Client/Server Architectures for Business Information Systems

A Pattern Language

Klaus Renzel

sd&m GmbH & Co. KG
Project ARCUS[1]
Thomas-Dehler-Str. 27
D-81737 München, Germany
email: Klaus.Renzel@sdm.de
Phone: +49-89-63812-251
http://www.sdm.de/g/arcus

Wolfgang Keller

EA Generali
Reumannplatz 7
A-1100 Vienna, Austria
email: WofgangWKeller@compuserve.com
Phone: +43-1-53401-0

Abstract: This paper presents several patterns for distributing business information systems that are structured according to a layered architecture.[2] Each distribution pattern cuts the architecture into different client and server components. All the patterns presented give an answer to the same question: How do I distribute a business information system? However, the consequences of applying the patterns are very different with regards to the forces influencing distributed systems design.

1 Introduction

Distribution brings a new design dimension into the architecture of information systems. It offers great opportunities for good systems design, but also complicates the development of a suitable architecture by introducing a lot of new design aspects and trapdoors compared to a centralized system.

---

[1] This work is sponsored by the German Ministry of Research and Technology under project name ENTSTAND.

[2] The paper is part of a larger effort to collect patterns for business information systems, currently under development by the ARCUS team.

While constructing the architecture for a business information system, which will be deployed across a set of distributed processing units (e.g. machines in a network, processes on one machine, threads within one process), you are faced with the question:

How do I partition the business information system into a number of client and server components, so that my users' functional and non-functional requirements are met?

There are several answers to this question. The decision for a particular distribution style is driven by users' requirements. It significantly influences the software design and requires a very careful analysis of the functional and non-functional requirements. Therefore, we present a list of general forces that you can check against your users' requirements. Requirements determine the strength of these forces. We then discuss some solutions in pattern form. The consequences will guide you in checking whether a particular solution fulfills your requirements. Throughout the patterns we briefly motivate and illustrate each solution by using a library application as a running example.

2 Scope of the pattern language

2.1 General Context

You are designing a business information system, in which many (spatially distributed) users work in parallel on a large amount of data. The system supports distributed business processes which may span a single department, a whole enterprise, or even several enterprises. Generally, the system must support more than one type of data processing, such as on-line transaction processing (OLTP), off-line processing or batch processing.

Typically, the application architecture of the system is a *Three Layer Architecture*: Most architectures for information systems derive from a layered architecture [BMR+96, p. 31, Den91] as depicted in Figure 1.

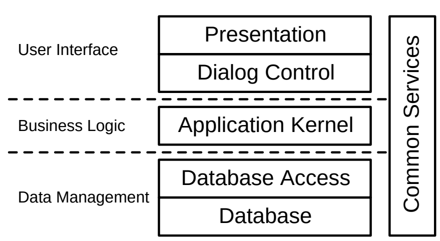


Figure 1: Three layer architecture for business information systems

The user interface handles presentational tasks and controls the dialog (e.g. see [CK97]), the application kernel performs the domain specific business tasks and the database access layer

[KC96] connects the application kernel functions to a database.[3] Our distribution view focuses on this coarse-grain component level.

In developing a distributed system architecture we mainly use the *Client/Server Style* (see [Wei97] for discussion of this style in pattern form), which defines a model for distributed processing. Within this model components of a distributed system are classified by two roles: client and server. Clients and servers communicate via a simple request/response protocol. We avoid to talk about hybrid (application) architectures, which combine different communication styles, even though the overall system architecture will often be a hybrid architecture.

2.2 General Forces

- *Business needs vs. construction complexity:* On the one hand allocating functionality and data to the places where it is actually needed supports distributed business processes very well, but on the other hand distribution raises system's complexity. Client server systems tend to be far more complex than conventional host software architectures. To name just a few sources of complexity: GUI, middleware, and heterogeneous operating system environments. It is clear that it often requires a lot of compromises to reduce the complexity to a level where it can be handled properly.

- *Processing style:* Different processing styles require different distribution decisions. Batch applications need processing power close to the data. Interactive processing should be close to input/output devices. Therefore, off-line and batch processing may conflict with transaction and on-line processing.

- *Distribution vs. performance:* We gain performance by distributed processing units executing tasks in parallel, placing data close to processing, and balancing workload between several servers. But raising the level of distribution increases the communication overhead, the danger of bottlenecks in the communication network, and complicates performance analysis and capacity planning. In centralized systems the effects are much more controllable and the knowledge and experience with the involved hardware and software allows reliable statements about the reachable performance of a configuration.

- *Distribution vs. security:* The requirement for secure communications and transactions is essential to many business domains. In a distributed environment the number of possible security holes increases because of the greater number of attack points. Therefore, a distributed environment might require new security architectures, policies and mechanisms (e.g. encryption, authentication protocols).

---

[3] The architecture represents a simplified view, which does not cover all possible components of an information system (e.g. interfaces to other applications, workflow components, batch interface).

- *Distribution vs. consistency:* Abandoning a global state can introduce consistency problems between states of distributed components. Relying on a single, centralized database system reduces consistency problems, but legacy systems or organizational structures (off-line processing) can force us to manage distributed data sources.

- *Software distribution cost:* The partitioning of system layers into client and server processes enables distribution of the processes within the network, but the more software we distribute the higher the distribution, configuration management, and installation cost. The lowest software distribution and installation cost will occur in a centralized system. This force can even impair functionality if the software distribution problem is so big that the capacities needed exceed the capacities of your network. The most important argument for so called diskless, internet based network computers is exactly software distribution and configuration management cost.

- *Reusability vs. performance vs. complexity:* Placing functionality on a server enforces code reuse and reduces client code size, but data must be shipped to the server and the server must enable the handling of requests by multiple clients.

2.3 Pattern Language Overview

To distribute an information system by assigning client and server roles to the components of the layered architecture we have the choice of several distribution styles. Figure 2 shows the styles which build the pattern language.

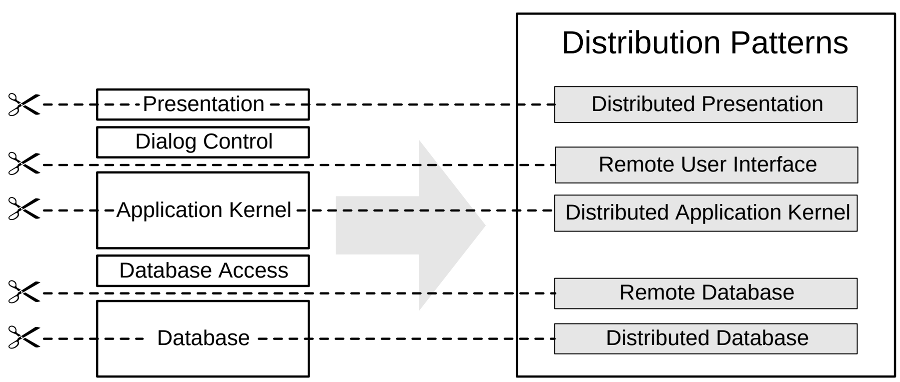


Figure 2: Overview of the patterns resulting from different client/server cuts

To take a glance at the pattern language we give an abstract for each pattern:

- *Distributed Presentation:* This pattern partitions the system within the presentation component. One part of the presentation component is packaged as a distribution unit and is processed separately from the other part of the presentation which can be packaged together with the other application layers. This pattern allows of an easy implementation and very thin clients. Host systems with 3270-terminals is a classical example for this approach. Network computers, internet and intranet technology are modern environments where this pattern can be applied as well.

- *Remote User Interface:* Instead of distributing presentation functionality the whole user interface becomes a unit of distribution and acts as a client of the application kernel on the server side.

- *Distributed Application Kernel:* The pattern splits the application kernel into two parts which are processed separately. This pattern becomes very challenging if transactions span process boundaries (distributed transaction processing).

- *Remote Database:* The database is a major component of a business information system with special requirements on the execution environment. Sometimes, several applications work on the same database. This pattern locates the database component on a separate node within the system's network.

- *Distributed Database:* The database is decomposed into separate database components, which interact by means of interprocess communication facilities. With a distributed database an application can integrate data from different database systems or data can be stored more closely to the location where it is processed.

The next figure summarizes the effects of the patterns on the design objectives mentioned in chapter 2.2:

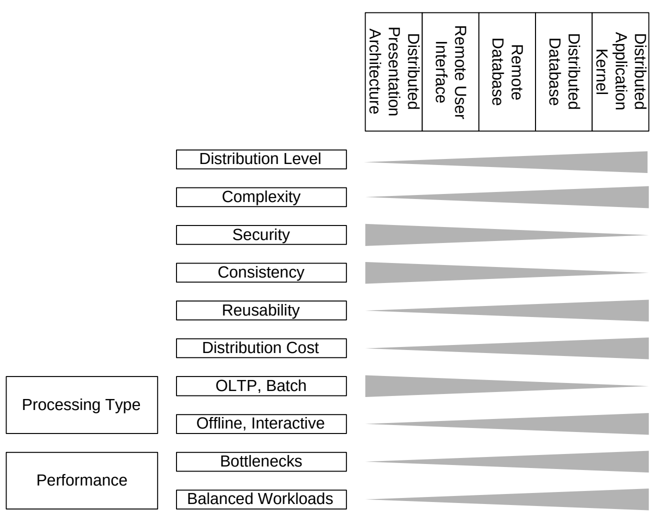
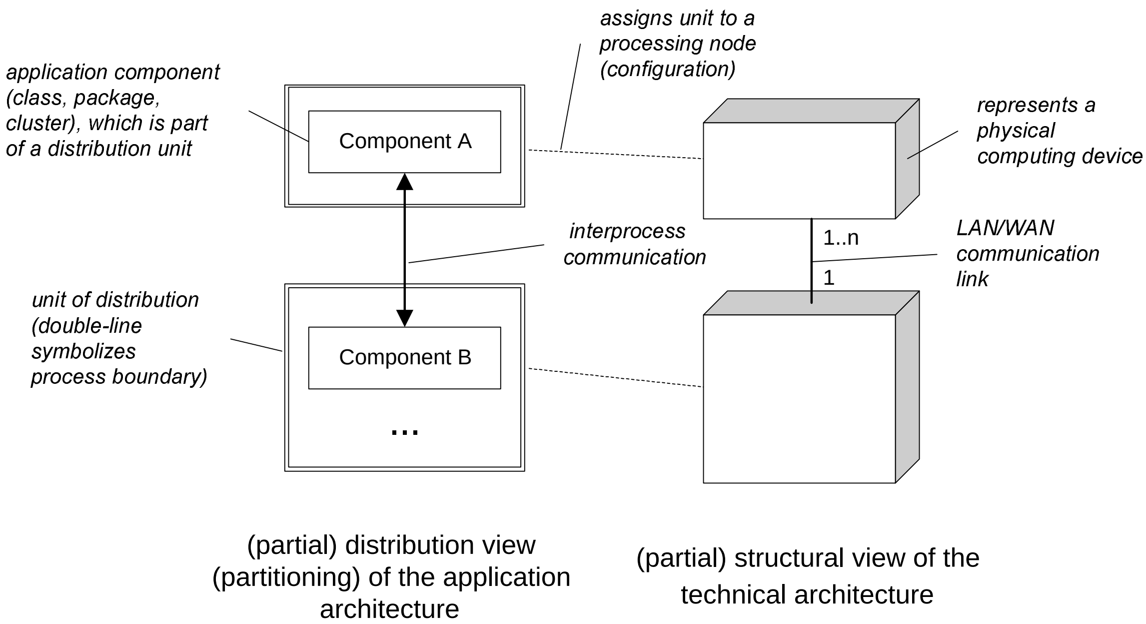


Figure 3: Patterns' resolution of the architectural forces.

In the patterns and examples we use the following notation to visualize solutions' structures:



Each pattern presents a partial distribution view of the application architecture and maps this to a physical system structure. As a solution addresses only a part of the application components the whole distribution model may result from applying more than one distribution pattern. Instances of application components can „live“ on different nodes of the technical architecture, because of components being part of more than one distribution unit or distribution units mapped to different nodes.

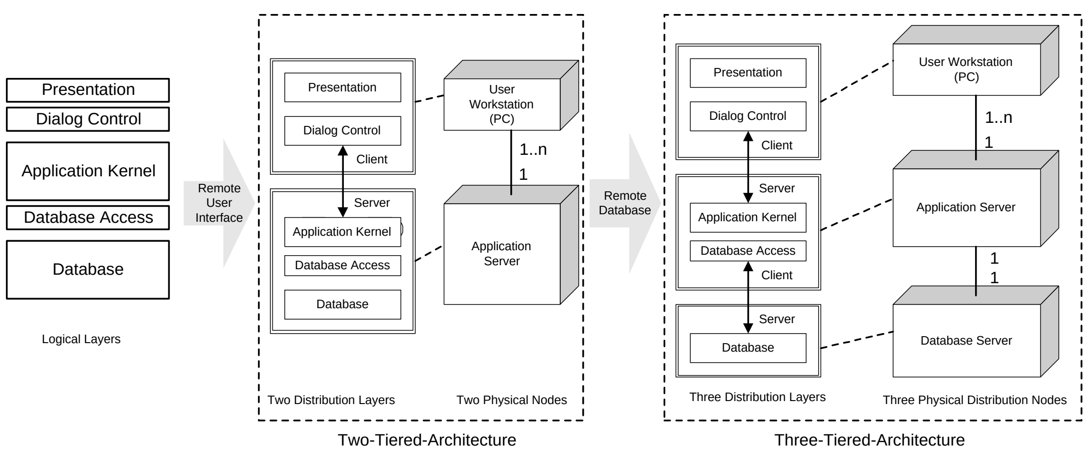


Figure 4: Building a Three-Tiered-Architecture applying the Remote User Interface pattern and the Remote Database pattern.

The patterns are orthogonal to each other. They may therefore be combined within a single design of a distributed information system (e.g. to build a Three Tier Architecture [Hir96] as illustrated in Figure 4).[4]

3 Description of the Solutions

Each solution presents a macro-level architecture for system distribution. These architectures can not resolve every force impacting client/server design. Many of them are only partially resolved and have to be reified or resolved further in the implementation of the architecture (e.g. by applying other patterns and use of technical infrastructure).

3.1 Pattern: Distributed Presentation Architecture

Also Known As

Host-Terminal Style

Solution

Partition the presentation layer. One part of the presentation layer is packaged as a distribution unit and is processed separately from the other part which can be packaged together with the other application layers:

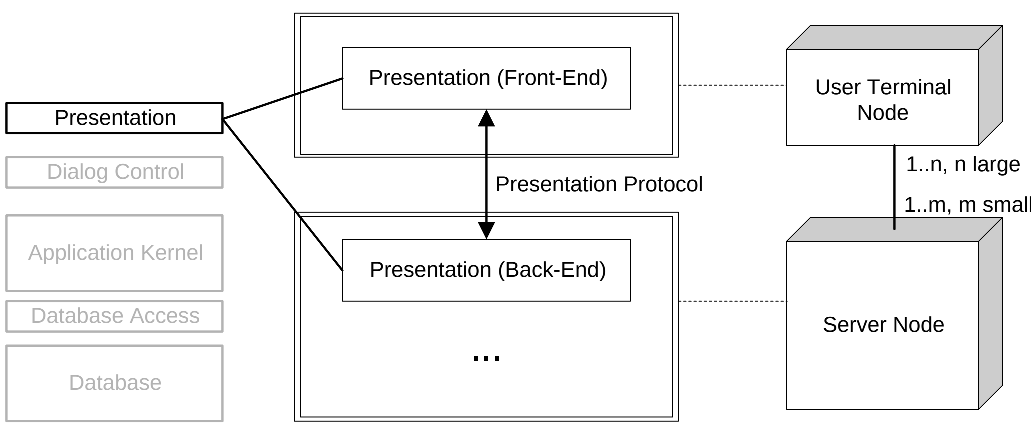


Figure 5: Structure of a Distributed Presentation Architecture

The back-end presentation (see Figure 5) is an application component which mediates between the dialog control and the technical front-end presentation. The latter provides an interface for

4 The pattern language can be used to generate a number of different three-tiered architectures.

presentation services (e.g. graphical primitives for opening a window or displaying fields) and is typically realized as a user-interface toolkit, library or framework. The presentation components interact according to some presentation protocol propagating presentation service requests from the `Server Node` to the `User Terminal Node` and input action as well as data in the opposite direction.

Example

Imagine the design of a library system using internet technology. The end user can search for available books and can also see whether a book is borrowed by someone.

A HTML[5]-Browser and a HTTP[6]-server are the basic system software for the presentation layer of the library application (Figure 6). With a subset of HTML we can describe the structure and content of system's input and output. The HTML-Browser visualizes the HTML descriptions.

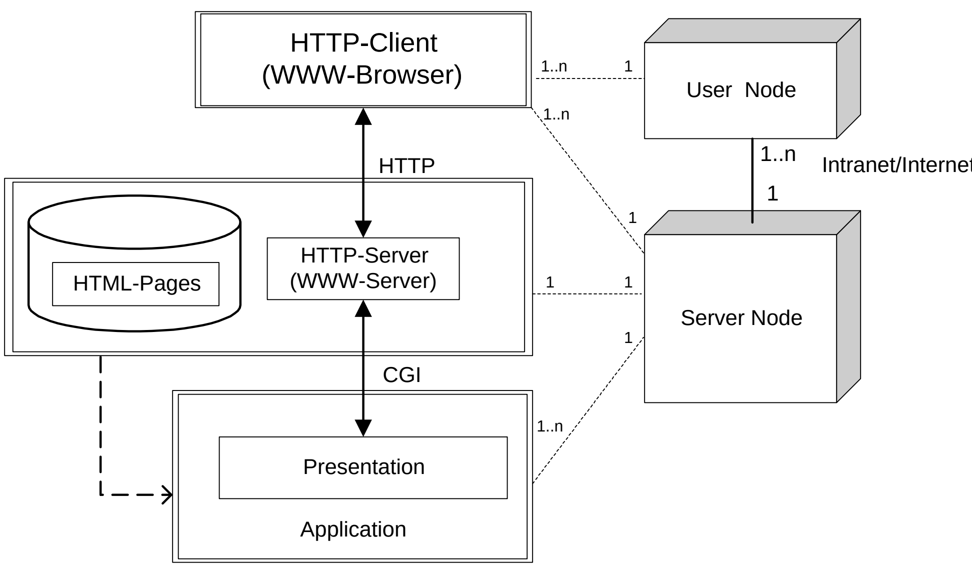


Figure 6: Example for a distribution view of the components responsible for the presentational tasks in a library system

The application programmer has to know about the Common Gateway Interface (CGI) to realize the communication with the HTTP-server. Most of the other communication and concurrency issues are handled by the system infrastructure. The HTTP-client can be running on the user node as well as on the server node (e.g. Lynx is a textual WWW-browser running under

5 **H**yper**t**ext **M**arkup **L**anguage

6 **H**yper**t**ext **T**ransfer **P**rotocol

unix). In the latter case the user node can be a simple terminal or a PC with a terminal emulation.

A search request on the library may cause the following interactions: When the user presses the submit button after filling a search form with the name of the author etc., the browser sends a request with a program identifier and the form data to the server. The server starts the program and passes the data via the Common Gateway Interface. The presentation component on the application side analyzes the data and deals with transformation of results in HTML. Finally, the HTTP-server forwards the HTML-page given by the application program to the client which displays the page to the user.

Consequences

- *Business needs vs. construction complexity*: Combined with elegant class libraries, the architecture look for the application programmer quite like a central system architecture. This reduces complexity of application development, because most of the distributed programming is done on the system level, so that the application programmer is not confronted with it. Besides it prevents complication of application's testability.

- *Distribution vs. performance*: Is a tradeoff between user interface functionality and communication bandwidth of the underlying network. Calling graphical presentation primitives may cause too much traffic within a slow WAN (Wide Area Network). For a X-Window System even in 10 Mbit LANs the usage is limited to roughly a hundred X-Terminals. In case of on-line processing the solution is best suited for form-based user interfaces in opposite to object-oriented user interfaces [CK97]. Thus, either the concrete technical infrastructure constrains the design of the user interface or vice versa.

- *Processing style*: A Distributed Presentation Architecture facilitates batch and transaction processing, because it does not split data processing and transaction control between different nodes of the system's network. However, it does not support off-line processing, because it requires continuous real-time communication between the user's node and the server.

- *Distribution vs. security*: Centralized processing and data eases security, but nevertheless user authentication, authorization and secure communication have to be considered carefully (e.g. internet application such as home banking).

- *Distribution vs. consistency:* As no data is stored on the client side consistency is not affected.

- *Software distribution cost*: As all domain specific software resides on the application nodes, distribution cost is limited to the system software required on the terminal nodes (e.g. installation and update of the X-Window System).

- *Reusability vs. performance vs. complexity:* The solution does not produce any reusable application components.

Design

- *Openness*: A Distributed Presentation Architecture can lead to very portable software. The front-end presentation requires little software and is enabled by a variety of technical infrastructure. For example, the X-Window protocol is open, machine-independent, and portable. X-Server emulators are available for many platforms. But note that portability is a product of the disciplined use of the architecture and not of the architecture itself. Communication APIs for the communication between graphical terminals and application servers tend to be complex. Therefore, shield the protocol with a application framework (e.g. ET++) , a toolkit, or a class libraries.

- *Reliability*: This depends on the choice of the hardware and software used on the application node and the middleware used to implement the presentation protocol. For example, the use of X-Window software on simple UNIX machines may result in poor reliability.

- *Scalability*: There is a limit to scalability due to high network traffic. You can tackle these problems with LAN segmentation and high speed LANs. The scalability of the server node depends on the system platform (hardware and system software).

Known Uses

X-Window Architecture: The most popular implementation of this architecture is the interactive network graphics provided by the X-Window System [SG86]. In the X-Window System the presentation protocol is known as the X-Protocol.

IBM 3270-Terminal: This may be seen as an early form of the Distributed Presentation Architecture in the world of alpha terminals.

HTML-Browser: Offer a modern alternative to 3270 terminals, terminal emulation or screen scraping (as shown in the example). Browser solutions are more economical in terms of network bandwidth than X-Window solutions, because functions such as mouse movements and screen updates cause no network traffic.

See Also

In context of a mainframe with connected terminals the pattern corresponds to the Host-Terminal Style [Wei97].[7] The X-Window System and HTML-Browsers use the Client-Server Style [Wei97].

3.2 Pattern: Remote User Interface

Also Known As

Thin Client

Solution

Apply a client server cut between dialog control and application kernel. On the client node the dialog control recognizes domain level actions and issues the necessary commands to the application kernel on a server node to perform these actions.

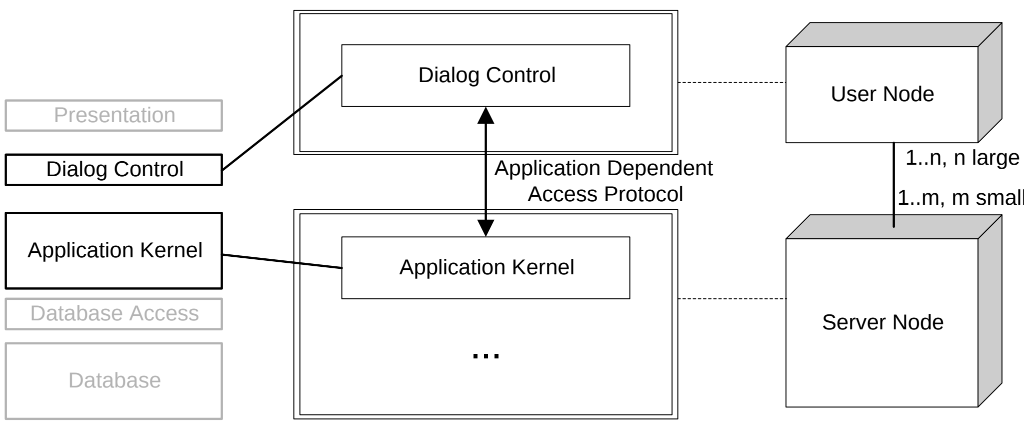


Figure 7: Structure of a Remote User Interface architecture

The Application Kernel Access Protocol is implemented by some middleware which propagates service requests from the client to an appropriate server and delivers the results.

Example

Instead of using HTML-forms for the user interface of the library application, we can implement the user interface as a Java applet. This applet can utilize the features of the *Abstract Window ToolKit* (AWT) to increase the usability of the application interface compared to the

[7] Sometimes this architecture is also called *One-Tiered-Architecture*, because all application code remains central on the host.

form-based variant. The application kernel can be implemented in Java as well, so that *Remote Method Invocation* (RMI) is applicable for the communication between the user interface running within the browser on the client side and the application kernel running on the server side.

Consequences

- *Business needs vs. construction complexity*: A Remote User Interface is more complex than a Distributed Presentation Architecture, because the client/server interaction can not be implemented by off-the-shelf system components and a standardized communication protocol. The communication issues have to be addressed by the application programmer. On the other hand this allows of advanced user interfaces more related to business need's.

- *Processing style*: A Remote User Interface runs smoothly with batches, as batches may run on powerful servers close to their data.

- *Distribution vs. performance*: Network traffic is low compared to a Distributed Presentation Architecture, because all interaction intensive processing can be performed on the client side. Therefore, the Remote Presentation Architecture has a potential for very good performance if the design of the application layers is done coherently (see design issues in the following section). In this respect the architecture constrains the design space and requires performance to be reified while implementing this architecture. For example, the user gets prompted with some information, enters some data, which is often subject to plausibility checks (single fields only, no checks depending on the application kernel), and finally hits some button that causes an application kernel command to be performed. To trigger a command in the application kernel, a Remote User Interface architecture transports just the data that are needed to identify the command and to provide it with parameters. In most cases much more than these data are needed from the database in order to perform the actual function. Only a Distributed Application Kernel can be tuned to cause even less network traffic by allocating parts of the application kernel to the client machine.

- *Distribution vs. s*ecurity: Does not differ very much from a Distributed Presentation Architecture, except that application code is spread to the clients and therefore more open to possible attacks.

- *Software distribution cost*: This architecture has higher software distribution costs than a Distributed Presentation Architecture. Often you can parameterize plausibility checking and dialog control using table driven meta systems (Reflection [BMR+96]. This will reduce the need for software updates, as tables are easier to distribute than dynamic link libraries and whole software releases. In any case, a Remote User Interface needs a full fledged operating system plus a graphical presentation system plus some communication software to cope with the middleware.

- *Reusability vs. performance vs. complexity:* The solution enables other application interfaces to access the same application kernel components. Thus, it facilitates the development of standardized domain-level components, but only on the level of the domain access interface offered to the remote user interface. A Distributed Application Kernel provides more flexibility and a more fine-grained level of reuse.

Design

- *Choose a dialog paradigm*: The architecture works best with a Form-Based User Interface [CK97]. Other forms of dialogs, especially direct manipulation may be better for some purposes in advanced applications, but will result in more network traffic. Therefore advanced information systems that cause heavy data transfer between user interface and data are better off with some form of off-line architecture. Off-line architectures load data onto a client machine once, may keep them for hours and write them back in a batch.

- *Have your primitive plausibility checks done on the client*: Plausibility checking may occur on a screen each time a user leaves a field. If plausibility checking is done entirely on the server machine, this results in heavy network traffic for remote procedure calls and also results in poor performance. Therefore, put as much plausibility checks on the client as possible.

- *Choose middleware*: You may choose any kind of middleware that allows some form of RPC to call application kernel functions remotely via the network. There are different RPC environments available on the market:

  - *Plain RPCs* like provided by all kinds of DCE compliant products and also some custom RPCs.

  - *Distributed transaction processing environments* that add transaction capabilities and administration features to plain RPC environments.

  - *Distributed object oriented Middleware (CORBA)* that add object orientedness to distributed transaction processing environments.

- *Load balancing*: Most distributed programming environments offer run time load balancing by replicating application servers at run time. Some distributed transaction environments [ACD+96] also offer advanced techniques such as priority queues, data dependent request routing and other scheduling techniques.

- *Manageability*: Depending on the middleware used, a Remote User Interface may be the best manageable architecture of all. Distributed transaction processing environments have brought many mainframe administration features to client server networks.

Known Uses

A lot of all client server architectures implemented today are Remote User Interface architectures with their share rising.

The OASIS framework [Moo97] implements a Remote User Interface within a Three-Tier-Architecture.

3.3 Pattern: Distributed Application Kernel

Solution

Apply a client server cut through the application kernel. The architect may distribute the application kernel freely at her or his whim, following certain rules. One part of the application kernel functionality is placed on a client node and another part on a server.

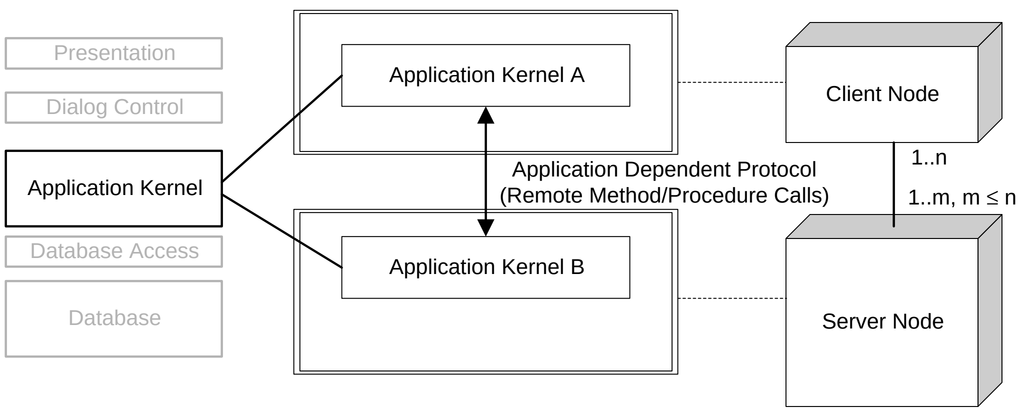


Figure 8: Structure of a Distributed Application Kernel architecture

The client server cut is bridged with some remote procedure call (RPC) based middleware, which propagates service requests from the client to the server and delivers results back to the client.

Example

Imagine, the application kernel of the library system is composed of several components building a DAG. Some of the low-level domain objects should become server objects shared by the more specific library objects running on the client nodes. The server part of the application kernel is implemented in C++, whereas the client part together with the user interface is an applet which uses an object request broker implemented in Java to access and interact with the application server.

Consequences

- *Business needs vs. construction complexity*: A distributed application kernel is a very flexible architecture. It is suited for complex, highly interactive applications (e.g. decision support systems with object oriented user interfaces), provides good utilization of the underlying hardware. Unfortunately, the solution is also the most challenging one for application design and implementation.

- *Distribution vs. performance*: The performance of the architecture depends on how well the cut has been chosen. A good cut offers excellent performance while a bad cut may have a devastating effect. There is no simple distribution rule for the application kernel objects. The distribution must be tailored to the call and traffic structures of the business logic. The criteria for this are similar to modularization: Minimum coupling and maximum cohesion.

- *Processing style*: Batches add additional complexity to the Distributed Application Kernel Architecture. On one side you want to avoid replicating parts of the application kernel. On the other hand the traffic and call structures for batches are completely different from dialog processing. To achieve sufficient performance for batches and dialog processing, you might be forced to replicate some portions of the application kernel.

  As a solution consider the three-tier-architecture discussed on page 19. Furthermore, some RPC environments, such as distributed transaction processing environments or CORBA, allow you to replicate functions making the redundancy problem at least better manageable.

- *Distribution vs. security*: Generally, this architecture is more sensitive to security holes than a Remote User Interface architecture, because of spreading data processing functionality over a client/server network. For example, think of Java applets or ActiveX in the context of internet applications.

- *Distribution vs. consistency*: In the context of transactions across clients and servers we enter the field of distributed transaction processing. For this field the normal RPC is in some environments extended to a transactional RPC, such that it becomes a unit of consistency (the PRC is either totally committed or rolled back).

- *Software distribution cost*: Depends much on the configuration within the network. If the client part is running on end-users' machines, you achieve management as good as in a Remote User Interface, but you still have a harder software distribution problem because of more specific and complex configuration dependencies. If the client part of the application kernel acts itself as a server for the user-interface and is distributed to a small number of server machines, distribution cost can be reduced.

- *Reusability vs. performance vs. complexity*: The solution promotes reusability of application components at the cost of complexity and possibly performance. In combination with open middleware application servers can be built which offer services (e.g. in form of common business objects) to a number of clients even for different applications.

Design

- *Openness*: Interoperability and portability depends on the openness of the middleware. For example, with CORBA you can use different programming languages on the client and server side and different CORBA implementations can communicate via the standardized Internet Inter-ORB Protocol (IIOP). To achieve portability in case of proprietary APIs (e.g. ActiveX, DCOM) requires much more discipline and work for the application programmer.

- *Stored procedures:* In combination with a Remote Database architecture stored procedures can be used to shift application functionality to the database server. The main force driving a stored procedure implementation is performance, but you possibly have to pay for it with a number of disadvantages such as proprietary language, no portability across vendor platforms and architectural mismatches. Besides performance stored procedures may also be attractive from a security point a view: many database systems support control and administration of access rights for stored procedures (e.g. rights to execute a stored procedure can be granted to specific database users).

- *Manageability*: Depending on the middleware, a Distributed Application Kernel is sufficiently manageable. You should consider using distributed TP environments for better manageability.

- *Scalability*: As you can use more than one application server, a Distributed Application Kernel Architecture is very scalable.

- *Load balancing*: There are many opportunities for performance tuning and for load balancing between clients and servers. The price you win for the high complexity of the architecture are very good performance tuning opportunities.

Known Uses

New client server development tools, such as Dynasty or Forté, support flexible partitioning of an application kernel into separate application processes and their deployment either on the client or server side. Forté even supports automated partitioning at run-time.

The OASIS framework [Moo97] allows persistent application kernel objects to be loaded on a client. This behavior comes close to Half-Object Assembly [Toe97].

3.4 Pattern: Remote Database (RDB)

Solution

Apply a client server cut below the database access layer. Persistence is provided by a remote (relational) database that is addressed via some remote database access protocol.

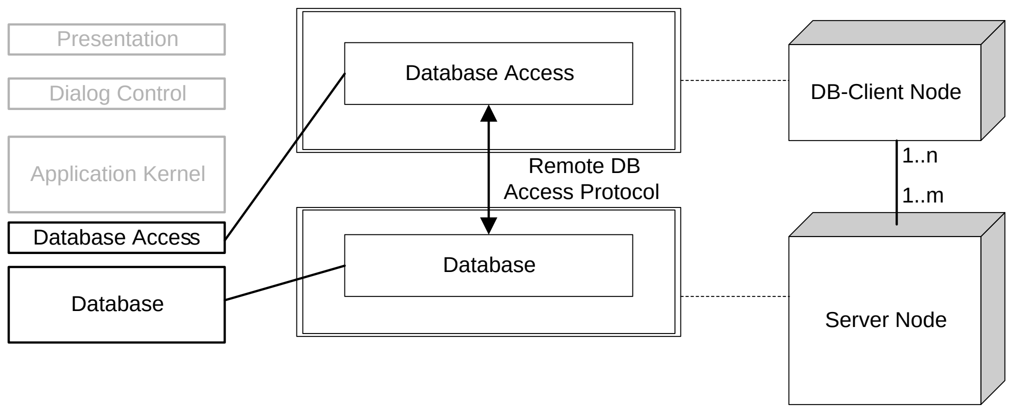


Figure 9: Structure of a Remote Database architecture

The client requests database services from the database server. The server node hosts a DBMS and offers services to the database access layer via some middleware. The services may comprise control (establishing an association between the client and server, managing database connections), transfer of database operations (e.g. in SQL) and parameters, transfer of resulting data, and transaction management.

Example

We implement the library application as an applet loading other java classes from the WWW-Server dynamically. The library data is stored in an SQL-Database which is accessed via a JDBC-API. The architecture is illustrated in Figure 10.

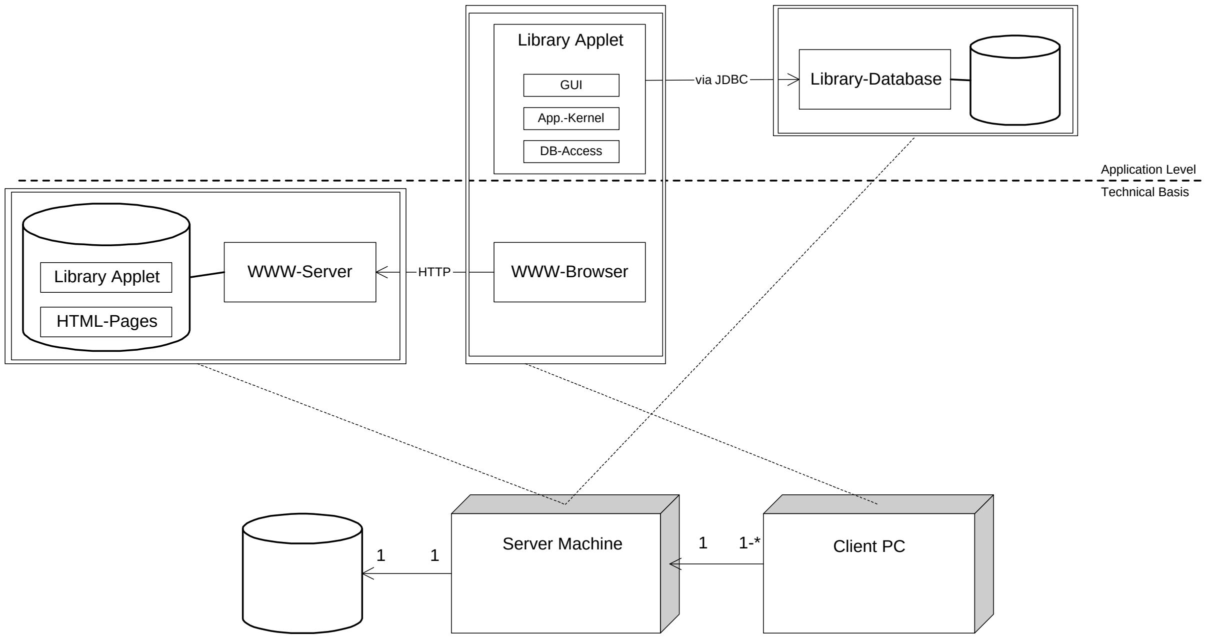


Figure 10: Sample architecture for a library system

Consequences

- *Business needs vs. construction complexity*: One of the main reasons for the success of RDB architectures is simplicity (e.g. no server software needs to be developed). Relational database vendors have started very early to bring their remote database protocols, proprietary stored procedure languages and 4GL tools to the market. The market is complemented by ODBC (Open Database Connectivity) protocols and ODBC adapters for nearly every client programming language. On the low end of the market, commodity products such as Visual Basic, and Microsoft Access, and Xbase products offer powerful support for simple dialog models.

- *Distribution vs. performance*: Network traffic is reduced compared to a Distributed Presentation Architecture but high compared to Remote User Interface and Distributed Application Kernel. The reasons for this have been discussed above. The architecture causes the least server load of all the C/S cuts discussed in this paper (as long as no stored procedures are used on the server). On the other hand, it causes the heaviest client load.

- *Processing style*: In a pure RDA architecture all data needed for batch processing have to be moved to a client machine. However, most client machines neither offer any support for batches nor have the I/O power you need. Additionally, the resulting traffic congests the network.

A solution for this problem is a Distributed Application Kernel. For example, we can replicate (parts of) the application kernel and run it on the database server (e.g. by use of stored procedures), or batches run on a separate batch server that is connected to the database server using a dedicated high speed LAN connection:

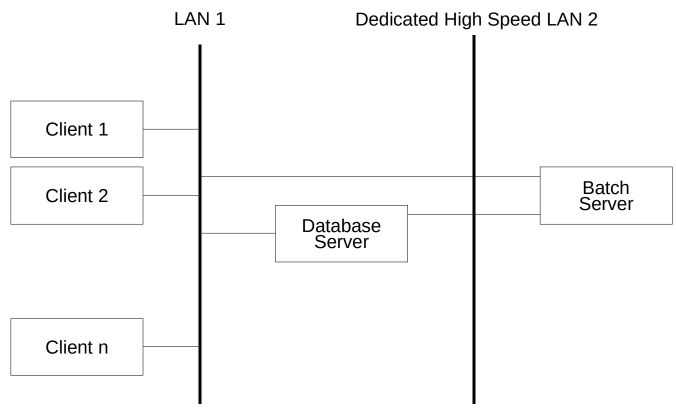


Figure 11: Solving the Batch Problem in RDB architectures

- *Distribution vs. security*: On the client as well on the server security depends on the operating system and on the quality of administration. Security inside a relational database is another problem. There are possibilities in SQL to `GRANT` access rights to users and user groups. These capabilities are seldom ever used as access control is much easier on a function level than on a data level. Hence you will often find slack security administration in relational databases.

- *Software distribution cost*: In a two-tier configuration you have to replicate the complete stack of a three layer architecture, except the database. This usually means moving Mbytes of software to every client machine on each major change of the installation. To tackle the problem you may use dedicated file servers for application software, automated software distribution, and configuration management tools. Still the data volume may exceed the network capacity, causing this solution to fail for large enterprises.

- *Reusability vs. performance vs. complexity*: The architecture does not promote reuse of application functionality but data can be shared by different client applications.

Design

- *Openness*: Most DBMS and protocols (e.g. ORACLE's SQL*Net) are proprietary products that do not support any portability. You may use a standard API, such as ODBC. However, there are several levels of ODBC compliance, also resulting in poor portability. Additionally, any use of stored procedures effects portability due to proprietary protocols and languages. RDA (e.g. [Lam94] provides an overview) is a communication protocol for remote database access that has been adopted as an international standard. It consists of two parts: The first part [RDA93a] specifies a generic model (services, protocol) for arbitrary database connection whereas the second part [RDA93b] specializes the model for connecting to SQL-databases. The RDA standard does not comprise any specification for an API, so it promotes interoperability but not portability.

- *Scalability*: Most RDBMS offer only pure support for multiple servers. We have found the following rules of thumb to be useful:
  - 4GL architectures should not be loaded with more than approximately 50 users.
  - RDB architectures without use of a transaction monitor should not be loaded with more than 100 users. Above that limit, a distributed transaction processing environment should be considered.

- *Existing legacy applications*: It is very hard to integrate existing legacy applications into a pure RDB architecture. You may couple old and new applications via a common database but this requires custom programming and is hard to do with plain 4GLs (see [KMW96] for more information).

- *Manageability*: SQL network protocols do not support management of large networks. You have to rely on the common products like SNMP.

- *Ease of creation:* Implementing a simple RDB architecture is easy compared to Distributed Presentation Architecture and Distributed Application Kernel. Especially 4GL tools will allow easy creation of prototypes and first cut applications. On the other hand, if it comes to maintenance, minimal code redundancy and elegant architecture, 4GLs are not the first choice. On the one hand, these tools may facilitate rapid application development, but on the other hand they are often restricted to two-tier architectures. Two-tier architectures are cut out for homogeneous environments and small span applications.

Related Patterns

Using stored procedures RDA can be combined with Distributed Application Kernel.

Known Uses

The RDB architecture can be found in the majority of all client server systems. It is broadly supported by existing database and 4GL products and tools. ORACLE Forms or Powerbuilder are two representatives of 4GL tools which allow easy realization of a RDB architecture.

sd&m has built various RDB architectures including the projects Champs, EASY C, EASY D1, and HYPO.

3.4.1 Pattern: Distributed Database

Solution

Apply a client server cut within the database component, so that application data can be distributed across several databases located on different nodes. The database access layer strives for transparency to hide distribution from clients' data requests. A distributed database management system (DDBMS) supports this approach very well.

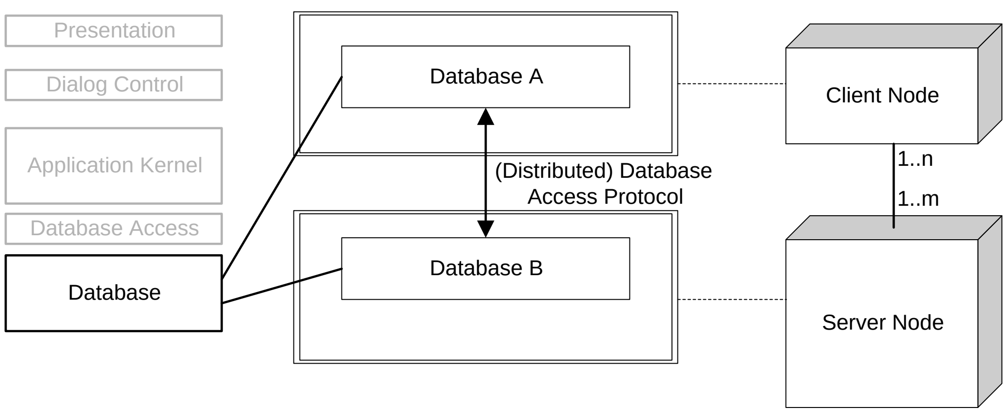


Figure 12: Structure of a Distributed Database architecture

Example

Consider a library system with a big central library and a number of local branches, which offer only the most popular books to their local clients. If clients search for a book not available on the local site it can be ordered from the central library. Instead of a single database server for the central library, local servers provide library databases for the administration of the locally available books. Other data such as common data of library users may be kept central.

Consequences

- *Business needs vs. construction complexity:* Organizations with a minor degree of shared data between geographically dispersed, autonomous departments may require a Distributed Database architecture although centralized data management is easier. Using a DDBMS offers a very elegant way to distribute application data without the application noticing it. Thus, the complexity for the application programmer can be reduced significantly.

- *Distribution vs. performance*: As with distributed file systems, you pay a certain performance penalty. The main factor that determines performance in a Distributed Database architecture is the ratio between data that are stored locally and data that are stored in remote (mounted) database tables. If all data are stored on the local machine, you have optimal performance. If all data are stored on a single remote machine, service times for simple queries (single table insert/update/delete) are longer than in a pure local configuration (a HYPO performance study on DDCS/2 with OS/2 and MVS RDA server shows an increase of 15-20%). The situation becomes worse for multiserver queries that span several remote databases. In this case, the local databases query optimizer may become pretty blind as he lacks the statistical data for an optimal query plan. Before you plan to implement queries that span multiple server, be sure the query plans are acceptable. The situation also becomes bad if the network is congested.

- *Processing style: A* Distributed Database is not suited for OLTP applications where a huge number of users share current data. Also for batch processing the architecture may result in poor performance.

- *Distribution vs. security*: Data being distributed among several nodes makes it more difficult to ensure data confidentiality, especially, if data is managed by different database systems.

- *Distribution vs. consistency:* If an application runs transactions which involve updates on distributed data, we need some transaction manager who ensures data consistency (e.g. via a two-phase commit protocol). Such a transaction manager can be part of a DDBMS or we can use a separate transaction monitor to coordinate updates on a number of (heterogeneous) databases.

- *Software distribution cost*: The cost of software distribution is even worse than in Remote Database architectures. Distributed Databases are heavyweight champions compared to some remote database access protocol proxies. This is the reason why you will seldom find a distributed database on an end user’s client machine but more likely on a department server.

- *Reusability vs. performance vs. complexity:* The same as in a Remote Database architecture.

Design

- *Openness*: Protocols like DRDA do not define a common API. Many vendors use proprietary SQL-APIs and protocols and offer DRDA-gateways to translate their protocols to DRDA.

- *Existing legacy applications*: Integrating existing legacy applications into a Distributed Database Architecture can be achieved mounting the old application's database tables. Powerful remote database architectures like IBM DRDA even provide gateways for old hierarchical IMS databases.

  On the other hand, integrating legacy applications via their data structures is harder than integrating them via wrappers.

- *Scalability*: Distributed Database architectures are not known for their above average scalability. There are strategies that help you improve overall performance when using a Distributed Database Architecture:
  - *Replicate read-only data*
  - *Compress data*
  - *Keep local data local*
  - *Data partitioning via business objects*

Related Patterns

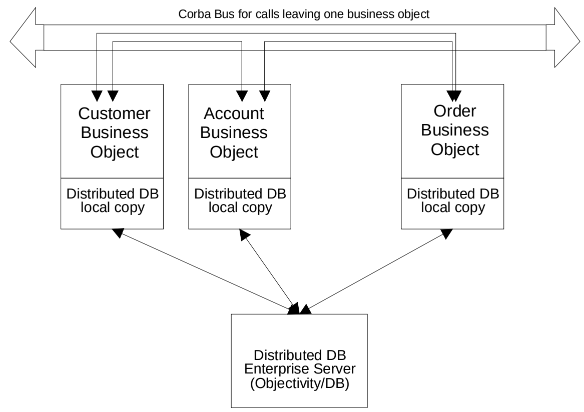
A pure Distributed Database architecture is hard to find in practice. It is more common to use a distributed database in combinations with Remote User Interface or Remote Database to build a three-tier architecture.

Known Uses

IBM's DRDA [DRDA] defines an interaction protocol for the communication between database clients and a database server. Client and Server databases complying to DRDA (e.g. via some gateway) can be combined to a distributed database system.

Hypo Bank in Munich, Germany uses OS/2 department servers, DDCS/2 and a MVS enterprise server together with the DB2 family of RDA products.

The IRIDIUM architecture [IRIDIUM], a very known project for a satellite based mobile phone system uses the following trick, to improve distributed database system performance: business objects are connected via an Object Request Broker bus and not via the distributed database. Internal business object traffic is done using the distributed database. This approach reduces database traffic with the Enterprise Server.



4 Acknowledgements

Thanks to António Rito Silva, the PLoP’97 shepherd for this paper.

5 References

[ACD+96] **Juan M. Andrade, Mark T. Carges, Terence J. Dwyer, Stephen D. Felts:** *The TUXEDO System, Software for Constructing and Managing Distributed Business Applications;* Addison Wesley, 1996; ISBN 0-201-63493-7

[BMR+96] **Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal:** *Pattern-Oriented Software Architecture - A System of Patterns;* John Wiley & Sons Ltd., Chichester, England, 1996; ISBN 0-471-95869-7

[CK97] **J. Coldewey, I. Krüger:** *Form-Based User Interface - The Architectural Patterns*; in **Frank Buschmann, Dirk Riehle (Eds.)**: *Proceedings of the 1997 European Pattern Languages of Programming Conference*, Irsee, Germany, Siemens Technical Report 120/SW1/FB, 1997

[Den91] **Ernst Denert:** *Software-Engineering - Methodische Projektabwicklung;* Springer-Verlag, Berlin Heidelberg New York; 1991, ISBN 3-540-53404-0

[DRDA] **IBM:** *Distributed Relational Database Architecture (DRDA*) - An Open Database Solution;* http://www.software.ibm.com/data/drda.html

[Gam92] **Erich Gamma:** *Objektorientierte Software-Entwicklung am Beispiel von ET++: Design-Muster, Klassenbibliothek, Werkzeuge;* Springer-Verlag Berlin Heidelberg New York, 1992; ISBN 3-540-56006-8

[Hir96] **Robert Hirschfeld**: *Three Tier Distributed Architecture*; Proceedings PLoP 96, Allerton Park, IL, 1996.

[IRIDIUM] **IRIDIUM Homepage**: http://www.iridium.com/

[KC96] **Wolfgang Keller, Jens Coldewey**: *Relational Database Access Layers: A Pattern Language*; Proceedings PLoP ‘96, Allerton Park 1996. Actual version can also be downloaded from http://www.sdm.de/g/arcus/cookbook/.

[KMW96] **Wolfgang Keller, Christian Mitterbauer, Klaus Wagner:** *Objektorientierte Datenintegration über mehrere Technologiegenerationen;* Proceedings ONLINE, Kongress VI, Hamburg, 1996

[Lam94] **Winfried Lamersdorf:** *Datenbanken in verteilten Systemen: Konzepte, Lösungen, Standards;* Vieweg , Braunschweig, 1994; ISBN 3-528-05467-0

[Moo97] **Conor Mooney**: *A Practical Framework for Distributing Business Objects*; Object Expert, Vol 2(2) Jan/Feb 1997.

[Ren96] **Paul E. Renaud**: *Introduction to Client/Server Systems*; Second Edition, Wiley 1996; ISBN 0-471-13333-7

[RDA93a] **International Organization for Standardization (ISO):** *Remote Database Access (RDA) - Service and Protocol*; International Standard 9579-1, ISO/IEC JTC1/SC21, 1993

[RDA93b] **International Organization for Standardization (ISO):** *Remote Database Access (RDA) - SQL Specialization*; International Standard 9579-2, ISO/IEC JTC1/SC21, 1993

[SG86] **R. W. Scheifler, J. Gettys**: *The X Window System*; ACM Transactions on Graphics 5(2), pp. 79-109, April 1986.

[Sim95] **David Simpson**: *A UNIX Server Is No Mainframe*; Datamation, December 15, 1995.

[Toe97] **Fridtjof Toenniessen:** *Half-Object Assembly - a Pattern System for Distributed Domain Objects in Business Applications*; in **Frank Buschmann, Dirk Riehle (Eds.)**: *Proceedings of the 1997 European Pattern Languages of Programming Conference*, Irsee, Germany, Siemens Technical Report 120/SW1/FB, 1997

[Wei97] **Charles Weir:** *Architectural Styles for Distribution . Using macro-patterns for system design;* in **Frank Buschmann, Dirk Riehle (Eds.)**: *Proceedings of the 1997 European Pattern Languages of Programming Conference*, Irsee, Germany, Siemens Technical Report 120/SW1/FB, 1997

[X/O96] **X/Open**: *Distributed TP: Reference Mode;* Version 3, X/Open Company Ltd., 1996; ISBN 1-85912-170-5

[zAPP] **Rogue Wave Software Inc.**: *zAPP Developer's Suite*; http://www.roguewave.com/products/zapp/index.html