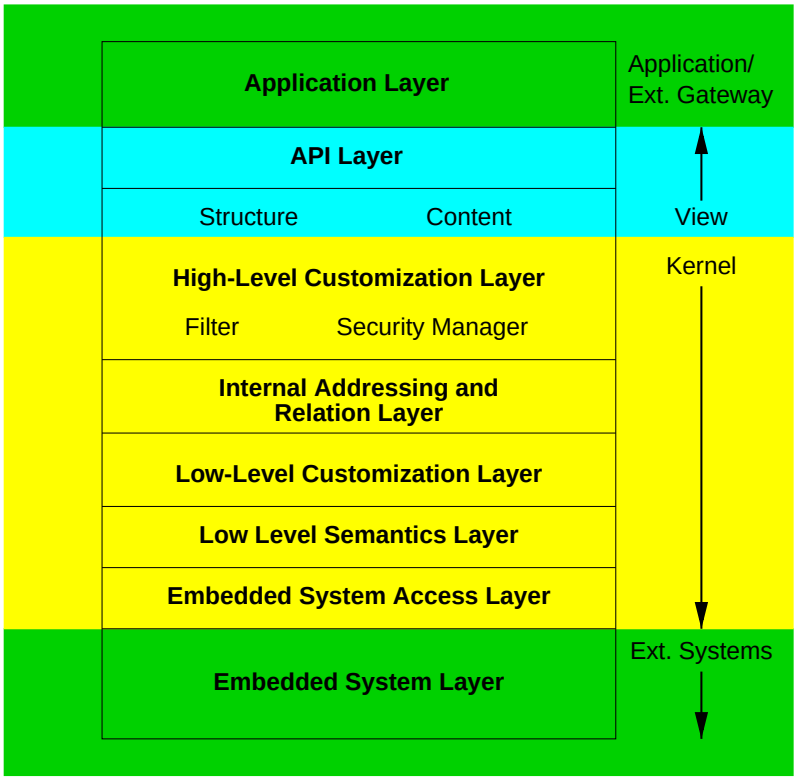


Figure 5.1: The Dino layer model shows the middleware abstraction layers between the clients and the embedded server systems.

This includes External Gateways that provide the content of the Dino system to (network) clients by the use of a specific (network) protocol. Gateways that offer a different API are imaginable as well, for example a simulation of the Java class `java.io.file`, so Java applications can be adapted easily to read/write from Dino instead of the local filesystem.

API Layer This is the common point of access for all applications. The API has functionality to administrate Dino and its embedded systems, request objects from the embedded systems, and manipulate these objects.

Additionally, Views create their representation of the system's structure in this layer. Depending on the View, different objects and their relations are visible to clients. There might for example exist one View for hierarchical relationships that uses only

Dino Relations of this specific type. Another View, using the "successor-predecessor" relations, gives a graph-like impression of the system.

High-Level Customization Layer The top-level layer of the Dino Kernel holds the Security Manager to check the rights of all operations invoked in the API-Layer and gives the system administrator the chance to add some high level customization filters (therefore the name of the layer) to the system.

Internal Addressing and Relation Layer Navigation in the Dino system is only possible by use of Relations, as the addressing scheme does not provide any information about relationships between objects. Therefore, the Internal Addressing and Relation Layer is responsible to map Relations to Dino addresses.

Low-Level Customization Layer This layer is responsible to combine several embedded systems to a bigger system inheriting functionality from all of the combined systems. This can be used for example to add meta-data support to a filesystem that is not able to store meta-data.

Low-Level Semantics Layer This level is an abstraction layer for the underlying embedded systems and maps the semantics of an embedded system to be used by Dino. For example the hierarchical structure (parent-children relationships of directories and files) of an embedded filesystem are mapped to the appropriate parent-children Relations in the Dino system.

Embedded System Access Layer The low-level driver[1] provides information about the embedded systems and encapsulates the access protocol/API of the external system. No interpretation of data is done in this layer, only basic in/output functionality like "write data" is provided.

Embedded System Layer Here the external systems reside. The only requirement they must fulfil is that they must be accessible by a network protocol or a programming interface (API).

Comparing the internal design model to other middleware systems is not trivial, as systems differ widely or internals are unknown. ActiveWorks for example, uses a completely

[1] The modules in this layer a referred to as "embedders" in this chapter.

different approach. Adapters[2] are not declared to be "client" or "server", although they might fulfil the task of a server or client. An Adapter serving HTTP clients is seen from the Information Broker equally righted as an adapter responsible for database queries, even though they have completely different jobs to do.

5.2 History of Dino

The development of Dino was started in the year 1996 at the IICM. Since then, it was continuously approved, changed, ripped apart, and put together again. In every phase the Dino system was successfully used in other projects of the IICM and these experiences flew back into the design process of the next version. The writer of this thesis started to use and to help modelling Dino from version two onwards.

5.2.1 Dino V1 - Message Based

The original goal of Dino was to facilitate navigation on different kinds of information systems. It wanted to relieve the applications from knowing about different protocols and to create a meta-protocol with a superset of features of all underlying protocols. If a new protocol was to be added to the system, just one driver would have to been written and all applications using the meta-protocol could have immediately used the new protocol without having to rewrite the whole application.

The first version of the Dino library was message based. Any internal or external events put a message in one global message queue and any interested components in the system listening to that queue could react. This design which is still used in a lot of systems (Windows[3], Sun's jini[4], Smalltalk, ...) was also the weakest point in the system. As all internal tasks were achieved through the use of messages, higher complexity of the system resulted in a vast amount of messages, so the performance became unacceptable and the overview got lost.

Drawbacks using the message based system:

[2] Adapters are responsible for handling the communication to the embedded systems.

[3] `http://www.micrososft.com`

[4] `http://www.sun.com/jini`

- Messaging was abused to define the structure of the objects on the server (e.g. the tree structure when using a file system or an LDAP server).
- Messages could not be kept robust. If a message had to be changed, all applications had to be updated.
- There was no central place where messages were defined.
- The applications or the users are required to define the data-structures. But in Dino V1, the system already had unchangeably predefined them.
- The superset of all integrated protocols is not a given standard, but open to new features. So it is extremely difficult to foresee what will be needed. Adding new protocols having completely new possibilities was difficult due to limits in the design of Dino.
- The meta-protocol was defined to be a protocol, but the usage was more like an API.

5.2.2 Dino V2/V3 - Tree Model

The next stage in the development of Dino eliminated the messages and to took a tree as the main underlying data structure. Another possibility would have been to use a graph, but as this concept is always kind of confusing and bad experiences had been made (e.g. multiple parent) this idea was dropped.

A tree-view was easier to understand and much more common in the programming community (e.g. file system). To keep the API as close as possible to well known interfaces, it followed the Java class that is used for file access[5] as close as possible. Different (network) protocols could be used, but the underlying data structure was always mapped to a tree. In figure 5.2 the layer model of the Dino V2 is shown.

The Dino Virtualization Layer is responsible to prepare any structures coming from the underlying protocols for the mapping into a tree.

As there exist structures that cannot be mapped to trees, a link management was added to add graph-like functionality. This link management was also used to create relations

[5]Java class `java.io.file`

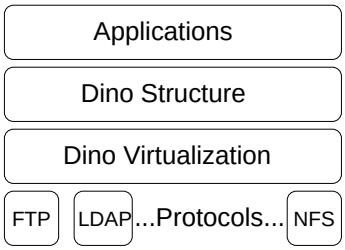


Figure 5.2: The Dino V2 layer model: The underlying systems are "normalized" and mapped to a tree structure. This tree-like data structure is then available to applications by the use of an API.

between two objects having their origin in different protocols. Like this, links of any type (e.g. annotation, reference, ...) could be made, even from a file originating in a filesystem on the local hard disc to an LDAP entry residing on a remote server [Blü99].

New features of this version of Dino were:

- A Dino specific low level protocol was introduced. Using this Dino-Dino protocol it was possible to connect two Dino systems in a peer-to-peer manner.
- Globally sent messages were replaced by events. These were only sent to specific destinations to minimize message traffic. This approach is also used by various other middleware systems like ActiveWorks or NetDynamics.
- To add special functionality not covered by system events, *Dino methods* were introduced. They enabled applications to invoke network transparent methods of (remote) modules.
- *Dino services* were visual representatives of Dino Methods. As Services as well as Methods could be sent over a network connection (Dino-Dino), the first steps from a distributed file system to a distributed application were taken. Services made it easier for applications to interact with the system, because the application could use components provided by the system, but as new Services could also be provided by the application they made it harder for the system to keep control.

Though this system was quite adaptive to new protocols to be embedded, there were clear drawbacks in some of the design decisions made.

- The mapping of all structures of underlying systems to a tree-like structure was a limitation which could be bypassed by the use of the link management, but this solution still was a bypass that did not satisfy all needs.
- The addressing scheme was closely related to navigation. This prohibited the creation of unique Ids, as it could not be guaranteed that they would be human readable.

The design decision to map all structures of underlying systems to a tree was too limiting and

Dino V3 was just an interim version for IICM internal use and is mainly identical to Dino V2.

5.2.3 Dino V4

Dino V4 is the current version and still under development. Most of the problems discussed in this thesis were taken out of the design process of this version. Some of them are unsolved at the time of writing and are presented in chapter 6 'Conclusion and Outlook'.

In the following sections, different aspects, ranging from the "first contact" to an external system to the tasks to be performed to provide the content of these systems to (network) clients, are discussed.

5.3 Embedding Systems

Embedding external systems into an environment that abstracts a variety of accesses and functionality poses some problems. The most mattering of them are discussed in this section.

5.3.1 Connection

The first step to integrate an external system is to create a connection to it. In case of a filesystem this might be an easy task that the operating system has already accomplished, but at least all systems accessed across the network need an explicit connection. Mostly connecting to a foreign system implies authentication by the user or by the system.

Generally, there are different possible combinations of connections and authentication that Dino must be able to handle:

- **without any authentication:** Taking the number of connects made in the Internet these days, this is the majority, as an HTTP-connect is made usually without any authentication. Other examples would be a connection to a SMTP server or to the filesystem.
- **authentication using system parameters:** Some systems authenticate others on system level, so the permission to connect may depend on the IP-address, the subnet, or even the time of day. Examples for this kind of authentication is an NFS server, a WWW-proxy, or a firewall. The server configuration indicates which systems are allowed to connect or denied.
- **authentication on user level:** These systems decide, if a connection will be allowed by demanding a username password combination or similar methods.
- **combinations of the above:** It is possible that a system first checks if the foreign system is allowed at all, then establishes a connection and allows anonymous/guest access with restricted possibilities, and finally might provide a possibility to change the users' status from anonymous to identified. As an example a Hyperwave server might be taken.

As can be seen from above, even the first act to connect or to embed an external system is not a straightforward process and offers different ways to be accomplished.

If the system does not require authentication at all, the task is easily solved or at least authentication is not the problem. If a server identifies a client (The embedding system is seen as a client from the external system) by system parameters, it might accept a connection or not. In either way, our system might not influence the server's choice as system parameters are not to be changed by our system.

So the main problems in this context are authentication on user level and changing the authentication after a connection has been established. For possible solutions to this problem, please see [Hau99].

There are some problems to be solved concerning connection and timeouts, as can be seen in the example of a Post Office Protocol Version 3 (POP3) server: This kind of server has a very short (10 minutes) timeout [BLFIM98]. All embedders have to be able to handle connection-timeouts. Embedders can be configured to reconnect on their own after having lost the connection or to wait, until they are told to do so. In both cases it is desirable that the former status is reestablished and no data is lost.

The second problem with the POP3 server is that it usually does not allow multiple connections under the same username. But as there is only one connection, it is impossible to execute more than one command at a time, e.g. read one email and delete another one at the same time.

Connecting to a relational database needs a very detailed configuration (Please see section 5.3.4), which might even ask for interaction with the user. So the embedder modules must be built extremely flexible to support all possible cases.

5.3.2 Security

The first aspect of this topic (security issues at connection time) was already discussed in section 5.3.1. But even after having embedded one or more external systems, the security aspect must not be neglected! It must be assured that every access concerning internal objects, data on external systems, or operations on either of them must be checked and then be allowed or denied.

Security Manager

The Checking is done by the *Dino Security Manager* which is part of the Dino Kernel. The layer model of the Dino Kernel in figure 5.3 illustrates that the Security Manager acts as a filter between the layer where the user accesses the system and the underlying Dino Kernel. This concept guarantees that no operation request reaches the Kernel without being checked by the security manager. The security manager determines which user set off the request and then calls the user manager to compare the request with the set of rights of the given user.

The user identification is done through thread-identification: Every thread is identified by

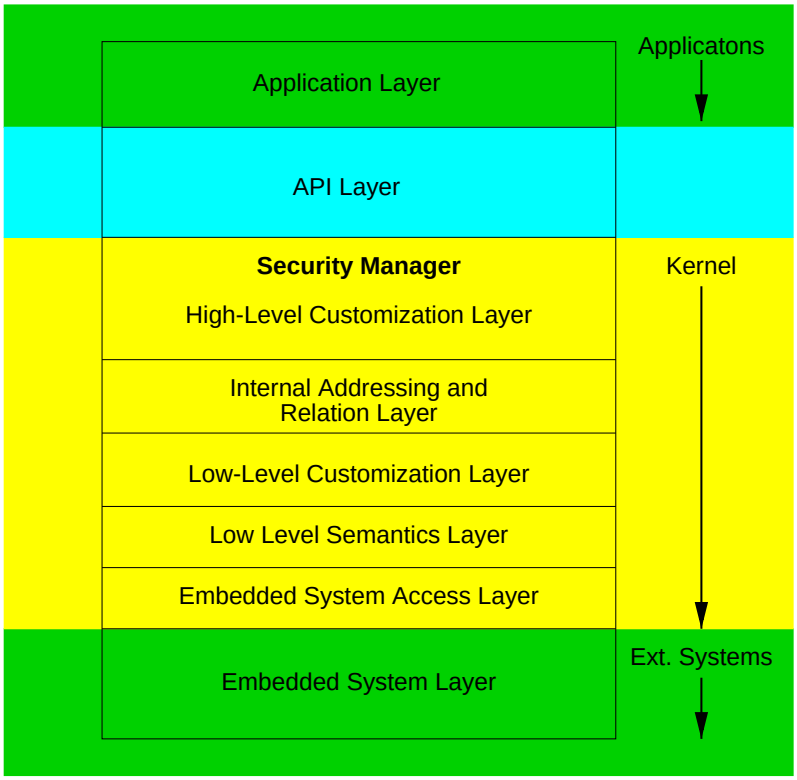


Figure 5.3: In the Dino Layer model, the Security Manager resides on the top of the Kernel, so it cannot be bypassed by any operation requests.

a unique Id and this Id is mapped to a user-Id/name. As the security manager is called inside the very same thread that requested the operation to be executed, it suffices to find the Id of the current thread.

Finding the user-Id that belongs to the thread without searching linearly in a list asks for a map and a (unique) key, but the Java implementation of a thread does not offer anything usable as a unique key.

One solution found was extending the thread class and adding a unique Id, but this has the drawback that all applications wanting an identified thread would have to use this special thread class instead of the 'normal' java threads. So the better choice is to use the hash code of the thread as the primary key, which can be searched for in a logarithmic time and only if there are two different threads returning the same hash value, the reference is

taken into account. By using the combination of hash value and reference it is absolutely ensured, that all threads can be mapped to a user unambiguously and as fast as possible.

A new Java2 feature[6] ensures that threads created from inside another thread can be given an Id automatically.

Callback from the Kernel

Dino offers the possibility to inform any registered listeners of events, like removal or changes of an object. In this case, the Kernel calls code written by any user. Using only the thread-identification method described above would result in user code being executed with system privileges. So the Kernel thread is reidentified as the user that registered the listener before and reidentified as Kernel after the call of the listener.

User Management

As the description of the complete functionality of the user management in the Dino system is beyond the scope of this thesis, only a short overview will be given here. For more details, please see [Hau99].

Dino uses access rights that can be assigned to users/groups, content-objects or to both. They are stored as Dino-Relations of a special type. This allows a combination of access control lists (ACLs) and capabilities [Hau99]. Using an access control matrix as shown in figure 5.4, access control lists may be seen as all non-empty cells of a *column* of this matrix and capabilities may be seen as all non-empty cells of a *row*. [Mil92]

Additionally, user rights are defined as rule-sets, using arbitrary attributes of the document and/or of the user and a wide range of operators.

Figure 5.5 shows the capabilities of rules. The first rule given allows all doctors and nurses and the author(s) of the given document to read and to write it during daytime, the second rule denies any write or delete access except for users belonging to one of the groups 'admins', 'co-admins' or 'doctors'.

By using a combination of ACLs and capabilities and additionally the highly dynamic

[6] Java class `java.lang.InheritedThreadLocal`

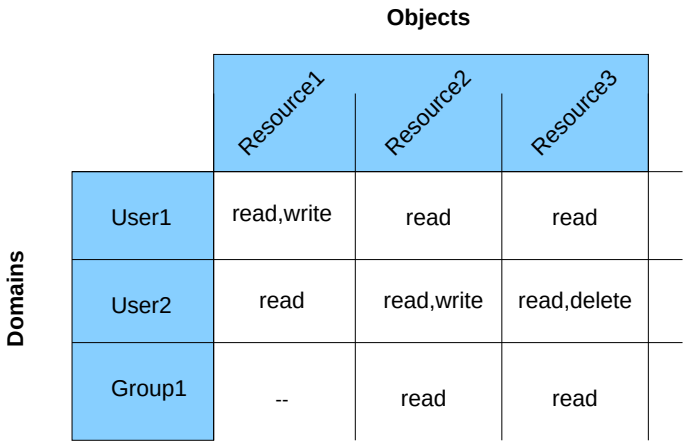


Figure 5.4: The access control matrix is used to map userrights to objects and vice versa.

```
ALLOW (read, write)
  ((user.metadata.profession IN ('doctor','nurse')
  OR (user.username == document.metadata.author))
  AND ((system.time > '6.00') AND (system.time < '20.00')))

DENY (write,delete)
  (user.group NOT IN ('admins','co-admins','doctors'))
```

Figure 5.5: A rule set for user rights allowing read/write access for the author, doctors and nurses during day time and denying write/delete access for all users except administrators, co-administrators and doctors.

concept of rule based access rights, it is possible to define a fine-grained system of access rights.

5.3.3 Relations

Relations helped to solve lots of problems during the design process of Dino. For this reason, a short description of this extremely flexible technology is given.

Following the Dino Software Engineering Document ([IIC99a]), Relations represent "interconnections" between arbitrary objects or even parts of objects (which are object again)

in the Dino system.

Relations are typed to describe the species of the interconnection. The type of a relation between two objects in an embedded filesystem might be *parent-children*, whereas a link connecting two documents could be *annotation*. It depends on the source of the relation, if a type can be arbitrarily chosen or if it is defined by the Embedder[7] of the external system:

- **Embedded-System-Immanent Relations** exist because of a certain structure that an embedded system supports natively, e.g. parent-children relations in a filesystem. These relations are persistent per definition and need not to be stored explicitly.
- **Explicit** or **logical Relations** are attached explicitly to objects. Explicit relations can be stored persistently, but it depends on the configuration, in which embedded system they are stored. It is possible to create a relation between two objects residing in an embedded filesystem but to store the relation itself in an embedded relational database server.

Please note that a specific relation can be *system-immanent* on one embedded system and *explicit* in another. As an example take an embedded filesystem which per-se has parent-children relationships because of its storage structure. Further consider an embedded relational database server that does not implicitly have the same storage structure. Therefore parent-children relationships have to be made *explicit* on the relational database server and are stored in a different location.

Storing the information about the relation not inside the system where the end-point-objects are located, but in a second, independent system (described more in detail in [Blü99]) allows the user to create relations on systems which do not even support relations at all or to create relations between objects located on the two different external systems.

[7] The *Embedder* is the module responsible for embedding the external system, not the user that invoked the embedding process.

Stability

After moving an object to a different location, the relations linking to/from the object must not be broken - so following these relations must still lead to the desired object.

Relations store the unique handles of their source and destination and as long as these are not changed, the relations are stable. Theoretically a unique handle never changes and is independent of the object's location. In practice however, some external systems are not capable of designating a unique Id to an object. A FTP server for example uses the path and the name of a file as an Id. Whenever a file is moved, the Id changes.

While system-immanent relations are stable per definition, explicit relations need additional effort to keep them stable whenever the unique handle changes. In these cases, a mechanism provided by Dino informs all Relations pointing to/from the changed object to update their source/destination handles.

5.3.4 Structure

One of the main problems in the development of Dino was the differently structured data on external servers. Some servers hold their data in a graph (e.g. Hyperwave), some provide them hierarchically (e.g. HTTP, filesystem), and others use tables (relational databases). The first approach of holding all objects internally in a tree-like datastructure[8], and using this structure for navigation as well as an addressing scheme, was too restrictive and had to be changed.

In the current version of Dino, any kind of external structure is modelled using Relations and the addressing scheme is completely independent from the structure. Please see section 5.3.10 for a detailed discussion about the addressing scheme and section 5.3.3 for a description of Relations in the Dino system.

Nevertheless is the process of embedding a relational database still not a trivial task. Relational databases store data in tables. If the Dino system wants to model the structure of more than one table in the database, it has to know the exact coherences of the tables. Figure 5.6 shows a simple address database.

Embedding a combination of tables into the Dino system requires the knowledge about

[8]Please see section 5.2.2 for the historical development of Dino.

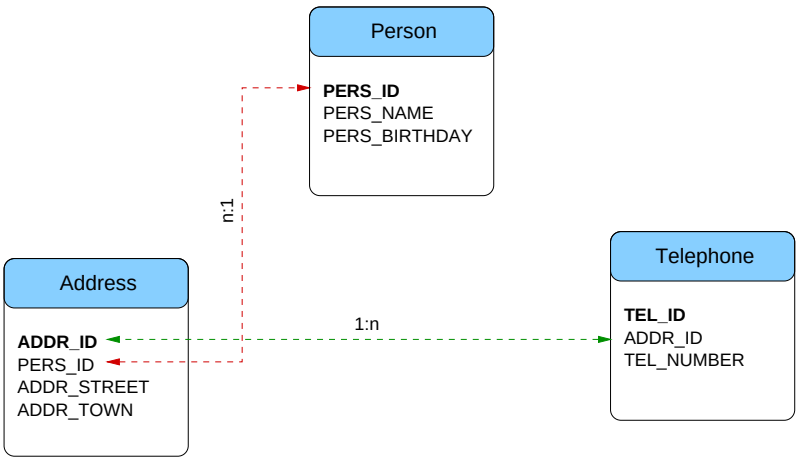


Figure 5.6: Even simple relations between three tables of a relational database need additional configuration.

the primary keys used to express the relations within the tables. This configuration information must be given to the embedder to allow the correct creation of Dino Relations. A general database embedder might detect relations within tables without any external help, but this cannot be assumed generally. For a completely unknown database the embedder might offer help to the user, but usually additional manual configuration is needed. This configuration can be done interactively at connection time or predefined for a known database.

5.3.5 Access Type

Using an (external) system requires any kind of physical access, either

- an API (e.g. LDAP is usually accessed via an API, because the underlying protocol is too low level to be used directly),
- a network socket (e.g. HTTP),
- or by using the system's resources directly (e.g. filesystem).

Often, more than one solution is possible, as normally for every network protocol, there exists a programming library offering an easier-to-use interface. The point is, that these

three methods require completely different approaches when transferring data.

Block vs. Stream

As Dino is designed to be network transparent, information is primarily sent as a byte-stream. This poses the problem that streams need to be converted to data-structures (for usage with APIs) or blocks of data (for file operations) and vice versa.

Figure 5.7 shows an external NFS gateway providing NFS clients with the content of embedded Dino servers.

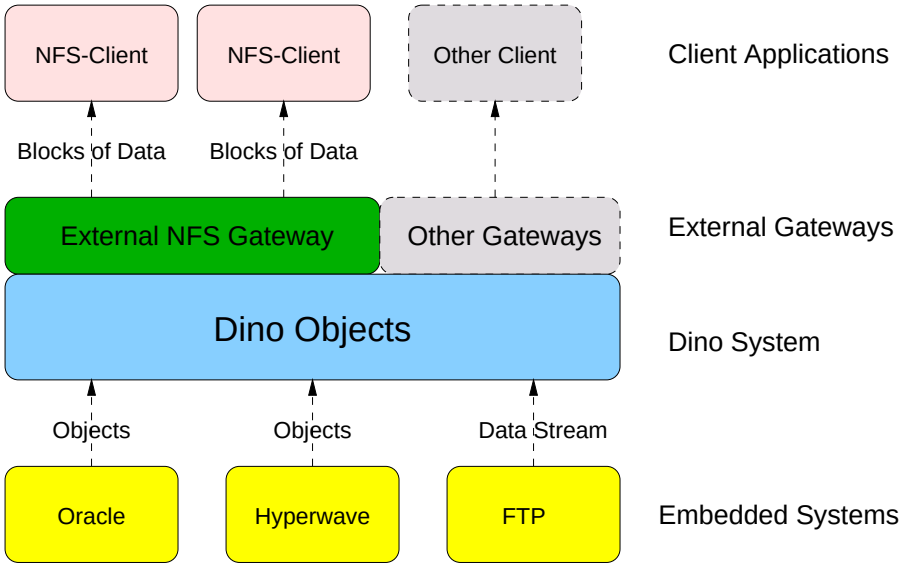


Figure 5.7: The Dino system transforms data structures from external systems to appropriate access types needed by clients.

The NFS clients read and write data block-wise, while Dino communicates with its embedded systems via an API using objects, via a network socket using a byte-stream, but very rarely using the random access operations the operating system offers for filesystem access. Therefore data blocks have to be buffered and converted from one type to the other.

Multiple Access - One Connection

In a multithreaded and multiuser system like Dino more than one user might access different objects of an embedded system. Using a network protocol as a connection to an external server inhibits multiple accesses at the same time. Even if more than one connection is used, this does not always solve the problem, as for example a POP3 server usually denies another connect when identifying with a username already logged in. In this case, the complete embedded POP3 server (or at least the connection for the given user) would be blocked by one request. In the worst case, the client hangs while downloading an email and so the POP3 server is busy and unreachable for this user, not even opening another connection would solve the problem.

As Dino hides the embedded systems behind a general interface, the user cannot be aware of the problems of a specific embedded server. At the time of writing, this problem is still unsolved.

5.3.6 Order of Commands

Different server systems sometimes require a different order of command execution. As an example, the creation of a document and the setting of meta-data can be taken: on an LDAP or a Hyperwave server it is impossible to create an empty LDAP-entry or an empty Hyperwave-document. These two server systems require a minimal meta-data set, otherwise the creation of an entry/document is denied. On the other hand, the process of creating a file in the local filesystem only takes the name of the file as a parameter. If the file should contain any additional information, the user has to do this in a second step.

The Dino system, respectively the module responsible for embedding the system, hide these differences, so the user doesn't need to know the order of commands for the given embedded system.

5.3.7 Content

All servers provide some kind of information to their clients. They all do it using different ways, following different standards or even worse, defining their own "standards". Dino tries to unify the set of different formats to a general document model. Taking this general

model, other document formats can be easily provided by simple conversion. The general document format used internally must not loose any information provided, so a model is used, that allows a broad range of extensions, but still is very well defined and supported.

Object Model

The Dino object model follows the Document Object Model (DOM) specification [W3C98] of the W3C[9]. A short summary of the specification is given in section 3.5.

The usage of DOM as a mediator for different document formats (illustrated in figure 5.8) can be compared to the use of Dino as a mediator for different protocols. The first enables the user to convert documents from a specific format to another, the second "converts" protocol commands or API calls.

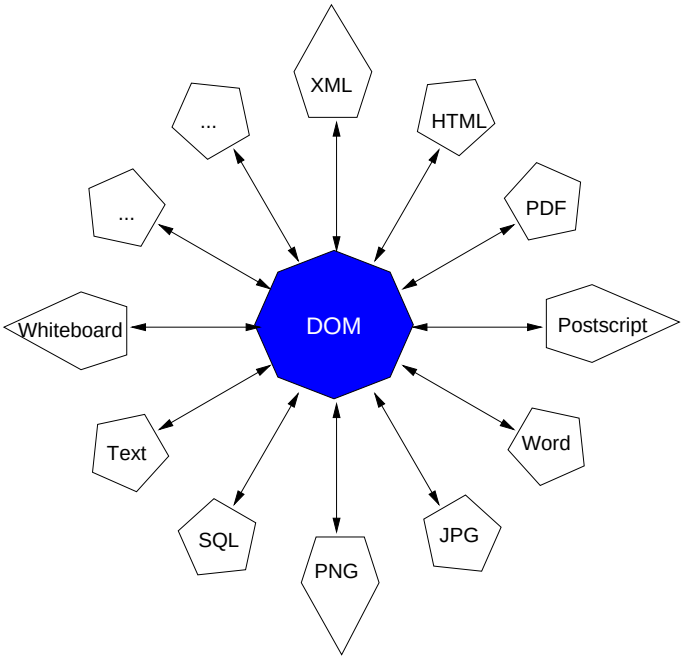


Figure 5.8: DOM, a mediator between different document formats

DOM and XML are closely related, so the use of this model offers Dino the wide range of XML tools and broad acceptance in the server world. Additionally, converters for

[9] `http://www.w3.org`

XML to/from any proprietary document formats like Microsoft's WinWord or Staroffice[10] are provided by the companies themselves, so integration of these formats can easily be achieved.

Active Content

Objects managed by the Dino system can be actors, not only passive data. The whiteboard shown in the medical report (figure 1.2) can be taken as an example. It enables users to communicate and to exchange text or graphics. Despite of the activity, the Document Object Model can still be used. The only difference to "normal", passive documents is, that some of the objects in the document object structure are communicating with other (active) documents. The use of the same Document Object Model ensures that even dynamic content can be converted to any supported document format at any time.

Content Type

Some clients (e.g. a Web browser) or some server systems (e.g. a Hyperwave server) need to know the content type of a document for being able to handle it correctly. On the other hand, there is only a very small number or systems that provide this information. Dino might use this information, but generally the detection of the content type has to be done by Dino itself. For this task it uses a magic number test, like the `file` command on Unix systems: based on specific series of bytes or string occurrences in the document, a content type is determined. For a short example of a magic number configuration, please see appendix A.

The magic number test is a very reliable method to find the content type, but it is not usable in all cases. If servers allow the documents to be read exactly once, this test cannot be applied, as the determination of the content type would make it impossible to read the document afterwards - it would be read for the second time then. Another problem occurs, if the server providing the document to be tested is very slow or does not allow reading a part[11] of a document. In this case, the complete document has to be read.

Even if the magic number test normally provides the highest quality of determination

[10] `http://www.stardivision.com`

[11] Usually the content type can be determined by reading just the first couple of bytes of a document.

of a content type, other methods can be used in the Dino system: a very popular, although nevertheless not very reliable method, is to map document name extensions to content types, for example a document named `index.html` is mapped to the content type `text/html`.

As the content type can be stored as a meta-data-entry together with the document, the process of determining it has to be done only once at creation time of a new document or when a new system is embedded.

Naming Standard

The naming of the content type follows the Multipurpose Internet Mail Extension Standard (MIME) [FB96]. As this standard is well known, only a short description is given here. The content type declaration is defined in the form of *top-level-media-type/subtype*, followed by set of parameters, specified in an attribute/value notation, for example `text/plain; charset=iso-8859-1`.

5.3.8 Meta-data

Meta-data provide additional informations to arbitrary objects. For example the author, creation date, modification date, content length, etc. are often used as meta-data keys. Generalizing the access to meta-data across server systems arises a lot of problems:

- **different key names** for the very same meaning: The modification date for example might be retrieved from a server as `modification date`, `modified at`, `mod.date`, etc.
- **different meaning** for the very same key: The request for example for the `date` of a document might return the modification date, the creation date or whatever date the author wanted it to be.
- **different data types** for the very same key: Using the (modification) date of the example above, the value could be returned as a string (`November 1st, 1999`), as a number (seconds from the beginning of the century), or as an object. Even when the same data type is used, it cannot be guaranteed that the data format is the same as well. Especially a date can be written in hundreds of formats, depending on the country, or the creativity of programmers, etc.

- **editors/viewers** for meta-data: As meta-data can be arbitrarily complex, special tools might be necessary to edit or to view the values (the keys are usually characters).
- **special kind of meta-data** only available on a specific system, like the `Object Id` of a Hyperwave document, or the `Access Control Information (ACI)` in an LDAP entry.

The Dino system normalizes the meta-data keys and values in respect to name, type and functionality. This normalization process is still under development.

5.3.9 External Systems' Features

Most external systems offer some special features that cannot be mapped to standard functionality. Nevertheless, they must not be lost due to embedding them in Dino. The Dino system offers two concepts for these special features:

- **Dino-methods**: Embedded systems may provide special methods that do not fit into the general user interface of Dino. These methods are checked against the security rules and then forwarded to the embedded system.
- **Dino-services**: A service is a GUI[12] component that visualizes special features of an embedded server system.

This concept was used in version 3 of Dino to provide the version control mechanism of a Hyperwave server to the clients. A *Dino-service* lets the users enter the necessary information in a comfortable and interactive way and *Dino-methods* afterwards execute the check-in, check-out commands, retrieve the log information, or commit the changes.

Even though Dino tries to offer the superset of possibilities of all connected systems, there will still some remain uncovered by its standard interfaces. The use of Dino-services/methods closes this gap.

[12]Graphical User Interface

5.3.10 Addressing Scheme

In a lot of different systems the addressing scheme is closely related to the navigation through the system. As an example, a filesystem can be taken: the "address" `/home/cdaller/text/latex/introduction.tex` is the identifier of the given file, but can also be used as a path for navigation through the file system as well as it contains the information about parent-children relations. A Hyperwave server is a good example for a system that doesn't use this approach. It identifies each object by a unique Id, a 64 bit number like `0x423a4fe30b4f36c1`. The navigation is completely independent from this Id and uses additional information stored in the database.

Addressing in the Dino System

Dino now[13] uses unique Ids called *Globally Unique Dino Handles (GUDHs)* that are also independent from navigation. These handles do not contain any kind of information about relations between objects. Therefore they may never be used for browsing and navigation through the systems, but only for directly accessing objects in the system.

A GUDH contains a resolution hierarchy of so called *Handle Chunks (HCs)*. These HCs can be of different types. A short example will be given to make this fact clearer: The attribute `author` of a document in the Dino system is to be addressed. The document's handle might be for example `[handle]-[of]-[document]`. This sequence of chunks is followed by a sequence of meta-data-HCs, for example `[meta-data]-[author]`. The resulting Globally Unique Dino Handle is `[handle]-[of]-[document]-[meta-data]-[author]` and resolving it, leads directly to the meta-data object holding the author's name. A handle consisting only of the first four HCs of the example given above would resolve to the whole set of meta-data of the node (if available). [IIC99a]

Each Handle Chunk might be of different type but nevertheless it is responsible to resolve the next hierarchy in the GUDH (written from left to right). In the example given, the Node addressed by the sequence of Handles `[handle]-[of]-[document]`, has to resolve the Handle Chunk `[meta-data]`, and this one is in his turn responsible to return the correct object when resolving the Chunk `[author]`.

[13] Up to version 3 the filesystem approach was used.

5.4 External Gateways

External Gateways provide objects stored inside Dino, or inside the systems embedded by Dino respectively, to clients using a specific protocol. There is no need that Gateways use all of the information available and support all features of embedded systems. A HTTP Gateway for example uses the relations, the meta-data, and the content of the objects, while an FTP Gateway provides only the content and a very limited set of meta-data to its clients.

External Gateways are treated as applications from the Dino layer model's (figure 5.1) point of view. They are not part of the Dino Kernel, but use the Dino API to access the system.

5.4.1 Provide Content

Probably the main usage of External Gateways is providing the content of objects located on different embedded systems to its clients. As all content-objects, independently of their origin, are internally held using the Document Object Model (DOM)[14], they are easily transformed into the desired document format. The Dino system supplies converters for most document formats. This functionality can be compared to the *Parsers/Renderers* of the Oracle8*i* system [Ora99].

As the content in the Dino system also holds meta-data, External Gateways choose which meta-data they want to include in the data they provide to the client. HTTP Gateways for example wrap Dino meta-data into the appropriate HTML-tags (`<META>`). In addition to this, the meta-data keys have to be "de-normalized" to match the specific protocol demands[15].

5.4.2 Additional Functionality

Some Gateways might add functionality, as an HTTP Gateway could for example provide user authentication and then customize the content for different users: It might provide

[14] Please see section 3.5 for a detailed description of this object model.

[15] The meta-data key for the date when the document was published is '`date`' in HTTP, while in Dino '`date.creation`' or '`date.modification`' might be used.

different entry points for different users, show the documents in a special layout, or limit the available documents to a selection the user has previously chosen.

5.4.3 Views

[IIC99a] states, that *Views* allow navigation through the Dino object space by means of arbitrary relations. Different Views exist for different types of relations and different purposes. For example a "hierarchical View" allows to navigate through parts of the system that define strictly hierarchical parent-children relationships. Another example is a "graph View" that allows to navigate through parts of the system that define successor-predecessor relationships. Figure 5.9 illustrates that the very same Dino system might get different looks, depending on the View used.

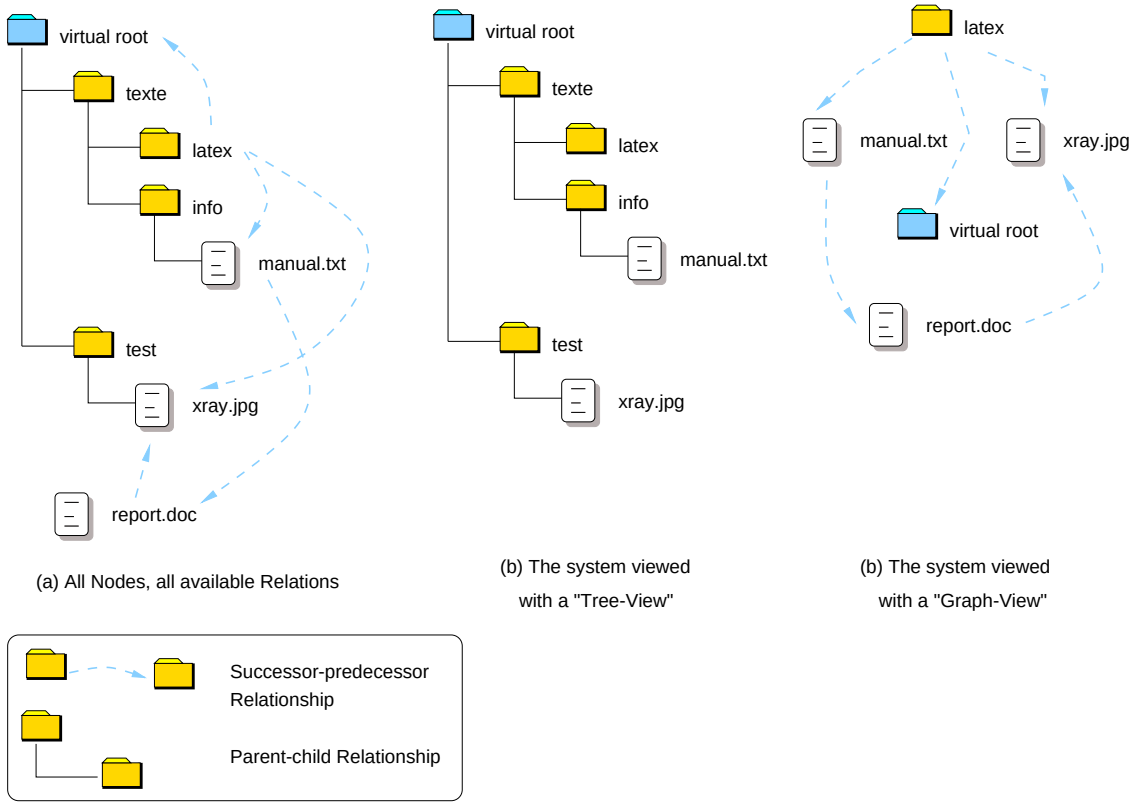


Figure 5.9: Different Views offer different Relations and/or Nodes to the user.

A View uses the internal Dino data structure and provides Dino Nodes[16] to the API layer. Nodes implement a slim general Dino Node interface and provide functionality as appropriate for the special View.

5.4.4 Name Space

The internal addressing scheme used by Dino might not be appropriate for clients. For example, a Hyperwave server uses 64bit numbers as Ids. These Ids are most probably used for the internal Dino addressing as well (remember, the *embedder* is responsible for providing/resolving the *Handle Chunks*), so a user accessing the Dino system by an FTP-Gateway will be quite confused to see the Id `0x745a8b3d6ce3f2a7` instead of a (hopefully self-explaining) name.

Applications access the Dino system by use of its API and its Views. In the case described above, the FTP Gateway would use a hierarchical View, as this structure matches the File Transfer Protocol best. The internal Id is used to address objects in the Dino space, but the system provides a human readable name (stored in meta-data) as well. This name is used by the View so an FTP-client does not need to fiddle around with Hyperwave object-Ids.

[16] see clarification of term in chapter 2

Chapter 6

Conclusion and Outlook

This chapter is structured into two parts. The first part gives a short conclusion and answers the questions posed in the definition of problems in chapter 1. The second part gives an overview of future work.

6.1 Conclusion

Since no other middleware systems fulfilling the requirements are available, a new system was designed and different aspects of its implementation were discussed in this work.

Most of the concepts described in this thesis have been implemented and tested in earlier versions of Dino. These tests proved the functionality of the concept and now the questions of section 1.5 can be answered:

- Are there other middleware systems that satisfy the requirements? How do they work?

  There is quite a number of systems on the market or in development stage that allow to connect easily relational databases to a Web browser or remote object systems like CORBA. There are very few real middleware systems that allow to customize the connection modules and so enable the clients to retrieve data from a wider range of external systems.

  For the clients accessing the middleware systems, it is even worse. The only choice

the user normally has, is to use a Web browser or a CORBA-aware application. A Web browser is a very general tool, but offers limited possibilities, whereas the design and implementation of a CORBA application is quite a time consuming process. Other clients, even if they would be specialized for the task the user wants to have done, are not served and so cannot be used.

But even the systems that allow more adaptation lack quite a few of the requirements stated in chapter 2. Especially the requirements concerning content handling, like distributed documents or active content are hardly fulfilled. Oracle8*i* is the only system that is able to provide the containing documents in different formats using its parsers and renderers.

Another weak point is relation management. This type of functionality seems to be completely ignored by other manufacturers. None of the products allow the creation of relations across system boundaries.

Surprisingly most of the compared systems do offer a good approach to scalability problems. To keep the system load acceptably low, they allow clustering and load balancing, some of them even different methods of this technique.

- Is there one product that satisfies all requirements?

  None of the examined products could satisfy all of the requirements set up in chapter 2. Most systems are not flexible enough to integrate different external systems, all of them are incapable of really "integrating" the content of the connected systems to make the user believe to work with a system made in one piece.

- What standards are used by middleware systems?

  All middleware systems allow client-access to the content by use of a standard (HTTP or CORBA). The CORBA approach is taken with caution, because not all ORBs can be used interchangeably without changes of the code.

  The other side of the middleware layer uses preferable JDBC or proprietary RDBMS-protocols to connect to relational databases. Other types of server systems are rarely integratable at all and require customization or programming of the customer.

- How platform-independent is the concept of Dino?

As Dino is completely written in Java, all platforms with an available Java Development Kit are supported. Testing was done with JDK1.2 under Linux, Sun Solaris, and Windows.

- How secure is the system?

  A rudimentary security system is part of the Dino system which nevertheless supports simple rules to define user access rights. Future plans include the integration of the user management of embedded external systems.

- How universal is the concept?

  The design of Dino is kept as universal as possible. Dino is able to embed a broad range of different types of systems. Various use cases were investigated, but as the current version of Dino is still under development, no proof of the universality can be given here.

- What problems occured when integrating new systems into Dino.

  As experience from the work on former versions of Dino flew into the design of the current version, no problems of concept have been found until now. External system specific problems existed and were solved as described in chapter 5.

- How can new clients access the content of Dino and what aspects have to be paid attention to?

  Different clients may access the system by External Gateways. These modules can be added/removed to Dino at runtime and are implemented as normal applications using the Dino API. Therefore no security leaks can come into existence by badly programmed gateways, as any operation is checked by the internal Security Manager.

  Generally it can be said, that providing information contained in the Dino system to other applications/systems is by far easier than integrating external systems. The general approach of using the Document Object Model internally guarantees to be open to future development.

6.2 Future Work

As Dino is under heavy development at the time of writing, some parts of the system still need to be defined in their final form.

User Management of embedded systems has to be integrateable, as most of the systems have their own proprietary user management. Oracle stores its user access rights in tables, Unix keeps its user information directly in the file system's inode structure, and LDAP servers put their access control information directly in the LDAP entries. All of them use a completely different structure of access rights and access rules, but they all should be integrated by the Dino access rule checker, so existing user rights do not have to be created twice.

Meta-data is one of the unsolved problems. An equally general approach to handle meta-data comparable to the Document Object Model (DOM) for content is needed. The *Dublin Core Meta-data Standard* [WKLW98] might be a good choice, but has to be examined carefully as the definitions are extremely loose [Cla97].

Additional system embedders Dino must be able to connect to more external systems, especially the databases of the big players like PeopleSoft, SAP, etc. will help Dino to gain respect in the computer-science world.

User authentication needs additional support using Smartcards, i-buttons, or other authentication techniques.

The concept of Dino enables universal access both to and from a broad range of systems. It is developed by a rather small team of the IICM, nevertheless it offers a functionality that surpasses most of the commercially available systems.

Appendix A

Magic Numbers

Only a short example of a magic number configuration file is given here. For a description of the use of a magic number test, please see section 67 or the manual pages for the `file` command on Unix systems. The example shows the magic numbers for JPEG files, bitmaps and gzip files.

```
#
# JPEG images
#
0 beshort 0xffd8 JPEG image data
>6 string JFIF \b, JFIF standard
# HSI is Handmade Software's proprietary JPEG encoding scheme
0 string hsi1 JPEG image data, HSI proprietary

# PC bitmaps (OS/2, Windoze BMP files)  (Greg Roelofs, newt@uchicago.edu)
0 string BM PC bitmap data
>14 leshort 12 \b, OS/2 1.x format
>>18 leshort x \b, %d x
>>20 leshort x %d
>14 leshort 64 \b, OS/2 2.x format
>>18 leshort x \b, %d x
>>20 leshort x %d
>14 leshort 40 \b, Windows 3.x format
>>18 lelong x \b, %d x
```

```
>>22 lelong x %d x
>>28 leshort x %d
0 string IC PC icon data
0 string PI PC pointer image data
0 string CI PC color icon data
0 string CP PC color pointer image data

# gzip (GNU zip, not to be confused with Info-ZIP or PKWARE zip archiver)
0       string          \037\213        gzip compressed data
>2      byte            <8              \b, reserved method,
>2      byte            8               \b, deflated,
>3 byte &0x01 ASCII,
>3 byte &0x02 continuation,
>3 byte &0x04 extra field,
>3 byte &0x08 original filename,
>3 byte &0x10 comment,
>3 byte &0x20 encrypted,
>4 ledate x last modified: %s,
>8 byte 2 max compression,
>8 byte 4 max speed,
>9 byte =0x00 os: MS-DOS
>9 byte =0x01 os: Amiga
>9 byte =0x02 os: VMS
>9 byte =0x03 os: Unix
>9 byte =0x05 os: Atari
>9 byte =0x06 os: OS/2
>9 byte =0x07 os: MacOS
>9 byte =0x0A os: Tops/20
>9 byte =0x0B os: Win/32
```

Bibliography

[Bac98] Dieter Bachmann. A Distributed Image Processing Back-end. Master's thesis, Graz University of Technology, March 1998.

[Ber99] Cliff Berg. The state of Java application middleware, Part 1. *Javaworld*, March 1999. available online `http://www.javaworld.com/javaworld/jw-03-1999/jw-03-middleware.html`.

[BLFIM98] T. Berners-Lee, R. Fielding, U.C. Irvine, and L. Masinter. Uniform Resource Identifiers (URI). RFC 2396, August 1998. available online `http://www.faqs.org/rfcs/rfc2396.html`.

[Blü99] Karl Blümlinger. Dino Link Management. Technical report, IICM, Graz University of Technology, 1999.

[Bra99] Dr. Bracho, Rafael. Integrating the eBusiness. Technical report, Active SoftWare, 1999. available online `http://www.activesw.com/bin/litregFile.cgi?file=eBusiness_A4.pdf`.

[Cla97] Roger Clarke. Beyond the Dublin Core: Rich Meta-Data and Convenience-of-Use Are Compatible After All. Technical report, Xamax Consultancy Pty Ltd, Canberra, 1997. available online `http://www.anu.edu.au/people/Roger.Clarke/II/DublinCore.html`.

[Deu] Prafulla S. Deuskar. Scheduler Agent using Voyager. Technical report, Indiana University, Bloomington Computer Science Department. available online `http://www.cs.indiana.edu/hyplan/pdeuskar/voyager.html`.

[FB96] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME). RFC 2046, November 1996. available online `http://www.faqs.org/rfcs/rfc2046.html`.

[Gla99] Graham Glass. Overview of Voyager: Objectspace's Product Family for State-of-the-Art Distributed Computing. Technical report, ObjectSpace, 1999. available online: `http://www.objectspace.com/products/documentation/VoyagerOverview.pdf`.

[Hau99] Heimo Haub. User Management of Heterogenous Server Systems. Master's thesis, IICM, Graz University of Technology, 1999.

[How99] Tim Howes. Directory Services; LDAP: Use as Directed. *Data Communications*, February 1999. available online `http://www.data.com/issue/990207/ldap.html`.

[IIC99a] Dino Team IICM. Dino V4 Software Engineering Document. Technical report, IICM, Graz University of Technology, 1999.

[IIC99b] Dino Team IICM. Dino V4 Software Requirements. Technical report, IICM, Graz University of Technology, 1999.

[Lug99] Wolfgang Lugmayr. Distributed Object Computing with CORBA, 1999.

[Mil92] Milan Milenkovic. *Operating Systems, Concepts and Design.* McGraw-Hill International Editions, second edition, 1992.

[Mor99] JP Morgenthal. Understanding Enterprise Java APIs. *Component Stategies*, August 1999.

[Obj97] Object Management Group. *The Common Object Request Broker Architecture and Specification (Revision 2.1)*, July 1997.

[OMG97] Object Transaction Service 1.1 Specification. Technical report, Object Management Group, 1997. available online `ftp://www.omg.org/pub/docs/formal/97-12-17.pdf`.

[OPR96] Randy Otte, Paul Patrick, and Mark Roy. *Understanding CORBA: The Common Object Request Broker Architecture.* Prentice Hall PTR, 1996.

[Ora98a] Oracle Application Server 4.0 White Paper: Product Overview. Technical report, Oracle, September 1998. available online `http://technet.oracle.com/products/oas/pdf/oas40wp.pdf`.

[Ora98b] Oracle8*i* Enterprise Edition. Technical report, Oracle, November 1998.

[Ora99] Understanding the Oracle8*i* Internet File System (*i*FS) Option. Technical report, Oracle, February 1999. available online `http://www.oracle.com/database/documents/understanding_o8i_ifs_fo.pdf`.

[Sad97a] Darleen Sadoski. Client/Server Software Architectures–an Overview. Technical report, Carnegie Mellon, Software Engineering Institute, 1997. available online `http://www.sei.cmu.edu/str/descriptions/clientserver_body.html`.

[Sad97b] Darleen Sadoski. Three Tiew Software Architectures. Technical report, Carnegie Mellon, Software Engineering Institute, 1997. available online `http://www.sei.cmu.edu/str/descriptions/threetier.html`.

[Sun99a] Introduction to NetDynamics for NetDynamics 5.0. Technical Report Part No.: 806-0999-10, Sun Microsystems, Inc., February 1999. available online `http://www.netdynamics.com/support/docs/nd50/Nd50DocSer/res/intro/intro\%.pdf`.

[Sun99b] Sun Microsystems, Inc. JNDI Java Naming & Directory Interface - Executive Summary, 1999. available online `ftp://ftp.javasoft.com/docs/j2se1.3/jndiexecsumm.pdf`.

[Sun99c] Object Framework Overview. Technical Report Part No.: 806-0528-10, Sun Microsystems, Inc., March 1999. available online `http://www.netdynamics.com/support/docs/nd50/Nd50DocSer/res/objframe/io\%f.pdf`.

[W3C98] Document Object Model (DOM) Level 1 Specification. Technical report, W3C, October 1998. available online `http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001`.

[Web99] PC Webopedia Definition and Links, 1999. `http://webopedia.internet.com`.

[WFC+99] Seth White, Maydene Fisher, Rick Cattell, Graham Hamilton, and Mark Hapner. *JDBC API Tutorial and Reference: Universal Data Access for the Java 2 Platform*. Java Series. Addison-Wesley Pub Co, second edition, 1999. ISBN-0201433281.

[WKLW98] S. Weibel, J. Kunze, C. Lagoze, and M. Wolf. Dublin Core Metadata for Resource Discovery. RFC 2413, September 1998.

[Wri99] Guy Wright. What is an Application Server? Technical report, February 1999. available online `http://webreview.com/wr/pub/sections/appserver/index2.html`.

Index