TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

Master's Thesis

Extending Enterprise Applications with Valorized Edge Servers

carried out at the
Distributed Systems Group
Information Systems Institute
Vienna University of Technology

under the guidance of
Ao. Univ. Prof. Dr. Schahram Dustdar

by
Erich Liebmann <erich.liebmann@gmx.net>
Matriculation Number 9625216
Kalmanstrasse 63 – 1130 Vienna – Austria

Vienna, 06 March 2004

Location, Date *Signature*

Extending Enterprise Applications with Valorized Edge Servers
by Erich Liebmann

To my parents who always supported me

Abstract

The increasing number of service oriented collaborations, among dispersed business partners, poses new challenges for enterprise solutions. The conventional multitiered application model, advocated by current enterprise software architectures, exhibits insufficient performance and scalability in large scale deployments. The emerging edge server architecture aims to overcome these limitations and shortcomings, by introducing geographically distributed information brokers. Key challenges in such systems include an efficient decomposition of enterprise applications and a transparent and flexible integration of data dissemination mechanisms. Most current research proposals and commercial products are tailored merely towards content delivery networks and web application scenarios. We argue that a combination of the edge server architecture with application specific data dissemination, caching, and expiration techniques can result into a comprehensive architecture for service centric computing. This thesis report proposes strategies and guidelines for an efficient decomposition of enterprise applications and a seamless integration of a flexible data dissemination infrastructure. We eventually demonstrate the architecture, design, and implementation of a data service layer for the efficient dissemination of business data in large scale enterprise applications. The thesis report ultimately exemplifies business processing at the edge of the internet in two diverse application scenarios.

Keywords
multitiered enterprise applications, edge server architecture, decomposition, data dissemination, data caching, integration, design, push, pull, data service layer

Technologies
J2EE, EJB, JMS, Java RMI, JNDI, JDBC

Zusammenfassung

Die steigende Anzahl an servicebezogenen Interaktionen, von verteilten Geschäftspartnern, resultiert in neuen Herausforderungen für Unternehmenslösungen. Die herkömmlichen mehrstufigen Anwendungsmodelle, der derzeitigen Unternehmenssoftwarearchitekturen, weisen in stark verteilten Anwendungszenarien unzulängliche Geschwindigkeit und Anpassungsfähigkeit auf. Die hervortretende Edge Server Architektur versucht diese Einschränkungen und Mängel durch eine geografische Verteilung von Informationsbrokern zu umgehen. Besondere Herausforderung stellt in solchen Systemen die effiziente Dekomposition von Unternehmensanwendungen, sowie die transparente und flexible Integration von Datenverbreitungsmechanismen dar. Die derzeit gängigen Forschungsansätze und kommerziellen Produkte sind lediglich auf Inhaltsverbreitung und Web Anwendungen ausgerichtet. Unserer Meinung nach kann durch eine Kombination der Edge Server Architektur mit anwendungsspezifischen Verfahren zur Datenverbreitung, zum Daten Caching, und zum Datenverfall, eine umfassendere Lösung für serviceorientierte Anwendungen erzielt werden. Die vorliegende Diplomarbeit erläutert Strategien und Richtlinien für eine effiziente Dekomposition von Unternehmensanwendungen und eine nahtlose Integration einer flexiblen Infrastruktur zur Datenverbreitung. Es wird schließlich die Architektur, das Design, und die Implementierung einer Daten Service Schicht für die effiziente Verbreitung von Unternehmensdaten in weit verteilten Systemen demonstriert. Die Diplomarbeit erläutert abschließend die verteilte Verarbeitung von Unternehmensdaten anhand von zwei Applikationsszenarien.

Schlagworte

mehrstufige Unternehmensanwendungen, Edge Server Architektur, Dekomposition, Daten Verbreitung, Daten Caching, Integration, Design, Push, Pull, Daten Service Schicht

Technologien

J2EE, EJB, JMS, Java RMI, JNDI, JDBC

Acknowledgement

While there is only one author on the title page of this thesis report credit for its development is shared by many individuals. I would like to thank my supervisor, Schahram Dustdar, whose insight and expertise was invaluable in directing this research effort. His comprehensive industry and research experience has done much to improve the outcome of this thesis. I am especially grateful that he gave me the opportunity and eventually support in authoring a research paper on the results of this thesis together with him. Special thanks also for his guidance in scientific writing and his advices on searching through various digital libraries for appropriate research articles. I would also like to thank Andreas Lang at Progress Software for organizing an extended evaluation license of the message oriented middleware provider Sonic Software SonicMQ.

I would like to thank Gottfried Szing for interesting and sometimes controversial but always informative discussions that ultimately lead to the fundamental inspiration for this research effort. His industry experience was crucial to the technical and conceptual accuracy of vendor offerings and methodologies utilized. I would also like to thank my friends and study mates for tolerating the flurries of research and writing and their support in times of trouble. I owe an extra thanks to Alois Dornhofer, Christopher Dräger, Ulrike Eisenmann, Martin Eller, Sascha Frühwirth, Franz Grabner, Michael Kalkusch, Bernhard Mecl, Jürgen Mischker, Christopher Neufeld-Chalupa, Bernhard Schiemann and my sister Karin Liebmann for aiding me during the sometimes difficult times of this endeavor. Of special note is Bernhard Schiemann who reviewed parts of this work and whose constructive criticisms ultimately helped to improve this report, Sascha Frühwirth for discussions on several conceptual details and providing valuable feedback, Alois Dornhofer for motivating discussions and practical advices, and Bernhard Mecl for numerous inspiring online conversations and remote assistance with minor technical issues.

Finally I extend the most sincere gratitude to my girlfriend, Maliwan Pawata, for supporting and assisting me through almost a whole year of research and writing. Without her unfailing support and love it would have been much more difficult to cope with this occasionally difficult time. Her gentle encouragement, spirit, and confidence are an inspiration.

Table of Contents

List of Figures

Chapter 1

Introduction

The increasing number of global business collaborations performed over wide area networks and the internet pose new communication, availability, and scalability issues for traditional centralized architectures. The conventional client and server architecture and modern multitiered enterprise applications exhibit insufficient performance and scalability in large scale deployments. Appropriate performance for widely dispersed clients can usually not be achieved. The emerging *edge server architecture* aims to overcome these limitations and shortcomings by extending the traditional centralized approach with additional geographically distributed servers. As a result, expensive communication across wide area networks can be minimized and clients experience improved service response time and availability.

1.1 Motivation

The deployment of the edge server architecture is intriguing, but at the same time raises numerous open issues and research questions. Key challenges in such systems include a comprehensive decomposition of enterprise applications and a transparent integration of efficient and flexible information dissemination infrastructures. A lack of guidelines and strategies limits most recent research approaches and commercial products to content delivery networks and web application scenarios. Moreover, merely plain services are currently delegated to edge servers while processing of complex business tasks remains at the central backend system.

We argue that a combination of the edge server architecture with application specific data dissemination, caching, and expiration techniques can result into a comprehensive architecture for service centric computing. With the evolution of distributed systems a lot of research has been conduct with the development of efficient dissemination and caching techniques. Numerous research papers discuss the development of caching algorithms and the improvement of dissemination schemes. Further activities involve the evaluation of application specific solutions that leverage the domain knowledge of applications to improve data exchange and caching. Unfortunately a lack of concepts for the integration of dissemination and caching approaches into modern multitiered enterprise applications can be identified. Furthermore, most application specific solutions are merely applicable to a small number of application scenarios and can not be transferred to distributed systems in other domains.

To reveal the full potential of the edge server architecture it is crucial to define guidelines and strategies for the efficient decomposition of multitiered enterprise applications. Moreover, an architectural approach is required for a seamless integration of application specific data dissemination, caching, and expiration techniques into enterprise applications.

1.2 Goal

This thesis report aims to provide an architectural proposal for the extension of multitiered enterprise applications with valorized edge servers. The focus is on establishing strategies and guidelines for a seamless integration of flexible data distribution infrastructures into existing and newly built enterprise applications. We research into the definition of a transparent communication layer that:

- supplies a flexible and efficient connectivity among dispersed systems
- enables execution of business tasks directly at the edge of the internet
- allows dynamic and personalized content creation at edge servers
- preserves value added services provided by modern application servers
- streamlines the definition of application specific algorithms
- facilitates direct comparison of dissemination and caching algorithms
- influences the development of business logic merely slightly

Our primary goal is a significant performance and scalability improvement of service centric applications in large scale deployments. The emphasis is on the expansion of the current edge server architecture towards service oriented computing. The thesis report explores a novel decomposition of enterprise applications which allows distribution of valorized edge servers to diverse geographical regions. We exemplify the implementation of an application and system model which allows the execution of business tasks directly at the edge of the internet. We aim at providing a flexible interconnectivity of dispersed enterprise systems which facilitates service collaborations across wide area networks. The emphasis is on the design of an application agnostic communication layer which transparently handles data replication and dissemination. We discuss best practices on successful integration of flexible data replication, dissemination, and caching techniques into enterprise applications.

The emphasis is on decomposition and integration issues rather than the definition of tailored solutions for particular application scenarios. The thesis researches into the establishment of a general foundation which aids software engineers in implementing application specific dissemination, caching, and expiration solutions. Particular emphasis is laid on the seamless integration with existing and widely adopted enterprise technologies. We focus on the preservation of value added services and aim to minimize the impact on the business logic. The thesis suggests mechanisms which allow a seamless combination of complete and partial data replication, speculative data dissemination, and application specific caching and expiration logic.

1.3 Outline

The remainder of this thesis is organized as follows. The first two chapters cover the state of the art and provide background information for the ongoing discussion. The first part of chapter 4 elaborates the problem statement while the second part provides the ideas, methodologies, and concepts that lead to the given solution. Chapter 5 covers the architectural proposal and is essential to the development of the reference implementation. In chapter 6 we discuss the design and implementation of the reference implementation. Finally, in chapter 7 we evaluate the presented architecture, show two concrete application scenarios, and place the work in the context of related approaches. A concluding summary of the presented work can be found in chapter 8.

Chapter 1, *Introduction*
This chapter gives a general overview of the problem domain, describes the motivation, and defines the goals of this work.

Chapter 2, *Enterprise Software Architectures*
This chapter provides a short overview of the evolution of distributed systems, explains the term *multitiered enterprise applications*, defines common value added services, and shows how they form the underlying technology of the presented architecture.

Chapter 3, *Dissemination and Caching*
This chapter walks the reader through the development of dissemination and caching techniques and defines the term *edge server architecture*.

Chapter 4, *Extending Enterprise Applications with Valorized Edge Services*
This chapter covers a conceptual analysis of the expectations and issues of an extension of enterprise applications with edge services. Furthermore, the suggested application and system model is discussed and design considerations are depicted.

Chapter 5, *Architecture*
This chapter outlines the architectural proposal for the extension of enterprise applications with valorized edge services.

Chapter 6, *Design and Implementation*
This chapter contains a detailed analysis of the design and implementation of the reference implementation.

Chapter 7, *Evaluation and Comparison*
This chapter describes two concrete application scenarios for the presented architecture, analysis the forces and benefits of the presented approach, places the work in the context of related research, and outlines future work.

Chapter 8, *Conclusion*
This chapter provides a brief summary of the presented work.

Chapter 2

Enterprise Software Architectures

Distributed computing allows parts of a business system to be located on separate computers. In other words, distributed computing facilitates the collaboration with business logic and data at diverse locations. Business logic is frequently defined in terms of a server side component model and subsidized by value added services of an application server. The following sections in this chapter provide the conceptual foundation for the extension of multitiered enterprise applications with valorized edge servers. The focus is on a general introduction of the technologies and concepts utilized for the design and implementation of the proposed architecture.

2.1 Enterprise Computing Technologies

Enterprise applications are usually deployed in an environment made up of heterogeneous networks consisting of computers ranging from large mainframes and supercomputers down to generic personal computers. This would not be possible without the same fundamental communication infrastructure and standardized server side component models.

2.1.1 Distributed Object Architectures

In contemporary distributed applications, communication is usually performed with *distrusted object technologies* such as Sun Microsystems Java RMI [52], OMG CORBA [92], and Microsoft DCOM [120]. These technologies allow applications on one machine to use objects on a different computer by communicating across the network.

The basis of every distributed object system is the remote method invocation (RMI) protocol* which is responsible for providing location transparency. It is beyond the scope of this chapter to describe the details of RMI protocols including such concepts as stub and skeleton. In general location transparency enables clients to communicate with server objects regardless of the location of the object. Clients may be located on the same machine or on a different network; the basic interaction is always the same. The main difference between local method calls and remote method calls is performance.

* The term RMI protocol is used to describe distributed object protocols in general while the Java language version of a distributed object protocol will be referred to as Java RMI.

The architectural proposal in the following chapters aims at providing design strategies that minimize remote communication while maximizing the distributed system's performance.

In addition to location transparency distributed object technologies offer error and exception handling, parameter passing, and other services such as passing of transaction and security context. An object request broker (ORB) usually provides a naming system for the location of remote objects and supports many other features such as reference passing, distributed garbage collection, and resource management. The remaining chapters show the importance of these additional services for the extension of enterprise applications with edge servers.

It should be noted that most ORBs are merely facilitating communication between distributed objects. Support for additional services such as transaction management and security is not managed automatically and needs to be implemented by application developers. In many application scenarios this is, however, a cumbersome task which limits reusability, flexibility, and extensibility.

2.1.2 Enterprise Messaging Systems

In addition to direct synchronous communication, outlined in the previous section, most modern enterprise application architectures offer support for asynchronous message oriented communication.

Applications connected with asynchronous messaging systems exchange information in form of messages. A message, in this context, is a self contained package of business data, message properties, and network routing headers. Enterprise messaging systems are usually utilized to exchange information on business transactions and to inform an application of some event or occurrence in another system. Each message is treated as an autonomous unit and is assumed to carry all data and state needed by the recipients to process it independently.

The proper distribution of asynchronous messages among applications is performed by a message oriented middleware (MOM). This message broker usually provides built in value added services such as transactional support, fault tolerance, and load balancing. Numerous MOM vendors, such as Progress SonicMQ [108], IBM WebSphere MQ [63], Microsoft Message Queuing [82], and Fiorano FioranoMQ [44] offer products ranging from centralized architectures that depend on a message server to perform routing, to decentralized architectures that utilize multicast protocols at the network transport layer for message distribution. Furthermore, each product uses different message formats and network protocols and supports a variety of different payload types to exchange messages. Nevertheless, the basic semantic is always the same; a vendor supplied API is used to construct messages, specify the payload (business data), define message properties, set quality of service attributes, assign routing information, and hand the message off to the MOM provider for delivery.

All modern MOM providers utilize virtual channels, usually referred to as *destinations*, for the exchange of messages. Virtual channels allow a sending application to address a specific destination instead of a particular target application. The receiving application, on the other hand, can register or subscribe to a destination of interest instead of listing for incoming messages from a particular sender. In this way, systems communicating through MOM are abstracted and decoupled from each other. In particular,

the sender of an asynchronous message is not required to wait until the message is received or handled by the recipient; it is free to continue processing directly after dispatching the message. Consequently, messages can be delivered to systems that are not currently running and processed when it is convenient. Furthermore, the sender of asynchronous messages is not required to be aware of the number of receivers and the receiver of asynchronous messages does not need to be aware of the number of senders. Sending and receiving asynchronous messages to and from destinations involves interactions between the client and the MOM provider. Further message routing and delivery is handled exclusively by the MOM provider. In general, enterprise messaging systems allow developers to build flexible distributed systems without the requirement to tightly coupling.

Two messaging models can be identified in enterprise messaging systems, publish-and-subscribe for one-to-many broadcast of messages and point-to-point for one-to-one delivery of messages. Some MOM providers support both messaging paradigms while others rely on a single model. Typical areas of application for the deployment of enterprise messaging systems include:

- enterprise application integration
- business-to-business solutions
- prevailing geographic dispersion

The following chapters present an architecture which is concerned with the extension of enterprise applications with geographically distributed edge servers. We leverage the benefits of enterprise messaging systems to bridge geographic dispersion and to avoid tightly coupling.

2.1.3 Component Models

A flexible, extensible, and reusable design can be implemented by taking advantage of object oriented programming (OOP) languages. Accordingly, most enterprise applications utilize OOP concepts to implement business logic and to define the presentation logic of services and products offered. With short lifecycles of commercial services and products and rapidly changing business models of today's organizations the encapsulation of business logic into *business objects* is crucial. The objectified version of business processes, products, and services ensures flexibility, extensibility, and reusability and therefore facilitates business evolution.

A component model defines a set of contracts between the component developer and the environment that hosts the component. It defines how a component should be developed, configured, packaged, and deployed. A component is a single class or a set of cooperating classes that make up a reusable design for a specific purpose. Components can be described as an independent piece of software tailored to be distributed and reused in several applications.

While client side components models, such as Microsoft ActiveX [81] and Sun Microsystems JavaBeans [53], handle presentation and user interface issues, server side component models, such as Microsoft COM+ [119], Sun Microsystems EJB [86], and OMG CCM [93], facilitate the development of complex enterprise solutions. The architecture

presented in the following chapters is built on top of a server side component model. Most server side component models define an architecture which combines the benefits of business objects with the accessibility of distributed objects. Server side components are usually deployed in an application server which manages the components at runtime and makes them accessible to local and remote clients. The utilization of server side component models makes it significantly easier to develop distributed business objects and to assemble them into business solutions. In addition, several modern server side component models define standardized deployment attributes for each component. Thus, runtime behavior of components can be adapted without modifying the source code or redefining application specific configuration options.

2.1.4 Transaction Processing Monitors

The growing needs of business applications resulted in the development of server platforms that provide automatically managed value added services. This kind of software platform is usually referred to as transaction processing (TP) monitor.

The TP monitor industry dates back to early days of distributed computing when procedural programming languages such as COBOL and C and remote procedure calls (RPC) where the prevailing case. Successful TP monitoring products, such as IBM CICS [61] and Bea Tuxedo [16], automatically manage the entire environment in which business applications process tasks and offer services. Application programmers can rely on the TP monitor to handle transactions, perform resource management, and ensure fault tolerance.

The automatic management of value added services not only simplifies the development of business applications, it makes them less error prone, more flexible, and increases reusability. The following chapters illustrate that preserving the automatic management of value added services is one of the key challenges in extending enterprise applications with edge servers.

The long evolution made TP monitors to one of the most reliable server platforms available. Hence there are still many mission critical systems in use that relying on the infrastructure provided by these software products. The non object oriented design (OOD) comes, however, with extensive limitations and drawbacks. Business solutions developed for TP monitors are required to express business logic in terms of procedures that are not as flexible, extensible, and reusable as business objects. Furthermore, the advantages of RMI, including support for exception handling, the negotiation of transactional scope, and security context, are not available with traditional RPCs. Moreover, the benefits of component models, that built upon object oriented software, can not be leveraged and need to be defined in terms of application specific solutions. Most importantly, however, a RPC is like executing a static method, there is no concept of object identity and therefore no built in support to keep track of conversational state.

2.1.5 Application Servers

The most recent server platforms for business solutions combine object request brokers, server side component models, and transaction processing monitors into a comprehensive enterprise software development environment. While more precise terms

such as component transaction monitor (CTM), object transaction monitor (OTM), component transaction server, and distributed component server are occasionally used to refer to this kind of server platform, the presented work uses the general term *application server*. Successful application server products with a large market share include Microsoft Transaction Server [84], Bea WebLogic Server [17], IBM WebSphere Application Server [62] and Oracle Application Server [96].

An application server coalesces enterprise computing technologies to provide a robust yet flexible infrastructure for mission critical work. Modern application servers (1) provide a communication backbone that enables client applications to locate and utilize distributed objects, (2) implement a robust server side component model that streamlines the development, configuration, and deployed of business objects, and (3) supply an infrastructure that can automatically manage transactions, concurrency, security, persistence, resource management, and failover. Standardized deployment attributes, supplied by the server side component model, are frequently used to define how an application server manages business components at runtime and provides access to local and remote clients.

In addition to the above mentioned features, an application server usually provides access to enterprise information systems and incorporates connectivity to enterprise messaging systems. Furthermore, a web server is included in a number of application servers that can be used to present a web based interface to clients.

In general, application servers come with several application programming interfaces (APIs) that allow developers to work with backend databases, perform remote method invocation, access naming and directory services, and utilize enterprise messaging systems. The primary business object component model is usually extended with a web component model for the development of comprehensive presentation logic and an application client component model for the implementation of flexible client applications. Depending on the application server and the enterprise software architecture additional APIs for, performing parsing and transformations of XML, managing distributed user transactions, supporting web services, handling authentication and authorization, supporting standardized resource adapters, and sending emails are supplied. In essence, application servers implement a surrogate architecture that works by intercepting method calls and inserting services based on attributes defined at deployment time. The surrogate mediates between clients and components providing value added services transparently.

The remaining chapters present an architectural proposal which provides a flexible and efficient communication infrastructure among several application servers. It is one of the key design principles to preserve the sophisticated features and services offered by modern application servers. To achieve a seamless integration the utilization of several enterprise computing technologies outlined in this chapter is crucial.

2.2 Multitiered Enterprise Applications

Comprehensive enterprise solutions divide application logic into several functional software components. A multitiered distributed application model enables application components and enterprise information systems (EIS) that make up an enterprise application to be deployed at several distinct machines. In general a modern enterprise

application consists of a client tier hosting application client components, a web tier hosting web components, a business tier hosting business logic components, and one or more backend tiers hosting enterprise information systems. The individual tiers of a multitiered enterprise application are usually distributed over three distinct locations (client machine, application server, and backend system) and are therefore frequently referred to as three tiered applications. Three tiered and multitiered applications extend the standard two tiered client and server model by placing a flexible and powerful application server between the client application and the backend system. The intermediate tier, among other things, allows a reduction of the size and complexity of client programs, enables caching and control of data flow for better performance, and can provide security for both data and user traffic.

Client Tier
: The clients of a multitiered enterprise application are either web clients or application clients. Web clients utilize a web browser to access a web server and ultimately web components deployed in the application server. The web components generate dynamic web pages consisting of various types of markup languages, such as XHTML and XML, which are rendered at the client side by the web browser. All interactions between the client and the enterprise application are performed with this page oriented graphical user interface. More sophisticated interactions can be supported with the use of client side scripting languages and browser plug-ins. An application client, in contrast, provides a regular graphical user interface or a command line interface and directly accesses the business components hosted in the application server.

Web Tier
: Most enterprise software architectures provide two different types of web components, one tailored towards web designers and one tailored towards software developers. While the former provides a mechanism to insert dynamic elements, custom tags, and programming constructs into static markup pages the latter provides an API for handling of request and response based protocols. In both cases a web browser request is processed and responded with dynamically generated markup documents.

Business Tier
: The business logic components running in the business tier solve or meet the needs of a particular business domain. In general a business object receives data from client applications, processes it, utilizes the backend tier for data storage and retrieval, and sends processing information back to the client program.

Backend Tier
: The backend tier is made up of one or more enterprise information systems, such as relational database management systems (RDBMS), object relational database management systems (ORDBMS), object database management systems (ODBMS), enterprise resource planning (ERP) systems, mainframe transaction processing systems, and other legacy information systems. Usually the entire data processed by the business tier is spun off to EISs that guarantee data consistency and handle housekeeping tasks required for data storage.

The structure implied by the multitiered distributed application model for enterprise applications is outlined in figure 2.1. It might be anticipated that an efficient connec-

tivity between the business tier and the backend tier is crucial to the overall performance of the business solution. Without a sufficient network connection, data storage and retrieval operations will soon become the bottleneck of the enterprise application. It comes as no surprise that in most real word scenarios the application server and the backend systems get deployed geographically close to each other, usually within the same building. The increasing number of global business collaborations emphasizes the drawbacks of this approach. Client applications that need to access business services across wide area networks usually experience slow response times and long data transfer delays. This thesis is precisely concerned with this issue and proposes the extension of enterprise applications with edge services. A concrete architectural proposal and a detailed analysis of the consequences and implications can be found in the following chapters.

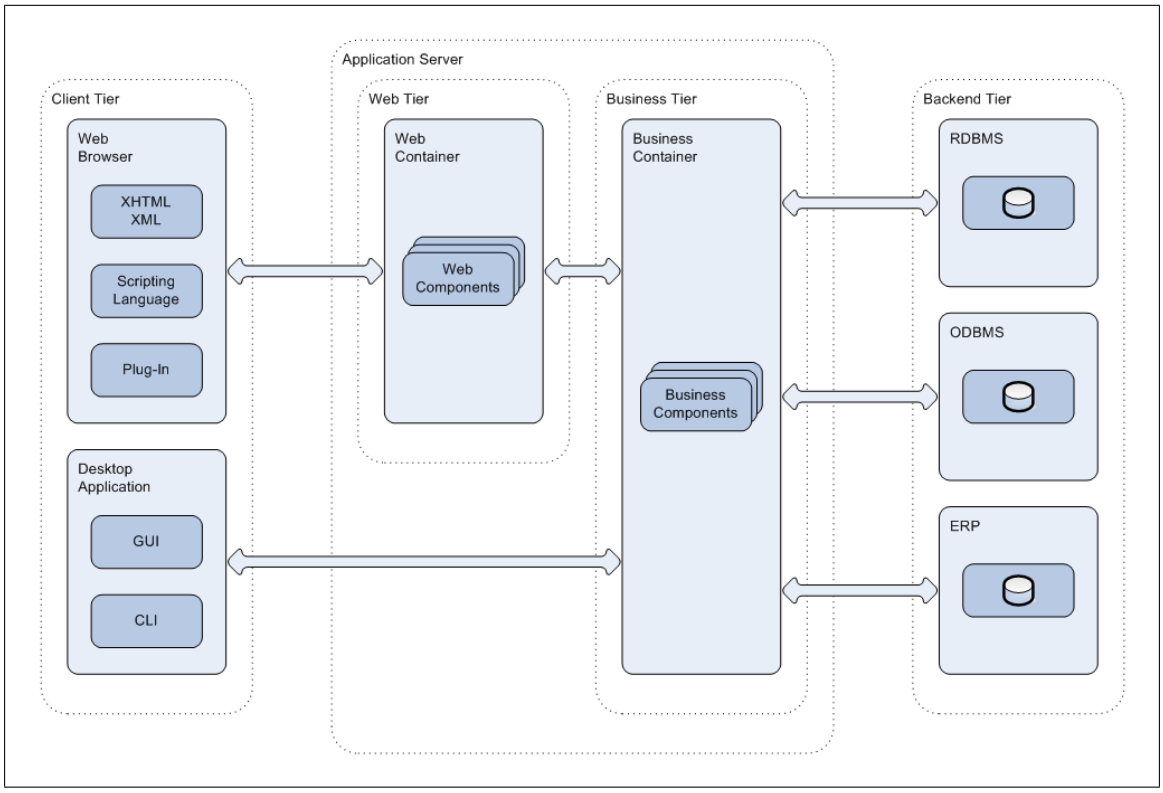

Figure 2.1: Multitiered Enterprise Applications

Multitiered distributed application models for enterprise applications together with the broad range of supporting APIs that come with modern application servers and middleware providers will be referred to as *enterprise software architecture*.

In essence, there exist two competing enterprise software architectures today, Sun Microsystems Java 2 Enterprise Edition (J2EE) [43] and Microsofts .NET framework [83]. While J2EE is open standard based, programming language centric, and platform neutral, .NET is proprietary, operating system centric, and programming language neutral. In particular developers are restricted to the Sun Microsystems Java programming language in J2EE and to the Microsoft Windows operating system in .NET. It is interesting to observe that Sun Microsystems managed to rally the entire industry behind its J2EE standard, including the top vendors of application servers and

transaction processing monitors, such as IBM, Bea, Oracle, and TIBCO, while .NET is Microsoft's sole effort to grab the enterprise software market share. The architectural proposal and the reference implementation outlined in the following chapters present a solution for the extension of J2EE based enterprise applications with edge servers.

2.3 Enterprise Application Services

One of the fundamental benefits of using application servers is that they handle resource management and manage value added services such as concurrency, transactions, persistence, naming and security automatically. It is evident that any extension of enterprise software architectures, such as the extension of the application model with edge servers, has to preserve these primary services. The following description provides a short overview of the most critical services and the way they are supported by modern applications servers.

Resource Management
: Most application servers support two mechanisms, *resource pooling* and *resource swapping*, to increase performance and reduce resource consumption. The concept of resource pooling reduces the number of object instances by introducing object sharing. It minimizes the overhead inherent in creating and destroying business objects and administrative objects, such as database and middleware connections. Resource swapping is a technique which allows to passivate idle business object instances to secondary storage and to activate them on subsequent requests.

Concurrency
: In distributed systems concurrency occurs when several client applications access a given shared resources at a time. Application servers handle concurrency automatically according to component attributes set by the software developer at deployment time. The component configuration defines if the application server ensures thread safety or allows reentrance for a given business object.

Transactions
: In traditional enterprise systems software developers used specific APIs to declare transactional scopes and to commit or rollback transactions. Virtually all application servers provide business component developers with the possibility to set transactional scopes and attributes within the component configuration. In addition some enterprise software architectures support explicit transaction management where automatic management is not sufficient or not applicable.

Persistence
: The slow adoption of ODBMS together with the pervasiveness of legacy systems that support relational data access demands for flexible solutions for object-to-relational persistence mechanisms in enterprise solutions. This is where modern applications servers come into play, managing business object persistence automatically. The software developer uses the component configuration to define the mapping of relational database tables to the fields of a business class, everything else, including database connection pooling and transactional database access, is handled by the application server.

Naming

All distributed object architectures include a directory or naming service that provides clients with a mechanism for binding and locating distributed objects or resources. While *object binding* is the association of a distributed object with a natural language identifier, *object lookup* allows clients to request a remote reference to a specific object. The naming and directory service is so fundamental in distributed objects systems that virtually every application server comes with an integrated naming service and usually provides connectivity to several foreign directory services that store references to enterprise resources.

Security

A secure enterprise solution usually involves three mechanisms: *authentication*, *authorization* (access control), and *secure communication*. Authentication and secure communication is usually provided by dedicated enterprise APIs and implemented on a per application basis. It is not uncommon that vendor specific management tools for user roles and user management are provided together with the application server. Access control, on the other hand, is often ensured on a component basis, once again through the definition of attributes in the component configuration.

The terms *primary services* and *value added services* will be used to refer to the features of application servers outlined above.

Chapter 3

Dissemination and Caching

The communication characteristics among machines participating in a distributed system are the determining factor for the efficiency and applicability of distributed software solutions. An elaborate decomposition into individual nodes and a careful design of the communication infrastructure is required to assure efficient information exchange and to enable the adaptation to changing requirements.

With the evolution of the internet and the increasing number of large scale distributed systems a number of dissemination and caching techniques have been devised and analyzed. This chapter provides a brief summary of communication patterns, explains the basic concepts of application specific dissemination and caching solutions, and introduces the edge server architecture as the theoretical foundation for the extension of enterprise applications with edge servers.

3.1 Communication Patterns

In general, propagation of information and events can be achieved by pushing or pulling data either periodically or on demand (aperiodic) using unicast, multicast, or broadcast communication.

The prevailing communication model in distributed systems, request-response, involves an explicit client request followed by a corresponding server response. This client initiated communication pulls information from the server, usually as a result of a user interaction or an application event. In contrast, push based data delivery originates from the server application that informs one or more clients of a state change or propagates newly available data items.

Both push and pull can be performed either on demand (aperiodic) or periodic, depending on the requirements of the distributed system. Aperiodic data delivery is event driven, whereas a pull request is usually triggered by a user interaction or a lack of necessary data while a push based transmission is frequently initiated upon data updates and application events. A periodic data delivery is triggered by a timer sometimes with some degree of randomness.

Unicast, multicast, and broadcast communication indicate the number of machines involved in a particular data delivery. While unicast is a point-to-point communication, and multicast addresses a specific subset of clients that indicated an interest in the

data provided by a server application, broadcast distributes information to all nodes in a distributed system.

The following description provides a short definition of the individual communication patterns including the introduction of additional terms required for the discussion within the following chapters. A more detailed characterization of data delivery mechanisms including references to sample applications, typical usage scenarios, and further research can be found in related research papers [45, 46].

Aperiodic Pull
: Traditional request-response communication exhibits an aperiodic pull over a unicast connection. If, however, the information requested by a specific client application is sniffed or snooped by other clients a pull based multicast or broadcast is performed.

Periodic Pull
: The periodic pulling of information is often referred to as polling which is usually performed by monitoring applications and messaging systems that lack support for push communication. It is uncommon but possible that the polled data is snooped by other client applications in a multicast or broadcast fashion.

Aperiodic Push
: In most application scenarios aperiodic push is based on the publish-and-subscribe paradigm that allows client applications to indicate an interest by subscribing to a specific topic. A server application is responsible for pushing or publishing data updates for a specific topic to all subscribed clients. Depending on the number of clients subscribed to a specific topic the data is unicast (for a single subscribed client), multicast (for several subscribed clients), or broadcast (if all clients are subscribed).

Periodic Push
: Periodic push is frequently used to broadcast constantly changing information to a large number of client applications. In contrast to aperiodic push, the time triggered delivery, usually relieves client applications from the requirement to subscribe to a specific topic of interest. All data available at the server is pushed according to a predefined schedule to clients available in the distributed system.

Distributed systems that rely on pull based communication impose a large communication overhead, in terms of the number of request-response interactions, especially when the number of client applications is large. Moreover, a pull based approach lacks adequate fidelity in systems with frequent data updates or in scenarios with stringent coherency requirements. In contrast, push based communication offers high fidelity for rapidly changing data and satisfies stringent coherency requirements. However, the infrastructure requirements for efficient push based data delivery are complex and difficult to implement on a large scale [122, 75]. The publish-and-subscribe [42] paradigm requires servers (or at least the connecting middleware) to keep state of subscribed clients and their interests. Furthermore, an efficient data delivery can be achieved only with the support of the underlying network infrastructure. Internet protocol (IP) multicast, for instance, is well suited for push based data delivery [95, 28]. Unfortunately the support of push enabling technologies in large scale networks is insufficient

and usually limited to individual autonomous systems (AS) managed by a single administrative authority such as an internet service provider (ISP). The total number of ASs (presented in the internet today and graphically illustrated by the skitter AS internet graph [23]) indicates, however, that a large scale push based system cannot rely on supporting technologies of the underlying network infrastructure. Consequently, several existing push systems on the internet are actually implemented using polling in order to emulate push behavior, which is inherent inefficient.

A further minimization of communication and data dependency among individual computers in a distributed system can be attained by caching data items. Virtually every distributed system relies, in addition to a dissemination infrastructure, on some sort of caching mechanisms to increase overall performance and availability.

3.2 Application Specific Solutions

The limitations and drawbacks of pure push and pull based data exchange outlined in the previous section lead to the development of more flexible dissemination and caching strategies for distributed systems. Several proposals [2, 38] suggest the utilization of a mix of both push and pull for data propagation. Depending on dissemination logic the distributed system will favor push or pull to minimize the number of network traffic and data dependency among individual computers. Further research suggests the commitment to sophisticated server initiated dissemination schemes [20] including the use of periodic broadcast push [1], on demand data broadcasting [72], and continuous multicast [101]. Moreover, speculative data propagation is introduced by a number of research papers [36, 19, 22] that focus on establishing mechanisms for the prediction of data requirements and the proactive dissemination of data in order to reducing expensive roundtrips to the origin site.

The diverse requirements, data characteristics, and usage patterns combined with the deviating decompositions of distributed systems result in application specific, dissemination and caching solutions. A majority of proposals are tailored to a particular application scenario or software product and consequently are difficult to compare with, reuse in, and port to different distributed systems. In general, most distributed systems presented in research papers, are specifically designed to incorporate *application specific dissemination* (ASD) strategies which renders the integration into existing systems difficult and complex. Furthermore, the implications and benefits of the integration of ASD strategies into real world distributed applications are difficult to predict and analyze. With all this in mind, it comes as no surprise that most ASD proposals are only present in prototype and reference implementations and never reached wide acceptance.

In particular, we identify a lack of guidelines and blueprints for the integration of ASD into existing and newly built distributed applications. The following chapters are concerned with the seamless combination of ASD and enterprise applications in order to facilitate the extension of multtiered enterprise applications with edge servers. The main aspect is on the preservation of enterprise application services (see section 2.3 for details) provided by modern enterprise software architectures.

Despite the application specific nature and diverse realization of flexible dissemination and caching strategies for distributed systems we identify the following common

characteristics, challenges, and goals.

Characteristics
Application specific dissemination leverages data usage and application domain characteristics of distributed systems to provide a more efficient data exchange solution as compared to pure pull and pure push based dissemination. The basic building blocks are (1) *dissemination logic* that decides which data items are disseminated at a given time and in which way, (2) *flexible infrastructure* that supports the semantics required by the dissemination logic, and frequently (3) *caching logic* that allows the caching of data items, often in close cooperation with the dissemination logic.

Challenges
The (1) seamless integration into existing and newly built distributed systems together with the (2) definition and analysis of dissemination and caching algorithms form the key challenges of application specific dissemination. It is crucial to ensure a transparent integration to relieve application developers from understanding the distribution and consistency semantics introduced by ASD, which complicates the process of application development. Furthermore, it is important to support existing technologies in the application domain addressed by the ASD strategy to make a wide adoption possible. In addition, most ASD approaches require metadata, context information, or other parameters to tune dissemination and caching logic. Clear guidelines need to be supplied for the gathering of information required for the successful deployment.

Goals
Several ASD proposals aim to (1) minimize network traffic and reduce data dependency among individual computers in a distributed system, (2) avoid latency by diminishing expensive roundtrips to the origin site, (3) optimize fidelity in systems with frequent data updates and compulsive coherency requirements, and ultimately (3) improve performance and availability.

The communication infrastructure for ASD is required to be flexible and adaptable in the sense that it (1) supports several different communication patterns to choose from, (2) allows dissemination logic to influence communication behavior among individual machines, and (3) supports a seamless combination with distributed applications without impact on existing value added services.

Most ASD solutions are designed for use in large scale distributed systems such as the internet and limited to the dissemination of static content, such as web pages. The distributed data is usually cached or processed without further interaction with the origin site.

3.3 Edge Server Architecture

The internet is a communication infrastructure that interconnects a global community of service providers, content providers, and end users. It consists of a huge number of interrelated autonomous systems (AS) [23] operated by companies ranging from large

backbone providers to local internet service providers (ISP). The interconnection of individual ASs is realized by internet traffic exchange facilities (switching centers) that are usually referred to as peering points. These public exchange services allow the peering of several ISPs without the need for a dedicated link between each of them. The individual routers connected to peering points perform data packet routing with an exterior gateway protocol (EGP), such as the border gateway protocol (BGP). In contrast, the routing within individual autonomous systems is performed with interior gateway protocols (IGP), such as open shortest path first (OSPF) and routing information protocol (RIP). This configuration of the internet as a network of networks allows service providers, content providers and, end users to reach a global community by purchasing a single network connection from an ISP.

The structure of the internet, briefly outlined above, illustrates the following types of bottlenecks:

- first mile connectivity
- peering point performance
- backbone capacity
- last mile connectivity

In the past few decades experts in the field of distributed computing have developed mechanisms, techniques, and guidelines to avoid these internet bottlenecks [4] and to reduce the costs involved in the transmission of data across wide area networks (WAN). Current research is concerned with the emerging edge server architecture [48] and related approaches [34, 106] that promise to improve performance and availability of web services by deploying a network of servers at geographically distributed sites.

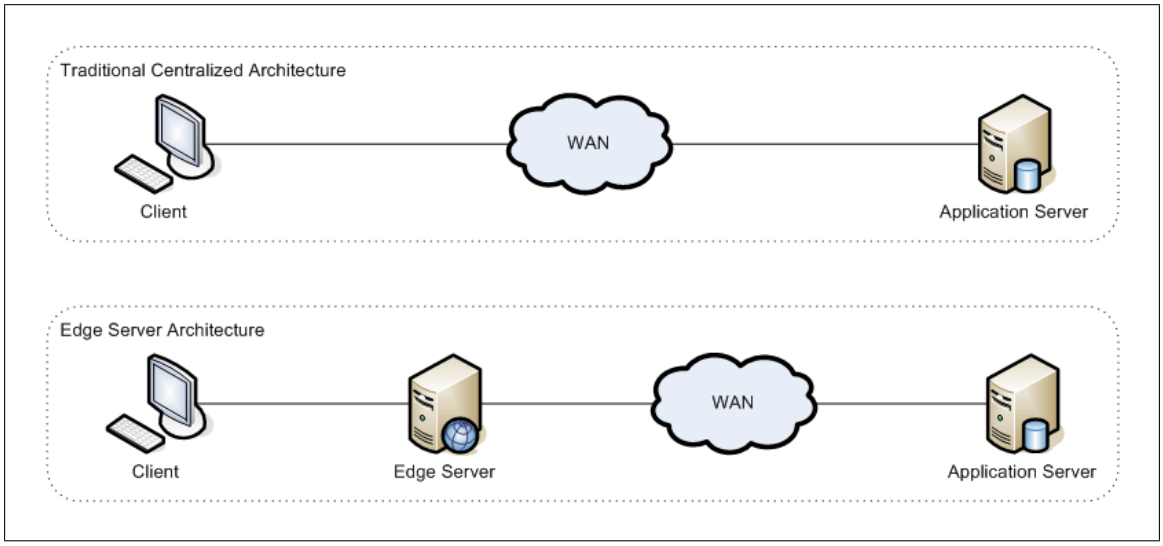

Figure 3.1: WAN Service Architectures

The edge server architecture, outlined in figure 3.1, is currently tailored towards content centric applications and enables end users to experience high performance by interacting with topologically close edge servers. In general, the edge server architecture

allows a partitioning of content generation, which takes place at the origin site, from content arrangement, which can take place at edge servers. Each edge server caches content fragments and assembles them for delivery on behalf of the origin site, while expired and missing fragments are retrieved from the backend system via an optimized connection. In this way, client applications can be served from the edge of the internet which improves access and reduces traffic across wide area networks. A comprehensive distributed edge network with servers at diverse geographic locations can evade or at least diminish peering point performance, backbone capacity, and last mile connectivity bottlenecks.

The edge server architecture is usually realized by deploying the web tier at several edge servers while the business tier remains at the origin site. The partitioning of web and business tier (sometimes referred to as splittier) significantly increases performance by spreading user access to multiple edge locations that handle assembly and delivery of content. Presentation components are deployed on individual edge servers in a way to minimize the communication with the origin site and therefore reducing expensive roundtrips across WANs. The individual components are distributed based on data access patterns, in order to offload infrequently updated content and the assembly of web interfaces. Consequently parts of the user interface, such as dealer information, product catalogs, and shopping carts can be cached and served directly from the edge of the internet. In this way load on the application server at the origin site is reduced and more resources are available for the completion of backend operations. In addition, the connectivity between the edge server and the backend system can be leveraged to improve reliability and communication negotiation latency by maintaining a persistent connection. Furthermore the data transferred across the WAN connections can be compressed to reduce network traffic and transmission delays.

Currently the edge server architecture is primarily utilized in content delivery networks (CDN) that serve web pages, streaming data, and file downloads to a large user population. The company that probably put most effort into the emerging market segment is Akamai Technologies, which operates a globally distributed edge network. The edge side includes (ESI) [8] standard specification developed by Akamai Technologies in cooperation with companies such as IBM, Bea, and Oracle allows the dynamic assembly [6, 5] of several web page fragments at edge servers. The ESI technology enables edge servers to arrange web pages consisting of both static (cacheable) and dynamic (not cacheable) content and therefore supports whole site delivery, streaming delivery, and digitalized downloads. Furthermore, Akamai Technologies teamed up with IBM to provide edge computing support for the IBM WebSphere software platform [64]. As a result, IBM offers the Edge-Computing Toolkit for WebSphere Studio [60] and the Cache Invalidation Adapter for WebSphere Application Server [59]. While the former integrates support for the splittier application model into IBM's integrated development environment (IDE), the latter enables cache expiration at edge servers.

In the context of the edge server architecture we define the following terms that will be used throughout this thesis:

Access Traffic

The term *access traffic* will be used to refer to traffic among client applications and edge servers. Depending on the number of edge servers in a given application scenario the communication is performed over local area networks, metropolitan area networks, or wide area networks. In any case, the connectivity is assumed to

be better, in terms of latency and throughput, than the connectivity to the origin site.

Transit Traffic
The term *transit traffic* will be used to refer to traffic caused by the communication among edge servers and the origin site. The connectivity is assumed to be transferred across wide area networks probably involving several backbones and peering points.

Dynamic Content
The term *dynamic content* will be used to refer to content that *cannot* be cached at edge servers. This type of data is usually retrieved directly from the backend database as it is the case for highly personalized web pages and in response to a user requested query operation.

Dynamic Assembly
The term *dynamic assembly* will be used to refer to the dynamic arrangement of static and dynamic content upon client request at the edge server. Depending on the number of expired and missing content fragments the assembly process may or may not result into transit traffic.

In contrast to application specific dissemination the edge server architecture relies on pull on demand to spread data to individual edge server. In this way, the distributed edge network acts like a huge reverse proxy cache. Depending on the business scenario it might be sufficient to deploy a few edge servers in diverse regions or to rely on a comprehensive infrastructure such as operated by Akamai Technologies.

Current edge server implementations are inherently content centric, optimized for static content, and build upon a web based user interface. The benefits of the pull based caching model are vanishing when it comes to dynamic content such as highly personalized web sites which is, however, the prevailing case in modern enterprise applications. Furthermore, the adoption of the splittier application model and the design of adequate presentation components that provoke a reduction of the number of expensive roundtrips across WANs is a complex task. Integration into existing enterprise solutions, that usually exhibit a tight coupling between the web and business tier, is unlikely to be successful. In general, short running and seasonal high volume web applications, as well as polling and product configuration applications, are good candidates for the deployment of edge servers.

The following chapters provide an edge server approach which utilizes application specific dissemination techniques to evade the partitioning of web and business tier. We aim to design a flexible communication layer that enables edge servers to handle dynamic content and highly personalized web sites more efficiently. In addition, we enable the commitment to the edge server architecture for more general purpose distributed enterprise applications that are not concerned with content delivery and do not rely on web interfaces. In particular, our goal is to move the edge server architecture to new areas of application beyond content delivery networks.

Chapter 4

Extending Enterprise Applications with Valorized Edge Services

The preceding chapters skimmed over the main concepts of enterprise software architectures and outlined dissemination and caching techniques for distributed systems. In particular, chapter 2 explored the most important enterprise computing technologies that are incorporate into modern multitiered enterprise applications and introduced several core enterprise application services. In addition, chapter 3 discussed fundamental communication patterns and explained application specific dissemination in distributed systems. Furthermore, chapter 3 identified several internet bottlenecks and walked the reader through the aspects of current edge server architectures.

This chapter analyses the limitations and drawbacks of conventional multitiered enterprise architectures, current application specific dissemination realizations, and the emerging edge server architecture in the context of large scale distributed systems. It discusses the requirement for an architectural proposal that provides an efficient infrastructure for service oriented computing across wide area networks. Moreover, an application model and a system model are proposed that facilitates the extension of universal enterprise applications with valorized edge services.

4.1 Service Centric Architectures

The recent focus on service oriented enterprise applications and the commitment to business-to-business (B2B) collaborations in general, entails a novel type of loosely coupled interfaces (web services) that support complex business interactions. Moreover, web based applications exhibit service oriented aspects by introducing highly personalized web pages, supporting sophisticated user profiles, and providing context sensitive and location based features and services. It is evident that the shift to service oriented computing changes the way customers, suppliers, and partners interact and ultimately results into different usage scenarios.

The enduring globalization of business processes comes with a significant increase of collaborations performed over wide area networks and results into content and processing intensive tasks. Consequently, internet based business solutions experience the communication characteristics and internet bottlenecks outlined in section 3.3. The changing requirements render conventional multitiered enterprise applications

insufficient and leads to the development of highly distributed enterprise applications. In general, comprehensive internet scale enterprise solutions deviate from the system and application model advocated by modern enterprise software architectures and distribute servers to multiple geographical locations. The main challenges for the deviation of traditional multitiered applications include (1) the definition of strategies and guidelines for the efficient decomposition and (2) the integration of a flexible communication infrastructure that handles interactions among dispersed components efficiently. The decomposition determines the responsibility of the individual participants and consequently influences the requirements for the connecting infrastructure.

The edge server architecture discussed in section 3.3 is a comprehensive solution for content delivery networks that spreads content assembly and delivery to the edge of the internet. The architecture is optimized for the delivery of static content and comes with support for dynamic content assembly. Service oriented requests that involve business processing or the generation of dynamic content are, however, delegated back to the origin site. With a growing number of service requests, the edge server architecture will consequently experience the same limitations and shortcomings already inherent in multitiered enterprise applications. The current design of the edge server architecture is tailored towards content delivery networks and cannot accommodate the requirements for a global service market.

The rapidly increasing number of service oriented solutions raises the exigency for valorized edge servers that perform business processing and serve dynamic content. The distribution of valorized edge servers is, however, significantly more complex than the content delivery solution supported by current edge server architectures. The distribution of servers with business processing capabilities requires a sophisticated decomposition and a flexible replication solution that can be realized across wide area networks. This chapter provides a detailed analysis of an architectural proposal for the extension of multitier enterprise architectures with valorized edge servers. We argue that current approaches are not well suited to accommodate the requirements for a global service market and identify the urgent need for distributed service oriented enterprise solutions.

4.1.1 Content and Service Brokers

In general a distributed enterprise system that deviates from the multitier architecture can be classified into (1) enterprise information systems (data source) which provide the base data for the enterprise solutions, (2) backend systems (origin site) that directly interact with the EIS and perform critical business tasks, (3) client applications (clients) which are the net consumers of the services offered, and (4) service integration and delivery systems (brokers) which perform business processing, deliver content, provide an client interface, and mediate between the clients and the origin site. In contrast to traditional three tiered enterprise applications the brokers are deployed at several diverse locations with varying degree of duties and responsibility.

The following terms will be used to identify the type of tasks performed by service integration and delivery systems (brokers):

Content Broker
The term content broker will be used to refer to service integration and delivery

systems that are responsible for the assembly and delivery of content. This type of information broker usually relies on caching to serve static content and communicates with the origin site to fulfill the delivery of dynamic information. In essence, the edge server architecture introduced in section 3.3 distributes content brokers to improve performance and scalability of content delivery networks.

Service Broker
The term service broker will be used to refer to service integration and delivery systems that add value to data, which usually involves business logic. Service brokers either communicate directly with the origin site or rely on data replication to perform business processing and the distribution of dynamic content. The terms valorized edge server and service broker will be used interchangeably throughout this thesis.

In multitiered enterprise applications the functionality of both content brokers and service brokers is provided by the origin site that offers an interface to client applications and satisfies information and service requests. The edge server architecture presented in section 3.3 spreads content brokers to handle content assembly and delivery at the edge of the internet. The processing of service requests is, however, delegated to the origin site that performs the business tasks and returns the result to the edge server.

Turning away from content centric applications results into an increasing number of business interactions that require enterprise systems to perform complex business processing. It is evident that better performance and scalability of service oriented applications demands for the distribution of service brokers and not the spreading of content brokers. The decomposition of enterprise systems into several service brokers is, however, not possible with conventional caching systems. The interconnection of service brokers requires a more sophisticated communication infrastructure that handles dynamic data dependencies and provides adequate coherency.

4.1.2 Caching and Replication

The prevailing solution for performance and scalability improvements in content delivery networks is the caching of static content fragments. The current edge server architecture further improves performance by adding dynamic assembly which involves pull on demand to spread data to individual edge servers. Aperiodic pull comes, however, with limited flexibility in scheduling the order of data delivery and continuously interrupts the server to handle data requests. Clients in a caching system try to locate the required data items in their local caches and upon cache miss contact the appropriate data source. It is important to note that this client initiated technique does not provide servers with the ability to propagate notifications of data changes. Therefore, newly added or updated data items may go unnoticed at clients, unless they periodically poll the server. Another shortcoming of conventional caching techniques in the presented problem domain is the lack of knowledge about the available data spectrum. Without the presence of a data index client applications can not directly access the data stored at the server and are therefore unable to perform sophisticated business processing and query requests. While caching is well suited for content brokers it is definitely not sufficient for service brokers that perform business processing in combination with data distribution.

Distributed applications that spread business processing to several servers usually rely on data replication to provide a consistent view of the available data to each node in the system. Data replication results, however, in the transmission of every data item to all participants and requires processing and storage resources to handle the increasing data volumes of current business solutions. Application scenarios that experience a significant number of updates usually rely on a dedicated network connectivity to handle the immense network traffic resulting from the stringent coherency requirements. In essence, replication is aperiodic push which occurs in the absence of specific client request. Sending irrelevant data to client applications is a waste of bandwidth and especially costly across wide area networks. In general, servers are, however, unable to accurately predict the data requirements of client applications. The publish-and-subscribe paradigm finds remedy by allowing clients to provide a profile of interests that servers can use to constrain data dissemination. While replication is well established in small scale distributed systems the inherent consistency overhead is not feasible for internet based solutions and therefore not suited for the institution of service brokers.

The analysis of caching and replication reveals that data dissemination and caching is one of the key challenges in large scale distributed systems. The next section discusses the applicability of application specific dissemination for the distribution of service brokers to diverse geographic locations.

4.1.3 Application Specific Dissemination

In conventional distributed systems caching is the prevailing mechanism to increase performance, while replication introduces redundancy to guarantee fault tolerance and availability of interrelated systems. The communication characteristics of wide area networks come, however, with various challenges for the interconnectivity of distributed systems. While several intriguing concepts for application specific dissemination (refer to section 3.2 for details) have been proposed, almost all of them possess one or more of the following shortcomings:

- tailored to specific applications and complex to adapt and modify
- difficult to integrate into existing and newly built applications
- require application developers to handle distribution and consistency semantics
- substitute existing technologies in the addressed application domain
- affect and constraint value added services of enterprise applications

The limitations enumerated above illustrate the requirement for an innovative approach that streamlines the utilization of application specific dissemination for service brokers. The conservation of the benefits of modern enterprise software architectures, outlined in chapter 2, is key to the success of the service broker approach. The extension of enterprise applications with valorized edge services must not influence exiting technologies, constrain value added services, or significantly complicate the development of business solutions. Otherwise, a wide adoption of the proposal is unlikely to be valuable and successful.

4.2 Valorized Edge Services Applied

In the preceding discussion we identified that modern enterprise applications are inherent service oriented and participate in global business collaborations. The conventional multitiered enterprise application model advocated by current enterprise software architectures is, however, insufficient for global business interactions and suffers from several internet bottlenecks. The edge server architecture, on the other hand, is optimized for large scale distributed systems but the commitment to content brokers limits this approach to content delivery networks.

The following enumeration summarizes the core shortcomings and drawbacks of current edge server architectures in the context of service oriented solutions:

- tailored towards content delivery networks
- dynamic information generation not possible at edge servers
- access is constrained to a web based user interface
- execution of business tasks not feasible at edge servers

The lineup in figure 4.1 illustrates various deviations of the traditional multitier architecture. The following description analysis the benefits and consequences of each architectural approach and characterizes *coherency overhead*, *dissemination complexity*, number of system nodes, and the communication aspects between client applications, brokers, and the origin site.

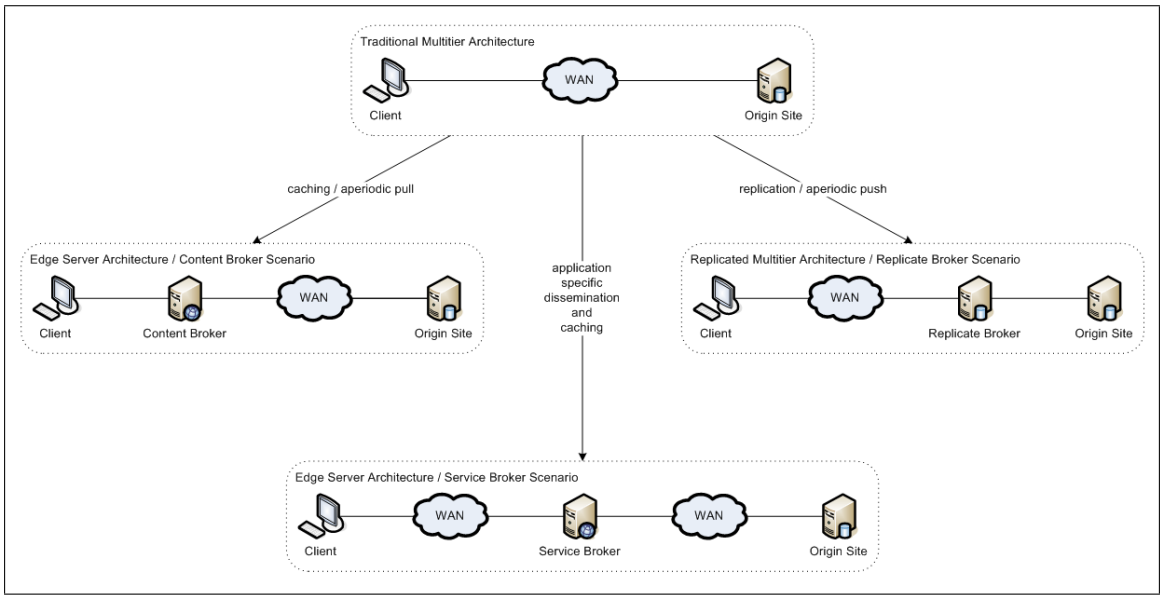

Figure 4.1: WAN Service Architectures Comparison

Content Broker Scenario

The edge server architecture outlined on the left hand side of figure 4.1 introduces caching (aperiodic pull) to bring content brokers topologically closer to client applications and business partners. The content broker scenario is characterized

by minor coherency overhead, low dissemination complexity, and a large number of system nodes (edge servers). The communication between client applications and content brokers usually involves just a few autonomous systems while transit traffic has to be performed across wide area networks. The application model is inherently content centric and the utilization of caching limits content brokers to the assembly and delivery of information fragments.

Replicate Broker Scenario
The replicated multitier architecture outlined on the right hand side of figure 4.1 introduces replication (aperiodic push) to increase scalability, reliability, and fault tolerance. The replicate broker scenario is characterized by major coherency overhead, low dissemination complexity and a small number of system nodes (replicate brokers). The communication between client applications and replicate brokers is usually performed across wide area networks. The coherency overhead requires the deployment of dedicated connections among server nodes and limits the distribution of replicate brokers. Since each replicate broker is equally capable of handling service requests the access traffic can be distributed by a load balancing mechanism.

Service Broker Scenario
The edge server architecture outlined on the bottom of figure 4.1 introduces application specific dissemination and caching to bring service brokers topologically closer to client applications and business partners. The service broker scenario is characterized by moderate coherency overhead, high dissemination complexity, and a medium number of system nodes (valorized edge servers). The coherency overhead results into a manageable number of edge servers and consequently deducts that both access traffic and transit traffic have to be performed over wide area or metropolitan area networks. In contrast to the content broker scenario, internet bottlenecks cannot be avoided completely they are rather reduced by the deployment of several edge servers in diverse geographical regions. The application model enables edge servers to generate dynamic content, process business tasks, and provide a set of different service oriented interfaces.

We consider the service broker scenario a valuable tradeoff for internet scale enterprise applications that exhibit global service oriented business collaborations. We suggest deploying web service enabled edge servers topologically close to business partners and business suppliers. In this way, service requests can be handled by a dedicated edge server and communication is narrowed down to this specific geographic region.

We argue, that the extension of the edge server architecture with service brokers allows enterprise applications to participate more efficiently in business interactions performed over wide area networks. We suggest the extension of multitiered enterprise applications with valorized edge services to streamline the deployment of service oriented business solutions in large scale distributed systems. The key challenges in the suggested application model include the definition of strategies and guidelines for the efficient decomposition into service brokers and the implementation of a flexible infrastructure for data dissemination and caching. It is crucial that each service broker can access shared data locally and thereby avoid the latency introduced by wide area networks that would occur in a traditional approach. We explore the design space of the suggested application model and highlight key issues for the decomposition into service

brokers. Moreover, we propose the utilization of a component model based application specific dissemination scheme that preserves value added services and extends rather than substitutes exiting enterprise computing technologies. In other words, the main contribution of this thesis is the definition of guidelines and strategies for the seamless integration of a flexible communication infrastructure (for application specific dissemination) into enterprise applications.

Our focus is on the design and implementation of a scalable service oriented enterprise application architecture that is well suited for wide area network interactions. We introduce a novel decomposition of multitiered enterprise applications and utilize application specific dissemination to overcome the shortcomings of current edge server architectures. The resulting architecture aims to support more general purpose enterprise applications that are neither constrained to web based user interfaces nor restricted to content delivery networks. In addition to the features provided by current edge server architecture the distributed service brokers are designed to:

- allow dynamic content generation, assembly, and delivery
- perform presentation logic and business logic processing
- enable access with web services, desktop applications, and mobile devices
- support distribution of workflow and business tasks

The distribution of service brokers moves business processing closer to client applications and business partners. Consequently, business-to-business interactions such as web service calls and remote method invocations can be carried out directly by valorized edge services. In this way, the amount of service requests across wide area networks can be reduced significantly and ultimately increases performance and scalability of business collaborations. In addition to the delivery of dynamic content the proposed architecture can facilitates the implementation of service oriented enterprise solutions, such as e-marketplaces, distributed workflow services, and custom internet telephony products.

4.2.1 Design Considerations

The introduction of application specific dissemination for the interconnectivity of service brokers leaves a lot of design, integration, and implementation scope. It is crucial to establish design principals to ensure a valuable result that satisfies the requirements discussed in the preceding sections.

Transparency

The most essential question is how much transparency the communication infrastructure provides to application developers. Distribution and location transparency are mechanisms which hide complexity and isolate applications form the underlying hardware and software details. It ensures that the location of software components has minimal impact on the design and implementation of distributed systems. The distributed application environment of internet based software solutions differs, however,

from typical enterprise application scenarios. The shared communication infrastructure operated by independent autonomous organizations does not provide quality of service (QoS) guarantees and might change dynamically and unpredictable from one interaction to the next. In this organic, irregular environment a slight reduction of transparency [49] that supplies developers with current QoS conditions can enable the implementation of more efficient and reliable software solutions. With the feedback on the expense involved with a particular remote operation the behavior of applications can be adapted dynamically to the changing environment. On the other hand, the additional complexity increases the burden on developers and diminishes reusability and flexibility of the resulting software solution.

The evolution of distributed systems discussed in chapter 2 shows that the transparent support of value added services is key to the success of enterprise software technologies. We observe, however, that the support for explicit management of primary services (as it is the case for user transactions) is important for scenarios where the automatic mechanism is not sufficient. We aim to apply this concept to the application specific dissemination infrastructure that interconnects service brokers and the origin site. By default the dissemination logic will select the communication pattern automatically. In complex situations where the built in mechanism is not sufficient the application developer can, however, customize the communication behavior. Depending on the QoS feedback provided by the communication infrastructure the processing of complex service requests can be either performed locally at the service broker or diverted to the origin site. In essence, this allows increasing overall performance by executing simple tasks at service brokers while complex tasks that would result into overwhelming transit traffic can be diverted to the backend system.

Integration

The analysis of existing solutions shows that it is complicated to integrate application specific dissemination into existing and newly built enterprise applications without affecting or constraining value added services. The main emphasis of this thesis is, therefore, on a seamless integration of a flexible communication infrastructure into enterprise applications. The focus is on the service broker scenario outlined in figure 4.1 that spreads service brokers across the internet and consequently requires an efficient connectivity among edge servers and the origin site. In other words, the emphasis is on supplying a flexible infrastructure that allows the extension of conventional multitiered enterprise applications with valorized edge servers.

The requirement for a general reusable architecture has already been recognized and lead to the development of a toolkit approach [45] and the definition of a component and communication model [55] for data dissemination systems. The presented work is concerned with the definition of an architectural proposal for the integration of a flexible communication infrastructure into enterprise applications. The suggested design advocates the extension rather than the substitution of enterprise technologies and aims to satisfy the stringent requirements of modern business solutions.

The server side component model is an integral part of enterprise software architectures and decisive to the features available to application developers. The suggested architecture, described in more details in the following sections, presents a component model based application specific dissemination strategy. An additional *data service*

layer is introduced that mediates between the persistence mechanism provided by the server side component model and the application code. This surrogate is responsible to fulfill data requests either by accessing the local data source or, if necessary, by issuing a remote data operation. The integration of the data service layer, discussed in more details in the following sections and chapters, aims to barely constraints the development of applications. The design of the surrogate builds upon existing technologies such as the suggested persistence mechanisms provided by the enterprise software architecture in use. This ensures that value added services are preserved and can be used without restrictions. In addition, the reliance on well established blueprints and best practice advices ensures a seamless integration into a majority of existing and newly build enterprise solutions.

Applicability

The analysis of application specific dissemination in section 3.2 stresses that most proposals are tailored towards a specific application scenario. Consequently, the communication infrastructure and the dissemination logic are hard to adapt to changing conditions and varying application domains. This fact is further pointed out by a detailed analysis [48] which concludes that no single strategy is optimal for a universal set of data in edge server architectures. The lack of a comprehensive dissemination strategy for general service broker scenarios renders the integration of specific algorithms into the architectural proposal inappropriate.

The suggested architecture offers another approach that provides a general framework for the definition, evaluation, and ultimately deployment of application specific dissemination algorithms. It supplies several communication patterns to choose from and multiple extension points with flexible and extensible interfaces for the definition of dissemination and caching logic. In other words, the integration of the data service layer provides a comprehensive infrastructure for data dissemination and caching in service broker scenarios.

The commitment to a general purpose infrastructure comes with several advantages for the deployment of application specific dissemination in service broker scenarios. The flexible interfaces alleviate the definition of several dissemination algorithms that can be changed at deployment time and at runtime to adapt to changing conditions. As a result it is possible to dynamically selecting the optimal distribution strategy [98] to improve overall performance and scalability. Moreover, a standard framework for the definition of algorithms and caching schemes in service broker scenarios allows the direct comparison of application specific dissemination in diverse enterprise solutions.

The suggested procedure for the utilization of application specific dissemination in service broker scenarios is the integration of the data service layer in combination with a straightforward default dissemination algorithm. This ensures a fast time to market and enables the analysis of access patterns in a real world scenario that aid to tweak the algorithms to the application scenario.

Techniques

Service oriented processing requires an accurate view of the whole data spectrum and operates on data that is assumed to come with high fidelity and temporal coherency. In

other words, to enable the processing of business tasks each service broker is required to have access to an accurate and complete view of all data available. This requirement conflicts, however, with our basic assumption to avoid complete data replication. The solution offered by the presented architecture is the use of indexing in combination with application specific data dissemination.

The service broker scenario shown in figure 4.1, has characteristics in between the content broker and the replicate broker scenario. Consequently, the data service layer will provide aperiodic pull and aperiodic push for data dissemination. The dissemination logic is responsible for determining an optimized mix of push and pull to improve the performance of business processing at edge servers by minimizing transit traffic.

In general, enterprise solutions delegate data retrieval and storage operations to enterprise information systems that usually rely on some sort of indexing to administer data items. The same mechanism will be used by the data service layer to manage dissemination, caching, consistency, and coherency of data items required for business processing. The separation of business data into an index and data items for partial replication is, however, a complex task that includes the identification of the right granularity of data units and the definition of optimal consistency strategies [106]. The outcome of the data segmentation can significantly influence the performance of the service broker scenario and has to be performed with the particular application scenario in mind. The presented architecture offers seamless integration of an infrastructure that is designed to work with all types of index information and is tailored towards efficient data dissemination for business processing.

The ubiquity of databases in enterprise systems leads to the assumption that most applications can use data records as the basic information unit. The TPC-W [123], a transactional web e-commerce benchmark, distinguishes between several scenarios (workload mixes) for a general e-business application. The operations performed during the performance evolution can be answered to a large degree (over 90%) by indexed data inquiries. For the remaining data the granularity of data units has to be defined higher or a delegation of processing to the origin site has to occur. It remains to be seen how restrictive the index based caching actually is in general application scenarios.

The presented architecture enables the outsourcing of business tasks to service brokers by utilizes index replication while data items are propagates based on dissemination logic. In essence, the index is automatically pushed to service brokers while data items are either pulled or pushed aperiodically depending on the strategy of the dissemination logic. Service brokers request information units from the data service layer that determines if the data item has already been pushed or pulled to the local cache. If the required data is locally available it is directly returned, otherwise a remote data operation pulls the information from the origin site. Business operations and queries which cannot be satisfied by an index based access have to be diverted to the origin site which processes the request and returns the result back to the service broker. Dynamic operation diversion may occur when the number of data items or the size of data that needs to be fetched from the origin site exceeds a predefined limit.

The deployment of a communication infrastructure that relies on push and pull for data dissemination comes with a number of benefits. It offers flexibility in the negotiation of data updates that can be combined with invalidations [74] and the introduction of leases [51]. The pushing of invalidations as an integral part of the communication infrastructure allows ensuring high fidelity and temporal coherency between service

brokers and the origin site. The utilization of leases, on the other hand, helps to muck out expired and stale data items. Moreover, the utilization of speculative data dissemination is straightforward with the supplied interconnectivity of edge servers and the backend system. Additional flexibility is guaranteed by the deployment of asynchronous middleware that offers publish-and-subscribe based dissemination and will be discussed in more details within the remaining chapters.

Despite the large number of research papers on the application of push based data dissemination only a few successful and mature products exist that leverage this technology. As a result, it might be argued that the introduction of push for data dissemination is inadequate and should be revised. A detailed analysis reveals, however, that push is absolutely necessary to fulfill the fidelity and coherence requirements of service oriented edge computing. The index replication, the sending of invalidations, and the option for server initiated speculative data propagation is most efficiently realized by push based dissemination. In contrast to various other approaches that rely solely on speculative push the presented architecture provides multiple communication patterns to choose from. It depends entirely on the application scenario how many data items are pushed in addition with the index information. Furthermore, it should be considered that the data service layer is transparent to the enterprise application. In other words, the internal communication strategies influence the efficiency of data operation, but not the type of operations available to the application developer.

Push is assumed to be used for index replication and the sending of invalidations to reduce cache staleness. Data propagation, on the other hand, is performed by selecting the most appropriate communication patterns supplied. Depending on the application scenario data can be delivered by a pull, push, or an application specific dissemination strategy such as adaptive hybrid delivery [110].

4.2.2 System Model

The functional requirements for the valorization of edge servers lead to the system model presented in figure 4.2. In contrast to content centric edge server architectures each service broker is equipped with an application server that hosts both a web tier and a business tier. The omission of the splittier deployment simplifies the development of presentation logic and ultimately enables business processing to take place directly at the edge of the internet. Moreover, the web tier can avoid distracting remote operations by access the locally available business tier and is therefore more flexible and adaptable to changing requirements. Depending on the business scenario it may, however, be necessary to delegate the processing of complex tasks back to the origin site. The need to divert tasks arises for inquiries that can not be answered by the indexing scheme utilized and for business processes with stringent concurrency requirements. The access of non indexed data can be detected automatically by the data service layer and allows seamless diversion of business tasks to the backend systems.

The delegation of business tasks is discussed in more details in the application model description. In general, the application developer defines the diversion logic in the business objects and relies either on deployment configuration or runtime information to decide if a given business operation has to be diverted to the backend system. The most efficient solution depends on the business scenario and the data requirements of individual tasks.

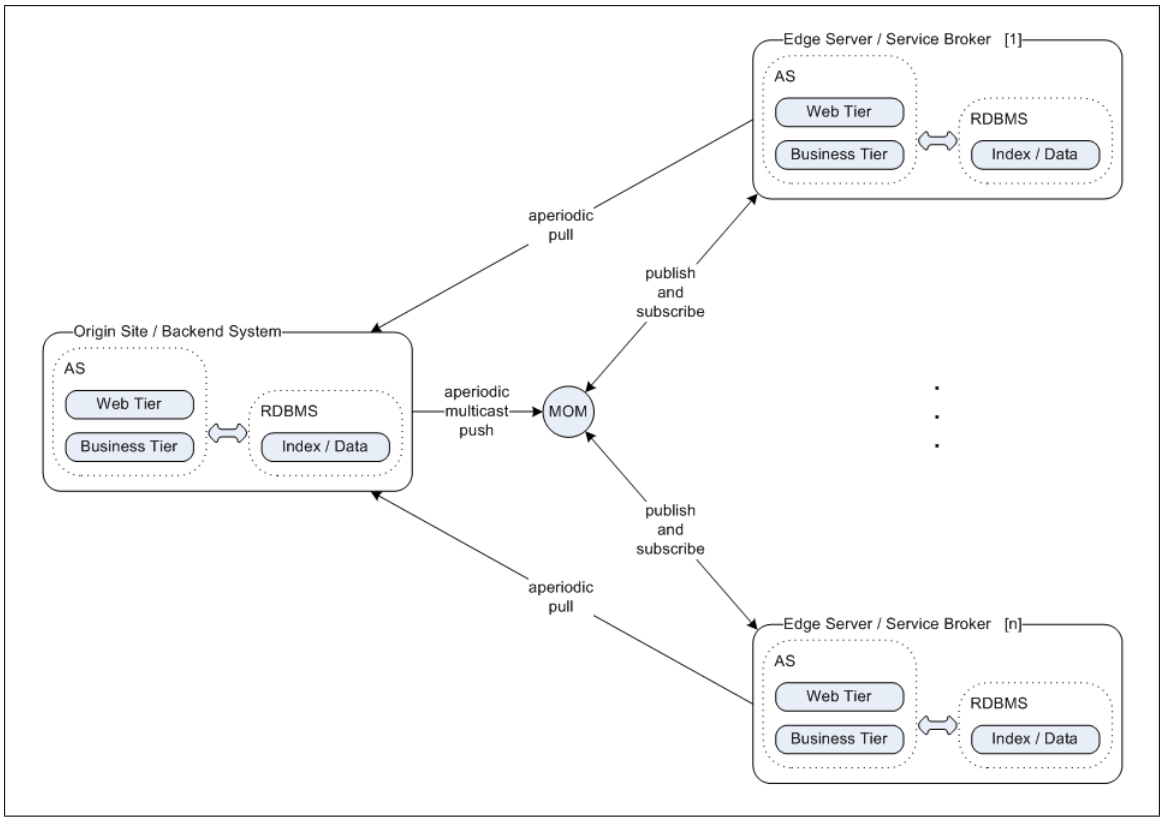

Figure 4.2: Edge Service Architecture – System Model

Avoiding the splittier approach allows service brokers to process business tasks at the edge of the internet and enables the use of more than a web based user interface. The presence of business objects enables edge servers to provide a comprehensive set of client interfaces, such as web services, message oriented interfaces, and remote method invocation endpoints. The support for several interfaces facilitates the use of edge servers in business-to-business collaborations and becomes especially important in a global service market.

The business processing capabilities of service brokers require the deployment of a relation database management system at each edge server. The RDBMS stores a complete data index, cached data items, and metadata for the data service layer. The metadata is automatically maintained by the data service layer and provides information to manage dissemination, caching, consistency, and coherency of data items. While moderate data volumes are expected to be cached at service brokers and the use of a RDBMS may seem excessive there are several reasons for this decision. Most importantly, the migration effort to the edge server architectures can be alleviated by ensuring that service brokers and the origin site provide a comparable environment for business processing. Since the backend system will most likely build upon data from a RDBMS the service brokers have to provide the same data access and query mechanisms. In addition, the implementation of the data service layer is significantly more complicated if different types of data sources have to be supported. The deployment of a RDBMS allows the data service layer to rely on value added services such as, resource pooling and transactional access to administer cached data items. Furthermore, the

data service layer can delegate the storage of business object to the persistence mechanism of the server side component model rather than implementing the functionality all over again. Moreover, application servers are optimized to access databases and offer a variety of in memory caching techniques to accelerate data storage and retrieval. The presented approach can take advantage of these features to further increase cache performance. The utilization of a database that can be tightly embedded into Java based solutions, offers transactional access, and is fully supported by the application server in use will yield to the best solution for the service broker scenario.

The preceding sections determined that the communication patterns aperiodic push and aperiodic pull are most appropriate for data dissemination in the service broker scenario. Accordingly the system model supplies a synchronous communication channel for pull on demand and supports aperiodic unicast, multicast, and broadcast push with an asynchronous enterprise messaging system. The message oriented middleware is crucial to the design of the service broker scenario. The messaging infrastructure allows an efficient multicast delivery of data index and modified data item. Furthermore, the decoupling allows the addition of edge servers without any changes to the data service layer or the backend system. The publish-and-subscribe paradigm allows service brokers to announce an interest in particular data categories and therefore provides a flexible configuration of data dissemination directly at the middleware layer.

At first glance the system model in figure 4.2 may look overloaded due to the deployment of an application server and a relational database management system at each edge server and the interconnectivity with a message oriented middleware solution. Comparing the proposed configuration to the distributed edge network operated by Akamai Technologies reveals, however, that the extensions introduced by the service broker scenario are not significant. An analysis of the content delivery network developed by Akamai Technologies in cooperation with IBM shows that content brokers are also equipped with an application server and rely on the IBM Cloudscape [65] database for content caching. In essence, the valorization of edge services is accomplished by a different decomposition approach (application model) and a flexible interconnectivity that is backed by a message oriented middleware provider.

The system model presented in figure 4.2 provides a solid foundation for the implementation of valorized edge servers. The interconnectivity comes with multiple communication patterns (both synchronous and asynchronous) to ensure a maximum of flexibility for the design of dissemination logic. The RDBMS equipped edge servers provide the data service layer with a proven backend for the storage of data index, cached data items, and metadata information. The omission of the splittier approach provides service brokers with the powerful server side component model of the enterprise software architecture and therefore potentiates complex business processing at the edge of the internet.

4.2.3 Application Model

Modern enterprise software architectures provide a powerful programming environment for multitiered enterprise applications and extend it with remote method invocation, message oriented middleware, and web services to facilitate communication among business partners. Despite all these services the distributed nature of the proposed service broker architecture combined with the communication latency of wide

area networks and the costs involved in transferring information across the internet demand for more efficient collaborations. The application model in figure 4.3 shows the mechanisms introduced by the presented architecture to overcome these shortcomings. The contributions include a data service layer, a flexible data dissemination infrastructure, the facility to delegate tasks to the origin site, and a locally available database that provides a complete data index, cached data items, and metadata information.

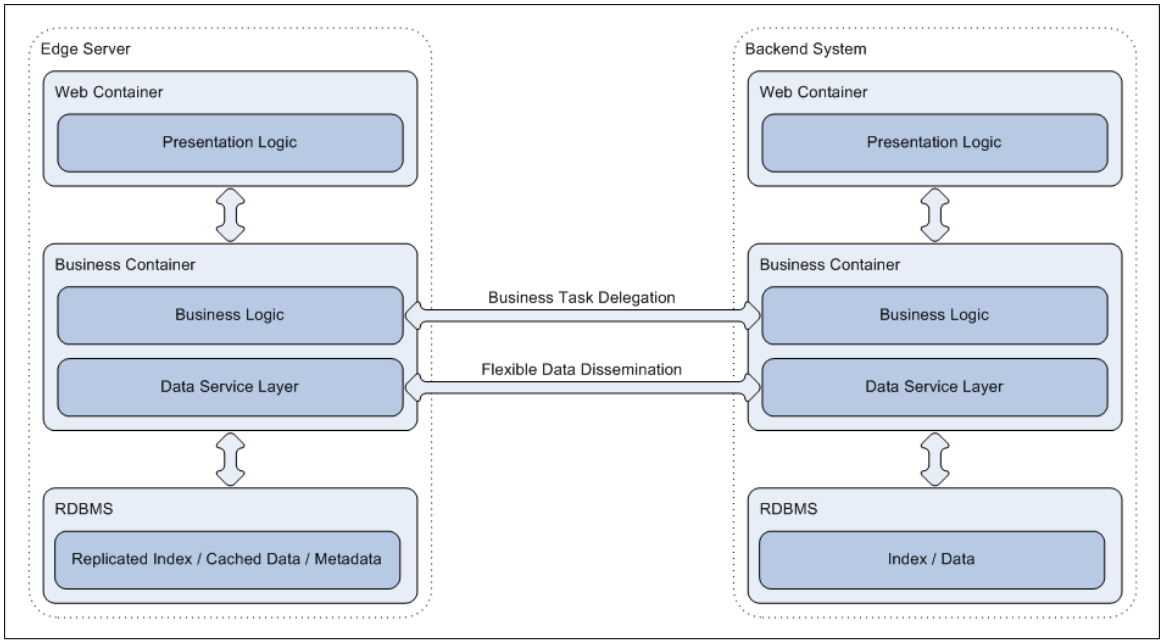

Figure 4.3: Edge Service Architecture – Application Model

The following description skims over the building blocks of the application model while an analysis of the interactions between the edge server and the backend system is discussed below.

Data Service Layer

The *data service layer* (DSL) deployed in combination with the business logic is the central component in the application model. The additional layer acts as a surrogate of the data source and mediates between the business logic, the local database and remote hosts. The deployment within the business tier is crucial since it enables the DSL to rely on the persistence mechanisms and database access services provided by the container. Furthermore, it guarantees a seamless integration with the business logic that runs within the same environment.

Business Task Delegation

The *business task delegation* (BTD) allows business logic at the edge server to divert processing to the backend system. In this way, data intensive business tasks and those with the requirement to access data items without index can be delegated to the backend system. The origin site will perform the diverted business task and return the result to the edge server. There are essentially two types of business task delegations (1) design time delegation and (2) runtime delegation. The latter is supported by the feedback mechanism of the DSL that triggers an exception when a predefined number of data items required for a

specific operation is not cached at the edge server. In this case the developer can either delegate the business task to the backend system or force the edge server to continue the business processing locally.

Flexible Data Dissemination

The dissemination logic implemented within the data service layer requires a flexible communication infrastructure to replicate the data index, support the dissemination of data items, and facilitate the sending of invalidations. The requirements and the technologies have been discussed in the context of the system model in the preceding section.

With the application model outlined in figure 4.3 the application code (presentation logic and business logic) can be developed as usual. The design of the data service layer follows well established blueprints and best practice guidelines for the abstraction of database access. The design ensures a seamless migration to and from conventional multitiered enterprise applications and does not impose restrictions on the implementation of business logic. The business task delegation requires, however, additional effort that may complicate the code slightly. It remains to be seen how stringent the consequences are in comprehensive business solutions.

The realization of read requests and the resulting interactions between edge servers and the backend system have already been discussed. If a required data item is not available on the edge server it will be pulled directly from the backend system. What has not been addressed yet is the procedure in case of write operations. In essence, a write operation can result into adding, updating, or deleting a data item. If a data item is added at the origin site, the dissemination logic decides if the index or the whole data item is pushed to the edge servers. On the other hand, if a data item is updated at the origin site the dissemination logic decides if the update data item or an invalidation message is propagated. In case of a delete operation the propagation of the index information is sufficient. The speculative multicasting of modified data items can be useful for frequently accessed data categories and for database hotspots. The handling of write operations at edge servers can be performed similar to a write through cache by forwarding every data modification to the database directly to the origin site. More sophisticated solutions for partially replicated data can be implemented by assigning consistency and coherency policies [106] to individual data categories. The definition of update strategies for specific data sets at edge servers can improve the overall performance and minimize transit traffic.

The past few paragraphs discussed the building blocks of the application model, the consequences on the development of business logic, and the interactions between edge servers and the origin site. Depending on the client application, a typical usage scenario involves a service request to either the web tier or the business tier. The service request is then delegated to the business logic that interacts with the data service layer to fulfill the business task. The data service layer mediates between the business objects, the local database, and the origin site to carry out data retrieval and storage requests. In rare cases it might, however, be necessary to completely divert operations to the origin site and forward the result back to the client application. The deployment of the business tier at edge servers allows the presented application model to serve a rich set of service oriented requests directly at the edge of the internet. Furthermore, it is possible to interact with desktop applications, embedded devices, and business part-

ners that require a direct communication with the business container through remote method invocation, enterprise messaging systems, or web services.

4.2.4 Service Brokers Analyzed

The preceding sections presented design considerations, a system model, and an application model for extending multitiered enterprise applications with valorized edge services. This section compares the presented approach to the current edge server architecture, highlights open issues, discusses the definition of dissemination logic, and analysis implications of context aware applications.

Comparison

The presented architecture transfers application specific dissemination developed for conventional distributed systems to multitiered enterprise applications. The resulting communication infrastructure allows a partial replication of data to edge servers and therefore enables the execution of both presentation logic and business logic at the edge of the internet. This is in contrast to the current edge server approach that utilizes a caching system to store static data fragments for assembly and delivery.

Despite the diverse requirements the system model is surprisingly similar in both scenarios. Content brokers and service brokers rely on an application server and access shared data that is stored in a locally available database. It is intriguing to observe that the introduction of a flexible communication infrastructure and the extension of the application model allow the valorization of edge servers. The communication infrastructure deviates mainly in respect to the availability of several communication patterns that can be leveraged to meet the data requirements of business tasks. The different application model on the other hand results from the omission of the splittier deployment inherent in the content broker scenario. Consequently, a data service layer is required that mediates between the business objects, the locally available database and the remote host. The complex access patterns and the query requirements of business objects require a comprehensive data service layer solution. A detailed discussion of the architecture, design, and implementation can be found in the following chapters.

With the presented architecture, service brokers can assemble and deliver content as efficiently as content brokers. Moreover, they are enabled to create dynamic content and to perform complex business tasks. The presented architecture should therefore be seen as an extension rather than a modification of the current edge server architecture. The valorization of edge servers adapts the content broker scenario towards service oriented architectures to fulfill the fast pacing requirements of modern enterprise applications.

The deviation of traditional multitiered enterprise applications inevitably adds complexity to the creation of business solutions. While the distribution of content brokers complicates the development of presentation logic, the deployment of service brokers renders the implementation of business logic more intricate. In both scenarios more effort has to be put into the definition of a flexible design in order to guarantee adaptability to changing requirements.

Open Issues

The presented work presents an architectural proposal that facilitates the extension of multitiered enterprise applications with edge servers. The main emphasis is on the definition of guidelines and strategies for the development of a flexible data service layer. There are, however, several open issues that are discussed in related research papers and have to be elevated in the context of the proposed architecture. Furthermore, a detailed analysis in real world applications is required to verify the applicability, benefits, and consequences of the suggested architecture.

The delegation of data intensive business tasks is discussed in the application model description in section 4.2.3. The number of business tasks that can be satisfied with partial replicated data is, however, undetermined. In other words, further research is required to figure out how many business tasks have to be diverted to the backend system in a general business scenario. Delegation is necessary if the requirements of business task can not be satisfied with index accessible data and for business logic that inquiries a large amount of the totally available data spectrum. An analysis of the transactional web e-commerce benchmark TPC-W [123] reveals that the majority of requests can be satisfied with index based queries. A comprehensive study for more general business solutions is required to estimate the impact on the service broker scenario. Furthermore, the delegation of business tasks may require the transfer of context information, such as session data, to the origin site. The exchange of outsized session information can however, introduce a significant overhead on business task delegation. Further evolutions in real world scenarios are required to approximate the consequences of this issue.

The amount of compulsory business delegation determines the added complexity of the presented architecture. If edge servers can perform a majority of service requests autonomously the consequences are not stringent. If developers are, however, required to arrange dynamic diversion for a large number of business tasks then the development process gets both cumbersome and error prone. Unfortunately the diverse requirements of business solutions make it impossible to give a general answer to this question. A detailed analysis is required to expose task categories that are especially affected by business task delegation and therefore not well suited for the service broker scenario.

The presented architecture, introduces partial replication to reduce the communication overhead among service brokers and the origin site. The dissemination scheme depends upon the division of business information into an index and data items. Determining the correct granularity of index and data units has an important performance implication and significantly influences the type of business tasks that can be performed on top of the available set of data. Choosing the granularity too coarse may reduce the benefit of partial replication since the individual constituents of a given data unit might have different access patterns. With data units that are too fine grain it may, however, become difficult to track them individually. A promising solution [106] to this problem suggests the use of fine grain data units that are automatically clustered for dissemination according to similarity of access patterns. It remains to be seen if this approach is useful for the presented service broker architecture.

Dissemination Logic

The architecture outlined in this work supports the deployment of service brokers by integrating application specific dissemination into multitiered enterprise applications. It is, however, unfortunate that universal dissemination logic can not be supplied together with the data service layer. To overcome this shortcoming the data service layer is equipped with well defined interfaces that enable the extension with tailored dissemination logic. An application developer can use these interfaces to define any kind of dissemination logic and optimize it to the application scenario. The use of design patterns enables changing the dissemination logic at deployment time and runtime and therefore allows dynamically selecting an optimal distribution strategy [98] for individual data categories.

The architectural proposal provides a rich set of communication patterns, well defined interfaces and, multiple extension points to facilitate the definition of tweaked dissemination logic. The synchronous communication channel, driven by the dissemination logic at the edge server, supports both pull on demand and speculative pull. The asynchronous communication channel is backed by a message oriented middleware provider and the publish-and-subscribe paradigm. Consequently, unicast, multicast, and broadcast push are available to the dissemination logic at the backend system in order to propagate invalidations and data items. With dissemination logic at both the edge server and the backend system it is possible to use context information of all servers involved to achieve a more sophisticated dissemination strategy. While edge servers have knowledge of cache hits the backend system can keep track of cache misses and database hotspots to optimize the dissemination behavior.

A comprehensive dissemination algorithm will use access patterns and domain knowledge to define an efficient dissemination strategy. While access patterns may vary from time to time and require the algorithm to adapt to changing conditions, domain knowledge is usually static and specific to a particular application scenario. For example, profiles of north american users are usually not required to reside on european servers while information on products exclusively shipped in asian are not relevant to customers in the middle east.

The interfaces for the definition of dissemination logic are discussed in the following chapters. To assist the implementation, evaluation, and deployment of efficient algorithms the dissemination strategies can either be changed at deployment time or at runtime. In addition, the data service layer provides metadata that can be exploited by application developers to define more efficient dissemination algorithms. The presented architecture can be seen as an ideal testing environment for the development of application specific dissemination. It provides several communication patterns to choose from, a comprehensive communication infrastructure, and the possibility to use business applications with stringent data requirements for testing. Moreover, the well defined interfaces enable a direct comparison of several algorithms in diverse application scenarios.

Dynamic Environments

The growing complexity of distributed systems combined with the commitment to shared and dynamically changing communication infrastructures (the internet) results

into extensive middleware challenges. Current middleware solutions aim to provide complete location transparency to facilitate the development of distributed applications. The application developer is usually exposed to middleware interface without the opportunity to inquiry additional context information. The internet and other shared communication infrastructures are, however, characterized by constantly changing quality of service (QoS) conditions that effect available bandwidth, current error rate, available security context, and costs involved in transferring data. The dynamically changing QoS attributes of the underlying communication infrastructure may, however, require the application to adapt its behavior. A mobile device, for instance, that experiences significant decreasing network bandwidth due to network congestion may want to decline processing power to save energy.

The fundamental mechanism that enables applications to react to changing conditions is the exchange of metadata or context information between the application code and the middleware solution. The middleware has to provide some sort of feedback that can be interpreted by applications and used to determine the appropriate actions in order to adapt the behavior to the new situation. The term *context aware application* will be used to refer to software solutions that perform *dynamic system configuration* to accommodate varying QoS conditions.

It is evident that context aware applications which have to interpret and react to changing conditions complicate the development process significantly. A key challenge is the design of an interface that streamlines the negotiation of QoS properties between applications and the connecting middleware without drastically influencing the application model. Although the integration of QoS management into middleware solutions is essential, a procedure [15] for doing so has yet to be agreed upon. In general, application developers should be able to define the kind of feedback changes that trigger a notification to the application code. Moreover, the negotiation and handling of environment changes should be optional and dependent on the application scenario.

The presented architecture supports context aware applications in the sense that the data service layer provides feedback on individual data retrieval and storage operations. A metadata record is provided for each data operation that contains additional details on the accomplishment and may be used to dynamically tweak the dissemination strategy. Moreover, the data service layer can trigger an event when a business task results in the transmission of more than a predefined limit of data items from the origin site. The application can use this feedback to cancel the current operation and divert processing to the backend system. Carrying the task out at the origin site and returning only the result back to the edge server usually results into a significant decrease of transit traffic.

Chapter 5

Architecture

The preceding chapter analyzed the state of the art of the edge server architecture and determined the requirement for valorized edge servers that perform service oriented computing. The presented solution suggests a decomposition into service brokers that builds upon a flexible infrastructure for partial replication. The design considerations lead to the system and application model presented in section 4.2.2 and 4.2.3 respectively.

This chapter discusses the architecture of the data service layer and the integration into enterprise applications built with the Java 2 Enterprise Edition (J2EE). The individual components of the data service layer are introduced and the incorporation with the technologies supplied by the J2EE is discussed. The focus is on describing the collaboration between the data service layer and the infrastructure provided by the enterprise software architecture. The design and implementation of the data service layer and the interfaces to the business logic are discussed in more details in the next chapter.

5.1 Environment and Context

Enterprise software architectures come with different programming models, varying value added services, specialized distributed object protocols, and incompatible server side component models. Unfortunately, these diverse features, emphasis, and characteristics make it impossible to provide a universal design for the data service layer. This section defines the environment for the realization of the data service layer and the reference implementation outlined in the next chapter. Section 5.1.1 argues why the Java 2 Enterprise Edition has been selected while section 5.1.2 describes the software products used for the implementation.

5.1.1 Enterprise Software Architecture

There exist two competing and widely adopted enterprise software architectures today, Sun Microsystems Java 2 Enterprise Edition (J2EE) [43] and Microsofts .NET framework [83]. The unmanageable amount of promised features and benefits renders, however, a comparison and ultimately a commitment to a particular enterprise

software platform a cumbersome and complex task. The identification of technical requirements for a given software solution is therefore crucial in determining the most suitable enterprise software architecture.

The data service layer is expected to mediate between the business logic, the data source, and remote systems. It is therefore necessary to guarantee a seamless integration into the business tier and a flexible communication infrastructure. In particular, the data service layer requires a robust, flexible, and powerful server side component model and the support for both synchronous and asynchronous communication. With this information in mind, it should now be possible to determine the most suitable enterprise software architecture for the realization of the data service layer.

While both enterprise software architectures seem similar at first glance the features and merits vary significantly. COM+ [119] is based on COM [85] originally designed for use on the desktop and eventually pressed into a server side solution. Consequently the COM+ services in the .NET framework are focused on stateless components and do not provide built in support for persistent transactional objects. Synchronous distributed access is possible with DCOM [120] while asynchronous communication may be achieved with Microsoft Message Queuing (MSMQ) [82]. Although the .NET framework provides many sophisticated features as an open standard it falls short. Essentially all technologies are Microsoft's proprietary implementations and therefore lack interoperability and third party support. The EJB [86] component model on the other hand, is an open standard that provides support for stateless, stateful, and persistent components. Synchronous distributed access is possible with Java RMI/IIOP [52] while asynchronous communication can be achieved with any Java Message Service (JMS) [87] compliant MOM provider. The standard based approach of the J2EE allows customers to choose among several vendor products, ensures interoperability, simplifies migration, and pushes third party products and solutions.

We argue that the J2EE fulfills the requirements for the data service layer better than the .NET framework. The exigency to perform synchronous and asynchronous interactions across the internet is better satisfied by the Java based technologies. Java RMI/IIOP is a language independent distributed computing protocol implementation based on the CORBA Internet Inter-Operability Protocol (IIOP) [91] and especially designed for internet based interactions. On the other hand, several JMS implementations are particularly crafted for internet scale deployments and are widely adopted for e-marketplaces and other large scale distributed environments. Moreover, the EJB component model supports stateful and persistent components which significantly decrease the development effort for the data service layer.

The remaining sections provide a more detailed discussion of the technologies utilized and explain the integration of the data service layer more precisely. It will be explained how the data service layer is able to leverage the persistence mechanism of the server side component model. Moreover, an in depth analysis of the integration of asynchronous messaging and the resulting consequences and benefits is provided.

5.1.2 Software Products

The standard approach of the Java 2 Enterprise Edition promises business solutions to work in any application server that supports the complete specification. Obviously, care must be taken when using proprietary extensions and vendor specific facilities.

The architecture of the data service layer aims to solely build upon standardized features of the J2EE. The implementation described in the next chapter leverages, however, some minor vendor specific features. Nevertheless the presented design should be straightforward to migrate to competing J2EE compliant products and solutions. The use of proprietary extensions will be stated within the corresponding description whenever possible. The following enumeration outlines the software products used for the design of the architecture and the realization of the reference implementation.

Bea WebLogic Server 8.1
: The Bea WebLogic Server 8.1 [17], a widely adopted J2EE 1.3 [104] compliant application server, provides a reliable platform for the implementation and evaluation of the data service layer. According to the system model an application server is deployed at each edge server and the backend system.

Progress SonicMQ 5.0.2
: The Progress SonicMQ 5.0.2 [108] broker, a JMS 1.0.2b [54] compliant message oriented middleware provider, supplies a flexible communication infrastructure for the dissemination of data. According to the system model the MOM provider is used for asynchronous server initiated unicast, multicast, and broadcast propagation.

PostgreSQL PostgreSQL 7.3.3
: The PostgreSQL PostgreSQL 7.3.3 [99] server, a SQL-92 [11] compliant RDBMS, serves as a reliable open source enterprise information system. According to the system model the database is deployed at each edge server and the backend system.

Sun Microsystems J2SE 1.4.2
: The Sun Microsystems J2SE 1.4.2 [115] platform, a complete environment for developing Java applications, is used in combination with the Sun Microsystems J2EE 1.3 platform to implement the data service layer.

JetBrains IntelliJ IDEA 3.0.5
: The JetBrains IntelliJ IDEA 3.0.5 [70] integrated development environment (IDE) provides extensive refactoring automation and enterprise development support for the design and implementation of the data service layer.

Apache Ant 1.5.3
: The Apache Ant 1.5.3 [13] tool supplies a platform independent extensible markup language (XML) [21] based build utility that is utilized to compile, deploy, stage, and test the data service layer.

Apache Log4j 1.2.8
: The Apache Log4j 1.2.8 [14] library provides advanced logging facilities that are utilized in the reference implementation.

Erich Gamma Kent Beck JUnit 3.8.1
: The Erich Gamma and Kent Beck JUnit 3.8.1 [41] framework provides flexible unit testing for the verification of the data service layer.

5.2 Java 2 Enterprise Edition

The brief comparison of enterprise software architectures in the preceding section asserted that the Java 2 Enterprise Edition is well suited to fulfill the technical requirements of the data service layer. This section provides a concise overview of the J2EE and skim over the most essential enterprise APIs required for understanding the architecture, design, and implementation of the data service layer. It is far beyond the scope of this thesis to provide a comprehensive comparison of enterprise software architectures or an in depth presentation of the technologies supplied by the J2EE.

The Java 2 Enterprise Edition defines a standard that unites several Java enterprise technologies into a complete platform for implementing multitiered enterprise applications. The specification outlines how these technologies work together to form a comprehensive solution for designing, developing, deploying, and managing applications in an enterprise environment. The J2EE establishes mechanisms for the configuration of application properties and resources at deploy time and supplies various automatically managed value added services.

The current Java 2 Enterprise Edition version 1.3 [104] is built on three component models, namely Enterprise JavaBeans 2.0 (EJB) [86], Servlets 2.3 [58], and JavaServer Pages (JSP) 1.2 [18]. In addition, the J2EE unifies many other connecting technologies, such as Java Remote Method Invocation (RMI) [52], Java IDL (CORBA) [43], Java Database Connectivity 2.0 (JDBC) [100], Java Naming and Directory Interface 1.2 (JNDI) [43], Java Message Service 1.0.2 (JMS) [87], JavaMail 1.2 [43], Java Activation Framework 1.0 (JAF) [24], Java API for XML Parsing 1.1 (JAXP) [88], J2EE Connector Architecture 1.0 (JCA) [113], Java Authentication and Authorization Service 1.0 (JAAS) [112], and Java Transaction API 1.0.1 (JTA) [30]. Please note that at the time of this writing vendors are soon expected to incorporate the recently released Java 2 Enterprise Edition version 1.4 [105] specification into their enterprise offerings. The J2EE 1.4 comes with many novel features and aims to overcome several shortcomings and limitations of the current specification. As outlined in section 5.1.2 the reference implementation is, however, still based on a J2EE 1.3 compliant application server.

The J2EE platform is designed to provide a component based approach to the development, assembly, and deployment of server side and client side enterprise applications. The multitiered distributed application model offers reusable components, a unified security model, flexible transaction control, web service support, a standardized naming context, built in resource management, and automatically managed concurrency and persistence. The Java 2 Enterprise Edition is based on the following fundamental building blocks:

Components
: A component, in the J2EE application model, is a self contained functional software unit that is assembled with other components into a comprehensive enterprise software solution. The J2EE specification defines application client components, web components, and business components.

Containers
: A container, in the J2EE application model, mediates between clients, components, and low level platform specific functionality. Containers provide a run-

time environment that transparently supplies clients and components with value added services, such as transaction support, resource pooling, and lifecycle management. The surrogate architecture allows the configuration of component behavior at deployment time rather than in program code. The J2EE specification defines client application containers, web containers, and business containers.

Connectors
: A connector, in the J2EE application model, defines a resource adapter that is usually supplied by a tool vendor or system integrator to support access to an enterprise vendor offering. Connectors can be plugged into any J2EE product to access and interact with underlying resource managers, such as databases and enterprise information systems. Since a resource adapter is specific to its resource manager the pluggable connector architecture promotes flexibility by enabling a variety of implementations for a specific type of service.

While components are the key concern of application developers, containers and connectors are implemented by system vendors to conceal complexity of the underlying platform specific facilities.

The J2EE provides two elementary mechanisms that allow a seamless collaboration and orchestration of components, resources, and service: a standardized naming service and deployment descriptors. The JNDI in general and the JNDI environment naming context (ENC) in particular, serve as a uniform way to locate and access components, properties, resources factories, and administrative objects. Deployment descriptors, on the other hand, define the relationship of components to their environment and ultimately the assembly of many components into enterprise applications. A deployment descriptor is virtually a complex set of configuration attributes that specifies a components runtime behavior including security context, transactional attributes, access lists, and lifecycle behavior. In fact primary the combination of deployment descriptors and JNDI ENC cause the J2EE to appear as a uniform platform and guarantees a seamless collaboration among all Java enterprise APIs.

After this rapid fire introduction of the Java 2 Enterprise Edition it's now time to skim over the Java enterprise APIs with particular impact on the data service layer.

5.2.1 Java Naming and Directory Interface

Virtually every computing environment relies on some sort of naming service or directory service* to identify, arrange, and locate various entities either in machine readable or in human understandable namespaces. The entities in a naming or directory system can range from files in a filesystem and names in the domain name system (DNS) [40] to user profiles in a lightweight directory access protocol (LDAP) [126] directory and Enterprise JavaBean (EJB) components in an application server.

The Java Naming and Directory Interface (JNDI) provides a unified implementation and protocol independent interface to naming and directory services. It allows Java

* Directory services are a natural extension of naming services that allow the association of attributes with objects. With a directory service it is possible to access the attributes of objects and to search for objects based on their attributes.

programs to utilize name and directory servers to associate names with objects, locate objects by name, and lookup objects by a set of specified attribute values. The JNDI architecture comes with two interfaces, the JNDI application programming interface (API) used by client applications to interact with the naming manager and the JNDI service provider interface (SPI) implemented by vendors for a specific naming or directory service. This structure allows Java applications to leverage a common interface, the JNDI API, to access arbitrary naming and directory services, provided that a JNDI SPI implementation is available. The client application is not exposed to implementation or protocol details and in fact can use any JNDI supported naming or directory service without modification. The support for new naming or directory services is achieved merely by installing the corresponding JNDI SPI implementation. Moreover, the JNDI specification allows names that span multiple namespaces and therefore may require service providers to cooperate in fulfilling a client JNDI operation. The current JNDI version comes with several service provider implementations [116] for the most common naming and directory protocols. The support ranges from object registries, such as Java RMI and CORBA Naming Service [94], to general purpose directory services such as LDAP, Network Information Service (NIS), and Novell Directory Service (NDS) [90].

In enterprise solutions naming and directory services are utilized to access entities such as distributed objects, enterprise information systems, employee profiles, printers, and security credentials. In general there exist to basic types of JNDI providers:

General Purpose JNDI Providers
: A general purpose JNDI provider can be used to store various types of objects on behalf of diverse clients. The objects are usually kept in a persistent store and may be accessed over a long period of time independent of server restarts. Typical examples of general purpose JNDI providers include LDAP, NIS, and NDS.

Vendor Specific JNDI Providers
: A vendor specific JNDI provider is usually integrated into some enterprise offering, such as an application server, and provides access only to a particular object type. In general, vendor specific service providers are utilized by a specialized set of clients that are able to retrieve objects but frequently can not add additional information. Typical examples of vendor specific JNDI providers include J2EE application servers and JMS providers.

The J2EE relies heavily on JNDI which is an integral part of every J2EE compliant application server and foreign JMS provider. The Java Naming and Directory Interface is used by components to access deployment information, collocated and remote components, data sources, enterprise messaging systems, and various other types of resources. In the data service layer implementation JNDI serves, among other things, to obtain EJB references, acquire JDBC access, retrieve JMS connection factories, and access environment entries.

5.2.2 Java Remote Method Invocation

Distributed object architectures, introduced in section 2.1.1, are a fundamental part of enterprise software architectures. The Java Remote Method Invocation (RMI) is the

distributed object system built into the Java platform. It extends the object oriented programming paradigm to distributed client and server architectures by allowing Java programs to invoke methods on remote objects. The Java RMI object services include a naming and registry service, an object activation service, distributed garbage collection, dynamic class loading of remote interfaces, and activation groups.

The primary communication protocol Java Remote Method Protocol (JRMP) is exclusively designed for interactions between Java programs. The limitation to a Java only distributed object scheme introduces simplicity since both client and server share a common set of data types and have access to the object serialization and deserialization features of the Java platform. The deployment in heterogeneous enterprise environments requires, however, interoperability with legacy code and the ability to communicate with distributed objects implemented in different programming languages. Consequently Sun Microsystems offers three options to incorporate Java RMI into enterprise solutions: Java RMI/IIOP, Java Native Interface (JNI) [76], and Java Interface Definition Language (IDL) [43].

Java RMI/IIOP
: Recent versions of the Java environment introduce an optional communication protocol, Java RMI/IIOP, to allow direct communication with CORBA based distributed objects. The interoperability with CORBA, a language independent distributed object scheme with bindings to most popular programming languages, bridges the language gap and enables collaborate with all major enterprise solutions.

JNI
: Access to legacy code can be achieved with JNI which makes it possible to wrap existing C [67] or C++ [68] code with a Java interface that can then be exported remotely as a Java RMI object.

Java IDL
: The interfaces to remote CORBA objects are described with a platform and language independent interface definition language (IDL). With the use of Java IDL it is possible to create an IDL mapping of Java RMI interfaces and therefore allow any CORBA client to access exposed Java remote objects.

The most appropriate solution for interfacing Java with legacy code and foreign distributed objects depends on the concrete application scenario. The CORBA approach offers the advantage that Java is not required on the server, presumable a legacy system, which is crucial since finding a stable Java implementation may be a problem. The JNI approach, on the other hand, omits CORBA entirely and offers native Java RMI remote communication.

The ubiquity of heterogeneous computer networks in enterprise solutions is the reason why the J2EE specification requires support for both Java RMI/JRMP and Java RMI/IIOP. The data service layer implementation utilizes Java RMI for synchronous pull on demand and the delegation of business tasks.

5.2.3 Java Database Connectivity

The Java Database Connectivity, JDBC[†], defines a set of interfaces for working with relational database systems and other tabular data sources, such as spreadsheets or flat files. It encapsulates major database functionality that allows Java programs to send structured query language (SQL) query and update statements to database servers. Moreover, JDBC establishes semantics for processing query results, determining configuration information, and executing stored procedures.

The JDBC API is a vendor and platform independent interface that relies upon a pluggable driver architecture to support multiple database products. The driver implementations hide the complexity of communicating with the database backend while the JDBC API provides a common interface for data retrieval and storage operations. The current version of the Java Database Connectivity supports a variety of facilities that go beyond basic SQL inquiries, including database connection pooling, batch updates, scrollable result sets, and storage of Java objects in databases. To guarantee a seamless integration into the J2EE database connection information can be obtained from a naming service, support for distributed transactions is provided, and database query results can be treated as JavaBeans components for direct reuse in presentation logic.

The data service layer uses JDBC for complex inquiries to the cache at edge servers and the data source at the backend system. The persistence mechanism of the server side component model is used, on the other hand, to perform simple queries and object to relational mapping.

5.2.4 Java Message Service

The Java Message Service (JMS) is a vendor agnostic Java API that abstracts access to enterprise messaging systems. It provides a programming model that allows applications to leverage message oriented middleware systems for sending and receiving notifications of events and data. The JMS specification defines the incorporation of asynchronous messaging into Java applications and specifies messaging delivery and retrieval semantics. The JMS API is implemented by enterprise messaging vendors to provide Java applications with a universal view of message oriented middleware providers. The common interface makes messaging clients portable across enterprise messaging products in the sense that the same API can be reused to access different MOM providers.

The JMS specification is an industry effort of several message oriented middleware vendors under the lead of Sun Microsystems. It unites the characteristics of enterprise messaging systems, already discussed in section 2.1.2, into a flexible API for Java applications. The asynchronous model of JMS allows systems to interact without the requirement to tightly coupling. The decoupled nature enables delivery of messages to systems that are not currently running and processing when it is convenient. Moreover, communication with other systems can take place without the knowledge of network addresses and routing information. The abstract communication model

[†] The string JDBC is not officially an acronym for Java Database Connectivity but a registered trademark of Sun Microsystems.

makes it possible to dynamically add and remove participants and to alter the routing of messages in already deployed enterprise systems. In contrast to synchronous communication, such as Java RMI, sender and receiver are not blocked until a service request is completed. The message is merely handed over to the message oriented middleware provider that is responsible for delivering the message to the appropriate recipients.

Since JMS serves exclusively as an interface to enterprise messaging systems it enables Java applications to communicate with systems using the native messaging API of MOM providers. The JMS connection is responsible for receiving native messages and converting them into the appropriate JMS message representation. The interoperability with native message clients makes JMS an excellent choice for enterprise application integration and heterogeneous business-to-business scenarios.

The JMS specification defines the publish-and-subscribe and point-to-point messaging models whereas the so called *messaging domains* are intended for one-to-many broadcast and one-to-one delivery of messages respectively. The messaging clients sending a message are referred to as *producers* while the clients receiving a message are called *consumers*. In addition, a publish-and-subscribe destination is named *topic* while a point-to-point virtual channel is specified as *queue*. The *JMS provider*, a compliant enterprise messaging system, is required to provide suitable tools for creating, configuring, and ultimately maintaining topics and queues referred to as *administrative objects*. The administrative objects are usually exposed to a naming service and retrieved by JMS clients with the Java Naming and Directory Interface.

The publish-and-subscribe messaging model allows producers to send a message to all consumers subscribed to a particular topic. It is actually a push based model where each message is automatically broadcasted to consumers that are not required to request or poll the topic for messages. With JMS, consumers can establish a durable connection with the messaging provider which guarantees that messages published during a disconnection are preserved. The enterprise messaging system will collect the messages on behalf of the disconnected consumers and deliver them upon reconnect. The point-to-point messaging domain, on the other hand, is used to send and receive messages both synchronously and asynchronously via queues. In traditional enterprise messaging systems, clients are required to utilize a pull or polling based model to retrieve messages sent to a queue. The JMS specification introduces, however, an option that allows messages to be automatically pushed to point-to-point consumers.

The decoupled asynchronous character of message oriented middleware makes JMS an extremely powerful and critical Java enterprise API. It comes as no surprise that version 1.3 of the J2EE requires any compliant application servers to supply a full JMS provider including support for both messaging domains. JMS is a crucial component in the design of the data service layer. It will be used by the origin site for asynchronous unicast, multicast, and broadcast delivery of notifications and data modifications.

5.2.5 Enterprise JavaBeans

The Enterprise JavaBeans (EJB) specification defines a standard server side *distributed component model* for the development and deployment of scalable, transactional, secure, and portable enterprise applications. The EJB framework provides an environment, in which enterprise application developers can efficiently take advantage of dis-

tributed transaction management, persistence, resource pooling, and security facilities provided by an application server. The server side component model introduced by EJB defines the interactions among components, containers, and the application server. In addition, it dictates conventions and policies for the development and deployment of enterprise beans.

The Enterprise JavaBeans architecture is a server side component model in the sense that enterprise beans are deployed in an application server, a standard component model in the sense that enterprise beans can be deployed in any J2EE compliant environment, and a distributed component model in the sense that enterprise beans can be accessed synchronously and asynchronously by collocated and remote components and applications. Enterprise JavaBeans can be developed and packaged separately and composed with other components to form J2EE solutions that solve any number of application requirements.

An enterprise bean consists of multiple interfaces that allow local and remote access, a bean class that implements the business and lifecycle methods, and a deployment descriptor that declares configuration attributes and runtime behavior. In addition, each enterprise bean is equipped with a default namespace, the JNDI ENC, which provides access to resources that are implicitly managed by the application server. In contrast to conventional distributed objects, the resources requested through JNDI ENC are automatically pooled by the application server and enrolled in transactions as needed. Moreover, the use of deployment descriptors allows adapting transaction policies, access control, and service availability at deploy time without modifying the program code. The flexibility provided by deployment descriptors in combination with JNDI ENC allows a seamless adaptation of enterprise beans to different conditions and application scenarios. In particular, resource configurations such as JMS administrative objects, JDBC database connections, and JavaMail settings can be changed without modifying the source code of the bean class.

The EJB component model significantly simplifies developing distributed components that are managed in a robust transactional environment. It allows developers to focus purely on the development of business logic and the definition of runtime attributes. The insulation from the underlying low level facilities not only reduces complexity but streamlines the adaptation of business components to different conditions and environments.

An EJB component is a unit of business logic and business data that is deployed in the business container of an application server. The application server is responsible for exposing the component as a distributed object and for automatically providing services such as resource pooling, lifecycle management, security, and naming services. In essence, EJB components can be used in any situation where conventional distributed objects are useful and require access to value added services. There are three fundamentally different types of EJB components, entity beans, session beans, and message driven beans:

Entity Beans

An entity bean models business concepts that combine business logic with persistent data. The state of an entity bean is made persistent across client sessions by storing it permanently in a database or some other kind of data source. Concurrent access to the data entity represented by an entity bean is automatically

handled by the business container. There are two types of persistence mechanisms supported for entity beans, *container managed persistence* and *bean managed persistence*. With container managed entity beans the synchronization of instance fields is automatically managed by the container. Bean managed entity beans, on the other hand, are designed to manage persistence manually, as is often required when dealing with legacy systems.

Session Beans
A session bean models business tasks that are executed on behalf of client requests. There are two types of session beans, *stateless* session beans that do not maintain state across method calls and *stateful* session beans that preserve instance fields for subsequent service requests by the same client. A session bean instance is typically accessed by a single client, exists for the duration of the client session, and gets removed by the application server upon expiration.

Message Driven Beans
A message driven bean is designed to provide a robust and scalable environment for the concurrent processing of asynchronous messages. It usually models business processes by coordinating interactions of other beans and resources according to the messages received. A message driven bean is stateless and therefore advocates coarse grain interfaces by requiring each message to contain all information necessary to perform a business task.

Both session beans and entity beans are synchronous server side components that are accessed by remote clients as distributed objects via the Java RMI/IIOP distributed object protocol. Message driven beans, on the other hand, are asynchronous server side components that act as stateless JMS consumers by concurrently processing messages sent by remote applications. The interactions among collocated enterprise beans can be performed via local interfaces that do not incur the overhead of remote Java RMI/IIOP based communication.

The server side component model provided by EJB constitutes a comprehensive solution for the development and deployment of business applications. Consequently, the data service layer relies on all three enterprise bean types to mediate between the business logic, the data source, and remote hosts.

5.2.6 Emerging Enterprise Platform

When Java was first introduced most developers focused on the competitive advantages it offered in terms of distribution and platform independence, user interface characteristics, and built in security mechanisms. In the last couple of years Java has, however, been recognized as a comprehensive platform for creating distributed server side enterprise applications. This shift has much to do with the emerging role of the Java platform in providing implementation and vendor independent abstractions for common enterprise technologies.

In fact, most Java enterprise APIs united in the J2EE introduce some sort of abstraction of existing enterprise solutions. The first and most familiar example is JDBC which provides a vendor independent interface for accessing relational databases. Moreover, the Java Naming and Directory Interface (JNDI) abstracts access to naming and

directory services, the Java Management Extensions (JMX) [114] abstract access to computer devices on a network, and the Java Message Service (JMS) abstracts access to enterprise messaging systems. Even the Enterprise JavaBeans (EJB) component model is an abstraction that allows components to be deployed in any J2EE compliant application servers.

The vendor agnostic approach of the Java 2 Enterprise Edition guarantees a broad industry acceptance and allows a seamless collaboration of diverse enterprise offerings. The recent focus on standard based web services may further promote interoperability [107, 79] of diverse enterprise software architectures and hopefully bridge the gap between the competing standards from Sun Microsystems and Microsoft.

5.3 Data Service Layer

The application model of the Java 2 Enterprise Edition, outlined in section 5.2, is based on three fundamental parts: components, containers, and connectors. The institution of connectors, as defined by the J2EE Connector Architecture (JCA), enables EIS vendors to provide standard based resource adapters for the connectivity between containers and resource managers. This abstraction is frequently highlighted by the introduction of an additional integration tier as illustrated in the logical separation of concerns diagram in figure 5.1.

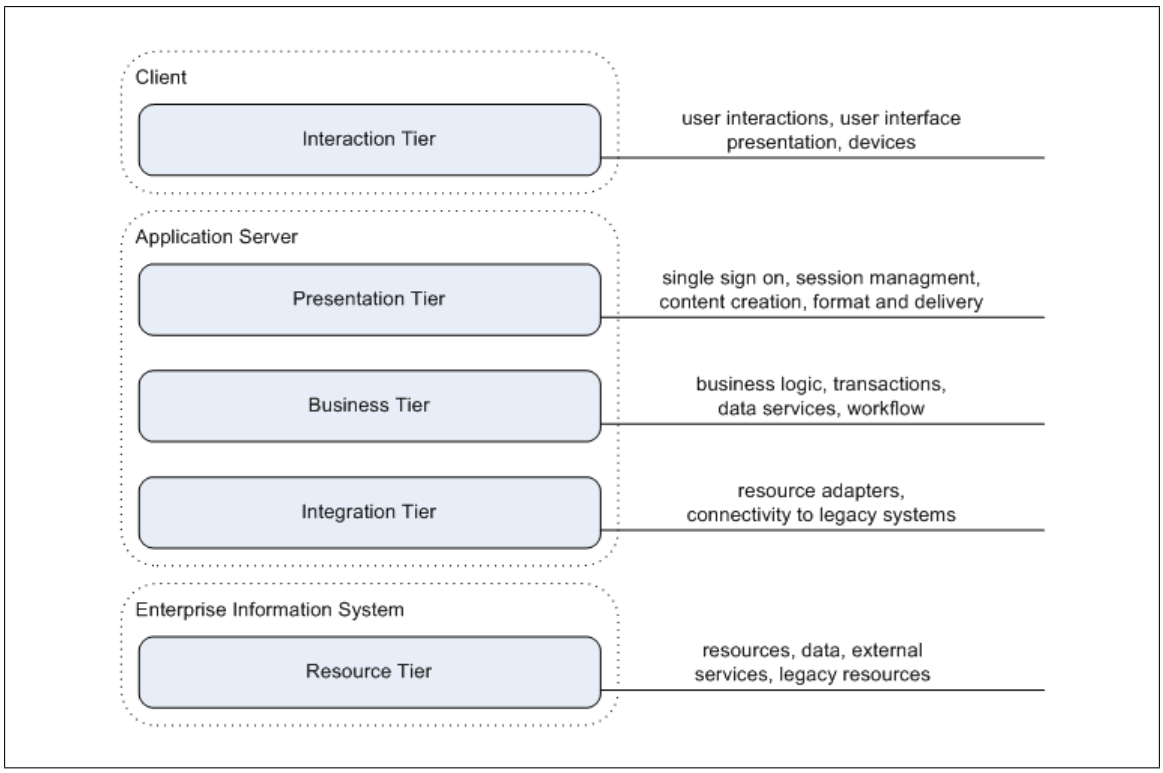

Figure 5.1: Logical Separation of Concerns

The data service layer (DSL) can be seen as an extensions that mediates between the business tier, the integration tier, and remote systems. The data service layer is deployed within the business container, as outlined in figure 4.3, and implemented by

application developers. The abstraction provided by the data service layer is, however, similar to the integration tier in the sense that both supply an infrastructure that controls access to the data source. The architecture of the data service layer and the integration into the J2EE platform is outlined in more details in the following sections.

5.3.1 Architecture Overview

The data service layer architecture described in this section aims to take account of the design considerations discussed in section 4.2.1. In addition, emphasis is on developing a general purpose architectural design that is not geared towards a particular application scenario. The information presented in this section and the next chapter is meant to establish strategies and guidelines for the development of an infrastructure that handles dissemination among dispersed systems efficiently. In other words, the goal is to provide developers with blueprints that aid in extending enterprise applications with valorized edge servers.

In general, the data service layer is responsible for the coordination of *data operations* and *data dissemination*. A data operation in this context is defined as an information retrieval or storage operation that is performed on the local data store. While data operations at edge servers access the locally available cache on the origin site they send inquiries to the backend database. Data dissemination, on the other hand, involves pulling and pushing data among the dispersed systems of the edge server architecture.

The suggested data service layer architecture realizes complex queries on top of JDBC and object-to-relational mapping of business objects with entity beans (EB). The deployment of standard J2EE technologies for data access guarantees a seamless integration into existing and newly built enterprise applications and allows the data service layer to take full advantage of value added services. Moreover, the reliance on entity beans ensures a smooth incorporation of the data service layer into the business container and preserves the data access semantics expected by business logic. The synchronous data exchange is implemented with Java RMI while asynchronous data dissemination relies on JMS. The utilization of JMS is crucial to the presented design of the data service layer and will be discussed in more details in the remaining part of this section.

An architectural overview is provided in figure 5.2 which shows the software components of the data service layer and the Java enterprise APIs utilized for data operations and data dissemination. In essence, the same architectural structure can be used at both the origin site and the individual edge servers. The data access and dissemination implementations vary, however, to accommodate context specific differences. The following discussion presents a design that allows deploying the same code base at all systems by solely varying configuration attributes within the deployment descriptors of the data service layer.

The components sketched in figure 5.2 are architecture level artifacts that model the overall structure of the data service layer. The discussion in this chapter is focused on describing the duties and responsibility of high level components. Furthermore, an analysis of the incorporation of the data service layer into the J2EE architecture and the collaboration with the Java enterprise APIs is provided. More details on the design and implementation of the data service layer can be found in chapter 6.

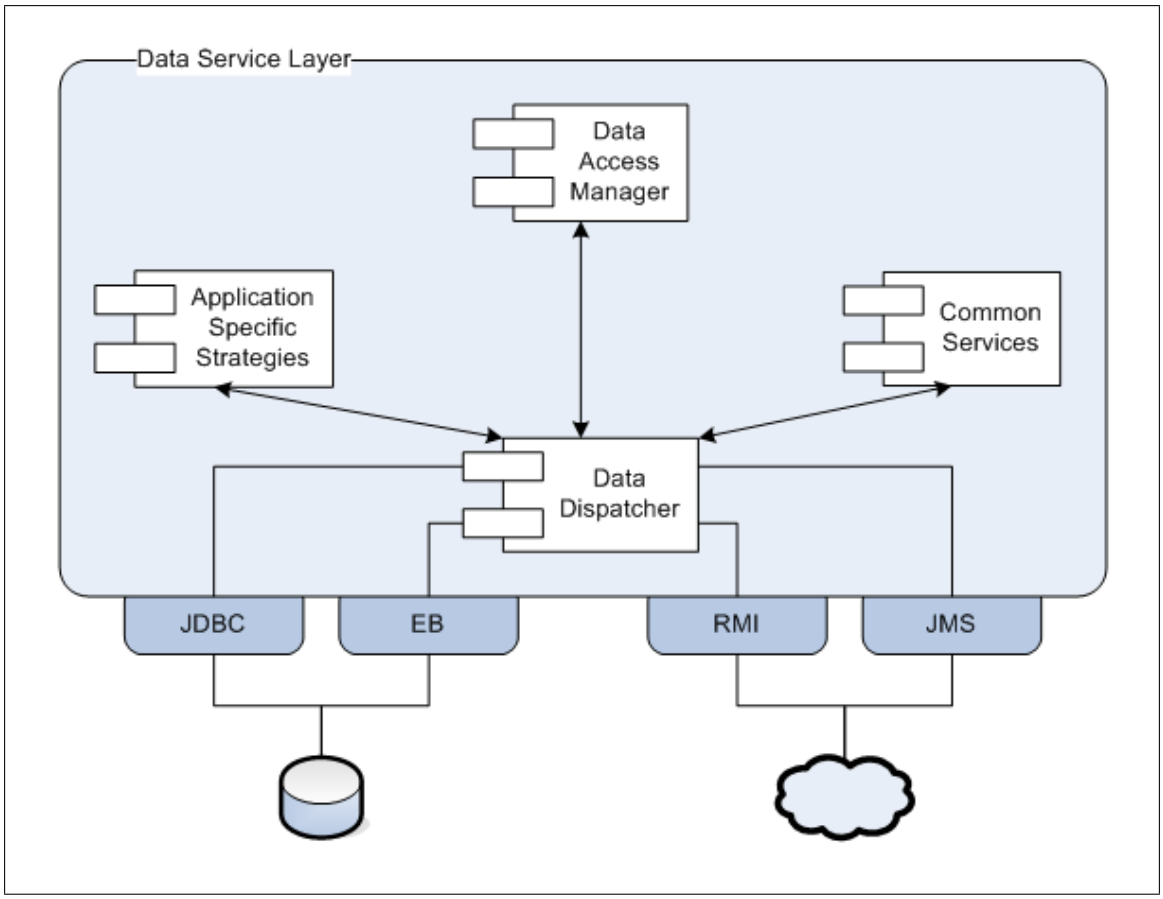

Figure 5.2: Data Service Layer

The data service layer consists of four fundamental components that collaborate to mediate between the business logic, the data source, and remote systems. The following description explains the duties and responsibilities of the individual architectural building blocks:

Data Access Manager

The data access manager (DAM) component encapsulates the complexity of interactions between the business objects and the data service layer. The data access manager provides a uniform coarse grain interface for data storage and retrieval operations. In addition, it can be utilized to change caching, dissemination, and expiration strategies at runtime and to access metadata information of data operations.

Application Specific Strategies

The application specific strategies (ASS) component defines the caching, dissemination, and expiration strategies that direct data operations and data dissemination performed by the data service layer. The component exposes well defined interfaces that facilitate the implementation of dissemination logic which is tailored towards a given application scenario.

Common Services

The common services (CS) component accumulates utility and helper classes, life-

cycle management, value objects, and exceptions that are utilized by the other components of the data service layer.

Data Dispatcher
The data dispatcher (DD) is the central component which manages data operations and data dissemination in collaboration with the DAM, ASS, and CS components. It interfaces directly with the data access and remote communication technologies provided by the J2EE. The data dispatcher mediates between the local data source and the remote systems to fulfill data requests and to disseminate date items.

While the architecture of the data service layer is the same at edge servers and the origin site functionality and runtime behavior of the application specific strategies and data dispatcher components vary significantly. The deployment descriptors configure the individual components and ultimately reflect if they are executed at the origin site or at an edge server. The application specific strategies component at the backend system determines which data items are proactively pushed while the same component at edge servers specifies which data items get cached and which are expired. The data dispatcher component, on the other hand, accepts pull requests and handles proactive push at the backend system while it issues pull request and receives pushed data at edge servers.

In general, data retrieval and storage operations are issues by the business logic by utilizing the interfaces provided by the data access manager. The data access manager applies the current configuration settings to the data operation and delegates it to the data dispatcher component. The data dispatcher is responsible for carrying out the operation and returns the result to the data access manager which in turn returns it to the business logic. The data dispatcher accomplishes data operations by interacting with the common services component, the application specific strategies component, the local database, and if necessary with remote data service layer instances. The data access manager raises an exception at edge servers if a data operation can not be fulfilled or results into the violation of a predefined quality of service constraint. The business logic is expected to handle data service layer exceptions by delegating the affected business task to the backend system.

The data dispatcher component attempts to fulfill data operations by accessing the local database. If this is not successful, as it might be the case on edge servers, a remote operation to the origin site is issued. The completion of data operations is directly followed by an inquiry to the application specific strategies component. The inquiry at the backend system specifies if the data dispatcher takes any further action such as sending data invalidation or proactively pushing data items to edge servers. The request at edge servers, on the other hand, determines if the data dispatcher is supposed to store the data item in the local cache.

The concrete tasks performed by the data dispatcher depend on (1) the type of system the component is executed at (edge server or backend system), (2) the origin of the data operation (data access manager or remote data service layer), and (3) the caching or dissemination decision determined by the application specific strategies component.

5.3.2 Dissemination Infrastructure

Data dispatchers form the actual dissemination infrastructure by carrying out data operations and data dissemination in coordination with the other components of the data service layer. Data dispatchers are further divided into several subcomponents that handle remote communication, access the local database, and collaborate with other parts of the data service layer. Since the functionality required at the origin site differs from the responsibilities expected at edge servers there are basically two configurations of the data dispatcher. The deployment descriptor configuration of the data dispatcher determines which components are deployed and defines the duties of the individual parts.

The overall dissemination infrastructure of two interconnected data dispatchers is outlined in figure 5.3. The component at the top of the diagram is configured for use on the origin site while the component at the bottom is set up for utilization at edge servers.

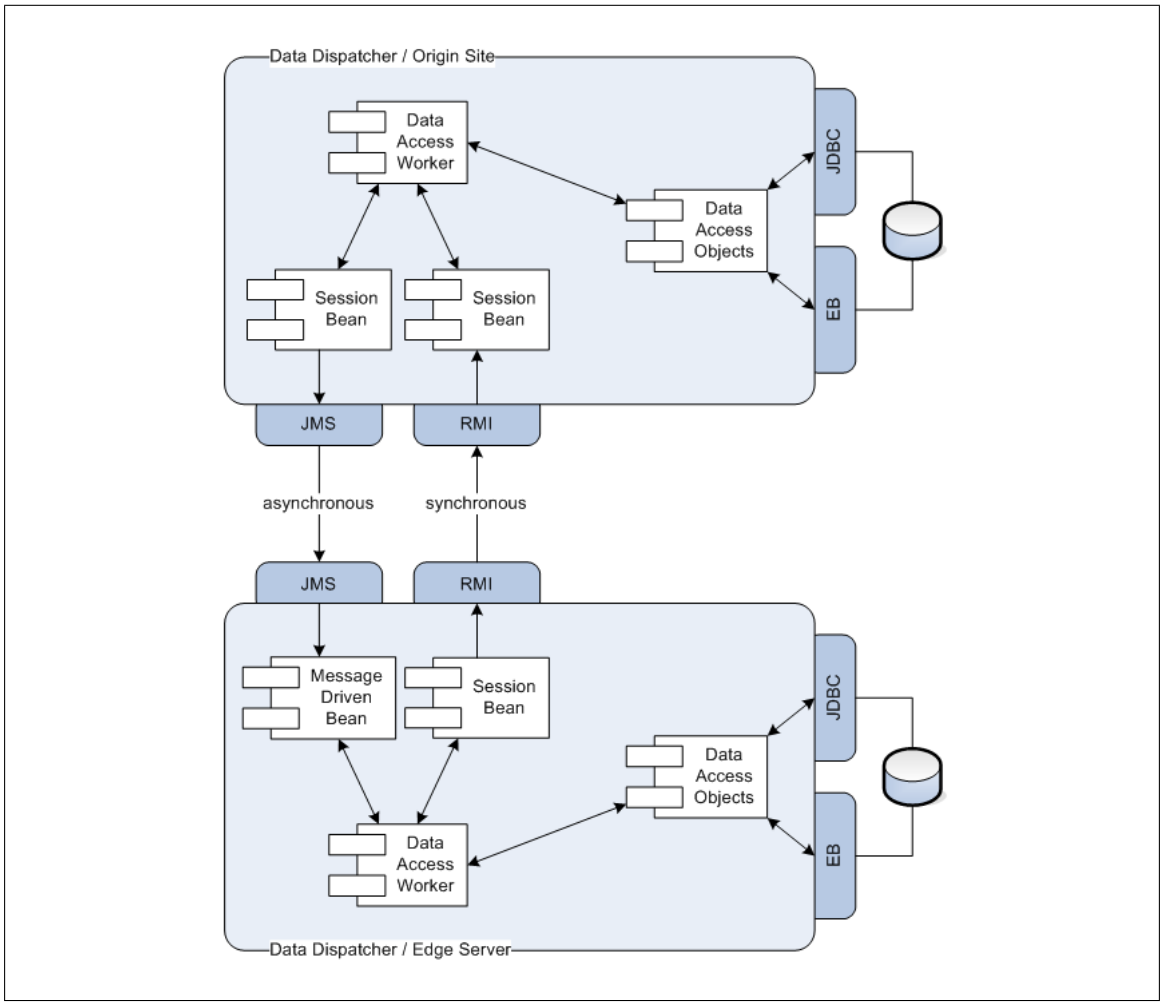

Figure 5.3: Dissemination Infrastructure

The data access worker (DAW) is the central component of the data dispatcher that interfaces with the other parts of the data service layer, outlined in figure 5.2, and coordinates both access to the local database and communication with remote data

dispatcher instances. The data dispatcher configuration at the origin site relies on two session beans for remote communication. One session bean acts as a JMS producer that dispatches data items and invalidations to edge servers while the other exposes a remote interface to serve as a Java RMI endpoint for pull requests. The data dispatcher configuration at edge servers, on the other hand, consists of a message driven bean and a session bean for interconnectivity with the backend systems. The message driven bean serves as a JMS consumer while the session bean sends Java RMI requests to the backend system.

Access to the local data source is encapsulated by the use of the data access object (DAO) [9] design pattern. The DAO manages the connection with the data source and provides access mechanisms to obtain and store data. It acts as an adapter between the data access worker and the database completely hiding the complexity of the data source implementation. The data access objects create and use so called *value objects* (VO) that can be obtained and modified by the data access worker. The data access worker acts as a transfer object assembler (TOA) [9] that constructs composite value objects (CVO) by retrieving data from local data access objects and remote data service layer instances.

The data access worker constructs new composite value objects based on data requirements of client requests. This involves locating the correct data access objects and, if necessary, contacting remote data service layer instances to fulfill an assembly request. At the origin site the data access worker solely relies on the data access objects to construct composite value objects. Client requests at the edge server may, however, require the data access worker to retrieve missing value objects from the backend system.

The data access worker component receives client requests from the data access manager. At the backend system the operation is first carried out at the local data source. The data access worker then constructs a composite value object with the result of the data request and inquiries the application specific strategies component to determine if data dissemination has to occur. It then returns the result to the data access manager and, if necessary, instructs the session bean to merely hand the affected value objects over to the JMS provider for dissemination. At the edge server the data access worker determines if the complete operation can be performed on the local database. If necessary parts of the operation are diverted by issuing a remote operation, otherwise the operation is completely carried out at the local database. The data access worker then constructs a composite value objects, as it is the case on the backend system, and returns the result to the data access manager. When the data dispatcher at the backend system receives a remote data operation request via Java RMI it performs the operation on the local database and, if necessary, broadcasts the data modification via JMS. Edge servers, on the other hand, accept remote operations via JMS and update the local database without taking any further actions.

The utilization of value objects is crucial to the design of the data dispatcher in particular and the data service layer in general. A value object, sometimes also referred to as transfer object or dependent value class, is a custom serializable object that is useful for packaging data and moving it between remote systems. In particular, it helps to separate the view of the data from the actual underlying persistence model. Furthermore, value objects are usually deployed to reduce network traffic and improve performance of remote operations by offering a coarse grain data transportation mechanism. The

data access worker further assembles individual value objects into composite structures and equips them with dissemination attributes. The attributes added by the modified transfer object assembler contain information that directs data dissemination and remote operations. The combination of composite value object and dissemination attributes will be referred to as *notification group* (NG) in the context of the data service layer. The data dispatcher relies on notification groups to perform remote data operations and to disseminate data among the dispersed systems of the edge server architecture.

The data access manager is responsible for converting client request into their appropriate notification group representation before delegating the operation to the data access worker. The data access worker analysis the notification group and depending on the type of operations included performs the operation merely on the local database or disseminates data among the systems of the edge server scenario. Upon successful completion of the data operation the data access worker returns a result notification group to the data access manager. The data access manager extracts the required value objects and ultimately returns the result to the client application.

To further complicate things, there are two types of value objects utilized by the data service layer: mutable value objects (MVO) and immutable value objects (IVO). The distinction of value objects provides a consistent interface that prevents clients from modifying value objects without committing the changes back to the data service layer. In addition, it enables data dispatchers to distinguish between modified and unmodified data which helps to significantly reduce the number of data operations and the amount of data dissemination.

The utilization of value objects is well established and the recommended way to abstract access to data sources. The data access manager encapsulates the complexity of the underlying data service layer implementation and provides clients with a regular interface for obtaining and modifying value objects. The client applications are completely unaware of the tasks performed by the data service layer and can be developed as usually. In other words, the reliance on a well established data access abstraction mechanism allows us to provide a transparent data service layer implementation with minimal impact on the business logic. Moreover, the characteristics of the transfer object assembler perfectly match with the requirements of the data access worker. This allows us to rely on a further core J2EE pattern to design a central component of the data service layer.

5.3.3 Deployment

The preceding sections provided an architectural overview and introduced the components of the data service layer. In addition, an introduction to the dissemination infrastructure and an explanation of the collaboration of the individual components is provided. The overall structure of the data service layer and the communication between edge servers and the backend system should now be evident. This section outlines configuration and deployment aspects of the data service layer.

The data service layer is designed as a conventional enterprise application that can be packaged and deployed in the same way as any other J2EE application. The primary steps involve the installation of the required libraries and the configuration of application server and message oriented middleware provider. The resources required to

deploy the data service layer include JDBC data sources, JDBC connection pools, JMS administrative objects, and JMS connection factories.

The configuration of the JDBC connection pool declares how to physically connect to the database. The set up is generally performed by installing an appropriate XA compliant [125] JDBC driver and configuring connection pool parameters with the administration interface of the application server in use. The JDBC data source, on the other hand, serves as an interface between the data service layer and the JDBC connection pool. The definition of JDBC data sources merely exposes JNDI names that can be obtained and ultimately used by the data service layer to access the RDBMS backend.

The administrative interface of the JMS provider allows the configuration of JMS administrative objects and JMS connection factories. It is crucial to assign JNDI names to the individual resources since it is the only way applications can access JMS administrative objects and JMS connection factories. The data service layer uses connection factories to create a connection to the MOM provider and relies upon administrative objects to send and receive messages to and from virtual destinations. When using a foreign JMS provider, as it is the case in the reference implementation, it has to be ensured that the JNDI names of JMS resources are accessible by the JNDI provider of the application server. This is usually assured by creating JNDI entries in the JNDI provider of the application server that are mapped to the bindings defined in the vendor specific JNDI provider of the JMS provider.

As discussed in the previous sections, the same data service layer implementation gets deployed at edge servers and the backend system. The deployment descriptors differ, however, to provide the individual components with an appropriate context that directs runtime behavior. The data service layer relies on environment entries which are available through the JNDI ENC and allow enterprise beans to customize their behavior by determining how they are deployed. The deployment descriptors of the data service layer provide each component, among other things, with the following information:

Node Name
: The node name provides components with information on the identity of the system. The data access worker declares the origin of notification groups with this value which allows the backend system to determine which edge server issued a remote operation.

Node Type
: The node type allows components to find out if they are deployed at one of the edge servers or at the backend system. It is the most crucial configuration parameter that influences the overall runtime behavior of the data access manager, data access worker, and application specific strategies components.

Strategies
: Several environment entries define the default strategies use by the application specific strategies component to determine dissemination and caching decisions. The data access manager allows client applications, however, to dynamically change caching and dissemination strategies at runtime.

The environment entries outlined above are utilized to configure various strategy [47], factory method [47], and template method [47] patterns that influence the overall behavior of the data service layer. More details on the design and implementation of

the individual components can be found in the following chapter. The deployment of the same data service layer implementation at edge servers and the backend system increases code reuse and consistence and diminishes development effort. In essence, major parts of the data service layer are identical for both environments purely the data access worker functionality varies significantly between edge server and backend system deployments.

5.3.4 Integration

Synchronous invocation of remote objects and server side components as well as transactional access to databases is common practice in enterprise applications. The seamless integration of asynchronous server side components and automatically managed transactional access to enterprise messaging systems is, however, something novel [117] in the world of enterprise computing. The data service layer relies heavily on the use of JMS as a means for data dissemination. This section explains the integration of the Java Message Service with application servers and analysis how message driven beans provide a robust and scalable environment for the concurrent processing of asynchronous messages. Moreover, the combination of application server and enterprise messaging system will be used to reveal details of distributed transaction management. The discussion of these low level facilities is intended as a clarification of some of the much lauded value added service. In addition, it aims to provide a better understanding of the infrastructure the data service layer depends upon.

Java Message Service Integration

The J2EE specification defines the connectivity between enterprise messaging systems and application servers which enables sending and receiving of asynchronous messages. The overall architecture of this interconnectivity is outlined in figure 5.4. The application server relies on a JMS connector to interact with the JMS broker and to coordinate the sending and receiving of asynchronous messages. The JMS connector instantiates several *message consumers* and *message producers* that correspond with the JMS broker to exchange messages. Message retrieval is handled by message consumers that are directly tied to message driven beans and merely hand over the message to the business container. Message dispatching, on the other hand, is initiated by session beans which are associated with a message producer to publish a message to the JMS provider.

The concurrent processing of asynchronous messages is, however, limited since each message producer is coupled with exactly one message driven bean instance. Thus message driven beans are directly dependent on individual message consumers and must wait for the next message to this particular consumer to continue processing. This lack of concurrency forces the application server to create both new message consumers and new message driven beans as load on the system increases. The attempt to increase scalability by creating new instances can, however, dramatically influence the overall performance of the application server. This is where *connection consumers* and *server session pools*, outlined in the middle of the diagram in figure 5.4, come into play. The use of server session pools provides message driven beans with a scalable mechanism for the concurrent process of asynchronous messages. The J2EE specification defines server session pools and connection consumers as an optional feature

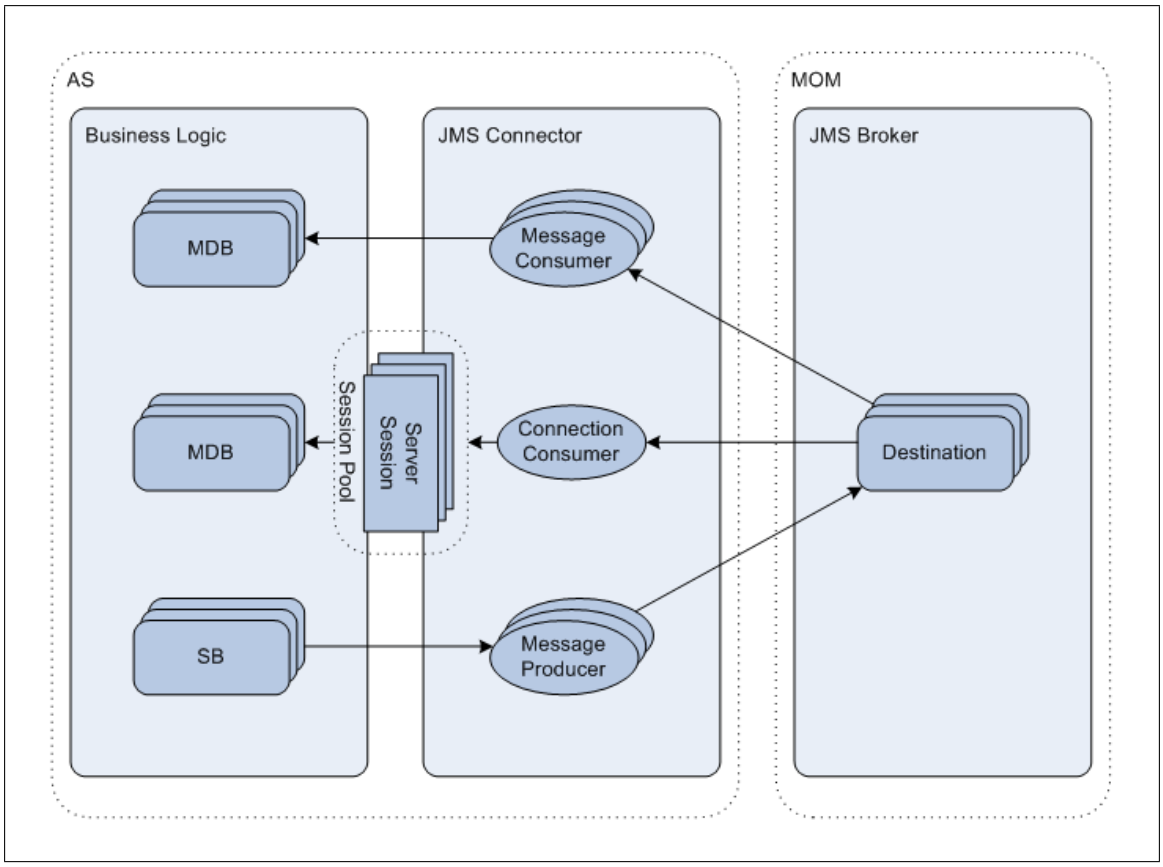

Figure 5.4: Java Message Service Integration

that might not be supported by all application server and message oriented middleware vendors. The technologies that support concurrent processing of messages as they arrive from the JMS broker are outlined in the following description.

Server Session Pool

The server session pool is implemented by the application server vendor and provides a mechanism to associate several server sessions with a single JMS session. The server session pool is actually a pool of threads for the parallel processing of messages received by connection consumers. Consequently, one or more message driven beans can incorporate with the server session pool and concurrently process messages received from the MOM provider.

Connection Consumers

The connection consumer, implemented by the message oriented middleware vendor, mediates between the server session pool and the JMS broker. It provides facilities that allow multiple server sessions to be bound to a single connection to the messaging infrastructure.

The server session pool provides concurrent message handling capabilities, a certain level of redundancy in case of server session failures, and the ability to fine tune the performance of application servers by dynamically adapting the number of server sessions. The connection consumer provides an efficient incorporation with the JMS

broker that minimizes thread context switching and resource utilization. The combination of server session pools and connection consumers can significantly increase performance of enterprise applications that rely heavily on the use of asynchronous messaging. The JMS connector serves as a good example how application servers relieve enterprise software developers from the implementation of resource pooling and failover mechanisms.

The data service layer benefits from the deployment of JMS in the sense that the data dispatcher implementation is more flexible and scalable. The data dispatcher at the backend system uses a session bean that merely delegates message dispatching to a message producer. The message producer encapsulates the complexity of communicating with the message broker and renders the data dispatcher completely independent of the actual network infrastructure. The data dispatcher at edge servers, on the other hand, relies on message driven beans for the concurrent processing of invalidation and data updates.

The asynchronous nature of JMS allows data dispatchers to send messages and immediately continue processing while the message is still in transit. In other words, the data dispatcher is free to accept new data operations and not required to wait until a message is handled by all remote systems. This concurrency improvement significantly diminishes the overhead introduced by the data service layer and consequently increases throughput [73] of data operations at the backend system. Moreover, by exploiting the guaranteed message delivery option of JMS it is sufficient to enlist the database update and message dispatching operation in a global transaction to assure transactional integrity. More details concerning the distributed transaction management required by the data service layer can be found in the following section.

Distributed Transaction Managment

The data service layer depends on a database and an enterprise messaging system to perform data operations and data dissemination. In order to guarantee transactional integrity, support for grouping of these enterprise resources into a distributed transaction is required. A *distributed transaction* spans multiple dispersed resources and therefore allows the completion of changes at one system to depend upon the changes at other systems. This enables the data service layer to treat processing of messages and database updates as a single unit of work, so that a failure to update the database or consume a message will cause the entire operation to fail.

Distributed systems usually rely on a *two phase commit* (2PC) protocol to define a contract with a transaction manager that takes care of coordinating the interactions of resources in a global transaction. The 2PC protocol is typically implemented using the XA [125] interface developed by The Open Group whereas the Java enterprise technologies mapping is provided by the Java Transaction API (JTA). In other words, every J2EE provider that implements the JTA XA APIs can therefore participate as a resource in a global transaction. The three essential components of a distributed transaction processing system are outlined in more details in the following description.

Resource Manager

A resource manager is a component or subsystem that provides access to shared resources or services and participates in distributed transactions using a 2PC

protocol, such as XA. Typical examples of resource managers in enterprise solutions include database management systems and message oriented middleware providers.

Transaction Manager
A transaction manager coordinates prepare, commit, or rollback of operations performed on all resource managers that participate in a global transaction of a given transaction domain.

Application Program
An application program defines operations that form part of a global transaction, such as updating a database and processing asynchronous messages.

An application program coordinates with the transaction manager to define transaction boundaries and communicates directly with the resource managers to access shared resources. It is, however, the responsibility of resource managers and the transaction manager to automatically exchange transactional details using the 2PC protocol. The transaction manager, ultimately, informs the application program of the current state of transactions. While this approach enables application programs to interface directly with resource managers it requires developers to work with the XA compliant version of the resource manager's API. Moreover, it requires application programs to enlist XA resources in global transaction and to handle prepare, commit, and rollback of operations. As a consequence of the distracting transaction handling code software solutions are rendered error prone and less portable.

Application server vendors provide remedy for transactional applications by integrating transaction manager capabilities into their enterprise offerings. A J2EE compliant application server uses the interfaces provided by JTA to coordinate distributed transactions among resource managers, the transaction manager, and the transactional business logic. In contrast to conventional applications the application server automatically enlists XA resources in transactions and with *container managed transactions* even manages prepare, commit, or rollback of operations. The transaction manager integrated into the application server uses the XA interfaces of resource adapters directly while application developers can communicate with the nontransactional version of the resource adapter APIs.

The distributed transaction management in an application server is illustrated in figure 5.5. In a typical scenario, the application server obtains a transactional resource (TR) object from a resource factory of the resource adapter. The resource adapter internally associates the transactional resource object with (1) an object that implements the resource adapter specific interface and (2) an object that implements the XA resource interface. Furthermore, the application server automatically enlists the XA resource to the transaction manager (TM) which associates the work performed by the resource manager with the current transaction assigned to the business logic. The important thing to understand is that after this initial setup the business logic interfaces with the nontranscational resource adapter API while the transaction manager communicates through the XA compliant API. Please also note that the XA resource interface is transparent to the business logic while the resource adapter specific interface is transparent to the transaction manager.

The data service layer directly utilizes the JMS API and JDBC API, as it would be the case in a non transactional scenario, to coordinate interactions with the JMS and

JDBC connection respectively. The XA resources associated with the JMS and JDBC connections are completely transparent to the data service layer. The transaction manager, on the other hand, relies on the JMS XA SPI and JDBC XA SPI to access the XA resources associated with the current connections of resource adapters. The connection based interface of resource adapters is transparent to the transaction manager. The application server is the only part that manages the transactional resource object that associates connection objects to XA resource objects. The application server encapsulates the complexity of enlisting XA resources to transaction managers and in the case of container managed transactions even manages the definition of transaction boundaries transparently.

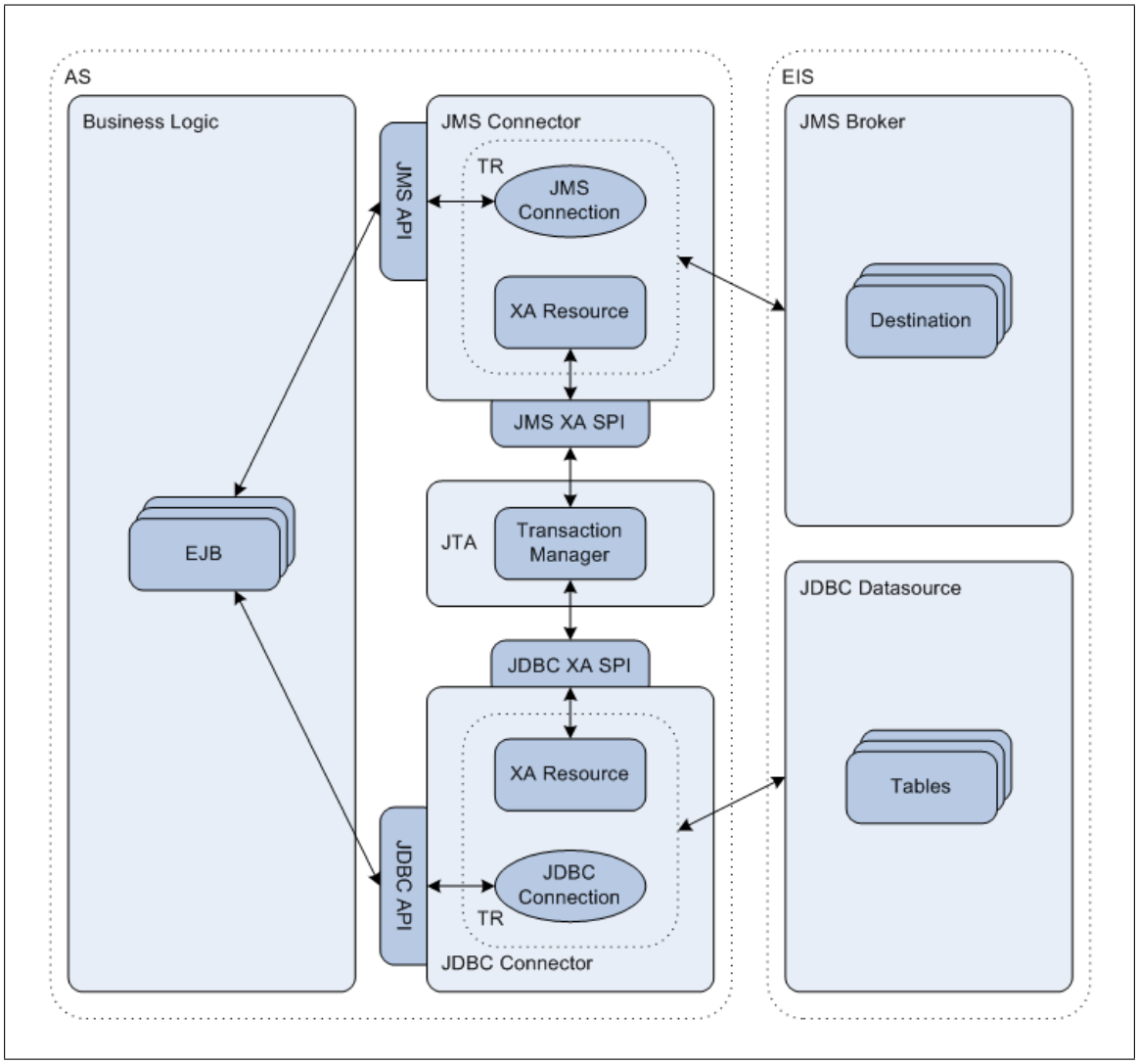

Figure 5.5: Distributed Transaction Management

The discussion above highlights that application servers provide a number of crucial value added services to the data service layer. Without the automatic management of distributed transactions and various other enterprise application services it would be significantly more difficult to implement the data service layer and to ensure a seamless integration into diverse application scenarios.

Chapter 6

Design and Implementation

The previous chapter introduced the architecture of the data service layer and discussed deployment and integration issues. This chapter explores the design and implementation of the reference implementation to effectively use the data service layer and to better understand the implications of the architectural proposal. The emphasis is on the overall design of the reference implementation as well as the discussion of control and data flow. The design depicted in this chapter aims to establish general guidelines and strategies for the implementation of the data service layer architecture. It might, however, be necessary to adapt various aspects of the implementation to deviating application scenarios.

6.1 Design Overview

The data service layer mediates between the business tier, the integration tier, and remote data service layer instances to fulfill data retrieval and storage requests. The partition into a dedicated layer is crucial to encapsulate the complexity of data dissemination and therefore facilitates the development of business objects. Without this separation of concerns application developers would be faced with the requirement to interweave business logic with distracting code for the handling of data dissemination. The resulting business solutions would be error prone, difficult to maintain, and almost impossible to reuse in other application scenarios. Introducing a surrogate for the data source can, however, have a considerable impact on the implementation of business objects. Consequently, every precaution has to be taken to guarantee a design of the data service layer that maximizes flexibility, extensibility, and efficiency. In addition, it is essential to provide application developers with a familiar and convenient environment for the design and implementation of business logic.

The design and implementation introduced in this chapter aims to fulfill these requirements by relying on well established blueprints that capture the preferred way to build enterprise applications for the Java 2 Enterprise Edition. The design of the data service layer combines several traditional design patterns [47] with the core J2EE patterns [9] published by Sun Microsystems to guide the development of flexible business solutions. The utilization of core J2EE patterns allows the data service layer to provide a familiar and convenient environment for the development of business logic while the combination with traditional design patterns further increases flexibility, extensibility, and efficiency.

Design patterns not only solve specific design problems and make object oriented software more flexible, elegant, and ultimately reusable. The use of design patterns improves documentation of software solutions by establishing a design vocabulary that provides an explicit specification of class and object interactions and their underlying intent. The utilization of design patterns helps to describe the design of the data service layer at a higher level of abstraction.

A successful deployment of the data service layer depends, apart from the design considerations outlined in section 4.2.1, on the flexibility, extensibility, and efficiency aspects outlined in the following description.

Flexibility
: The evolution of enterprise solutions is often characterized by changes to the underlying software products. The migration to different enterprise offerings may be necessary due to scalability problems, a lack of compatibility with other systems, licensing issues, and various other reasons. A comprehensive design of the data service layer should, therefore, be resistant against changes to the deployment environment. In addition, the data service layer is required to provide a maximum of flexibility for the configuration of dissemination, caching, and expiration logic.

Extensibility
: A successful enterprise solution frequently results into extensions to the data model as well as the expansion with additional enterprise systems. The design of the data service layer should, therefore, allow a seamless modification of the data model and ensure a straightforward addition of new edge servers and data sources. Furthermore, incorporating novel dissemination, caching, and expiration logic into the data service layer should be convenient and efficient.

Efficiency
: The data service layer, inevitably, introduces overhead to local data operations at edge servers and the backend system. It is crucial to devise a data service layer that provides adequate performance for local data operations in order to guarantee scalability at edge servers and the backend system.

The design aspects discussed above directly affect the development of business logic since the quality, portability, usability, and performance of data operations depends upon the design of the data service layer. The remainder of this chapter presents a design aimed at providing a high degree of flexibility, extensibility, and efficiency. A design pattern centric approach will be used to analyze design decisions and to verify the realization of design considerations.

The diagram in figure 6.1 shows the relationship of the main design patterns applied to the reference implementation of the data service layer. Further design patterns, such as iterator [47], adapter [47], and singleton [47], have been used but are not shown for the sake of clarity. The diagram addresses a recent concern of architects and designers who identified a lack of understanding of how to apply patterns in combination to form comprehensive solutions. The diagram emphasizes the fact that design patterns are not isolated artifacts but need to be incorporated into software solutions and combined with other patterns to realize design problems. The high level overview presented

in figure 6.1 can guide developers to better put the detailed description of individual design structures of the data service layer into perspective. The following discussion merely skims over the design of the data service layer implementation. A more detailed description of individual components can be found in the following sections.

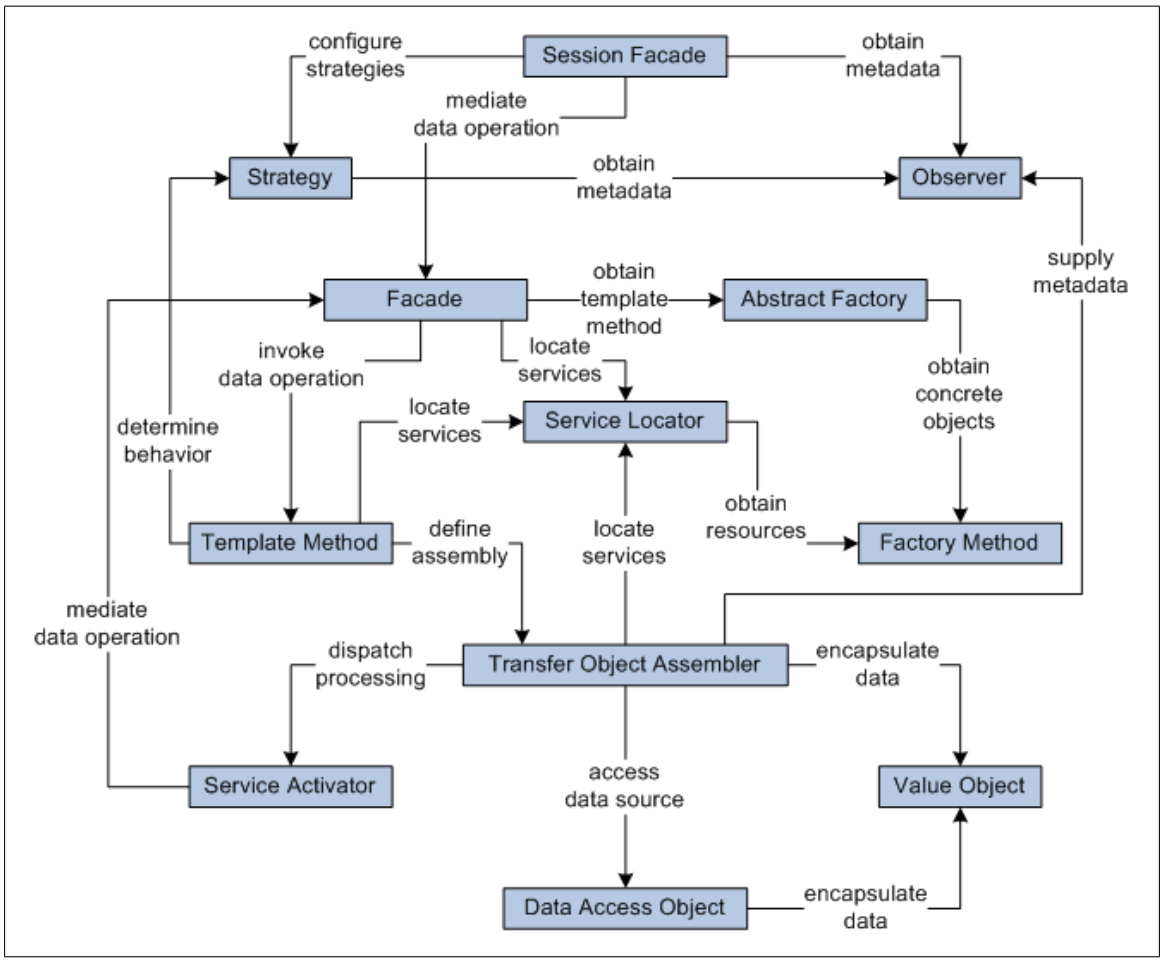

Figure 6.1: Design Pattern Relationship

The following description gives a brief overview of the duties and responsibilities of each design pattern in the data service layer implementation.

Session Façade

The session façade [9] design pattern servers as a uniform coarse grain interface to the data service layer. Business objects obtain a reference to the session façade to issue data retrieval and storage operations, configure application specific strategies, and to obtain metadata information of recent data operations.

Strategy

The strategy [47] design pattern is utilized for the implementation of caching, expiration, and dissemination algorithms. The structure of the strategy pattern provides well defined interfaces for a seamless extension with new algorithms and allows dynamically changing strategies at deploy time and runtime.

Observer
The observer [47] design pattern provides the infrastructure for provision and consumption of metadata. The data dispatcher component supplies metadata information on data operations to the observer pattern that can then influence caching and dissemination algorithms.

Façade
The façade [47] design pattern serves as a common interface for both local and remote data operations. The encapsulation of the complexity of the underlying data retrieval, storage, and dissemination tasks allows treating local and remote operations identically.

Abstract Factory
The abstract factory [47] design pattern determines the correct set of data operations for a given notification group. The pattern encapsulates the creation of concrete objects and therefore relieves the façade from the requirement to determine the correct template method implementations.

Service Locator
The service locator [9] design pattern provides a deployment environment independent access to resources and services. The abstraction introduced by the service locator eventually enables deploying the same data service layer implementation at all systems of the edge server architecture.

Template Method
The template method [47] design pattern defines common algorithms for data operations and data dissemination. The pattern facilitates centralized control of the data service layer behavior and at the same time increases code reuse.

Factory Method
The factory method [47] design pattern is used to instantiate service and resource objects base on requests from the service locator and to create concrete instances based on requests from the abstract factory.

Transfer Object Assembler
The heavily modified version of the transfer object assembler [9] design pattern carries out data operations and data dissemination. The transfer object assembler interacts with the local data source and remote data service layer instances to fulfill data storage and retrieval requests.

Service Activator
The service activator [9] design pattern is used to dispatch data operations synchronously and asynchronously to remote data service layer instances. Service activators encapsulate the complexity of communicating with remote data service layer instances.

Value Object
The value object [9] design pattern serves as an encapsulation of data obtained from the data source. The business objects rely on value objects as a coarse grain data access mechanism.

Data Access Object

The data access object [9] design pattern is utilized to abstract access to the underlying data source. The data access object pattern transfers value objects between the transfer object assembler and the locally available data source.

An overview of the activities performed by the data service layer in collaboration with the business logic is outlined in figure 6.2. Please note that the activities of the common services component are not included in the diagram for the sake of clarity. A detailed discussion of the duties and responsibilities of the individual components as well as an in depth analysis of the execution phase is provided in the following sections.

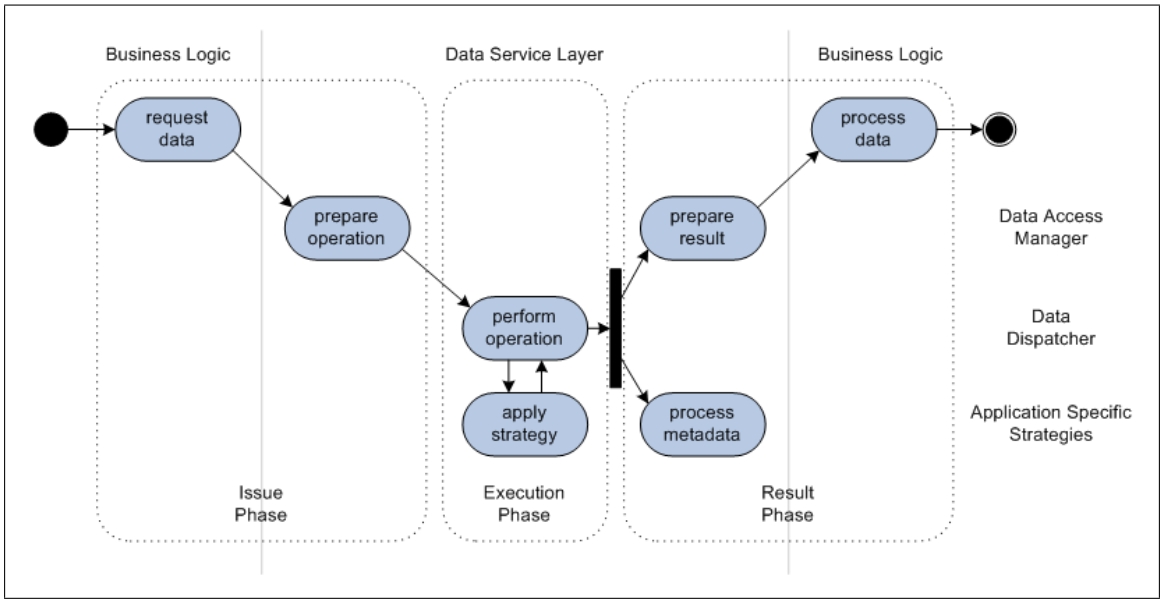

Figure 6.2: Activity Overview

6.2 Common Services

The reference implementation of the common services component provides general purpose services and resources that are used throughout the data service layer implementation. The component defines exceptions, provides an infrastructure for lifecycle management, implements a service locator and the corresponding resource factories, enables access to project constants, specifies several enumeration classes, and accumulates value objects. The common services component consists entirely of plain old Java objects (POJO) that are designed to execute in the context of EJBs implemented in other components of the data service layer. The discussion in this section focuses on the implementation of the service locator and the lifecycle management infrastructure. The implementation of exceptions, the access to project constants and property files, and the creation of enumeration classes is not outlined and assumed to be evident. The importance of value objects in the context of the data service layer, on the other hand, has already been discussed in section 5.3.2.

6.2.1 Service Locator

The data service layer relies upon several local and remote EJBs, accesses JDBC data sources, requires access to JMS connection factories, utilizes JMS administrative objects, and obtains information from environment entries. These components are, however, not directly accessible since the application model of the J2EE requires the use of JNDI to obtain references to services and resources. As a result, accessing a service component usually involves obtaining an initial context from the context factory of the JNDI API that can then be used to lookup the required object and retrieve a reference to the service or resource.

While the use of JNDI provides a flexible and unified access to all enterprise services and resources it comes with several drawbacks and limitations. The context factory used for the creation of the initial context is provided by the service provider vendor and therefore results into a vendor dependent process for the lookup and creation of service components. Moreover, with multiple service providers, as it might be the case with the combination of application servers and message oriented middleware providers, the context factory might even be dependent on the type of object being looked up. Furthermore, lookup and creation of services and resources could be complex and may be required repeatedly throughout a J2EE project. In addition, the initial context creation can be resource intensive and may impact the overall performance of the application.

The utilization of a service locator object can hide the complexity of initial context creation and lookup operations. Moreover, multiple clients can leverage the service locator to reduce code complexity, provide a single point of control, and improve performance by providing a caching mechanism. The service locator pattern abstracts access to services and resources and therefore renders J2EE applications independent of the lookup process. This allows us to combine the service locator functionality with a mechanism that enables the same data service layer implementation to be deployed at all systems of the edge server architecture.

The common services component provides an implementation of the service locator pattern that allows deployment specific service and resource lookups. The service locator implementation dynamically generates the JNDI names depending on information provided in the deployment descriptors. This mechanism enables the same request to the service locator to result into a different lookup operation at edge servers and the backend system. This context aware service locator implementation allows deploying a single data service layer implementation at all systems of the edge server architecture without modifications to the program code.

There are basically two different service locator implementations one for local and one for remote service and resource lookups. The local service locator implementation relies on the JNDI ENC to resolve lookup names while the remote service locator is used by client applications and to access remote data service layer instances. The service locator implementation of the common services component builds upon several factories that encapsulate the actual lookup and creation of the service and resource objects. The common services component provides factories for the creation of JDBC data sources, EJB home interfaces, JMS connection factories, JMS administrative objects, and environment entries. The use of a service locator object not only allows deploying the same implementation at all systems of the edge server architecture, it makes the data service layer resistant against changes to the deployment environment.

6.2.2 Lifecycle Management

The conscientious reference implementation of the data service layer introduces a periodically scheduled task that handles cache expiration. This expiration task is executed at specified intervals to query the cache at edge servers for expired data items. The task relies on an expiration algorithm, defined in the application specific strategies component, to determine which data items are expired and ultimately deletes them from the cache. This mechanism ensures that the cache at edge servers does not grow infinitely as more and more items get cached at the edge of the internet.

The realization of time based business logic execution comes, however, with several technical obstacles. The implementation of the expiration task requires a mechanism that allows a periodic execution of tasks within an application server and a way to initiate scheduling automatically after the data service layer is deployed. While both requirements may appear elementary the J2EE version 1.3 offers no standardized technique to utilize a timer service. The application model is inherent request based and requires a client application to initiate processing with a service inquiry. The EJB 2.1 specification [37] included in the upcoming J2EE version 1.4 [105] solves this shortcoming by defining a container managed timer notification service. This EJB Timer Service is, unfortunately, not available to the reference implementation of the data service layer which relies on a EJB 1.3 compliant application server.

The reference implementation of the data service layer overcomes the lack of a standardized timer service for EJBs by utilizing the Java Management Extension (JMX). The JMX API provides a timer service implementation that allows POJOs to register for timer callbacks to occur at a specified time, after a specified elapse time, or at a specified interval. The application specific strategies component contains several Java classes that utilize the JMX API to register a notification listener that executes the expiration task at a specified interval. The open issue is how to register the notification listener and start the time service upon deployment of the data service layer. At the time of this writing there exists, unfortunately, no standardized way to perform operations upon deployment events. The application server used for the reference implementation provides the following options to initiate processing upon deployment of the data service layer.

Application Lifecycle Listener
: An application lifecycle listener can be registered, in a proprietary way, to receive application lifecycle events from the application server. This mechanism allows developers to perform operations during deployment, undeployment, and redeployment of enterprise applications.

Startup Servlet
: The standard deployment descriptor of web applications can be used to declare a startup servlet that is initialized automatically when the application server starts.

Startup Session Bean
: The proprietary application server specific deployment descriptor of enterprise beans can be used to declare a startup session bean that is initialized automatically when the application server starts.

While all three options serve the same purpose each of them comes with different limitations and drawbacks. The startup session bean relies on a proprietary deployment descriptor element and therefore limits portability to other application servers. The startup servlet, on the other hand, provides a standardized approach but requires the presents of a web application which is not suitable for an intermediate component such as the data service layer. The application lifecycle listener, eventually, depends upon a proprietary way to register POJOs for lifecycle callbacks. Since most application server vendors provide a mechanism to register objects for lifecycle events and the startup class merely initializes the timer service of the expiration task it should be straightforward to port the startup class to deviating enterprise offerings. As a result, the reference implementation utilizes an application lifecycle listener to schedule the expiration task upon deployment of the data service layer.

The application lifecycle listener implemented in the common services component registers the expiration task for periodical execution upon deployment of the data service layer. As a result, the JMX timer service starts the expiration task at a specified interval during the lifetime of the data service layer. The expiration task accesses the data source and deletes expired data items according to expiration logic defined in the application specific strategies component.

Unfortunately, there is no great solution when a startup class requires access to EJBs as it is the case in the data service layer implementation. The problem stems from the hierarchical classloader architecture inherent in Java and the way parts of an enterprise application are loaded by the application server. In general, both the application server and startup classes are loaded by the system classloader while enterprise applications have their own context classloaders. Startup classes are therefore unable to directly access EJBs since the required interfaces can not be located by the system classloader. It is possible to include the EJB interfaces in the classpath of the system classloader but this prevents the hot deploy capabilities of application servers.

A better solution in the context of the data service layer is the use of shared interfaces for interclassloader communication. The common services component utilizes the `Runnable` interface provided by the J2SE as the public view of dynamically loaded classes that are not directly accessible to the classloader of the startup class. The application specific strategies component provides two classes that implement the `Runnable` interface to start and stop the timer service. The startup class dynamically loads these classes, instantiates them, casts them to the `Runnable` interface, and ultimately triggers processing within the scope of the enterprise application classloader. While the `Runnable` interface provides just one method without parameters it is sufficient to perform basic operations such as starting and stopping the timer service of the expiration task upon deployment events of the data service layer.

6.3 Application Specific Strategies

The overall performance of the edge server architecture depends primarily on the efficiency of the data service layer which is determined by the data dissemination and caching logic. It is therefore crucial to provide a flexible and extensible infrastructure for caching, dissemination, and expiration logic. The algorithms should be interchangeable at deployment time and at runtime while the development of new policies

should be convenient. The business logic should control the usage of algorithms but, nevertheless, stay independent of the complex algorithm details and data structures.

The reference implementation of the application specific strategies component aims to provide a flexible and extensible infrastructure for the development and deployment of decision logic by relying on the strategy design pattern. The data service layer at the backend system uses a dissemination strategy while edge servers depend on caching and expiration strategies. The strategy pattern allows the definition of a set of interchangeable algorithms that may vary independently from clients that use it. The individual algorithms are encapsulated and independent from each other and can therefore be developed concurrently and under different perspectives.

The strategy pattern streamlines the definition of new algorithms and enables dynamically adapting dissemination, caching, and expiration policies when appropriated. The reference implementation of the data service layer provides two mechanisms for selecting dissemination, caching, and expiration algorithms. The algorithms can either be set at deployment time by declaring an environment entry in the deployment descriptors or at runtime by utilizing the interfaces provided by the data access manager component. The expiration strategy is used by the periodically executing expiration task at edge servers. The dissemination and caching strategies, on the other hand, are accessed by the data dispatcher to determine if a given data item has to be cached or disseminated.

The strategy pattern consists of (1) a strategy class that defines a common interface to all supported algorithms, (2) several concrete strategy classes that implement the algorithms using the strategy interface, and (3) a context that is configured with a concrete strategy object and utilizes the strategy interface to execute the concrete algorithm. During the design of the strategy pattern it is essential to declare an efficient contract between the strategy and the context. This contract has to provide a mechanism that allows exchanging any data needed to perform the algorithm steps and to return the result. A context may pass all data required by the algorithm to the strategy or alternatively pass itself as an argument and direct the strategy to request data explicitly. The reference implementation of the application specific strategies component relies on the former approach. The data dispatcher passes the notification group of the current data retrieval or storage operation to the strategy. The concrete algorithm implementation will then extract information on the type of operations and the data items in order to determine dissemination or caching decisions.

Apart from the strategy patterns, the application specific strategies component provides an implementation of the expiration task. As discussed in section 6.2.2 it is necessary to implement a notification listener that can be registered to the JMX timer service. The registration of the notification listener is achieved by POJOs that implement the `Runnable` interface, are dynamically loaded by the startup class, and initiate the registration and deregistration process.

6.4 Data Access Manager

The data access manager component of the reference implementation defines a coarse grain data access interface that is used by business objects to perform data storage and retrieval operations. The interface can in addition be used to configure the dissemina-

tion, caching, and expiration strategies at runtime and to access metadata of recent data operations. The data access manager component implements the session façade design pattern to encapsulate the complexity of the data service layer implementation. The stateful session façade strategy has been chosen since several business processes are expected to require multiple method calls to complete. The conversational state is automatically saved by the stateful session bean and can eventually include the strategy configuration and the metadata information.

The reference implementation of the data access manager utilizes the session façade merely as the entry point to the data service layer. As outlined in figure 6.1 data operations are mediated to a further façade implemented in the data dispatcher component. The data access manager performs the critical conversion of regular value object based data operations to notification groups. This process converts client requests to the internal representation used by the data dispatcher component to direct access to the local database and to coordinate data dissemination.

When a new session bean, representing the session façade, is initialized it determines the default caching or dissemination algorithm by accessing the environment entries of the deployment descriptor. The initialization process then configures strategies of the application specific strategies component with a concrete algorithm implementation. The interface provided by the session façade can, however, be utilized to overwrite this default configuration and to dynamically change caching and dissemination algorithms when appropriated.

The data access manager is in addition responsible to provide an infrastructure for the provision and consumption of metadata. Metadata is available for each data operation and contains information on the actions performed by the data service layer to fulfill a data retrieval or storage request. The information reveals, among other things, if a given data operation resulted into a remote operation, if data items have been cached or disseminated, and if an invalidation message has been sent. The reference implementation of the data access manager utilizes the observer design pattern to provide a flexible mechanism for metadata propagation.

The observer pattern defines a one-to-many dependency between objects so that all participants are notified and updated when one object changes state. The structure of the observer pattern allows a collection of cooperating classes to maintain consistency without the requirement for tightly coupling. As a result, any number of objects in the data service layer implementation can declare an interest in metadata provided by the data dispatcher. The metadata can be used by caching, dissemination, and expiration algorithms to generate statistics on the behavior of the data dispatcher component. This information can be used to tweak the algorithm behavior at runtime or may be stored for manual analysis and revision of algorithms. Since usage can change over time, the session façade may analyze the metadata and dynamically replace dissemination, caching, or expiration algorithms with more appropriate ones during the lifetime of the data service layer. The observer pattern enables a seamless integration of new algorithms that depend on access to metadata information. The algorithm merely announces an interest to the observer without the requirement for changes to any part of the data service layer. Moreover, the decoupled metadata exchange allows any number of algorithms to attach to the observer.

6.5 Data Dispatcher

The data dispatcher is the central component of the data service layer which coordinates access to the local data source and communication with remote data service layer instances. The overall structure of the data dispatcher component has already been outlined in section 5.3.2. The focus in this section is on a discussion of the tasks performed by the data access worker subcomponent. A generalized view of the most essential classes implemented in the data access worker is provided in figure 6.3.

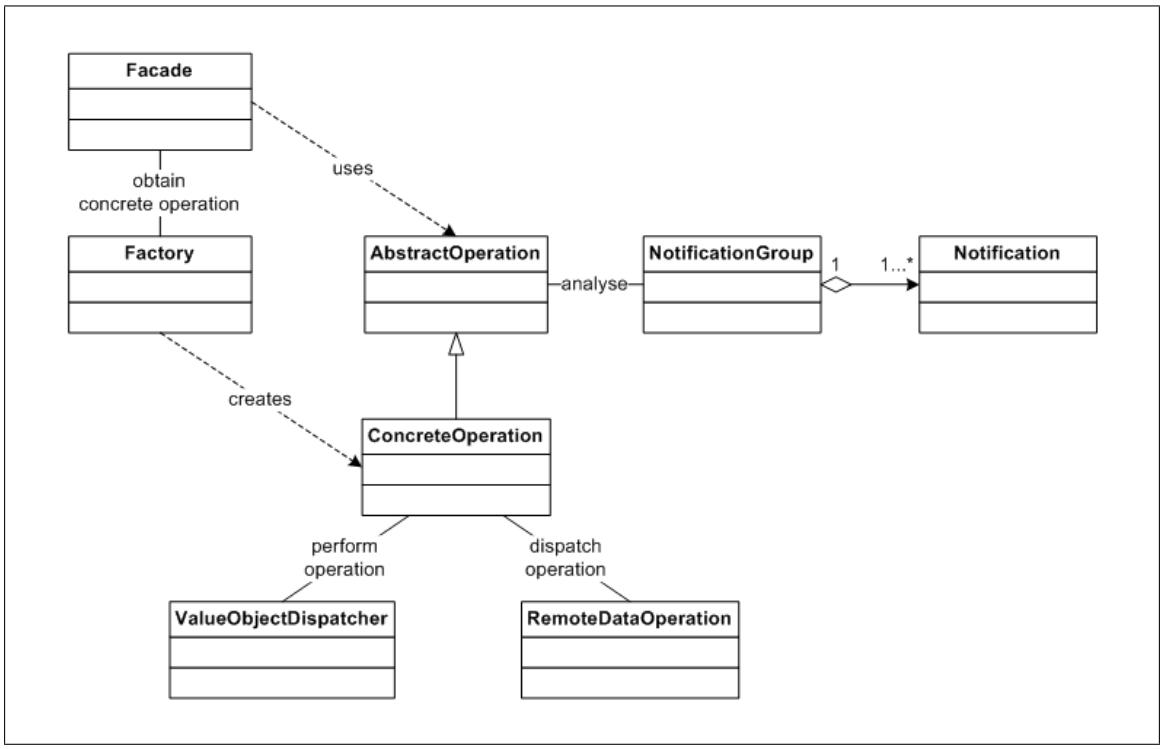

Figure 6.3: Data Access Worker Structure

The following discussion explains the duties and responsibilities of the artifacts outlined in figure 6.3.

Facade
: The Facade provides a coarse grain interface to the data access worker which is utilized by the data access manager and remote data service layer instances to request accomplishment of data operations. The requests are sent in form of NotificationGroup objects that encapsulate one or more data operations.

Factory
: The Factory is used by the Facade to obtain a reference to a ConcreteOperation object for a given NotificationGroup. The Factory analyses the NotificationGroup to determine which ConcreteOperation is able to perform the requested data operations.

AbstractOperation
: The AbstractOperation defines a data independent skeleton of an algorithm for

the accomplishment of data operations. The AbstractOperation separates the overall behavior of the data access worker from the actual data retrieval and storage details.

ConcreteOperation
The ConcreteOperation objects perform the actual work which is directed by the AbstractOperation and defined in the NotificationGroup object.

NotificationGroup
The NotificationGroup consists of one or more Notification objects as well as administrative information required by the data access worker to coordinate data access and dissemination.

Notification
A Notification object contains information on a particular operation and the corresponding data to perform the operation.

ValueObjectDispatcher
The ValueObjectDispatcher [56, 57] associates Notifications with data access objects to actually carry out the data operation on the local database.

RemoteDataOperation
The RemoteDataOperation issues remote operations by dispatching NotificationGroup objects to remote data service layer instances.

The important thing to understand is that data retrieval and storage requests issued by the business logic do not directly result into method calls to data access objects. The internal encapsulation of data operations in NotificationGroup objects allows the data service layer to transparently perform data operations either locally or remotely. The ConcreteOperation objects eventually determine if a local operation is sufficient or if a remote operation is required. The actual behavior depends on the type of operation and the entities operated on. The translation of Notification objects to data access object method calls is performed by the ValueObjectDispatcher. The Notification contains enough information to instantiate the correct data access object and to call the right method. The reference implementation makes use of the Reflection API [25] to translate Notification objects to concrete operations performed on data access objects.

The template method design pattern is applied to define the correlation between the AbstractOperation class and ConcreteOperation classes. As a result, all tasks performed by ConcreteOperation objects are directed by the common algorithm defined in the AbstractOperation class. The use of the template method pattern allows us to control the overall behavior of the data access worker in a universal way. The common algorithm ultimately streamlines the addition and definition of new ConcreteOperation objects that merely implement the variant parts of the algorithm and carry out the actual data storage and retrieval operations. The AbstractOperation class analyses the NotificationGroup object to determine the algorithm steps required for a particular data request. The algorithm eventually applies the caching and dissemination strategies to the NotificationGroup, analyses the result, and directs data operations, caching, and dissemination accordingly. The combination of AbstractOperation and ConcreteOperation classes roughly reassembles the functionality of the transfer object

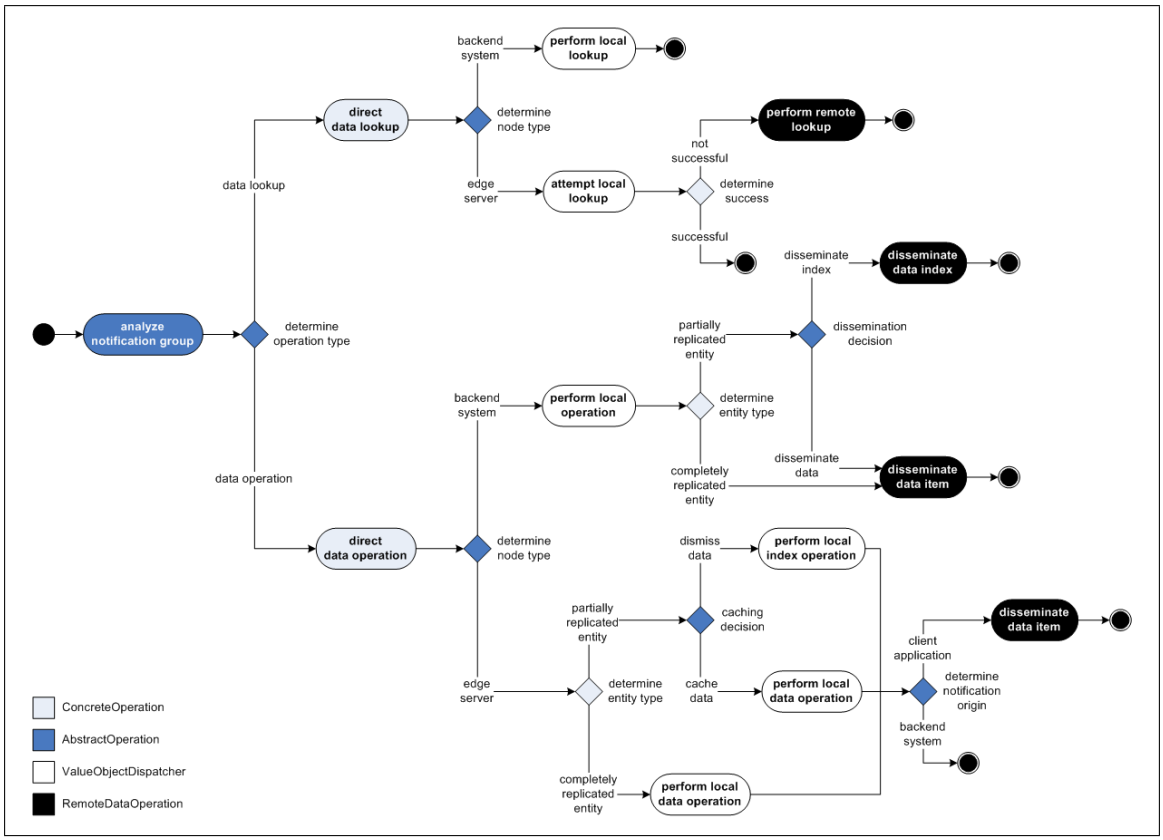

Figure 6.4: Execution Phase Activities

assembler design pattern. The template method directs the assembly of NotificationGroup objects upon data retrieval and storage requests as needed.

A detailed synopsis of the activities performed by the individual parts of the data access worker is provided in the activity diagram in figure 6.4. The diagram depicts the execution phase introduced in the activity overview diagram in figure 6.2. Please note that object flow and processing of metadata information is not sketched in the diagram for the sake of clarity. The initial state is triggered by the façade of the data access worker which in turn is either accessed by the data access manager for local data operations or by the service activator for remote data operations.

The utilization of the template method design pattern is liable for the alternating influence of AbstractOperation and ConcreteOperation on the overall control flow. The diagram further illustrates that the combination of AbstractOperation and ConcreteOperation eventually directs the work performed by the data access worker. The AbstractOperation determines processing steps by analyzing the NotificationGroup object while the ConcreteOperation inspects the data dependent and entity specific Notification objects. The AbstractOperation class is therefore defining the overall control flow while the ConcreteOperation objects declare the data specific workflow. Access to the local database and data dissemination, on the other hand, is carried out by the ValueObjectDispatcher and RemoteDataOperation objects respectively.

Chapter 7

Evaluation and Comparison

The data service layer presented in the preceding chapters describes a general concept for the extension of enterprise applications with valorized edge servers. Architecture, design, and implementation are intentionally discussed in an abstract, application scenario independent, way to facilitate the integration into diverse enterprise solutions. The focus is on the establishment of universal guidelines and strategies for the decomposition of enterprise systems rather than the discussion of implementation details of the reference implementation. This chapter aims to present the data service layer in two varied application scenarios, namely a digital library and a distributed workflow management system. The former represents the application scenario chosen for the reference implementation while the latter corresponds to current research activities.

7.1 Application Scenarios

The digital library application scenario, discussed in section 7.1.1, has been implemented by the reference implementation of the data service layer. The typical usage patterns and the obvious partitioning of index and data render digital libraries an ideal environment for the deployment of the data service layer. A distributed workflow application is the second integration scenario discussed in section 7.1.2. While the deployment in distributed workflow management systems is intriguing it raises a number of issues and significantly constrains the type of tasks that can be performed at edge servers. The discussion of two diverse application scenarios is intended to highlight the impact and consequences of the deployment of the data service layer.

7.1.1 Digital Library

The reference implementation of the data service layer is a prototype for an efficient worldwide digital library. Several edge servers are distributed to diverse geographical locations to serve client requests at the edge of the internet. The functionality of the reference implementation is similar to the services provided by the ACM Digital Library [3] and the NEC Scientific Literature Library [89]. Users can query the digital library with various search criteria to find and ultimately download research papers and journal articles. The design goal of the digital library prototype is a significant increase of query performance in terms of execution time and response time. In other

words, we want to move query processing directly to the edge of the internet while document downloads may result into data transfer from the backend system.

Implementation

The deployment of the data service layer allows us to flexibly combine complete data replication with partial data replication. The partitioning into completely and partially replicated data is obvious in the digital library scenario. Data required for query processing is replicated to edge servers while the rest remains on the backend system and is disseminated on demand. The prototype implementation utilizes partial replication for paper and article documents. The completely replicated data index, on the other hand, consists of author and category information as well as title, description, abstract, and keywords of papers and articles. The simplified database scheme used for the prototype implementation of the digital library is shown in figure 7.1.

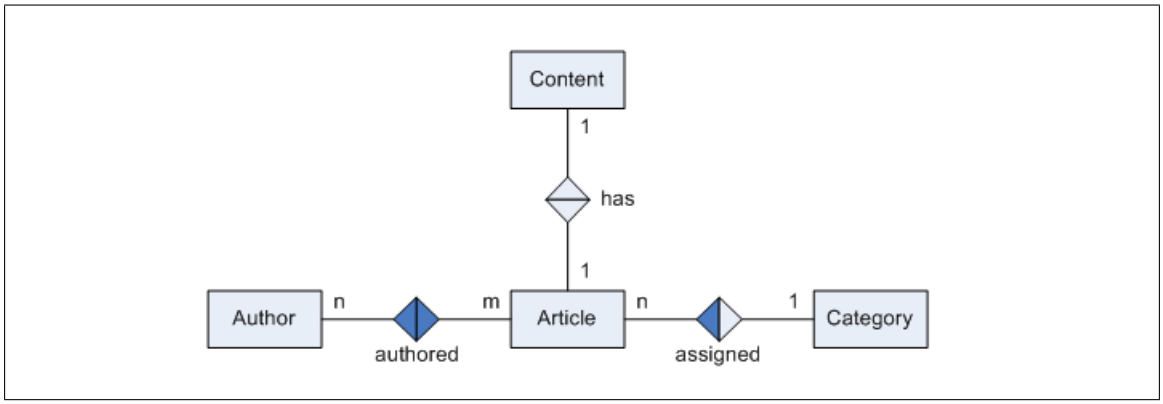

Figure 7.1: Digital Library Database Scheme

The Author, Article, and Category tables are replicated completely while the Content table is partially replicated to edge servers. Please note that the metadata tables maintained by the data service layer are not shown for the sake of clarity. The following description provides a brief overview of the individual tables.

Author
: The Author table contains information on authors who published a research paper or journal article stored in the digital library.

Article
: The Article table contains title, description, abstract, and keywords of all research papers and journal articles stored in the digital library.

Category
: The Category table contains information on categories used to organize research papers and articles stored in the digital library.

Content
: The Content table stores the research paper and journal article documents directly in the database by utilizing a binary large object (BLOB) field.

The design of the data service layer is capable of replicating individual columns of a database table while data of the remaining columns is subject to dissemination on demand. The Article and Content tables of the database scheme shown in figure 7.1 could therefore be merged into a single database table. The separation results from a design decision of the database scheme rather than a restriction of the data service layer. The database scheme of the digital library prototype further illustrates that the type of data that is partly disseminated is completely dependent on the application scenario. To enable convenient query processing at edge servers in the digital library application scenario it is necessary to replicate the information contained in the Author, Article, and Category tables. The documents stored in the Content table, on the other hand, can be transferred on demand when appropriated.

The distribution of edge servers allows presentation logic, query processing, browsing, and searching of the digital library to occur directly at the edge of the internet. The resulting distribution of database inquiries to multiple edge servers combined with the reduction of communication latency can significantly increase performance of typical user requests. The download of documents, on the other hand, may result into transit traffic if the documents are not already cached at edge servers. The deployment of the data service layer improves the performance of inquiries to the digital library in terms of execution time and response time. Since digital libraries typically exhibit a large number of browsing and searching requests and only a moderate number of downloads we can achieve an improvement of a majority of user interactions. Moreover, advance of performance is assured for the interactive process of finding an appropriate research paper or journal article rather than the mainly automated process of downloading the documents determined. Thus response time and execution time is increased for the type of requests that are most crucial to the users of digital libraries.

The size of the Content table at the backend system can be assumed to be several orders of magnitude larger than the information stored in the Author, Article, and Category tables. Dissemination on demand as introduced by the deployment of the data service layer helps to reduce storage requirements at edge servers and diminishes the amount of information transferred across wide area networks. The edge servers can be assumed to merely cache the most frequently accessed documents while relying on the backend system to deliver the remaining research papers and journal articles. Nevertheless, edge servers are capable of fulfilling inquiries for documents according to author name, article title, article description, article keywords, and categories information. Merely a full text search of research papers and journal articles requires business task delegation to the backend system.

The design of the data service layer does not constraint the utilization of the persistence mechanisms provided by the J2EE. The prototype implementation realizes object to relational mapping with conventional container managed persistence (CMP) entity beans while more sophisticated inquiries are implemented with JDBC. The database relationships are mapped to the object representation via container managed relationship (CMR) fields. The caching and dissemination functionality of the data service layer is implemented on top of the built in persistence mechanisms. Caching and dissemination decisions are determined by the entity specific implementation of the variant parts of the algorithm defined in the AbstractOperation template class (see section 6.5 for more details).

Interactions

The diagram in figure 7.2 depicts a typical usage scenario for the reference implementation of the digital library. The infrastructure consists of a backend system (dl.com) and four edge servers (america.dl.com, asia.dl.com, australia.dl.com, and europe.dl.com) which handle user requests of their geographical region respectively.

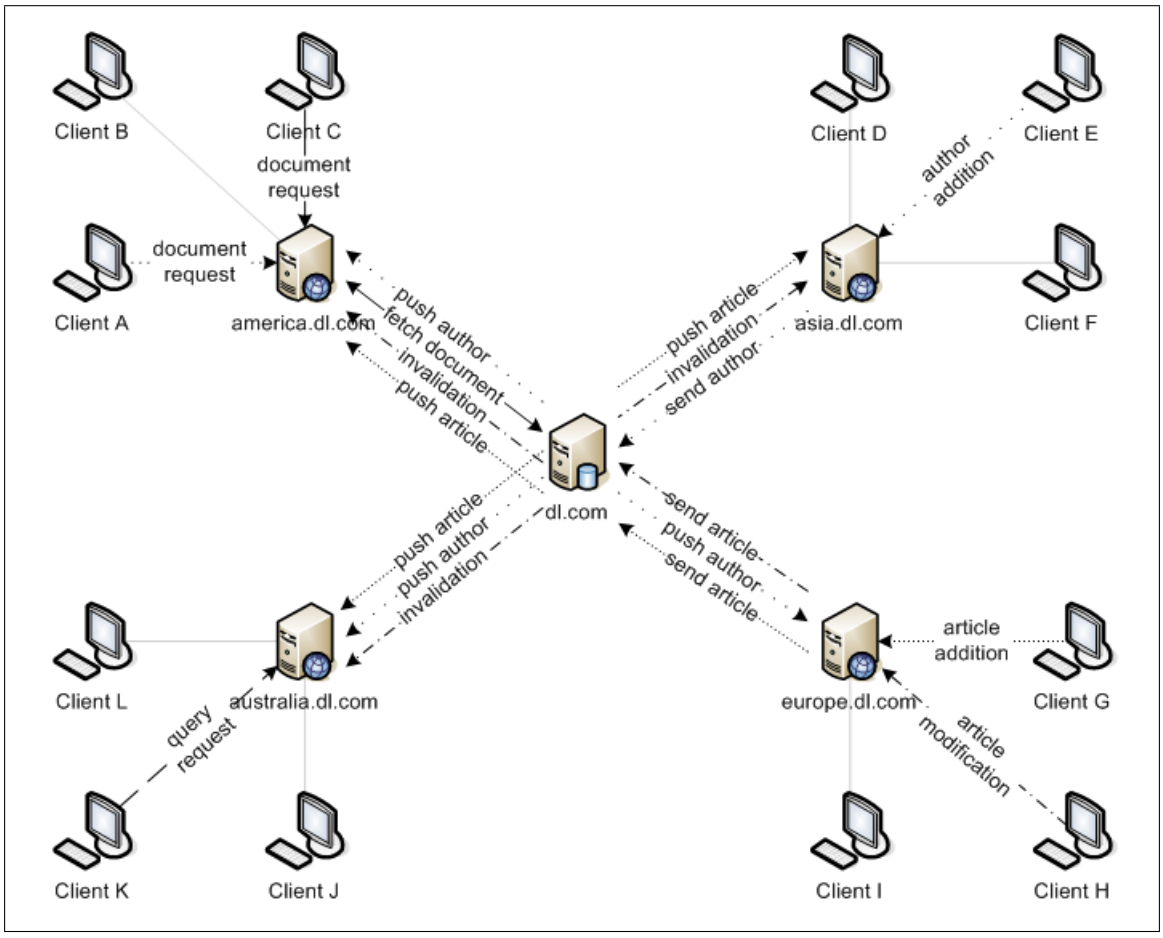

Figure 7.2: Digital Library Interactions

The following description outlines the use cases illustrated in figure 7.2 and discusses possible deviations which may result from different caching and dissemination decisions at both the backend system and at edge servers.

Query Request

The query request initiated by an Australian client (Client K) is directly processed by the Australian edge server (australia.dl.com) without the requirement to interact with the backend system. In essence, all inquiries can be processed by edge servers except for full text searches of documents which have to be redirected to the backend system.

Document Request

Two document request scenarios are shown for American customers (Client A and Client C). The request of Client A is directly fulfilled by the American edge

server (america.dl.com) while the request of Client C results into a cache miss and requires the edge server to fetch the document from the backend system.

Author Addition
The addition of an author is performed by an Asian client (Client E) in collaboration with the Asian edge server (asia.dl.com). The author information is sent to the backend system and pushed to all edge servers since complete replication is configured for author entities.

Article Modification
A European customer (Client H) modifies an article at the European edge server (europe.dl.com). The caching logic of the edge server decides if the modified document is cached and eventually propagates the article information to the backend system. The dissemination logic at the origin site ultimately comes to the conclusion that the modified article does not need to be pushed to all edge servers. The backend system sends an invalidation which enables edge servers to delete the cached article if necessary.

Article Addition
A European customer (Client G) adds an article at the European edge server (europe.dl.com). The caching logic of the edge server decides if the newly added document is cached and eventually propagates the article information to the backend system. The dissemination logic at the origin site ultimately comes to the decision that the newly added article should be disseminated proactively to all edge servers. The edge servers receive the pushed article and caching logic at each individual server determines if the article is going to be stored in the local cache.

The use cases illustrated in figure 7.2 and describe in the enumeration above highlight operation which:

- depend upon completely replicated data
- utilize partially replicated data
- benefit significantly from the edge server architecture
- profit moderately from the edge server architecture

Searching and browsing the digital library is significantly improved by the edge server architecture since expensive round trips to the origin site can be avoided. The full text search of documents, on the other hand, results into business task delegation to the backend system. Nevertheless, full text searches can be accelerated slightly because presentation of the query results is carried out directly at the edge of the internet. The impact of the edge server architecture on document requests mainly depends upon the efficiency of the caching algorithm. If the request for a particular document results into a cache miss it has to be transferred from the backend system. The data service layer may help to achieve a slight performance enhancement by utilizing a permanent connection between the edge servers and the backend system and by compressing the documents transferred. The creation, modification, and deletion of completely replicated entities results into a considerable overhead since changes have to be propagated to all edge servers.

The flexibility of the data service layer is most apparent for operations on partially replicated data. The creation and modification of articles outlined above illustrates that storage of data items at edge servers depends on dissemination decisions of the backend system and caching decisions of edge servers. By leveraging context information of the backend system and each edge server it is possible to fine tune the placement of data items. The distribution of caching and dissemination algorithms to all systems of the edge server architecture can significantly diminish network traffic and storage requirements at edge servers.

The deployment of the data service layer allows query processing, dynamic content delivery, and dynamic content assembly to occur at the edge of the internet. Moreover, the data service layer enables modifications and additions to the digital library to arise from the collaboration of clients with edge servers. In addition, please note that client applications are note required to access the web tier of edge servers. Business participants may directly communicate with the business tier when appropriate.

7.1.2 Distributed Workflow

The increasing number of collaborations performed among dispersed business partners poses new challenges for traditional centralized workflow management systems. Several research papers [29, 102, 124] have therefore evaluated the distribution of workflow over a number of systems and organizations. This section analyses the application of the data service layer for the efficient distribution of workflow data in distributed workflow management systems. The focus of our research is on the efficient distribution of task specific data to enable the processing of frequently executed workflow tasks directly at the edge of the internet. We assume the reader is familiar with workflow automation but may want to inspect a current research paper [111] which provides an extensive overview and outlines current technical, management, and organizational research issues.

The individual tasks of a decentralized or collaborative workflow are not executed by a centralized workflow engine, but by multiple engines collaboratively. The mechanism which coordinates the distributed workflow execution will be referred to as *collaborative process management* [29] while the distributed workflow itself is usually called *collaborative business process* [29]. One of the key issues in collaborative process management is the distribution of workflow data to all participating workflow engines. Our focus is therefore on the informational perspective of distributed workflow management systems. In other words, we research into the application of the data service layer for the efficient distribution of task specific data.

A successful deployment of the data service layer in distributed workflow management systems comes with several prerequisites. The *collaborative process definition* [29] has to be extended with additional metadata information to enable the implementation of efficient strategies for the dissemination of task specific data. In particular, precise data requirements and predicted execution frequency of individual workflow tasks have to be provided. With all this information in place it is finally possible to apply complete data replication merely to frequently executed tasks and partial data replication to infrequent tasks. The overall data replication overhead can therefore be reduced significantly since data for infrequent tasks can be transferred on demand when appropriate.

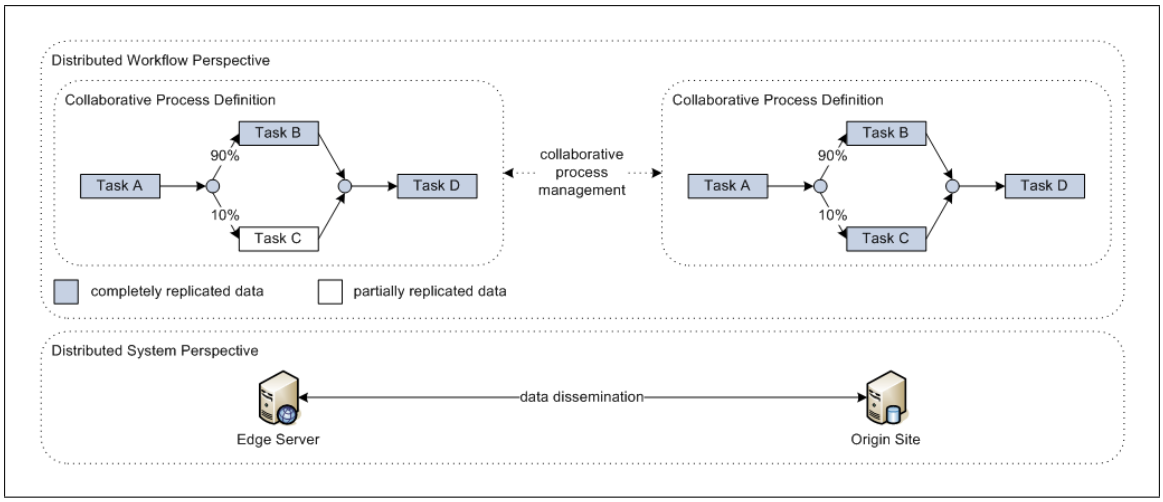

Figure 7.3: Distributed Workflow Scenario

The diagram in figure 7.3 illustrates a plain collaborative business process which consists of four separated workflow tasks (Task A, Task B, Task C, and Task D). The diagram further outlines that the predicted execution frequency of Task B is much higher than the execution frequency of Task C. The implementation of the data service layer may leverage this information in combination with the knowledge of the data requirements of individual tasks to define an appropriate dissemination strategy. As sketched in figure 7.3, data required by Task A, Task B, and Task D is completely replicated while data for Task C is disseminated on demand. As a result, network traffic is reduced significantly since data required by Task C is not automatically pushed to all edge severs. Nevertheless, a large number of possible workflows can be processed directly at the edge of the internet. For the rare case that a workflow requires to execute Task C two options exist. The first option requires the data service layer to throw an exception that can be handled by the business logic to perform business task delegation. The business logic can utilize the facilities of the collaborative process management to initiate execution of Task C at the origin site. If the data service layer is an integral part of the distributed workflow management system delegation of task execution may occur automatically. The second option involves the data service layer to request on demand delivery of all data items necessary for the execution of Task C and invalidated since the last execution of Task C. The transfer and substitution of all invalidated data items may introduce an extensive processing delay and a lot of network traffic. The second option is therefore advisable merely in situations where a further execution of Task C can be assumed to occur shortly.

A large number of distributed workflow scenarios may exhibit a varying, runtime specific, execution frequency of conditional workflow tasks. It might therefore be useful to supply execution frequency information to the application specific strategies algorithm of the data service layer that can then dynamically adapt the dissemination and caching strategies. Please note that the plain collaborative business process depicted in figure 7.3 consists merely of four workflow tasks. A better reduction of network traffic can be expected with bigger control flow graphs and a larger number of conditional workflow tasks.

Distributed workflow management systems require a flexible and robust communica-

tion infrastructure for the coordination of collaborative business processes. The information propagated among individual workflow engines usually includes process data, task return messages, task control information, and several types of workflow events [26]. In addition, programmers are usually provided with mechanisms for event notification and signaling among loosely coupled workflow tasks. As a consequence, distributed object architectures form a key infrastructure technology [50] for workflow coordination in distributed workflow management systems. Current research is concerned with the utilization of message brokers and the publish-and-subscribe paradigm for event dispatching [26]. It comes as no surprise that a number of commercial workflow products, such as IBM WebSphere MQ Workflow [66], are actually built on top of established enterprise messaging systems.

Most distributed workflow management systems merely propagate coordination and collaboration information but do not come with an infrastructure for the dissemination of operational data as required by workflow tasks. We argue that on demand based distribution of task specific data might be the next trend in distributed workflow management. The integration of the data service layer architecture into distributed workflow management systems may provide a flexible mechanism for the partial distribution of task specific data. We believe that the data service layer can reduce the data replication overhead of conditional workflow tasks in large scale workflow scenarios. We further argue that collaborations of dispersed enterprise systems can be accelerated notably by moving the executing of workflow tasks closer to business partners. Please note that the infrastructure provided by the data service layer may as well be leveraged for the coordination of collaborative business processes. It remains to be seen how difficult it actually is to extend the collaborative process definition with the required metadata information.

7.2 Analysis and Applicability

The design of the data service layer allows business solutions to utilize a mix of completely replicated and partially replicated data entities. Complete data replication is achieved with message oriented middleware; a proven solution for internet based data exchange. Partial replication, on the other hand, can be tweaked to the application requirements due to a flexible communication infrastructure and the possibility to define application specific dissemination strategies. The data service layer approach is based on the fact that a considerable amount of operations in a variety of application areas can be accomplished on top of a partially available data spectrum. The key requirement for a successful deployment and a performance improvement is, however, that data requirements of business operations can be defined at design time or predicted at runtime.

The digital library application scenario outlined in section 7.1.1 is an example for the assignment of replication schemes at design time. The data of Author, Article, and Category entities is always replicated completely while a more flexible dissemination scheme is applied to the data of the Content table. Please note that purely the assignment of replication schemes is performed at design time. The caching and dissemination strategies are still adapting dynamically as document request patterns change. The distributed workflow application scenario outlined in section 7.1.2 adapts both the entities which are partially replicated and the dissemination strategy applied. De-

pending on the execution frequency of conditional workflow tasks partial replication may be applied to a variety of different data entities. In the example sketched in figure 7.3 changes to the execution frequency may lead to partial data replication for Task B while data for Task C is completely replicated. In any case, the application specific dissemination strategies of the data service layer have to be supplied with appropriate metadata information.

The applicability of the data service layer is considerably dependent upon the particular application scenario. As a result, implementation of a general purpose data service layer for a range of enterprise solutions is not possible. The discussion in this thesis report is therefore tailored towards the definition of a general data service layer architecture and the establishment of strategies and guidelines for its implementation. The experiences of the authors have been summarized in this work to aid development of the data service layer in a diversity of enterprise solutions. The design of the data service layer aims to compensate the lack of a general purpose solution with a maximum of flexibility for the definition of data dissemination strategies and business task delegation. A detailed analysis in real world application scenarios may probably determine the fraction of operations that can be performed directly at the edge of the internet. Please note, however, that the data service layer has to be seen as an extension of the content broker approach. Virtually all tasks performed by content brokers, such as static content delivery, dynamic assembly, and execution of presentation logic, can as well be executed at service brokers relying on the data service layer. Compared to traditional centralized architectures, performance is therefore enhanced even in the unlikely case that all business tasks have to be diverted to the backend system.

The focus of the research community during the last couple of years illustrates that the traditional multitiered application model, advocated by current enterprise software architectures, is inappropriate for internet scale deployments. The increasing number of collaborations performed among dispersed business partners lead to numerous caching and dissemination proposals. The data service layer presents our current approach to diminish the expense of communication and collaboration across wide area networks. In contrast to a range of other proposals the design of the data service layer is tailored towards service oriented business solutions. In addition, we present a universal architecture and establish general strategies and guidelines instead of suggesting an optimized solution for a particular application scenario. It might be argued that the concepts presented are too unspecific and lack a detailed performance evaluation. This thesis is rather focused on contributing a general and application independent discussion on guidelines and strategies for the development of efficient, flexible, and reusable dissemination infrastructures for enterprise solutions. The data service layer may provide much demanded concepts to bridge the gap between numerous theoretical approaches and an actual implementation in modern enterprise solutions.

During the realization of the data service layer particular emphasis was laid on the creation of a flexible architectural design and the utilization of sound technologies. The deployment of well established design strategies and best practices has already been discussed in chapter 6. The subsequent discussion provides background information and references that played a decisive role in finding the most appropriate technologies and design decisions. This information might be especially useful with the growing number of competing technologies [80] for the implementation of B2B solutions. The information is intended for software architects and developers who are frequently confronted with the complicated process of determining the most appropriate technologies

for the implementation of business solutions.

The commitment to a message oriented middleware has the most extensive influence on the overall design of the data service layer. A precise assessment of the applicability of JMS is therefore unavoidable. We have already discussed several arguments for and the influence of the deployment of MOM throughout this thesis. In addition to the technical features and benefits outlined in various J2EE and JMS books we explicitly researched into case studies and performance evaluations. The interested reader might want to refer to an executive report [33] on the deployment of JMS in Commerce One [32], one of the largest current B2B solutions. The article analysis the middleware challenges of a global ecommerce solution with an extensive number of trading partners. The author provides several details on the challenges in transferring huge amounts of data in a transactional way among a large number of business systems. Further information is provided on the selection of an appropriate message oriented middleware vendor for the Commerce One scenario and the custom extensions requested. The requirements of the Commerce One e-marketplace are surprisingly similar to those of the data service layer. Both require a flexible internet based communication infrastructure and a transactional dissemination of data items. The deployment of JMS in Commerce One is a valuable prove for its applicability and suitability in internet based B2B collaborations. Please note that the MOM vendor selected for the implementation of Commerce One is especially suitable for internet scale deployments. The same MOM product has consequently been utilized for the implementation of the data service layer.

Additional affirmation comes from a related research paper [73] which demonstrates performance improvement of data access by exploiting asynchronous messaging and caching services. The article suggests the uses of JMS for components which are required to dispatch data and access a database within a single transaction. Please note that the data access worker component is also required to treat processing of messages and database updates as a single unit of work. The performance evaluation is, therefore, of particular interest for the implementation of the data service layer. The research paper confirms our basic assumption that asynchronous messaging can significantly improve throughput and scalability of data operations at the backend system. A deployment of JMS for on demand delivery of data items may improve performance at edge servers. It remains to be seen if a mechanism for the complete replacement of RMI based communication with JMS can be found for the implementation of the data service layer. The deployment of JMS for data dissemination is further acknowledged by a recent IBM article [39] on large scale file replication.

A comprehensive performance and scalability analysis [27] of J2EE applications reveals that performance of persistence logic is best when implemented merely with servlets or session beans. In fact JDBC based database access is implemented with session beans in the data service layer. One of the design goals of the data service layer is, however, the support of a wide range of diverse enterprise applications. It is therefore necessary to support entity beans in addition to JDBC based database access. According to the results of the performance study, throughput of both CMP and BMP entity beans can be improved significantly by the deployment of a session façade. As outlined in chapter 6 conversion of data retrieval and storage requests to notification group objects is performed by a session façade. The design pattern may, in addition, improve performance of local data operations especially since the advice to use local interfaces (as provided by the EJB 2.0 specification) has been followed.

In addition to excellent books on the discussion of conventional [47] and J2EE specific [9] design patterns we found the discussion of patterns for the development of maintainable ORB middleware [103] especially interesting. The authors provide a detailed study on the application of design patterns for the development of flexible, extensible, and ultimately efficient ORB middleware. An ORB is similar to the data service layer in the sense that both extend a communication infrastructure with additional configurable services. The ORB provides convenient interconnectivity for object oriented programming languages on top of conventional communication protocols. The data service layer, on the other hand, provides data dissemination services on top of distributed object architectures and enterprise messaging systems. We identified several coincident problems and eventually have been able to adapt several design decisions to the data service layer implementation.

7.3 Related and Future Work

The presented work defines concepts, strategies, and guidelines for the extension of enterprise applications with valorized edge servers. A detailed evaluation of the data service layer under real world conditions is one of the most curial areas of future work. The analysis may either be performed in a simulating environment as defined by the TPC-W [123] benchmark or, even better, by integrating the data service layer into a comprehensive business solution. In any case, of particular interest is the influence of the data service layer on the performance of local data operations as well as the fraction of business operations that can be processed directly at the edge of the internet. Further research is required to determine the impact of the suggested architecture on scalability, failover, and clustering support of modern application servers. It remains to be seen how difficult it actually is to provide sufficient metadata information for the deployment of efficient dissemination algorithms. Further investigation is needed to verify seamless adaptation of application specific caching and dissemination algorithms suggested in numerous related and mainly theoretical research papers.

A possible utilization of the data service layer for partial replication of task specific data in distributed workflows management systems is outlined in section 7.1.2. The current solution is limited to on demand distribution of data required by individual workflow tasks. Further research may reveal a mechanism for ad hoc delivery of task descriptions. The resulting distributed workflow management systems would, therefore, be in charge of dynamically distributing workflow tasks to arbitrary workflow engines.

Analogous to the deployment of the DAO design pattern, considerable effort is inherent in the implementation of the data service layer. There exist, however, several object-to-relational mapping tools, such as Oracle TopLink [97] and Thought CocoBase [121], which offer automatic code generation for DAOs. The utilities automatically generate the code once the mapping is complete, and may provide other value added services such as results caching and query caching. In theory, automatic generation of parts of the data service layer might be achieved by defining specialized deployment descriptors. Replication schemes could be defined similar to the configuration provided in vendor specific deployment descriptors of CMP entity beans. The data access manager, data dispatcher, and common services component could then be generated automatically leaving the application developer merely with the implementation of appropriate

caching, dissemination, and expiration algorithms.

An interesting alternative to JDBC and CMP entity beans is the novel Java Data Objects (JDO) [71] API. It provides persistent support for POJOs and supports a wide variety of data sources including relational databases, object databases, and filesystem storage. The JDO API has been designed to provide persistence support in either managed or nonmanaged environments. In the context of the data service layer we are especially concerned with the deployment of JDO in an application server. The most popular use of JDO in an EJB environment is to have session beans directly manage JDO objects, avoiding the use of entity beans. Further research is required to determine if the surrogate architecture of the data service layer can potentially be used to support JDO based data source access.

Issues involving web based deployment of conventional multitiered enterprise applications lead to the proposal of numerous solutions [78, 31, 10, 12, 35] in the context of large scale distributed systems. Most approaches are, however, either of theoretical nature or tailored towards a specific application scenario. Moreover, a lack of support for and a seamless integration with existing enterprise technologies is often identified for prototype implementations. The presented work aims to bridge the gap by introducing strategies and guidelines for the integration of application specific dissemination solutions into modern enterprise applications.

The application model of a CORBA based persistent state service prototype [77] is similar to the data service layer approach. The authors suggest the deployment of a flexible information dissemination infrastructure for a worldwide auction system. Similar to the data service layer a mix of aperiodic pull and aperiodic push is utilized for data propagation. In addition, a snooping mechanism is introduced to proactively fill caches at edge of the network auction systems. In contrast to the data service layer approach a complete unbundling of caching and object-to-relational mapping is introduced. The auction systems at the edge of the internet are, consequently, restricted to in memory caching. The type of operations that can be performed on top of cached data items is, therefore, limited significantly. Moreover, server restarts result into a complete cancellation of all cached items. The data service layer introduces a caching system based upon a relational database. Business logic at edge servers can therefore perform complex inquiries, such as SQL queries, on top of cached data items. Furthermore, access to cached data items is handled in the same way as it is the case for regular data items at the backend system. Please note that in memory caching of persistent data is a built in service of most modern application servers. Moreover, edge servers can take advantage of resource pooling and transactional access when operating on cached items. We consequently argue that the data service layer enables a richer set of operations to be performed at the edge of the internet. The CORBA based persistent state service prototype is rather an intelligent caching system than a comprehensive edge server architecture.

The introduction of a slightly relaxed consistency scheme for data replication in edge server architectures is the focus of another related research project [48]. The system aims to achieve a scalability improvement by introducing a weak consistency model for individual distributed objects. A reduction of consistency requires application developers to be aware of the application uniformity and distribution scheme. This is in conflict with our primary design constraint of keeping the data service layer simple and transparent to application developers. The research article provides a detailed and

general discussion of usage patterns in ecommerce solutions. This information may be leveraged for the development of efficient caching and dissemination algorithms for the data service layer. Moreover, the information can aid developers in deciding which data categories, of ecommerce applications, are most appropriate for partial replication.

A rather theoretical discussion on data replication in edge server architectures can be found in a current research paper draft [106]. The authors research into on demand based replication of data in web applications. The report proposes automatic data segmentation across replicates based on data access patterns. This is essentially similar to the approach promoted by the data service layer architecture. The focus of our work is on the integration of mechanisms for partial replication while the research paper explores the design space of data clustering and replication. The article is therefore of particular interest and may solve one of the open issues outlined in section 4.2.4. In particular, it may guide application developers in determining the correct granularity of index and data items and help defining optimal consistency strategies for data segments. Moreover, the detailed analysis of usage patterns may facilitate the development of efficient caching, dissemination, and expiration algorithms.

Several commercial products, such as SpiritSoft SpiritCache [109] and Tangosol Coherence [118], offer distributed caching solutions for J2EE based enterprise applications. Most of these products are based on the Java Temporary Caching (JCACHE) [69] API which is a current Java Specification Request (JSR) for distributed object caching. It is interesting to note that SpiritSoft SpiritCache leverages JMS as an intercache communication infrastructure. At first glance, the techniques deployed by these tools might look similar to the mechanisms supplied by the data service layer. Please note, however, that a pure caching system significantly limits the type of operations that can be distributed (see section 4.1.2 for more details). The probably most comprehensive commercially available product in the context of the edge server architecture is the Akamai EdgeComputing [7] platform. As outlined in section 3.3 the architecture enables static content caching and dynamic assembly directly at the edge of the internet. We already provided a detailed comparison of the data service layer with the Akamai EdgeComputing platform in the preceding chapters. The most crucial difference is that the Akamai approach is tailored towards CDNs while the data service layer is designed to support service oriented processing at the edge of the internet. We argue that the increasing number of collaborations among dispersed business partners requires a more service oriented solution than currently offered by Akamai.

The data service layer approach is notably the only application specific dissemination solution, known to the authors, which incorporates caching, dissemination, and expiration algorithms at both the backend system and edge servers. We believe that distribution of decision logic may result into increased dissemination efficiency. We argue that context information of all systems can be leveraged to determine a more efficient placement of data items.

Chapter 8

Conclusion

This thesis discusses the extension of multitiered enterprise applications with valorized edge servers. We explore the design space of the edge server architecture to enable distribution of business processing to geographically dispersed service brokers. We eventually committed our effort to the Java 2 Enterprise Edition and discuss the architecture, design, and implementation of a transparent data service layer. Our architectural proposal defines a system model and an application model for the efficient decomposition of enterprise applications into a central backend system and multiple dispersed edge servers. In addition, we provide strategies and guidelines for the integration of a push and pull based data distribution infrastructure. Moreover, particular focus is on the seamless incorporation of application specific dissemination, caching, and expiration algorithms.

The implementation of the data service layer assumes that a considerable amount of operations in a variety of application areas can be accomplished on top of a partially available data spectrum. A key requirement for a successful deployment is, therefore, the definition of data requirements of business operations. The design of the data service layer enables business solutions to flexibly combine complete and partial data replication as well as on demand based and speculative data dissemination. Application developers declare the replication scheme as well as dissemination, caching, and expiration strategies for individual data entities. The data service layer, on the other hand, provides a flexible infrastructure that ultimately allows a minimization of network traffic and a performance improvement of business operations.

The data service layer is a conventional J2EE application which is deployed in the business container of an application server. It consists of several EJBs which mediate between the business logic, the locally available database, and remote data service layer instances. Local data operations are performed on top of CMP or BMP entity beans and JDBC while remote communication relies on Java RMI and JMS. The data service layer performs data retrieval and storage operations on behalf of the business logic. The design automatically determines if local data operations are sufficient or a remote request has to be initiated. The data service layer encapsulates the complexity of local and remote data operations, speculative data propagation and on demand based data retrieval.

The data service layer provides an efficient interconnectivity for dispersed enterprise systems that can significantly improve performance and scalability of service oriented interactions.

Appendix A

Source Code Samples

This appendix provides source code samples which may aid developers in gaining a better understanding of the design and implementation of the data service layer. The following sections present selected code fragments of the reference implementation which illustrate crucial design aspects and implementation concepts. This appendix is not intended as a comprehensive source code documentation, but rather an attempt to provide further insight into the development of the data service layer. The source code samples can guide developers in determining the applicability of the data service layer for a given application scenario. Moreover, an estimation of constraints and impact of the data service layer on business logic is better achievable with an overview of decisive interfaces and source code parts.

The source code shown in the following sections is taken from the reference implementation of the digital library as discussed in section 7.1.1. The sample code follows the architectural proposal as well as the design and implementation strategies and guidelines discussed within this thesis. Most of the source code listings outline universal code which can be reused directly or with slight modification in various application scenarios. The entities operated on are, however, specific to the digital library application scenario. This is in contrast to the rest of the thesis which discusses concepts, design, and implementation in an application independent way and at a higher level of abstraction. It might therefore take some effort to set the code samples into context to the overall design discussed in the pervious chapters. Nevertheless, this appendix can be invaluable to gain an in depth understanding of the concepts introduced and discussed. Please note that debugging, logging, error handling, and exception handling statements presented in the source code of the reference implementation are not included in the samples within this appendix.

Data Access Manager Interface

This section discusses the coarse grain value object based interface of the data access manager component. The following code snippets outline how web and business components interact with the data service layer to request accomplishment of data storage and retrieval operations. The code samples provide valuable insight into the semantic and the abstraction introduced by the data service layer. The following explanation goes into details on (1) establishing a session with the data access manager, (2) per-

forming data lookup and storage operations, (3) obtaining metadata information, and (4) configuring the data service layer at runtime.

The data access manager implements the session façade pattern to provide a comprehensive set of business methods for data retrieval and storage operations. To interact with the data service layer web and business components have to create a new session bean instance. The procedure to obtain a reference to the the `DataAccessManager` EJB in the digital library implementation is illustrated in the following code snippet:

```
// obtain a reference to the service locator
ServiceLocator serviceLocator = ServiceLocator.getInstance();

// obtain a home local reference to the DataAccessManager EJB
DataAccessManagerHomeLocal dataAccessManagerHomeLocal = (DataAccessManagerHomeLocal)
        serviceLocator.getHomeLocal(JndiEncNames.EJB_DATA_ACCESS_MANAGER);

// obtain a local reference to the DataAccessManager EJB
DataAccessManagerLocal dataAccessManager = dataAccessManagerHomeLocal.create();
```

The sample code shown above obtains a reference to the `ServiceLocator`, uses the `ServiceLocator` to retrieve a home local reference to the `DataAccessManager` EJB, and finally creates a new session bean instance. The `DataAccessManagerLocal` reference can then be used to perform data storage and retrieval operations, to configure the data service layer, and to obtain metadata information. In other words, the methods of the bean class provide value object based access to the persistent data store. The following method illustrates the use of selected data lookup operations provided by the `DataAccessManager` EJB:

```
public void obtainArticleInformation(Integer id)
{
  // get the article immutable value object
  ArticleImmutableValueObject article = dataAccessManager.findArticleByPrimaryKey(id);

  // get the category/topic immutable value object assigned to the article
  TopicImmutableValueObject topic = dataAccessManager.getTopicForArticle(id);

  // get a set of author immutable value objects assigned to the article
  Set authors = dataAccessManager.getAuthorsForArticle(id);

  // get the content immutable value object assigned to the article
  ContentImmutableValueObject content = dataAccessManager.getContentForArticle(id);

  // process article, topic, authors, and content immutable value objects
}
```

The `obtainArticleInformation()` method outlined above uses a local reference to the `DataAccessManager` EJB and the `id` passed as a parameter to obtain information on an article. Please note that the `getContentForArticle()` method call attempts to retrieve a partially replicated data entity. The data access worker component may, therefore, issue a remote data lookup if the operation is executed at an edge server and the document assigned to the article is not available locally (not cached). The data service layer encapsulates the complexity of data retrieval and storage operations and provides the web and business logic with a convenient value object based interface. The application developer is not required to be aware if data is locally available or has to be pulled on demand.

In rare cases information on the internal activities and operations performed by the data service layer may be useful. This information can be requested from the data service layer in form of metadata objects. The following sample method illustrates how

metadata information can be obtained for an article update operation:

```
public void obtainMetaDataInformation(Integer id, String title, String desc, Date date)
{
  // construct a new mutable value object with the article information supplied
  ArticleMutableValueObject mvo = new ArticleMutableValueObject(title, desc, date);

  // update the article with the specified id with the given information
  ArticleImmutableValueObject ivo = dataAccessManager.updateArticle(id, mvo);

  // obtain metadata information for article update operation
  DataOperationMetaData dataOperationMetaData = (DataOperationMetaData)
          dataAccessManager.getMetaData();

  // analyze metadata information
}
```

The `obtainMetaDataInformation()` method sketched in the previous listing updates an article specified by the `id` parameter with the information provided in the `title`, `desc`, and `date` parameters. The method uses both mutable and immutable value objects to perform this task. While the `updateArticle()` method requires an `ArticleMutableValueObject` parameter the return value is an `ArticleImmutableValueObject`. The use of mutable and immutable value objects has already been discussed in section 5.3.2. In essence, the `DataAccessManager` EJB returns immutable value objects to prevent application developers from unintentionally updating transfer objects without committing the changes back to the data service layer. The mutable value objects, on the other hand, don't provide an `id` field to emphasize the fact that primary keys are automatically generated in the reference implementation. The interface would not be intuitive if insert and update operations accept value objects which contain an `id` value but don't take it into account.

The `DataOperationMetaData` object obtained with the `getMetaData()` method call can be used to analyze the operations performed by the data service layer during the article update. The metadata object reveals, among other things, if the data operation resulted into a remote operation, if the article has been cached or disseminated, and if an invalidation message has been sent. The information in the metadata object is dependent on the type of system (edge server or origin site) the update operation has been executed at. The `getMetaData()` method is intended to be used when metadata information is required occasionally as it is usually the case with web and business components. A class which requires metadata information for each data operation may implement the `MetaDataObserver` interface and announce an interest to the metadata subject. This mechanism is especially useful for application specific algorithms which are required to collect statistics on all data operations performed by the data service layer. The `MetaDataObserver` interface, shown in the following listing, is straightforward to implement:

```
public interface MetaDataObserver
{
  // gets called by the subjects of the metadata observer pattern
  public void updateMetaData(MetaData metaData);
}
```

The `updateMetaData()` callback method is invoked on each metadata observer implementing the `MetaDataObserver` interface and attached to the metadata subject. An arbitrary number of application specific algorithms can subscribe to the metadata subject to get updated automatically upon each data storage and retrieval operation.

The default configuration supplied in the deployment descriptors of the DataAccessManager EJB can be adapted dynamically at runtime to react to changing conditions. This is of particular interest for caching and dissemination logic which can, depending on the current circumstances, switch between several application specific algorithms. The following code snippet illustrates how the caching strategy of the current session can be configured by web and business components:

```
// create a new caching strategy object
CachingStrategy randomCachingStrategy = new RandomCachingStrategy();

// change the caching strategy for the current session
dataAccessManager.setCachingStrategy(randomCachingStrategy);
```

The brief introduction to the interfaces provided by the DataAccessManager EJB highlights how web and business components interact with the data service layer to request accomplishment of data storage and retrieval operations. The coarse grain value object based interface follows best practice advises and guarantees a seamless integration into existing and newly built enterprise applications. Moreover, the DataAccessManager EJB encapsulates the complexity of the underlying data service layer implementation. The web and business logic is unaware of the partitioning of data into fully and partially replicated data entities. The application developer may, however, request metadata objects to inspect the activities and operations performed internally by the data service layer. In other words, the metadata information enables application developers to circumvent the transparency of the data service layer.

Data Access Worker Implementation

The data dispatcher is the central component of the data service layer which actually carries out data operations at the local database and in collaboration with remote systems. The overall structure of the most essential classes has already been discussed in section 6.5. The AbstractOperation is the most crucial class which directs the overall behavior of the data access worker subcomponent. This section describes the implementation of the AbstractOperation template methods in the reference implementation of the digital library. The focus is on explaining how the data access worker analyzes NotificationGroup objects and delegates the work to individual ConcreteOperation objects.

The reference implementation of the digital library utilizes two distinct AbstractOperation classes to handle data operations and data lookups respectively. The DataOperation class handles data modifications such as inserts, updates, and deletes while the DataLookup class directs searches and resolving of entity relationships. The following discussion is specific to the more complex DataOperation class. The tasks directed by the DataOperation and DataLookup classes are summarized in the activity overview diagram in figure 6.4. Please be informed that the source code implementation performs some of the activities in a slightly deviating order.

The following discussion introduces the most essential template methods of the DataOperation class and concludes with a code snippet of the NotificationGroupAnalyser which dispatches NotificationGroup objects to the correct ConcreteOperation instances (subclasses of the DataOperation template class). The following template

methods use the prefix `do` for methods that have to be overridden by subclasses while the prefix `hook` is used for methods that may be overridden if necessary.

The following source code listing shows the `performOperation()` method of the `DataOperation` class of the reference implementation. The `activeMode` information is obtained from the deployment descriptors while the `sender` variable is extracted from the `NotificationGroup` object which encapsulates the data operation. The `ImmutableValueObject` is a superclass of all concrete immutable value object implementations utilized in the digital library prototype. The `performOperation()` method is merely analyzing the current node type (`activeMode`) the operation is executed at and the issuer (`sender`) of the data operation. Further processing is delegated to the additional template methods `acceptDataOnProvider()`, `disseminateData()`, and `acceptDataOnDistributor()` of the `DataOperation` class. The `doDataOperationRemote()` method, on the other hand, is declared abstract and has to be implemented by ConcreteOperation subclasses.

```
public final ImmutableValueObject performOperation()
{
  // the immutable value object created during the data operation
  ImmutableValueObject immutableValueObject = null;

  // if the data operation is executed at the provider/origin site
  if (ActiveModes.PROVIDER.equals(activeMode))
  {
    // perform data operation at the provider/origin site
    immutableValueObject = acceptDataOnProvider();

    // if the request is issued by on of the distributors/edge servers
    if (Issuers.DISTRIBUTOR.equals(sender))
    {
      // send/forward the data to the other distributors/edge servers
      disseminatData(Boolean.TRUE);
    }
    // if the request is issued by a client (directly attached to the origin site)
    else if (Issuers.CLIENT.equals(sender))
    {
      // send/dispatch the data to all distributors/edge servers
      disseminateData(Boolean.FALSE);
    }
  }
  // if the data operation is executed at the distributor/edge server
  else if (ActiveModes.DISTRIBUTOR.equals(activeMode)
  {
    // if the request is issued by the provider/origin site
    if (Issuers.PROVIDER.equals(sender))
    {
      // perform data operation at the distributor/edge server
      immutableValueObject = acceptDataOnDistributor();
    }
    else if (Issuers.CLIENT.equals(sender))
    {
      // perform remote date operation at the provider/origin site
      doDataOperationRemote();

      // perform data operation at the distributor/edge server
      immutableValueObject = acceptDataOnDistributor();
    }
  }

  // return the immutable value object resulting form the data operation performed
  return immutableValueObject;
}
```

The `disseminateData()` method of the `DataOperation` class directs the dispatch-

ing of NotificationGroup objects at the origin site. The method determines if the NotificationGroup object should be sent with the original node name (as it is the case for data operations received from edge servers) or with the node name of the origin site (as it is the case for data operations issued by client applications). The method then attaches completely replicated entities to the NotificationGroup object by calling the doAttachMandatoryEntities() method. The disseminateItemEnabled() method invocation bothers the application specific dissemination algorithm to determine if the current entities should be replicated partially or completely. The hookAttachItem() and hookAttachIndex() methods attach the data index or the data items to the NotificationGroup object. The sendMessage() method finally dispatches the NotificationGroup object to edge servers. Please note that most methods in this algorithm skeleton have to be implemented by concrete subclasses.

```
private final Boolean disseminateData(Boolean forward)
{
  // if the notification group should be forwarded with original node name
  if (Boolean.TRUE.equals(forward)
  {
    // forward the original notification group
    doForwardMessage();
  }
  // if the notification group should be dispatched with current node name
  else
  {
    // dispatch the modified notification group
    doPrepareMessage();
  }

  // attach completely replicated entities to notification group
  doAttachMandatoryEntities();

  // if dissemination algorithm determines that index and data should be disseminated
  if (Boolean.TRUE.equals(disseminateItemEnabled())
  {
    // attach full data item to the notification group for dissemination
    hookAttachItem();
  }
  // if dissemination algorithm determines that merely the index should be disseminated
  else
  {
    hookAttachIndex();
  }

  // send the notification group to distributors/edge servers
  sendMessage();
}
```

The acceptDataOnProvider() method of the DataOperation class specifies the algorithm for local database operations at the origin site. The doAcceptMandatoryEntities() method performs the operation for completely replicated data items while the hookAcceptItem() and hookAcceptRelationship() methods carry out the work for partially replicated data items (which are always completely available at the origin site). The doProviderOperationResult() method eventually returns the results of the data operation in form of an ImmutableValueObject object.

```
public final ImmutableValueObject acceptDataOnProvider()
{
  // perform data operation for completely replicated entities
  doAcceptMandatoryEntities();

  // perform data operation for index and data
  hookAcceptItem();
```

```
    // perform relationship operation for index and data
    hookAcceptRelationship();

    // return the immutable value object resulting from the data operation
    return doProviderOperationResult();
  }
```

The `acceptDataOnDistributor()` method of the `DataOperation` class defines the steps performed by the data service layer at edge servers upon data operation requests. The `doAcceptMandatoryEntities()` method performs the operation for completely replicated data items. The `acceptItemEnabled()` method call invokes the caching algorithm to determine if the data operation should be performed for the entire data item or the data index only. The `hookAcceptItem()`, `hookAcceptIndex()`, and `hookAcceptRelationship()` methods carry out the actual work while the `doDistributorOperationResult()` method returns the result of the data operation.

```
  public final ImmutableValueObject acceptDataOnDistributor()
  {
    // perform data operation for completely replicated entities
    doAcceptMandatoryEntities();

    // if caching algorithm determines to cache index and data
    if (Boolean.TRUE.equals(acceptItemEnabled())
    {
      hookAcceptItem();
    }
    // if caching algorithm determines to cache merely the index
    else
    {
      hookAcceptIndex();
    }

    // perform relationship operation for index and data
    hookAcceptRelationship();

    // return the immutable value object resulting from the data operation
    return doDistributorOperationResult();
  }
```

As outlined in section 6.5 a factory is responsible for determining the correct ConcreteOperation object for a given `NotificationGroup` object. The following method illustrates the `dataOperation()` method of the `NotificationGroupAnalyser` which instantiates the correct ConcreteOperation object for a given `NotificationGroup` object. The `DataOperationFactory` implements the abstract factory pattern and returns the correct ConcreteOperation object by analyzing the supplied `NotificationGroup` object. The `DataInsert`, `DataUpdate`, and `DataDelete` are subclasses of the `DataOperation` class that implement functionality for insert, update, and delete operations respectively. The `operation` information used to determine the type of operation is taken from the `NotificationGroup` object. The call to the `performOperation()` method ultimately results into the execution of the template method of the `DataOperation` class as discussed above.

```
  public ImmutableValueObject dataOperation(NotificationGroup notificationGroup)
  {
    // get data operation factory
    DataOperationFactory dataOperationFactory =
            DataOperationFactory.getDataOperationFactory(notificationGroup);

    // extract operation type from notification group object
    Operations operation = notificationGroup.getOperation();
```

```
    // if the current operation is an insert
    if (Operations.INSERT.equals(operation)
    {
      // get correct concrete operation object
      DataInsert dataInsert = dataOperationFactory.getDataInsert();

      // perform the insert operation (directed by template method)
      return dataInsert.performOperation();
    }
    // if the current operation is an update
    else if (Operations.UPDATE.equals(operation))
    {
      // get correct concrete operation object
      DataUpdate dataUpdate = dataOperationFactory.getDataUpdate();

      // perform the update operation (directed by template method)
      return dataUpdate.performOperation();
    }
    else if (Operations.DELETE.equals(operation))
    {
      // get correct concrete operation object
      DataDelete dataDelete = dataOperationFactory.getDataDelete();

      // perform the delete operation (directed by template method)
      return dataDelete.performOperation();
    }
  }
```

Please note that all source code samples are shown without error handling, exceptions handling, logging, and debugging statements. Moreover, the code constructing and updating metadata objects is not included in the methods discussed above.

Index

Bibliography

[1] Swarup Acharya, Rafael Alonso, Michael Franklin, and Stanley Zdonik. *Broadcast Disks: Data Management for Asymmetric Communication Environments*. In Proceedings of ACM SIGMOD Conference, pages 199–210, October 1994. http://citeseer.nj.nec.com/acharya94broadcast.html.

[2] Swarup Acharya, Michael Franklin, and Stanley Zdonik. *Balancing Push and Pull for Data Broadcast*. In Proceedings of the ACM SIGMOD97, pages 183–194, 1997. http://citeseer.nj.nec.com/19637.html.

[3] *ACM Digital Library - Website*. ACM, available as of Fri, 06 January 2004. http://portal.acm.org/dl.cfm.

[4] *Internet Bottelnecks: The Case for Edge Delivery Services - White Paper*. Akamai Technologies, November 2000. http://developer.akamai.com/pdf/Bottlenecks-Whitepaper1.pdf.

[5] *EdgeSuite Service Description - White Paper*. Akamai Technologies, August 2001. http://developer.akamai.com/pdf/EdgeSuite_Service_Description.pdf.

[6] *Turbo-Charging Dynamic Web Sites with Akamai EdgeSuite - White Paper*. Akamai Technologies, December 2001. http://developer.akamai.com/pdf/WP_TCD.-pdf.

[7] *Akamai EdgeComputing - Product Website*. Akamai Technologies, available as of Fri, 13 February 2004. http://www.akamai.com/en/html/services/edge-computing.html.

[8] *Edge Side Includes - Product Website*. Akamai Technologies and Oracle, available as of Mon, 05 January 2003. http://www.esi.org/.

[9] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, June 2001. ISBN 0-13-064884-1.

[10] Khalil Amiri, Sanghyun Park, and Renu Tewari. *A Self-managing Data Cache for Edge-Of-Network Web Applications*. In Proceedings of the 11th International Conference on Information and Knowledge Management, pages 177–185, September 2002. http://doi.acm.org/10.1145/584792.584824.

[11] *Information Systems - Database Language - SQL*. ANSI American National Standard X3.135-1992 and ISO/IEC International Standard 9075:1992, 1992.

[12] Jesse Anton, Lawrence Jacobs, Xiang Liu, Jordan Parker, Zheng Zeng, and Tie Zhong. *Web Caching for Database Applications with Oracle Web Cache*. In

Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 594–599, July 2002. http://doi.acm.org/10.1145/564691.564762.

[13] *Apache Software Foundation Ant - Product Website*. Apache Software Foundation, available as of Sun, 18 January 2004. http://ant.apache.org/.

[14] *Apache Software Foundation Log4j - Product Website*. Apache Software Foundation, available as of Sun, 18 January 2004. http://logging.apache.org/log4j/.

[15] Filipe Araújo and Luís Rodrigues. *On QoS-Aware Publish-Subscribe*. Department of Informatics, University of Lisbon, February 2002. http://www.di.fc.ul.pt/tech-reports/02-2.pdf.

[16] *BEA Tuxedo - Product Website*. BEA Systems, available as of Wed, 31 December 2003. http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/tux/.

[17] *Bea WebLogic Server - Product Website*. Bea Systems, available as of Wed, 31 December 2003. http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/server/.

[18] Hans Bergsten. *JavaServer Pages, 3rd Edition*. O'Reilly & Associates, December 2003. ISBN 0-596-00563-6.

[19] Azer Bestavros. *Speculative Data Dissemination and Service to Reduce Server Load, Network Traffic and Service Time for Distributed Information Systems*. In Proceedings of the Internetional Conference on Data Engineering (ICDE96), March 1996. http://citeseer.nj.nec.com/bestavros96speculative.html.

[20] Azer Bestavros and Carlos Cunha. *Server-initiated Document Dissemination for the WWW*. Bulletin of the Computer Society Technical Committee on Data Engineering, pages 3–11, September 1996. http://citeseer.nj.nec.com/bestavros96serverinitiated.html.

[21] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. *Extensible Markup Language (XML) 1.0 Recommendation*. World Wide Web Consortium, October 2000. http://www.w3.org/tr/rec-xml.html.

[22] Fabian Bustamante, Patrick Widener, and Karsten Schwan. *Scalable Directory Services Using Proactivity*. Georgia Institute of Technology, March 2002. http://citeseer.nj.nec.com/bestavros96speculative.html.

[23] *Cooperative Association for Internet Data Analysis - Skitter AS Internet Graph*. CAIDA, available as of Sat, 03 January 2004. http://www.caida.org/analysis/topology/as_core_network/AS_Network.xml.

[24] Bart Calder and Bill Shannon. *JavaBeans Activation Framework Specification, Version 1.0a*. Sun Microsystems, May 1999. http://java.sun.com/products/javabeans/glasgow/JAF-1.0.pdf.

[25] Mary Campione, Kathy Walrath, and Alison Huml. *The Java Tutorial Continued: The Rest of the JDK*. Addison-Wesley, December 1998. ISBN 0-201-48558-3.

[26] Fabio Casati and Ming-Chien Shan. *Event-Based Interaction Management for Composite E-Services in eFlow*. Information Systems Frontiers Journal, volume 4, issue 1, pages 19–31, April 2002. http://dx.doi.org/10.1023/A:1015374204227.

[27] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. *Performance and scalability of EJB applications*. In Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 246–261, October 2002. http://doi.acm.org/10.1145/582419.582443.

[28] Aslihan Celik, JoAnne Holliday, and Bindumadhavi Ramavarjula. *I-DG and HI-DG: Scalable and Efficient Techniques for Subscription-Based Data Dissemination via IP Multicasting*. Electronic Commerce Research Journal, volume 3, issue 1-2, pages 143–165, April 2003. http://dx.doi.org/10.1023/A:1021533512241.

[29] Qiming Chen and Meichun Hsu. *Inter-Enterprise Collaborative Business Process Management*. In Proceedings of the 17th International Conference on Data Engineering, pages 253–260, April 2001. http://csdl.computer.org/comp/proceedings/-icde/2001/1001/00/10010253abs.htm.

[30] Susan Cheung and Vlada Matena. *Java Transaction API Specification, Version 1.0.1b*. Sun Microsystems, November 2002. http://java.sun.com/products/jta/.

[31] Gregory V. Chockler, Danny Dolev, Roy Friedman, and Roman Vitenberg. *Implementing a Caching Service for Distributed COBRA Objects*. In Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms, pages 1–23, June 2000.

[32] *CommerceOne - Website*. Commerce One Operations, available as of Wed, 11 February 2004. http://www.commerceone.com/.

[33] Simon Cutting. *Middleware issues at Commerce One*. Middleware Spectra Journal, volume 15, issue 1, pages 21–30, February 2001. http://www.middleware-spectra.com/abstracts/2001_02_03.htm.

[34] Paul Dantzig. *Architecture and Design of High Volume Web Sites (A Brief History of IBM Sport and Event Web Sites)*. In Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, pages 17–24, 2002. http://doi.acm.org/10.1145/568760.568765.

[35] Anindya Datta, Kaushik Dutta, Helen Thomas, Debra VanderMeer, Suresha, and Krithi Ramamritham. *Proxy-Based Acceleration of Dynamically Generated Content on the World Wide Web: An Approach and Implementation*. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 97–108, July 2002. http://doi.acm.org/10.1145/564691.564703.

[36] Brian D. Davison and Vincenzo Liberatore. *Pushing Politely: Improving Web Responsiveness One Packet at a Time*. In Proceedings of PAWS00, June 2000. http://citeseer.nj.nec.com/davison00pushing.html.

[37] Linda G. DeMichiel. *Enterprise JavaBeans Specification, Version 2.1*. Sun Microsystems, November 2003. http://java.sun.com/products/ejb/docs.html.

[38] Pavan Deolasee, Amol Katkar, Ankur Panchbudhe, Krithi Ramamritham, and Prashant J. Shenoy. *Adaptive Push-Pull: Disseminating Dynamic Web Data*. In Proceedings of 10th International WWW Conference, pages 265–274, 2001. http://citeseer.nj.nec.com/452165.html.

[39] Daniel Drasin. *Get the message?* IBM, February 2002. http://www.ibm.com/-developerworks/java/library/i-jms/.

[40] Robert Elz and Randy Bush. *Clarifications to the DNS Specification, RFC 2181*. Internet Engineering Task Force and Internet Engineering Steering Group, July 1997. http://www.ietf.org/rfc/rfc2181.txt.

[41] *JUnit - Product Website*. Erich Gamma and Kent Beck, available as of Sun, 18 January 2004. http://www.junit.org/.

[42] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. *The Many Faces of Publish/Subscribe*. ACM Computing Surveys, volume 35, issue 2, pages 114–131, June 2003. http://doi.acm.org/10.1145/-857076.857078.

[43] Jim Farley, William Crawford, and David Flanagan. *Java Enterprise in a Nutshell, 2nd Edition*. O'Reilly & Associates, April 2002. ISBN 0-596-00152-5.

[44] *Fiorano FioranoMQ - Product Website*. Fiorano, available as of Wed, 31 December 2003. http://www.fiorano.com/products/fmq/.

[45] Michael Franklin and Stanley Zdonik. *A Framework for Scalable Dissemination-Based Systems*. In Proceedings of OOPSLA97, pages 94–105, October 1997. http://citeseer.nj.nec.com/franklin97framework.html.

[46] Michael Franklin and Stanley Zdonik. *Data in your Face: Push Technology in Perspective*. In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD98), pages 516–519, June 1998. http://citeseer.-nj.nec.com/franklin98data.html.

[47] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, January 1995. ISBN 0-201-63361-2.

[48] Lei Gao, Mike Dahlin, Amol Nayate, Jiandan Zheng, and Arun Iyengar. *Application Specific Data Replication for Edge Services*. In Proceedings of the 12th International Conference on World Wide Web, pages 449–460, 2003. http://doi.-acm.org/10.1145/775152.775217.

[49] Kurt Geihs. *Middleware Challenges Ahead*. IEEE Computer Magazine, volume 34, number 6, pages 24–31, June 2001. http://csdl.computer.org/comp/mags/co/-2001/06/r6024abs.htm.

[50] Dimitrios Georgakopoulos, Mark F. Hornick, and Amit P. Sheth. *An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure*. Distributed and Parallel Databases Journal, volume 3, issue 2, pages 119–153, April 1995. http://citeseer.nj.nec.com/georgakopoulos95overview.html.

[51] C. Gray and D. Cheriton. *Leases: an efficient fault-tolerant mechanism for distributed file cache consistency*. In Proceedings of the 12th ACM Symposium on Operating Systems Principles, pages 202–210, 1989. http://doi.acm.org/10.1145/-74850.74870.

[52] William Grosso. *Java RMI*. O'Reilly & Associates, October 2001. ISBN 1-56592-452-5.

[53] Graham Hamilton. *JavaBeans API Specification, Version 1.01*. Sun Microsystems, August 1997. http://java.sun.com/products/javabeans/docs/spec.html.

[54] Mark Hapner, Rich Burridge, Rahul Sharma, and Joseph Fialli. *Java Message Service Specification, Version 1.0.2b*. Sun Microsystems, August 2001. http://java.sun.com/products/jms/docs.html.

[55] Manfred Hauswirth and Mehdi Jazayeri. *A Component and Communication Model for Push Systems*. In Proceedings of the 7th European Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 20–38, September 1999. http://doi.-acm.org/10.1145/318773.318784.

[56] Nic Holbrook. *Value Object Dispatcher Pattern*. TheServerSide, February 2003. http://www.theserverside.com/patterns/thread.jsp?thread_id=17739.

[57] Nic Holbrook. *Value Object Dispatcher Pattern (Revised)*. TheServerSide, April 2003. http://www.theserverside.com/patterns/thread.jsp?thread_id=19020.

[58] Jason Hunter. *Java Servlet Programming, 2nd Edition*. O'Reilly & Associates, April 2001. ISBN 0-596-00040-5.

[59] *Cache Invalidation Adapter for WebSphere Application Server - Product Website*. IBM, available as of Mon, 05 January 2003. http://www.alphaworks.ibm.com/-tech/cia4was/.

[60] *Edge-Computing Toolkit for WebSphere Studio - Product Website*. IBM, available as of Mon, 05 January 2003. http://www.alphaworks.ibm.com/tech/edgetk/.

[61] *IBM CICS - Product Website*. IBM, available as of Wed, 31 December 2003. http://www-306.ibm.com/software/htp/cics/.

[62] *IBM WebSphere Application Server - Product Website*. IBM, available as of Wed, 31 December 2003. http://www-306.ibm.com/software/webservers/appserv/was/.

[63] *IBM WebSphere MQ - Product Website*. IBM, available as of Wed, 31 December 2003. http://www.ibm.com/software/integration/wmq/.

[64] *WebSphere Software Platform - Product Website*. IBM, available as of Mon, 05 January 2003. http://www.ibm.com/websphere/.

[65] *IBM Cloudscape - Product Website*. IBM, available as of Mon, 12 January 2004. http://www-306.ibm.com/software/data/cloudscape/.

[66] *IBM WebSphere MQ Workflow - Product Website*. IBM, available as of Wed, 11 February 2004. http://www.ibm.com/software/integration/wmqwf/.

[67] British Standards Institute. *The C Standard: Incorporating Technical Corrigendum 1*. John Wiley & Sons, September 2003. ISBN 0-470-84573-2.

[68] British Standards Institute. *The C++ Standard: Incorporating Technical Corrigendum 1*. John Wiley & Sons, December 2003. ISBN 0-470-84674-7.

[69] *Java Community Process Java Specification Request Java Temporary Caching API - Website*. JCP, available as of Fri, 13 February 2004. http://www.jcp.org/en/jsr/detail?id=107.

[70] *JetBrains IntelliJ IDEA - Product Website*. JetBrains, available as of Sun, 18 January 2004. http://www.intellij.com/idea/.

[71] David Jordan and Craig Russell. *Java Data Objects*. O'Reilly & Associates, April 2003. ISBN 0-596-00276-9.

[72] Kannan Kothandaraman. *On-Demand Data Broadcasting*. Texas A&M University, August 1998.

[73] Samuel Kounev and Alejandro Buchmann. *Improving Data Access of J2EE Applications by Exploiting Asynchronous Messaging and Caching Services*. In Proceedings of the 28th International Conference on Very Large Data Bases, 2002. http://citeseer.nj.nec.com/kounev02improving.html.

[74] Balachander Krishnamurthy and Craig E. Wills. *Piggyback server invalidation for proxy cache coherency*. In Proceedings of the 7th International World Wide Web Conference, pages 185–193, April 1998. http://citeseer.nj.nec.com/krishnamurthy98piggyback.html.

[75] W. Li, V. Penkrot, S. Roychowdhury, W. Zhang, P. K. Chrysanthis, V. Liberatore, and K. Pruhs. *An Optimized Multicast-based Data Dissemination Mhddleware: A Demonstration*. In Proceedings of 19th International Conference on Data Engineering (ICDE03), 2003. http://citeseer.nj.nec.com/li03optimized.html.

[76] Sheng Liang. *Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, June 1999. ISBN 0-201-32577-2.

[77] C. Liebig, M. Cilia, M. Betz, and A. Buchmann. *A publish/subscribe CORBA Persistent State Service Prototype*. In Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms, pages 231–255, June 2000. http://portal.acm.org/citation.cfm?id=338368.

[78] Qiong Luo, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Honguk Woo, Bruce G. Lindsay, and Jeffrey F. Naughton. *Middle-Tier Database Caching for e-Business*. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 600–611, July 2002. http://doi.acm.org/10.1145/564691.564763.

[79] Tim Mallalieu and Jeromy Carriere. *Enterprise Interoperability: .NET and J2EE*. Microsoft, January 2004. http://msdn.microsoft.com/library/en-us/dnbda/html/dotnetinteroperability.asp.

[80] Brahim Medjahed, Boualem Benatallah, Athman Bouguettaya, Anne H. H. Ngu3, and Ahmed K. Elmagarmid. *Business-to-business interactions: issues and enabling technologies*. The International Journal on Very Large Data Bases, volume 12, issue 1, pages 59–85, April 2003. http://dx.doi.org/10.1007/s00778-003-0087-z.

[81] *ActiveX - Product Website*. Microsoft, available as of Wed, 31 December 2003. http://www.microsoft.com/com/tech/ActiveX.asp.

[82] *Microsoft Message Queuing - Product Website*. Microsoft, available as of Wed, 31 December 2003. http://www.microsoft.com/windows2000/technologies/-communications/msmq/.

[83] *Microsoft .NET Framework - Product Website*. Microsoft, available as of Wed, 31 December 2003. http://www.microsoft.com/net/.

[84] *Microsoft Transaction Server - Product Website*. Microsoft, available as of Wed, 31 December 2003. http://www.microsoft.com/com/tech/MTS.asp.

[85] *COM - Product Website*. Microsoft, available as of Sun, 18 January 2004. http://www.microsoft.com/com/tech/com.asp.

[86] Richard Monson-Haefel. *Enterprise JavaBeans, 3rd Edition*. O'Reilly & Associates, September 2001. ISBN 0-596-00226-2.

[87] Richard Monson-Haefel and Dave Chappell. *Java Message Service*. O'Reilly & Associates, December 2000. ISBN 0-596-00068-5.

[88] Rajiv Mordani and Scott Boag. *Java API for XML Processing Specification, Version 1.2*. Sun Microsystems, August 2002. http://java.sun.com/xml/downloads/-jaxp.html.

[89] *NEC Scientific Literature Digital Library - Website*. NEC, available as of Fri, 06 January 2004. http://citeseer.nj.nec.com/cs/.

[90] *Novell Directory Service - Product Website*. Novell, available as of Mon, 19 January 2004. http://www.novell.com/products/nds/.

[91] *Common Internet Inter-Operability Protocol Specification, Version 3.0, formal/2002-12-06*. Object Managment Group, December 2002. http://www.-omg.org/technology/documents/formal/corba_iiop.htm.

[92] *Common Object Request Broker Architecture: Core Specification, Version 3.0, formal/2002-12-06*. Object Managment Group, December 2002. http://www.-omg.org/technology/documents/formal/corba_iiop.htm.

[93] *CORBA Component Model Specification, Version 3.0, formal/2002-06-65*. Object Managment Group, June 2002. http://www.omg.org/technology/documents/-formal/components.htm.

[94] *CORBA Naming Service Specification, Version 1.2, formal/2002-09-02*. Object Managment Group, September 2002. http://www.omg.org/technology/-documents/formal/naming_service.htm.

[95] Lukasz Opyrchal, Mark Astley, Joshua S. Auerbach, Guruduth Banavar, Robert E. Strom, and Daniel C. Sturman. *Exploiting (IP) Multicast in Content-Based Publish-Subscribe Systems*. In Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms, pages 185–207, April 2000. http://citeseer.nj.nec.com/opyrchal00exploiting.html.

[96] *Oracle Application Server - Product Website*. Oracle, available as of Wed, 31 December 2003. http://www.oracle.com/appserver/.

[97] *Oracle Application Server TopLink - Product Website*. Oracle, available as of Thu, 12 February 2004. http://otn.oracle.com/products/ias/toplink/index.html.

[98] G. Pierre, M. van Steen, and A. Tanenbaum. *Dynamically Selecting Optimal Distribution Strategies for Web Documents*. In Proceedings of the IEEE Transactions on Computers, pages 637–651, June 2002. http://citeseer.nj.nec.com/-pierre01dynamically.html.

[99] *PostgreSQL PostgreSQL - Product Website*. PostgreSQL, available as of Sun, 19 January 2004. http://www.postgresql.org/.

[100] George Reese. *Database Programming with JDBC and Java, 2nd Edition*. O'Reilly & Associates, August 2000. ISBN 1-56592-616-1.

[101] Pablo Rodriguez and Ernst W. Biersack. *Continuous Multicast of Web Documents over the Internet*, April 1998. http://citeseer.nj.nec.com/rodriguez98continuou.-html.

[102] Mehmet Sayal, Fabio Casati, Umesh Dayal, and Ming-Chien Shan. *Integrating Workflow Management Systems with Business-to-Business Interaction Standards*. In Proceedings of the 18th International Conference on Data Engineering, pages 287–296, February 2002. http://csdl.computer.org/comp/proceedings/icde/-2002/1531/00/15310287abs.htm.

[103] Douglas C. Schmidt and Chris Cleeland. *Applying Patterns to Develop Extensible ORB Middleware*. IEEE Communications Magazine, volume 37, issue 4, pages 54–63, April 1999. http://dl.comsoc.org/cocoon/comsoc/servlets/Get-Publication?id=180055.

[104] Bill Shannon. *Java 2 Platform Enterprise Edition Specification, Version 1.3, Final Release 7/21/01*. Sun Microsystems, Juli 2001. http://java.sun.com/j2ee/-j2ee-1_3-fr-spec.pdf.

[105] Bill Shannon. *Java 2 Platform Enterprise Edition Specification, Version 1.4, Proposed Final Draft 4/11/03*. Sun Microsystems, April 2003. http://java.sun.-com/j2ee/j2ee-1_4-pfd3-spec.pdf.

[106] Swaminathan Sivasubramanian, Guillaume Pierre, and Maarten van Steen. *Towards On-Demand Web Application Replication*, November 2003. http://www.-globule.org/publi/TODWAR_draft.html.

[107] James Snell and Tom Glover. *Portability and Interoperability*. IBM, March 2003. http://www.ibm.com/developerworks/webservices/library/ws-port/.

[108] *Sonic Software SonicMQ - Product Website*. Sonic Software, available as of Wed, 31 December 2003. http://www.sonicsoftware.com/products/sonicmq/.

[109] *SpiritSoft SpiritCache - Product Website*. SpiritSoft, available as of Fri, 13 February 2004. http://www.spirit-soft.com/products/cache/introducing.shtml.

[110] Konstantinos Stathatos, Nick Roussopoulos, and John S. Baras. *Adaptive Data Broadcast in Hybrid Networks*. In Proceedings of the 23rd Very Large DataBases Conference, pages 326–335, August 1997. http://citeseer.nj.nec.com/stathatos97-adaptive.html.

[111] Edward A. Stohr and J. Leon Zhao. *Workflow Automation: Overview and Research Issues*. Information Systems Frontiers Journal, volume 3, issue 3, pages 281–296, September 2001. http://dx.doi.org/10.1023/A:1011457324641.

[112] *Java Authentication and Authorization Service Specification, Version 1.0*. Sun Microsystems. http://java.sun.com/products/jaas/.

[113] *J2EE Connector Architecture Specification, Version 1.0*. Sun Microsystems, August 2001. http://java.sun.com/j2ee/connector/download.html.

[114] *Java Management Extension Instrumentation and Agent Specification, Version 1.2*. Sun Microsystems, October 2002. http://java.sun.com/products/Java-Management/reference/docs/index.html.

[115] *Java 2 Platform Standard Edition Specification, Version 1.4.2*. Sun Microsystems, June 2003. http://java.sun.com/j2se/1.4.2/download.html.

[116] *Java Naming and Direcotry Interface Service Providers - Product Website*. Sun Microsystems, available as of Mon, 19 January 2004. http://java.sun.com/-products/jndi/serviceproviders.html.

[117] Stefan Tai and Isabelle Rouvellou. *Strategies for Integrating Messaging and Distributed Object Transactions*. In Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms, pages 308–330, 2000.

[118] *Tangosol Coherence - Product Website*. Tangosol, available as of Fri, 13 February 2004. http://www.tangosol.com/coherence.jsp.

[119] Pradeep Tapadiya. *COM+ Programming: A Practical Guide Using Visual C++ and ATL*. Prentice Hall, September 2000. ISBN 0-13-088674-2.

[120] Thuan L. Thai. *Learning DCOM*. O'Reilly & Associates, April 1999. ISBN 1-56592-581-5.

[121] *Thought CocoBase - Product Website*. Thought, available as of Thu, 12 February 2004. http://www.thoughtinc.com/cber_index.html.

[122] Joe Touch. *The LSAM Proxy Cache - a Multicast Distributed Virtual Cache*. In Proceedings of 3rd International WWW Caching Workshop, June 1998.

[123] *TPC-W Transactional Web eCommerce Benchmark - Website*. Transaction Processing Performance Council, available as of Thu, 12 February 2004. http://www.-tpc.org/tpcw/.

[124] Wil van der Aalst. *Loosely coupled interorganizational workflows: modeling and analyzing workflows crossing organizational boundaries*. Information & Managment Journal, volume 37, issue 2, pages 51–100, March 2000. http://dx.doi.org/-10.1016/S0378-7206(99)00038-5.

[125] *X/Open CAE Specification - Distributed Transaction Processing: The XA Specification*. X/Open Company, February 1992. http://www.opengroup.org/public/-pubs/catalog/c193.htm.

[126] W. Yeong, T. Howes, and S. Kille. *Lightweight Directory Access Protocol, RFC 2251*. Internet Engineering Task Force and Internet Engineering Steering Group, December 1997. http://www.ietf.org/rfc/rfc2251.txt.