P2P INDEXING OF ENTERPRISE JAVA BEANS

By

Georgios A. Dementis

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE
DIPLOMA IN ELECTRONICS AND COMPUTER ENGINEERING
AT
TECHNICAL UNIVERSITY OF CRETE
CHANIA, GREECE
JULY 2005

© Copyright by Georgios A. Dementis, 2005

TECHNICAL UNIVERSITY OF CRETE
DEPARTMENT OF
ELECTRONICS AND COMPUTER ENGINEERING

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled "**P2P indexing of Enterprise Java Beans** " by **Georgios A. Dementis** in partial fulfillment of the requirements for the degree of **Diploma in Electronics and Computer Engineering**.

Dated: July 2005

Supervisor: ____________________
Prof. Vasilis Samoladas

Readers: ____________________
Prof. Manolis Koubarakis

____
Prof. Euripides Petrakis

TECHNICAL UNIVERSITY OF CRETE

Date: **July 2005**

Author: **Georgios A. Dementis**

Title: **P2P indexing of Enterprise Java Beans**

Department: **Electronics and Computer Engineering**

Degree: **Diploma** Convocation: **July** Year: **2005**

____
Signature of Author

To my family.

Table of Contents

Table of Contents v

Abstract vii

Acknowledgements viii

1 Introduction 1
- 1.1 Distributed Hash Tables 2
- 1.2 Indexing 4
- 1.3 Enterprise Java Beans (EJB) 6
- 1.4 Our contribution 8
- 1.5 Outline 9

2 Related Work 11
- 2.1 Peer-to-Peer Technologies 11
  - 2.1.1 GISP 11
  - 2.1.2 Project JXTA 12
- 2.2 J2EE technologies 14
  - 2.2.1 EJB Architecture 14
  - 2.2.2 JBoss Application Server 15
- 2.3 Other Projects 18
  - 2.3.1 PROST 18
  - 2.3.2 pSearch 19
  - 2.3.3 Data Indexing in Peer-to-Peer DHT Networks 20
  - 2.3.4 Coral 20
  - 2.3.5 OceanStore 21

3 P2P indexing of EJB: The Design 22
- 3.1 Defining P2P Indexing of EJB's 22
  - 3.1.1 Transactional issues 24
- 3.2 Basic components 26

3.2.1 Container issues . . . 27
3.3 The Indexing mechanism . . . 28
3.3.1 Insert mechanism . . . 28
3.3.2 Delete mechanism . . . 29
3.3.3 Lookup mechanism . . . 30

4 P2P indexing of EJB: The Implementation 32
4.1 DHT : GISP . . . 32
4.2 EJB Container : JBoss . . . 34
4.3 Implementing the Indexing mechanism . . . 35
4.3.1 Insert mechanism . . . 35
4.3.2 Delete mechanism . . . 39
4.3.3 Lookup mechanism . . . 40

5 Conclusions 43

A User's Manual 45
A.1 Prerequisites . . . 45
A.2 Step 1: Platform Setup . . . 47
A.3 Step 2: Extend your EJB . . . 48
A.4 Step 3: Define the indexed attributes . . . 51
A.5 Step 4: Create the Finder methods . . . 55
A.6 Step 5: Deploy your Bean . . . 61
A.7 Step 6: Running a client application . . . 61

B Javadoc 63

Bibliography 64

Abstract

As the Web continues to grow in both content and the number of connected devices, peer-to-peer (P2P) systems are becoming increasingly popular. The core operation in every P2P system is to efficiently locate the node that stores a particular data item. To address this problem lately a lot of research effort has been put into the design and analysis of Distributed Hash Tables (DHT), resulting in a variety of applications built on top of them.

This thesis proposes such an application and in particular presents a P2P infrastructure that allows an application based on Entity EJB's to be deployed in multiple Application Servers and form a P2P network, where it is possible to create indexes on the data of EJBs and thereupon implement corresponding finder methods that can locate EJBs, inside the P2P, that hold particular data.

Since Entiry EJB's are an object-oriented representation of data in a persistent store, such as the records in a database, our architecutre actually extendes the idea of indexing over a single database, to a P2P database. The DHT provides the required indexing functionality.

We outline the design specific characteristics of such an indexing mechanism and provide an implementation that can be easily adopted by an EJB developer on top of his application and through a given API, extend the standard EJB functionality with P2P indexing capabilities.

Acknowledgements

I would like to thank Vasilis Samoladas, my supervisor, mostly for his patience during this past year. His guidance, relevant or not to this Thesis, will always be appreciated.

I would also like to thank Costas Harizakis, Christos Vosnidis, Nikos Pallas and Panagiotis Paterakis. Each of them provided some help relevant to this work, not to mention their support and friendship.

Special thanks goes to my Dad. I will never forget how supportive he has been all this time I was a "lazy" son.

Finally, I wish to thank all those people that are next to me and support me all those years..and I am happy they are a lot; The friends for life i made here in Chania and we really had the best time together; Stavros, John, Dimitris(kyb), Dimitris (Kontokos), Fanouris, Nikos, Akis; and the ones I already had;Dimitris(cousin), Alexandros(!), Ali, Giorgos(cousin), Giorgos(Antonop), Eric Cartman (Ooops no he is just a cartoon) and last but not least Stella.

Chania, Crete | Georgios Dementis
July 18, 2005

Chapter 1

Introduction

As the Web continues to grow in both content and the number of connected devices, peer-to-peer (P2P) systems are becoming increasingly popular. Powerful PCs and broad-band networks made it possible to do calculations on the network edge and inevitably several recent distributed applications based on the P2P paradigm have drawn media headlines and industry attention. Formally P2P systems and applications are distributed systems without any centralized control or hierarchical organization, where the software running at each node is equivalent in functionality. Informally, P2P systems are composed of a collection of peer nodes that cooperate in order to perform some task and share their resources by direct exchanges. In the P2P model, each node may be both a provider and consumer of services (i.e., a peer), which differs from the client-server model where only a relatively small number of server nodes provide services to a potentially large number of client nodes.

The core operation in every peer-to-peer system is to efficiently locate the node that stores a particular data item. To address this problem lately a lot of research effort has been put into the design and analysis of Structured P2P overlay systems, which are virtual communications structures that are logically laid over an underlying physical network such as the Internet. They conform

to a specific graph structure that allows them to locate objects by exchanging O(logN) messages where N is the number of nodes in the overlay. The most common service abstraction implemented by them is Distributed Hash Tables (DHT). Exploring the notion of DHT as a starting point, in the rest of this chapter we briefly present all the relevant components that are necessary for this thesis.

1.1 Distributed Hash Tables

Distributed Hash Tables (DHT) research was originally motivated, in part, by peer-to-peer systems such as Napster and Gnutella. These systems were able to take advantage of resources distributed across the Internet to provide a single useful application. In particular, they took advantage of increased bandwidth and hard disk capacity to provide a file sharing service. Napster and Gnutella themselves were different solutions to a search problem - how to find files located on different computers around the world that have no knowledge of one another. Napster solved this problem by acting as an index and introduction service: when computers joined the Napster network, they would notify a central server of the files they held locally. Searches were performed on the server, which would refer the queerer to the machines that held files relevant to the search. This central component left the system vulnerable to attack. In response, Gnutella and similar networks moved to a flooding query model - in essence, each search would result in a message being broadcast to every other machine in the network. While avoiding a single point of failure, this method was significantly less efficient than Napster. Distributed hash tables attempt to find a more optimal method for organizing nodes while still avoiding the problems of Napster.

A DHT typically seeks to achieve some or all of the following properties:

- **Decentralized operation**: every node should be able to function independently and collectively form the complete system without any central coordination.
- **Scalability**: the system should function efficiently even with large number of nodes. That is, it should scale.
- **Load balance**: keys (i.e. data) should be distributed evenly among the different participants.
- **Fault tolerance**: the system should be reliable (in some sense) even if nodes fail or leave the system.
- **Performance**: Operations such as routing and data storage or retrieval should complete quickly.
- **Data integrity**: It should be easy to verify the correctness of data stored in or retrieved from the system.
- **Security/Robustness**: The system should continue to function "correctly" even if some (possibly large) fraction of the nodes are conspiring to prevent correct operation.
- **Anonymity**: The system should not allow observers to determine who is doing what inside the system.

It is difficult to achieve all of these properties simultaneously; research into achieving these goals is on-going.

Nodes in a DHT are organized in a network overlay (such as a circle or a hypercube) over some space. Each node has a logical identifier that determines its logical position in the overlay. A join protocol allows a new node to bootstrap

into the existing system, usually by contacting a node that is known to be in the system already. This protocol introduces the node to a set of neighbours and typically facilitates the construction of the new node's routing table .

Routing tables are used by DHT nodes to efficiently determine what other node is responsible for a given piece of data. Data is given a key (in the same identifier space) and assigned to the closest node in the overlay. The definition of closest varies depending on the DHT and the topology chosen and usually does not have to do with the physical distance between nodes. The routing table allows any node to find the closest node to any given key efficiently, often in O(logn) network hops . This style of routing is sometimes called key based routing. The routing algorithm that we used in order to achieve DHT functionality is called GISP (Global Information Sharing Protocol).The implementation of GISP is based upon JXTA, which is a set of protocols for building peer-to-peer networks. Both are discussed in the next chapter.

Currently, the idea of distributed indexes is adaptable to many peer-to-peer applications including distributed file systems[10],event notification[4], content distribution [5], e-mail delivery[11], web caches[6],indirection services[15] to name a few. This thesis proposes another P2P application built on top of a DHT and in particular an indexing mechanism for a specific type of persistent data called Entity Enterprise Java Beans (EJB).

1.2 Indexing

Indexing techniques are very well known as a basic feature of any database management system (DBMS). The classic analogy to database indexes is the index in the back of reference books. If we wanted to find everything in the book about a particular subject we could start at the beginning and scan every page, but it is

much faster to look in a smaller, alphabetized subject index that directs us to a list of pages. Then we need to scan only those pages to find information about our chosen subject. Not everything in the book is indexed, however, so if our subject is not mentioned in the index, we must still scan for it. Likewise, a database index is a look-up mechanism that helps a DBMS find the information we request faster than it could with a full scan. As with book indexes, not everything in the database is indexed, so an occasional scan may still be necessary.

The primary reason to build an index is to improve performance. But it is not the only reason to build an index. The second reason has to do with enforcing uniqueness among rows stored in a database table. Tables in a SQL database are usually designed with a primary key; that is, a set of columns with a unique value that identifies a row in the table. When a new row is inserted into a table defined with a primary key, it is up to the DBMS to ensure that the primary key value for that row is unique. Performance would be unacceptable if the DBMS had to scan the entire table each time a new row was inserted. Therefore, the accepted solution is to build a unique index on the primary-key columns and let the DBMS use that as the physical enforcement mechanism for the primary key uniqueness requirement.

Therefore while it's mandatory to build a unique index on a table's primary key (primary index), in order to enforce uniqueness among rows stored in a database table, there are other times when an index is desirable. For example for a query like :

```
SELECT *
FROM Customer
WHERE fname = 'Georgios' ;
```

where column fname is not part of a primary or foreign key, the system would have to read every record and check the *fname* column for the name 'Georgios'. If

this query that accesses the fname column was frequently used, the performance would be poor. To facilitate queries such as this one, we often create one or more *indexes* on a relation. An index is any data structure that takes as input the value of one or more columns and finds the records with that value "quickly", by exporting a small fraction of all possible records that we must check for results. The more indexed columns (secondary index) in a where clause, the more likely it is that a DBMS will be able to use an index to speed up query performance. But there is a trade-off here: Not enough indexes results in slow queries; too many indexes results in slow changes to the database.

There are many different data structures that are used in order to implement indexing techniques.The two basic kinds of indexes are :

- **Ordered indexes**. Based on a sorted ordering of values
- **Hash indexes**. Based on a uniform distribution of values across a range of buckets.The bucket to which a value is assigned is determined by a function, called a *hash function*

No one can be characterized as best solution in any occasion.Rather, each technique is best suited to particular database application, based on the nature of the actual data stored and the queries performed on them. For instance in the case of columns that don't change frequently and the there are commonly "equality" queries on them, hash indexing will probably boost performance.

1.3 Enterprise Java Beans (EJB)

Sun Microsystems' definition of the Enterprise JavaBeans architecture is:

> The Enterprise Java Beans architecture is component architecture for the development and deployment of component-based distributed

> business applications. Applications written using the Enterprise Java Beans architecture, are scalable, transactional, and multi-user secure. These applications may be written once, and then deployed on any server platform that supports the Enterprise JavaBeans specification.[1]

A somehow shorter definition is :

> Enterprise JavaBeans is a standard server-side component model for distributed business applications.

A server-side component model may define an architecture for developing *distributed business objects* that combines the accessibility of distributed object systems with the fluidity of objectified business logic. Server-side component models are used on the middle-tier application servers, which manage the components at runtime and make them available to remote clients. They provide a baseline of functionality that makes it easy to develop distributed business objects and assemble them into business solutions. Therefore the second definition means that EJB offers a standard model for building server-side components that represent both business objects (customers, items in inventory, and the like) and business processes (purchasing, stocking, and so on). Once you have built a set of components that fit the requirements of your business, you can combine them to create business applications. On top of that, as "distributed" components, they don't all have to reside on the same server. Components can reside wherever it's most convenient: a Customer component can "live" near the Customer database, a Part component can live near the inventory database, and a Purchase business-process component can live near the user interface. You can do whatever's necessary for minimizing latency, sharing the processing load, or maximizing reliability.

[1] Sun Microsystems' Enterprise Java Beans Specification, v2.1, Copyright 2002 by Sun Microsystems, Inc.

Enterprise Bean Type	Purpose
Session	Performs a task for a client
Entity	Represents a business entity object that exists in persistent storage
Message-Driven	Acts as a listener for the Java Message Service API, processing messages asynchronously

Table 1.1: Enterprise Bean Types

For several reasons, enterprise beans simplify the development of large, distributed applications. First, because the EJB container provides system-level services to enterprise beans, the bean developer can concentrate on solving business problems. The EJB container –and not the bean developer– is responsible for system-level services such as transaction management and security authorization.

Second, because the beans–and not the clients–contain the application's business logic, the client developer can focus on the presentation of the client. The client developer does not have to code the routines that implement business rules or access databases. As a result, the clients are thinner, a benefit that is particularly important for clients that run on small devices.

Third, because enterprise beans are portable components, the application assembler can build new applications from existing beans. These applications can run on any compliant J2EE server provided that they use the standard APIs.

In our case as an EJB container JBoss, a popular open-source J2EE Application Server was used.

The three types of enterprise beans are discussed in the following table:

1.4 Our contribution

In this current Thesis, we propose a flexible P2P infrastructure that allows an application based on Entity EJB's to be deployed in multiple Application Servers

and form a P2P network, where it is possible to create indexes on the data of EJBs and thereupon implement corresponding finder methods that can locate EJBs, inside the P2P, that hold particular data.

Since Entiry EJB's are an object-oriented representation of data in a persistent store, such as the records in a database, our architecutre actually extendes the idea of indexing over a single database, to a P2P database. The DHT provides the required indexing functionality, over the P2P network, with similar to hash indexing properties and usage.

We outline the design specific characteristics of such an indexing mechanism and provide an implementation that can be easily adopted by an EJB developer on top of his application and through a given API, extend the standard EJB functionality with P2P indexing capabilities.

1.5 Outline

The rest of the thesis is organized as follows:

The next chapter 2, briefly presents the Peer-to-Peer, J2EE Technologies, or other proposed technologies that are relavant to this particular work.

Then in Chapter 3, we present the reader with our proposal regarding the design that allows the indexing of Entity EJB's over a P2P network, regardless of the underlying technologies that can be used during the implementation phase.

Based on the above design in Chapter 4, we present a specific implementation we created, using JBoss as an EJB container and GISP as our DHT.

Finally, in Chapter 5 we present our conclusion.

Appendix A, serves as a user's manual, describing the necessary steps that an EJB developer should perform, in order to make use of our architecture and

therefore extend the functionality of his application with P2P indexing capabilities.

Appendix B, contains the Javadoc pages for our implementation.

Chapter 2

Related Work

This chapter provide a small introduction to each one of the technologies has some kind of relevance to our work, either because we used it or it shares similar ideas. References to the appropriate bibliography is given each time.

2.1 Peer-to-Peer Technologies

2.1.1 GISP

Recently a diverse set of DHT implementations have been proposed, such as Chord [16], CAN [12], Pastry [13], or Tapestry[18]. The implementation we used in order to build our DHT, as mentioned in the introduction, is based on GISP (Global Information Sharing Protocol)[8]. Usually a distributed hash table consists of (key, value) pairs of data that are shared among peers. GISP allows not only (key, value) pairs but also any kind of XML element to be shared and XPath is used to query the XML elements, which was particularly usefull for our purpose.

GISP defines among others how data are inserted, queried and replicated ,how message routing is performed and what local information each peer must store.

The Java interface it provides is the following :

- public void insert(String key, String str);
- public void insert(String key, String str, long ttl);
- public void insert(String key, byte[] xml);
- public void insert(String key, byte[] xml, long ttl);
- public void query(String key, ResultListener l);
- public void query(String key, ResultListener l, long timeout);
- public void query(String key, String xpath, ResultListener l);
- public void query(String key, String xpath,ResultListener l,long timeout);

The fact that it doesn't support data deletion, had an impact in our design as we will discuss in a later chapter. GISP started as a project of JXTA which provides most of the undelying P2P functionality, required to implement the GISP protocol. The main source for information about the GISP project is its homepage *http://gisp.jxta.org/*

2.1.2 Project JXTA

JXTA is an open network computing platform designed for peer-to-peer (P2P) computing. Its goal is to develop basic building blocks and services to enable innovative applications for peer groups. It provides a common set of open protocols and an open source reference implementation for developing peer-to-peer applications. The JXTA protocols, which are defined as a series of XML message formats, standarize the manner in which peers:

- Discover each other
- Self-organize into peer groups
- Advertise and discover network services
- Communicate with each other
- Monitor each other

The JXTA protocols are designed to be independent of programming languages, and independent of transport protocols. The protocols can be implemented in the Java programming language, C/C++, Perl, and numerous other languages. They can be implemented on top of TCP/IP, HTTP, Bluetooth, HomePNA, or other transport protocols. They enable developers to build and deploy interoperable P2P services and applications. Because they are independent of both programming language and transport protocols, heterogeneous devices with completely different software stacks can interoperate with one another. Using JXTA technology, developers can write networked, interoperable applications that can:

- Find other peers on the network with dynamic discovery across firewalls
- Easily share documents with anyone across the network
- Find up to the minute content at network sites
- Create a group of peers that provide a service
- Monitor peer activities remotely
- Securely communicate with other peers on the network

The Java programming language API was used to access operations supported by these protocols. A Programmers Guide along with other useful material about Project JXTA can be found in the following URL : *http://www.jxta.org/*

2.2 J2EE technologies

2.2.1 EJB Architecture

Enterprise Java Beans have been introduced in the previous chapter. However here we will provide a brief overview of the EJB architecture [3]. The EJB architecture endows enterprise beans and EJB containers with a number of unique features that enable portability and reusability:

- Enterprise bean instances are created and managed at runtime by a container. If an enterprise bean uses only the services defined by the EJB specification, the enterprise bean can be deployed in any compliant EJB container. Specialized containers can provide additional services beyond those defined by the EJB specification. An enterprise bean that depends on such a service can be deployed only in a container that supports that service.

- The behavior of enterprise beans is not wholly contained in its implementation. Service information, including transaction and security information, is separate from the enterprise bean implementation. This allows the service information to be customized during application assembly and deployment. The behavior of an enterprise bean is customized at deployment time by editing its deployment descriptor entries. This makes it possible to include an enterprise bean in an assembled application without requiring source

code changes or recompilation.

- The Bean Provider defines a client view of an enterprise bean. The client view is unaffected by the container and server in which the bean is deployed. This ensures that both the beans and their clients can be deployed in multiple execution environments without changes or recompilation. The client view of an enterprise bean is provided through two interfaces(Home and Remote). These interfaces are implemented by classes (enterprise bean classes) constructed by the container when a bean is deployed, based on information provided by the bean. It is by implementing these interfaces that the container can intercede in client operations on a bean and offer the client a simplified view of the component. Figure 2.1 illustrates the implementation of the client view of an enterprise bean.

  Besides the EJB specification an excellent reading about EJB's and J2EE in general is [9]

2.2.2 JBoss Application Server

JBoss's primary goal is to provide a full J2EE-based implementation. JBoss consists of JBossServer, the basic EJB container, and Java Management Extension (JMX)[1] infrastructure. It also provides JBossMQ, for JMS messaging, JBossTX, for JTA/JTS transactions, JBossCMP for CMP persistence, JBossSX for JAAS based security, and JBossCX for JCA connectivity. Support for web components, such as servlets and JSP pages, is provided by an abstract integration layer. Implementations of the integration service are provided for third party servlet engines like Tomcat and Jetty.

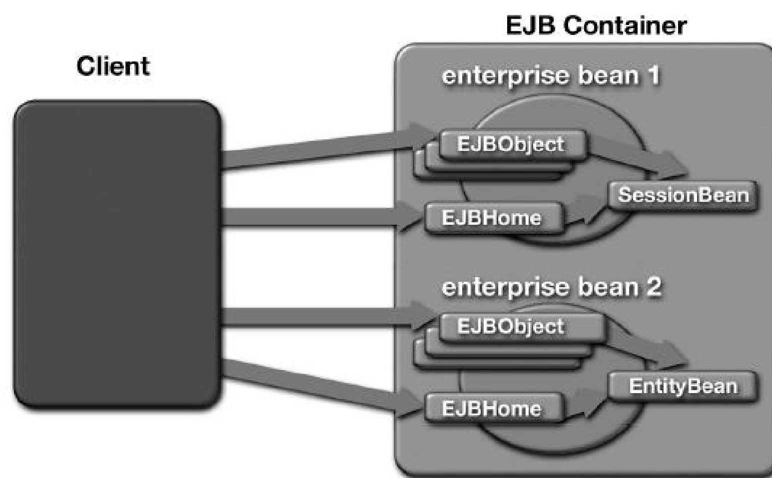

Figure 2.1: Implementation of Client View of Enterprise Beans

JBoss enables you to mix and match these components through JMX by replacing any component you want with a JMX compliant implementation for the same APIs. JBoss features depends on JMX or Java Management Extension, which is an ideal solution for software integration. JMX provides a common spine or a bus through which the components(modules, containers, and plug-ins) of the JBoss architecture interact. Components are declared as MBean Services that are then loaded into JBoss. The components may subsequently be administered using JMX.

Java Management Extenstions (JMX)

JMX architecture [1] defines three levels. The level closest to the application is called the *instrumentation level*. This level consists of four approaches for instrumenting application and system resources to be manageable (i.e., making them *managed beans*, or *MBeans*), as well as a model for sending and receiving notifications. The middle level of the JMX architecture is called the *agent level*. This level contains a registry for handling manageable resources (the *MBean server*) as well as several agent services, which themselves are MBeans and thus are manageable. The third level of the JMX architecture is called the *distributed services* level. This level contains the middleware that connects JMX agents to applications that manage them (*management applications*). An MBean is a Java object that implements one of the standard MBean interfaces and follows the associated design patterns. The MBean for a resource exposes all necessary information and operations that a management application needs to control the resource.

The four types of MBeans are :

- **Standard MBeans**: These use a simple JavaBean style naming convention and a statically defined management interface. This is currently the most common type of MBean used by JBoss.
- **Dynamic MBeans**: These expose their management interface at runtime when the component is instantiated for the greatest flexibility. JBoss makes use of Dynamic MBeans in circumstances where the components to be managed are not known until runtime.
- **Open MBeans**: These are an extension of dynamic MBeans.
- **Model MBeans**: These are also an extension of dynamic MBeans. Model MBeans simplify the instrumentation of resources by providing default behavior. There is a Model MBean implementation used by JBoss known as an XMBeans.

Our implementation is based on the construction of an XMBean, who mediates between the EJB and the DHT. Most of the material covered here and much more about JBoss can be derived from [14].

2.3 Other Projects

2.3.1 PROST

PROST[2] is a programmable infrastructure based on the key-based routing layer of a structured P2P network. Applications and services can be deployed in PROST by dynamically loading code modules onto nodes of the P2P overlay. These code modules, which we call peerlets, implement the application-specific functionality, while making use of the efficient lookup

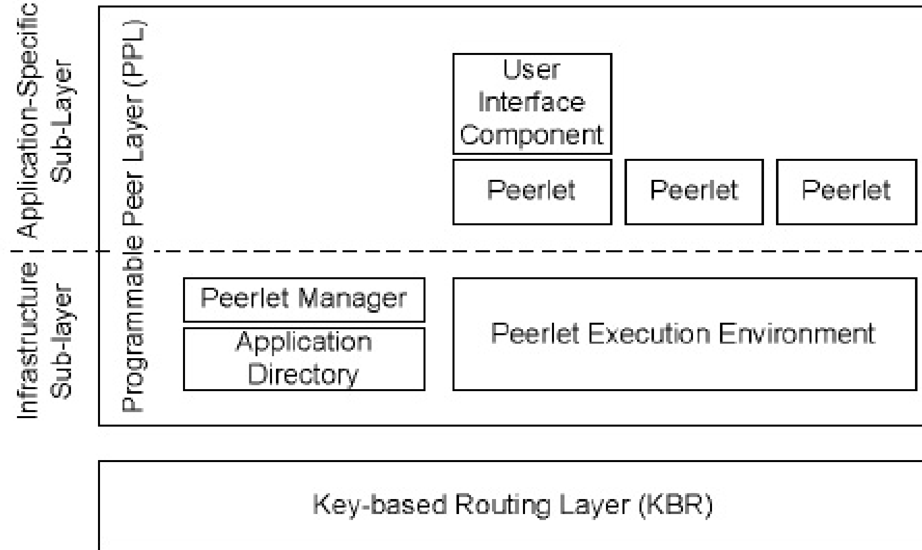

Figure 2.2: Node architecture

facilities of the shared KBR layer. Figure 2.2 illustrates the architecture of a programmable peer node in PROST.

Our design shares similar ideas with PROST in the sence, that both try to integrate the use of an Application level software, such as an Application Server with a Key-based Routing mechanism, such as a DHT.

2.3.2 pSearch

pSearch [17] is an effcient peer-to-peer information retrieval system, that supports state-of-the-art content- and semantic-based full-text searches. pSearch avoids the scalability problem of existing systems that employ centralized indexing, or index/query ooding. It also avoids the nondeterminism that is exhibited by heuristic-based approaches. In pSearch, documents

in the network are organized around their vector representations (based on modern document ranking algorithms) such that the search space for a given query is organized around related documents, achieving both effciency and accuracy.

2.3.3 Data Indexing in Peer-to-Peer DHT Networks

This paper [7], describes techniques for indexing data stored in peer-to-peer DHT networks, and discovering the resources that match a given user query. The system creates multiple indexes, organized hierarchically, which permit users to locate data even using scarce information, although at the price of a higher lookup cost. The data itself is stored on only one (or few) of the nodes. Experimental evaluation demonstrates the effectiveness of the indexing techniques on a distributed P2P bibliographic database with realistic user query workloads.

2.3.4 Coral

Coral *(http://www.coralcdn.org/)* is peer-to-peer content distribution network, comprised of a world-wide network of web proxies and nameservers. It allows a user to run a web site that offers high performance and meets huge demand, all for the price of a $50/month cable modem.

Publishing through Coral is as simple as appending a short string to the hostname of objects' URLs; a peer-to-peer DNS layer transparently redirects browsers to participating caching proxies, which in turn cooperate to minimize load on the source web server. These volunteer sites that run

Coral automatically replicate content as a side effect of users accessing it, improving its availability. Using modern peer-to-peer indexing techniques, Coral will efficiently find a cached object if it exists anywhere in the network, requiring that it use the origin server only to initially fetch the object once.

One of Coral's key goals is to avoid ever creating hot spots that might dissuade volunteers from running the software for fear of load spikes. It achieves this through a novel indexing abstraction we introduce called a distributed sloppy hash table (DSHT) [6], and it creates self-organizing clusters of nodes that fetch information from each other to avoid communicating with more distant or heavily-loaded servers.

2.3.5 OceanStore

OceanStore [10] is a utility infrastructure designed to span the globe and provide continuous access to persistent information. Since this infrastructure is comprised of untrusted servers, data is protected through redundancy and cryptographic techniques. To improve performance, data is allowed to be cached anywhere, anytime. Additionally, monitoring of usage patterns allows adaptation to regional outages and denial of service attacks; monitoring also enhances performance through pro-active movement of data. A prototype implementation is currently under development.

Chapter 3

P2P indexing of EJB: The Design

In this chapter we will present our P2P indexing mechanism for Entity EJB's. Our design will be regardless of the underlying technologies that will be used during the implementation phase. The analysis identifies the basic components needed, specify their responsibilities and demonstrate how the components connect to each other, in order to achieve the desired functionality. We begin our discussion by defining the exact nature of P2P indexing of EJB.

3.1 Defining P2P Indexing of EJB's

From the beginning of this thesis the term *P2P indexing of Entity EJB* has been used several times. However we haven't really analyzed the meaning of this term and what actions are required in order to achieve it. Therefore the most appropriate way to begin is by clarifying what the term actually means. A simple answer is that it stands for the existence of indexes on the value of a single(or combined) Entity EJB attribute(s). The EJB

Operation	**Purpose**
put/insert(key,value)	Stores the(key,value)pair to the appropriate peer, based on the hashed key
get/lookup(key)	Locate the peer(s) holding the (key,value) pair, by hashing the key

Table 3.1: The DHT API

developer will somehow define for which single (or combined) attributes he wants indexes to be created for their values.

Moreover, the P2P part in our definition means that the underlying structure that will be used to store the index is actually a P2P network. A traditional hash indexing mechanism, would hash the indexed attribute value(key) and the result would give the position in the hashtable, this key must be stored in. Afterwards when someone tries to find the key, he will again hash the key and the result will be the correct hashtable position, if it exists. In our case the notion of a hashtable position is replaced by a peer in the P2P network. The DHT used will provide the hashing functionality over the P2P network. As explained earlier, nodes (peers) in a DHT are organized in a specific network overlay and each node has a logical identifier that determines its logical position in the overlay. Then through the standard put/get operations, all DHT's provide, they can by hashing a key, store the key and retrieve it, in and from the appropriate peer, respectively. Table 3.1 summarizes the API available by all DHT's.

To conclude however, our discussion here, we must make some final remarks.

- Defining an index on a value entails three basic responsibilities:

  1. When a new value is created, the index must be populated with this value.

2. When a value is changed or deleted again the index must be updated accordingly.
3. When we have a query about an indexed value, the index is used in order to find the results.

In order to achieve these properties, our design makes use of the appropriate EJB call-back methods (*ejbStore*, *ejbRemove*)and some user defined finder methods, in a way that will be explained later.

– Entity EJB's are an object-oriented representation of data in a persistent store, such as the records in a database. During the deployment phase the Beans attributes will be mapped to specific database table attributes. However attributes are not directly accessed. Instead methods declared as part of the Bean's Home or Remote Interface, are used. These methods execute in a transactional manner (their invocation eventually is translated into some kind of SQL statement), which in turn means that only when the database "commits", the requested changes take place. In order for our indexing mechanism to be able to synchronize its state with this of the database, another Session EJB will be used. This Session Bean extends Interface *SessionSynchronization.*

3.1.1 Transactional issues

Here we must add some theory about EJB Transactions and the above Interface in order to understand what it is and why it solves our problem. In an enterprise bean with container-managed transactions, the EJB container sets the boundaries of the transactions. When deploying a bean, you specify

which of the bean's methods are associated with transactions by setting the transaction attributes (in our case we assume that all methods are associated with the *Required* attribute, which is the strickest requirement). A transaction attribute controls the scope of a transaction. Figure 3.1 illustrates why controlling the scope is important. In this figure, method-A begins a transaction and then invokes method-B of Bean-2. If method-B has also a *Required* transaction attribute then they are both executed within method's A transaction (TX=TX1). Therefore in our design an Entity Bean method, with *Required* transaction attributes calls a method in our Session Bean which is also declared as *Required*and as so the second method is executed in the same transaction with the first.

On the other hand the SessionSynchronization interface allows a Session Bean instance to be notified by its container of transaction boundaries. It provides three standard methods. The afterCompletion(boolean committed) method, which interest us, notifies a session Bean instance that a transaction commit protocol has completed and tells the instance whether the transaction has been committed or rolled back(by setting the value of parameter committed to true or false respectively).

Combining both observations it becomes obvious that the Session Bean is actually notified, if the transaction started by an Entity's Bean method has committed. Only if so we will perform the actions needed.

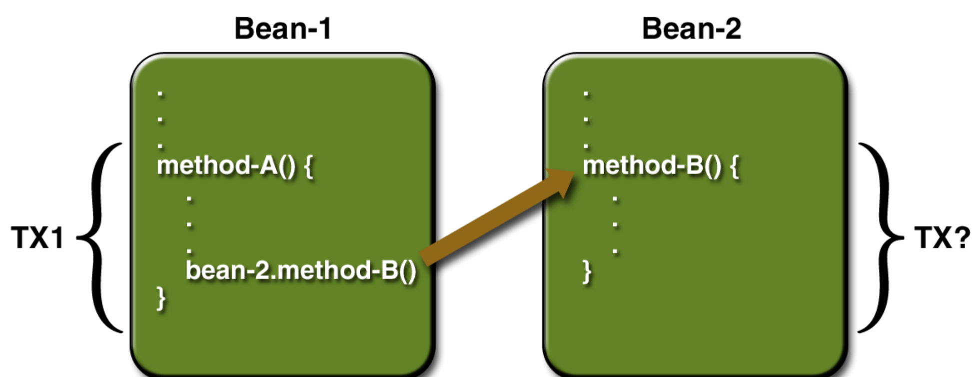

Figure 3.1: Transaction Attributes

3.2 Basic components

Having said all of the above we can now identify the basic components that participate in order to build our indexing mechanism. These are:

- Obviously the textbfEntity EJB whose attribute(s) values are those that we want the indexing to be performed. Container Managed Persistence is assumed as the most general case.
- The **Session EJB**, that extends the Synchronization Interface in order to synchonize the index with the actual database state.
- The **EJB container**. The Container besides the obvious responsibility to provide the enviroment where the above mentioned EJB's will be deployed, plays another very important role in our design. It provides an internal mechanism(some kind of software component)that will be responsible to maintain an active "connection" to the underlying DHT and delegate the EJB's methods invocations that want to access the DHT, to it. Therefore it will act as a mediator or a server that the

EJB instances will have to send their calls, through an appropriate API, and from where the results of their calls will be returned back to them. Moreover in order to cover the case of a System failure, this component must store the indexed values. Then, during the recovery phase a Boostrap operation must be invoked and insert back into the P2P network, previously inserted values. How the EJB container will provide the above functionality, clearly depends on the Container that will be used and what features he offers.

- A **database**, as the persistent storage, the attributes values are stored. Support for Transactions was assumed, in our design
- An **API**. This is actually the various methods defined in the Entity EJB Home and Remote Interface.
- A **DHT** implementation that is used to access the P2P network, through its standard API methods. However implementation specific features of the API will determine the nature of the key,value pair

3.2.1 Container issues

Before we continue we must make a notice about the EJB containers in general. The EJB specification (until version 2.1) defines a contract between an enterprise Bean and its container. This component contract describes, through a set of Interfaces, the responsibilities of the container, in regard to the required set of methods that must be implemented along with the desired behaviour. However the actual implementation is left to the EJB container. Therefore it provides great flexibility to each EJB container vendor, to choose how he will internally implement the various methods

and when and how they will be called, by him.

On the contrary it sets certain constraints in our design, as it must reflect the most general case. Because, although the end results of each method invocation is the one the component contract dictates, the internals may change the order certain operations are performed. As so it becomes essential to place our code carefully in order to eliminate the danger of incorrect execution, based on vendor specific implementation. Even the Configuration options inside the same EJB container can completely change its behaviour, causing some functions not to be executed and so on (for example declaring a specific cache policy in JBoss can cause *ejbLoad* not to execute). The methods particularly affected by this are the call-back methods that notify the bean class of life-cycle events. At runtime, the container invokes these methods on the bean instance when relevant events occur.

3.3 The Indexing mechanism

The final step in order to complete our design is defining the way the different components are connected, in order to perform the indexing. To achieve this we will follow the reverse course and discuss step by step how the indexing mechanism functions. This discussion will eventually unveil the aforementioned connections.

3.3.1 Insert mechanism

Part of the indexing mechanism resides inside the *ejbStore* call-back method. This method is called when the container is about to write an entity bean

instance's state to the database. That is, after a client application has called a *create* or *setter* method, although practically it is called after a *getter* too. Each time *ejbStore* is called, our mechanism checks if an indexed attribute value is created or updated. If so, the new or updated value along with a value that identifies the EJB instance this value belongs to, are sent through an insert method to an instance of our Session Bean.

The Session Bean implements the necessary Interface, in order to be notified if the transaction that called it committed. If so it will call the appropriate Container component method which will in turn process the attribute(s) value and the identifier. Thereupon, it will pass the processed values as parameters to the DHT put/insert method. Moreover the Component must somehow store the indexed pair, in case of a System failure, as mentioned earlier. Notice here that whether we have a new or updated value, method put/insert is invoked in the DHT. DHT's don't actually have an update method. A validation mechanism must be created in order to check whether the results of a finder method are correct.

3.3.2 Delete mechanism

We would like for a procedure similar to the above to be performed also whenever a *remove* method is invoked. Again DHT don't necessarily define a delete method and a validation mechanism is used to check that the bean exists. However since as said before the Container component stores indexed values, a proper delete function is provided by the Session Bean. In case the transaction committed a corresponding method of the Container Component will be called and erase this Beans values from its local storage.

As so we prevent the event, of a Bootstrap method, inserting back into the P2P network, previously deleted pairs.

3.3.3 Lookup mechanism

The reason we create indexes all along is in order to have finder methods that can make use of them. The corresponding to the indexes finder methods must be able to take as parameters specific attribute values and return the Remote References of those EJB's holding such value. The intermediate steps executed are:

1. The finder method collects the parameters and passes them to the appropriate Container Component method
2. The Component processes these parameters and creates a search key
3. This key is then passed as a parameter to a get/lookup DHT method
4. The results are returned, again through the Component, to the EJB instance that called the finder method
5. The returned values contain the necessary information, as mentioned earlier, to identify the specific EJB that holds this (key,value) pair
6. The validity mechanism checks if indeed the EJB's value is valid and hasn't changed or deleted. If so its Remote Interface is returned as a result

To conclude our design, Figure 3.2 gives us a graphical reprsentation of the above procedure:

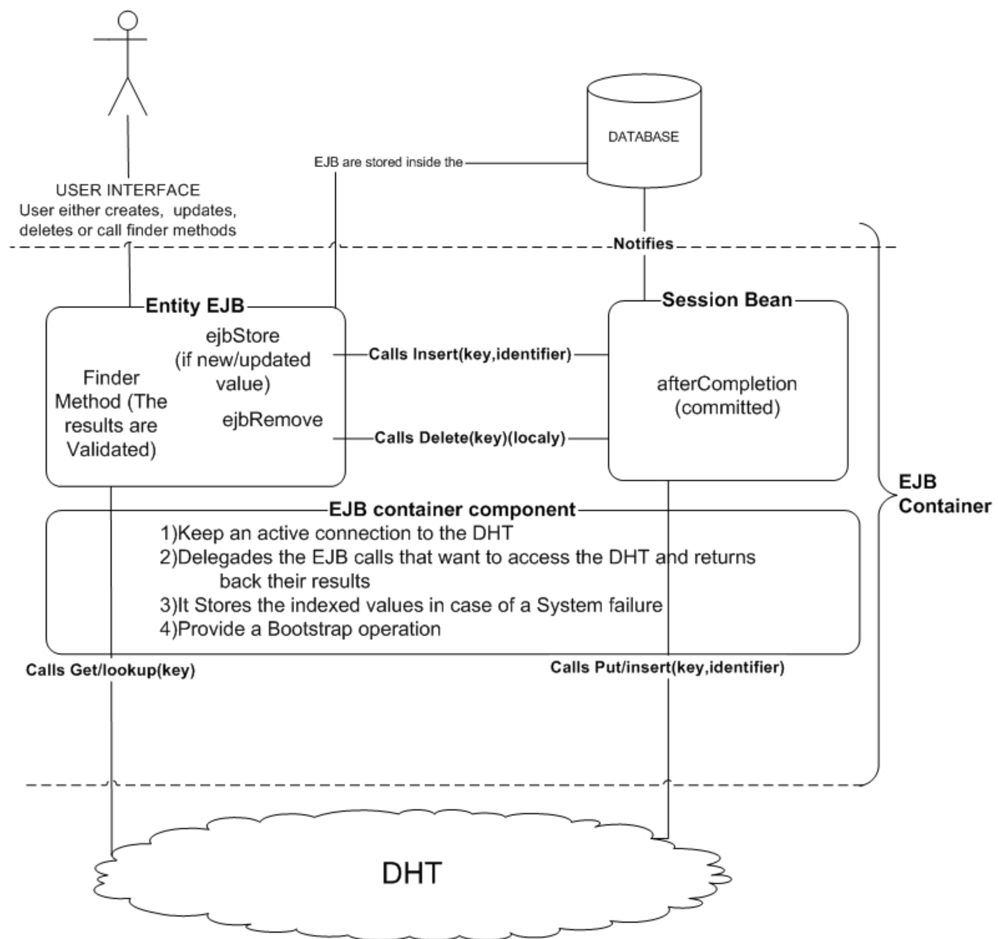

Figure 3.2: Indexing mechanism

Chapter 4

P2P indexing of EJB: The Implementation

As the title suggests in this chapter we will present an actual implementation, based on our Design, produced for the purpose of this current Thesis. The design identified the basic components that participate, in order to create the indexing mechanism. Among them, the EJB(Entity and Session) along with the database system, provide a well defined API, regardless of the implementation. As so the specific features of the remaining two components, are those that determine our implementation.

4.1 DHT : GISP

As mentioned several times the DHT implementation we used is called GISP. GISP's standard API provides an *insert(String key, byte[] xml),long ttl* function where the *xml* parameter is an XML style document. The interesting feature of GISP is that the corresponding *query (String key, String xpath, ResultListener l,long timeout)* method can actually provide, besides the standard search key and an XPath expression (*xpath*) that is

used in order to search inside the *xml*, for results. The *Result Listener* returns the result to the funtion any time an answer is found and *long* sets for how long he must search for results. Our implementation accessed GISP always through these two functions. A standard XML format, that will be discussed later, was used for the *xml* part. Why this was useful will become soon aparent.

Another point we must mention is the fact that GISP doesn't have a delete mechanism. The reason most DHT's don't have such a mechanism has to do with data replication. When an insert function is called the DHT sends the data to the appropriate peers that must hold the particular pair, along with some other peers, in order for data to be accessible, if any of them goes down. This means that the DHT looses control of the data and therefore it isn't possible to relocate them somehow, in order to update or delete them. The way it tries to solve this problem, is by setting times to live(ttl) for the inserted data. When data expire they will be removed from the DHT(all peers who keep them, will delete it from their local DHT store).

However what we did is extend GISP in order to delete data from the local DHT storage, in case they where deleted from the database for example. The idea was that if each peer deletes the data he knows are incorrect from his local storage, eventually the system load to process incorrect results and the network load to transfer them among the peers is reduced. Using the same mechanism GISP uses to query for results, based on an XPath, we could trace the appropriate results that needed to be removed. Inserting all data with appropriate information stored in the *xml*, makes this process work. Method *delete(String xpath)* with the above functionality, was added

to GISP API.

4.2 EJB Container : JBoss

The EJB Container we used in our implementation was that provided by JBoss, a very successfull Open Source J2EE Application Server. JBoss is much more than just an EJB Container. As explained the JBoss architecture is based on JMX. Everything in JBoss is actually a Managed Bean(MBean). The EJB container itself is an MBean. Jboss's role is similar to that of an MBean server on the JMX architecture, that is to register all the resources available and allow them to communicate with each other. Therefore, the functionallity a specific JBoss instance provides, at any time, is based on the deployed MBeans. Obviously, it is possible for anyone to develop an MBean and deploy it in JBoss. The methods your MBean provides can be accessed either by other MBeans inside the JBoss or from the outside, through proper adaptor(currently JBoss employs an HTTP and an RMI Adaptor).

Using the above JBoss characteristic, we creted an MBean (DHTxmbean) as the Server Component or the mediator between the EJB instances and the DHT, our design imposed. Because it is possible for the EJB's deployed into the JBoss to access our MBean, through the RMI adaptor available, this approach is suitable for our problem. To be more precise we created an XMBean which is a JBoss implementation of a Model Bean. The extra feature XMBeans have is that its attributes and methods can be described through XML style documents.

Each MBean provides standard life-cycle operations. Methods *startService* and *stopService* are automatically invoked every time an XMBean is deployed or undeployed (if it is removed), respectively. We used this particular feature, by placing the appropriate code to connect to GISP(in order to become part of it) along with the Bootstrap method inside the *startService* method. Furthermore each time DHTxmbean is deployed it establishes the required connection to GISP and if necessary inserts previous data in it.

Thereupon DHTxmbean through the methods it provides(and that we will see later in details) can be accessed from the EJB instances, using the RMI adaptor and delegate insert or query requests into and from GISP.

4.3 Implementing the Indexing mechanism

Using the above as a basis, in this section we will follow the exact same step-by-step presentation we used in the previous chapter(section Indexing mechanism), and analyze the methods invoked and their underlying functionality. Figure 4.1 is based on Figure 3.2, with the difference that the specific components and the exact method invocations performed, in our implementation, are indicated.

4.3.1 Insert mechanism

Specify indexed attributes

Suppose a user called a create, set or get method causing method *ejbStore* to be invoked. The first action *ejbStrore* does is call method *indexed_data()*

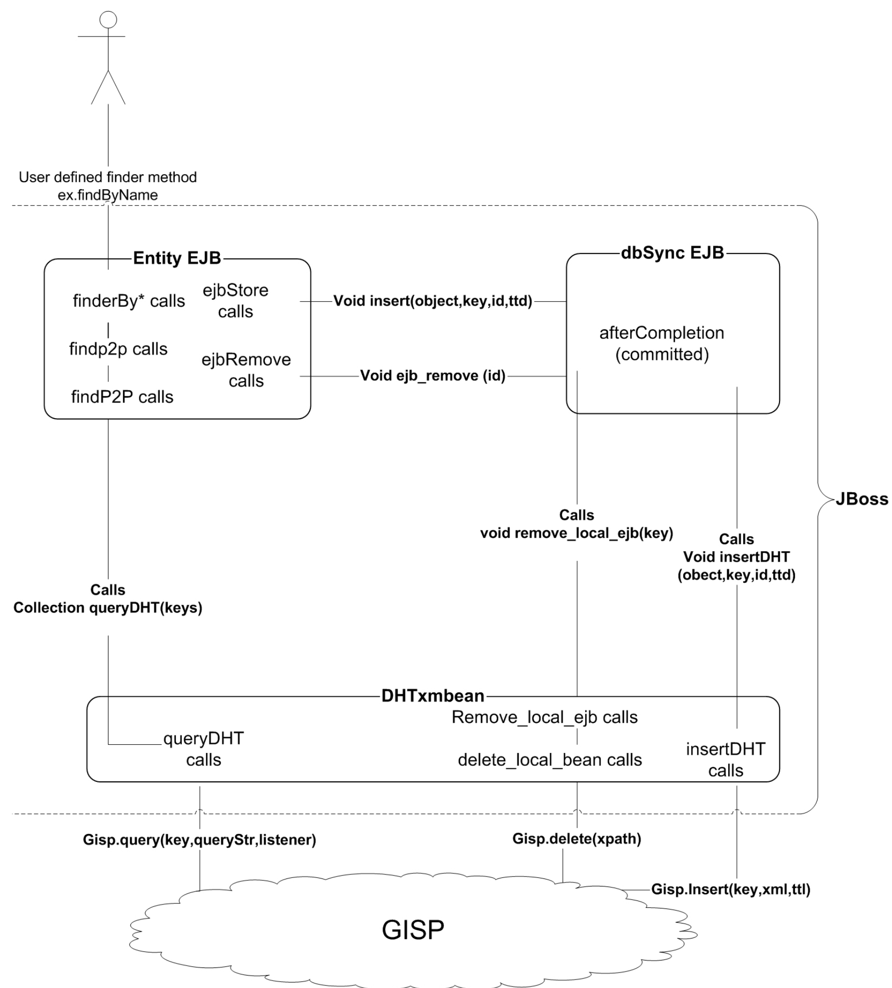

Figure 4.1: Indexing mechanism Implementation

which is defined by the EJB developer in order to create an array with (identifier,value) pairs. Identifier is a String name used to distinguish the various single or combined attributes that we want to create an index for and value is either their actual value(a String representation of it), that we get through getter methods, or a reserved String value in case any of the values was NULL(not set). For those pairs, a value actually existes *ejbStore* creates a Session Bean instance and calls method *insert* of its Remote Interface. The parameters passed in it are a concateneted String of the Bean's EJBObject and the identifier called object, the actual value, as a String, that we want to be indexed (key) , the primary key(String representation) of the Bean (id) and the time this indexed value must expire(ttd).

Synchronize to Database

The *dbSync*(our Session bean) instance temporary stores these values. The *SessionSyncronization* Interface that it extends provides method *afterCompletion(boolean committed)* that is automatically invoked by the Container to notify whether the transcation, the invoked method belongs, has committed(committed ==true). Inside its body we check this value and if its true, through the RMI adaptor we call DHTxmbean's method *insertDHT* with the same parameters we got.

XMBean Hashtable

DHTxmbean in order to maintain the indexes that will be stored, in case of a System failure as we said before, creates a temporary hashtable with (object,class Entry instance) pairs. The object uniquely identifies each one

of the hashtable elements while instances of class Entry store the primary key, id and ttd for each indexed value. Moreover, every time the hashtable changes, it is serialized into a file that will be used by the *Bootstrap()* function. Furthermore, the fact that object can uniquely identify the exact bean and the exact attributes(single or combined) this Entry belongs to, allows us to use the hashtable as a mechanism to identify whether the indexed value is new or not. Because, if an insert request arrives and its object exists inside the hashtable that means that the same Entity EJB has previously inserted a value for the same set of attributes.

Insert into GISP

Based on the above, inside method *insertDHT* our first step is to check whether object exists into our hashtable :

– If not, it means this is a new value(create or set method). We create a new Entry and store the (object,Entry) pair into the hashtable. Then we call method *create_value(primary key, id).* This method creates the required XML structured value we will pass as a parameter to GISP. The format we use is the following:

```
<item>
<key>The indexed value <\key>
<id>The EJB's primary key as a String<\id>
<ip>The ip of the machine that the bean is deployed<\id>
<\item>
```

The reason we store these particular values will become apparent later.

Thereupon, GISP's method *insert* is called with parameters the indexed value , the XML structured value and a time-to-live(ttl) value (time-to-die(ttd) - CurrentTime).

– If the hashtable contains the object and the key(indexed value) is the same to both of them, that means the method that called *ejbStore* was a getter and as so no action is performed.

– If finally the object exists and has a different value this means the method was a setter. We remove the previous (object,Entry) pair from the hashtable, call method *delete_local* that will be discussed later and follow the exact same procedure as if the object was a new value.

If the hashtable has changed during the above process it is serialized to the proper file. In case now of a *Bootstrap* operation during deployment time, the function deserializes the proper file and for every hashtable entry it checks to see if the value hasn't expired and if so it follows the process of inserting a new value into GISP. Notice here that the informations stored in each Entry are those needed in order to check if the value has expired(if not calculate new ttl) and call method *create_value* to construct the XML value.

4.3.2 Delete mechanism

Method *delete_local* that was mentioned earlier uses the delete mechanism we presented in section DHT :GISP. It takes as parameters the key and id of a value we know that exists into the DHT, since it exists into the hashtable. Inside the body an XPath expression is created with the following format:

```
/item[key='The indexed value' and id='The beans Primary key' and
ip='The machine this bean is deployed']
```

This String is passed as a parameter in our *delete(xpath)* GISP method which in turn locates the local, based on the ip, specific indexed values and remove them from the local DHT storage, which was our objective.

The delete mechanism must also be invoked and in case a Bean is removed. Then mehtod *ejbRemove* is invoked which passes the Primary key of the Bean to dbSync's method *ejb_remove*. If the transaction commits, that is the entry was deleted from the database, through the *afterCompletion* method we invoke DHTxmbean's method *remove_local_ejb*. This method removes from the local hashtable all the elements with the same Primary key field, that is all the indexes this Bean has inserted into GISP and calls method *delete_local_bean* with only parameter the Primary key. The logic of this method is the same with that of method *delete_local*. It creates an XPath expression with the format:

```
/item[id='The beans Primary key' and
ip='The machine this bean is deployed']
```

and passes it to GISP's *delete(xpath)* method. All the local, based on the ip, pairs that have been inserted into GISP form the specific EJB, based on the id , will be located and removed from the local DHT storage.

4.3.3 Lookup mechanism

Having described the process of inserting and deleting items from GISP it remains to analyze the process of locating correct Remote References

based on a given search key. Obviously a corresponding finder method will be available to the user from the Bean's Home Interface, for each set of attribute values, single or combined, that have already been indexed. This method will take as parameters the attribute's values that he wants to find(search key analogy) and that can be of any type and will return the to user that called it, a Collection(a Vector) with Remote References of EJB's that hold the search key.

Besides the finder method the EJB developer must also implement a corresponding check method, as part of the Beans Remote Interface. Each check method takes as a parameter a String value and returns the result of the equality check with the concatenated String representation of the current attributes values, participating in the finder method(through the use of the appropriate getters).

Query GISP

To achieve all of the above the first step is to query the DHT for possible correct results. The finder passes it's parameters to a findp2p method, which creates a single array of Objects and passes it to method *findP2P(Object[] keys)* . DHTxmbean's method *queryDHT(Object[] keys)* is then called. This method concatenates the String representation of each value into a single String Search key, that is used passed to GISP's query method. For each one of the results we get we extract the key, id and ip values from the XML value part and check if the same value has already been returned(from a peer which has replicate the value). If so we do nothing else we add these three values in the same order into a Vector and a concatenated version of

them into another Vector that is used to perform the initial check. After a period of 10 sec, which can be tuned, we return to *findP2P* the Vector with the results.

Validate results

Now starts the Validation mechanism which has two phases. Phase one is executed inside the *findP2P* method. Using the ip we connect with the appropriate JBoss instance and get a *jndiContext* which is used to locate the class method and as so get a Home Interface of the appropriate EJB. Afterwards we use the *findByPrimaryKey* each Entity EJB provides, passing it the id we got as a result and then add the Bean's Remote Reference that is returned into a Vector along with the key. The Vector with the References is returned through *findp2p* to the appropriate finder method, that was initially invoked.

In the second phase of the validation process the finder method calls for each one of the Remote References the corresponding to the finder, check method with the key as a parameter. If the method returns true, the specific EJB, indeed holds the correct value and the Remote Reference is added to the Vector the finder returns.

Chapter 5

Conclusions

In this thesis, we outlined the idea of a P2P indexing machanism for Entity EJB's. The use of Distributed Hash Tables (DHT) for such an application is a new concept. Therefore our main purpose was to identify the basic design principles, characteristics and limitations such a system has. The greatest challenge was to allow the P2P network act as the underlying structure the actual indexing would be performed, while at the same time preserve its basic functionality. Although DHT provide a very efficient mechanism to store and retrieve data over the P2P network it can not guarantee their consistency .

Moreover the fact that Entity EJB's represent data stored in a persistent storage, dictates that our indexing mechanism must be Synchronized at any time with the persistent storage.

Our design however manages to address both issues, in a simple and easily adopted manner. Based on our design we implemented such an indexing mechanism and verified it's efficient use. It is therefore possible for an EJB developer to adopt our design and be able to easily extend the functionality

of his EJB application. Selected Entity EJB attribute(s) can be automatically indexed over a P2P network and as so used answer queries based on their values. The design is such that the programming effort required is not a discouraging factor.

Appendix A

User's Manual

The following pages act as a user manual for the EJB developer who aims to extend the functionality of his EJB application by providing P2P indexing capabilities. We provide step-by-step instructions regarding the architecture "setup" procedure along with detailed examples of how the programmer can create data indexes and the corresponding finder methods. The material that will be used throughout this manual, including the jars and source code files exist in a single zip file named p2pindex.zip, which can be extracted anywhere. The files and folders, inside the p2pindex file and their usage are described briefly in Table A.1.

A.1 Prerequisites

In order to make use of the architecture certain programs must be properly installed into your system:

1. Sun JDK 1.4+ or higher must be installed and the necessary environmental variables must be set according to the OS in use to ensure that the java executables are in your CLASSPATH. During our tests JDK

folder	**Contains**
DHTxmbean	Source code for the XMbean that performs the P2P indexing Running ant inside the directory will build DHTxmbean.sar file and copy it into the JBoss deploy directory
dbSync	Source code for the EJB that performs the database synchronization Running ant inside the directory will build dbSync.jar file and copy it into the JBoss deploy directory
exampleBean	Source code of an example EJB Running ant inside the directory will build customer.jar file and copy it into the JBoss deploy directory
Gisp	Source code from altered GISP platform (with delete function)
jars	Contains the sar and jar files described above
lib	Jar files that must be copied into specific JBoss lib directories

Table A.1: p2pindex file structure

versions 1.4.2(02-05-07) were used. All relevant material is located at the following URL: Sun's Java official site

2. Apache Ant building (versions 1.5.4, 1.6.2) tool was used in order to easily build the source code and run the examples. Make sure you install Ant and set the necessary environmental variables. All relevant material is located at the following URL: Apache Ant official site

3. JBoss Application Server must be properly installed and the necessary environmental variables must be set. It is important to notice that our architecture can only work with JBoss versions 4.0.1 or higher, due to a feature (the ability to transfer a bean's transactional context between Jboss instances) missing in earlier versions. During our tests the latest version available, JBoss 4.0.2, was used. All relevant material is located at the following URL: JBoss official site

A.2 Step 1: Platform Setup

After you have correctly installed the above programs the "setup" procedure of our architecture is a very easy task. The first action that must be performed is copy all the jar files, which are necessary for our platform to function, from the *lib* directory of file p2pindex, into the following directories:

- $JBOSS_HOME/lib
- $JBOSS_HOME/client
- $JBOSS_HOME/server/default/lib/ (JBoss default configuration assumed)

where $JBOSS_HOME is the directory JBoss was installed. Then all you have to do is copy the files DHTxmbean.sar (JBoss XMbean) and dbSync.jar (Session EJB) into the JBoss deploy directory (typically $JBOSS_HOME/server/d
if you use the default configuration, like we assume you do). This is JBoss's hot-deployment directory and any file (jar,sar,ear etc) that is dropped in this directory is automatically deployed in JBoss. To achieve that you can:

- Copy the two files directly from the *jars* directory of file p2pindex into the JBoss deploy directory
- Use the Ant Build files (build.xml) provided. Open a shell (or command prompt) and go to where you have unzipped the p2pindex file. Change into directories *DHTxmbean* and *dbSync* and each time run the *ant* command. If everything worked fine the *ant* command will build the source codes and create files DHTxmbean.sar and dbSync.jar respectively, which then will be automatically copied into the

$JBOSS_HOME/server/default/deploy directory.

Notice: The Ant Build files (build.xml) might not work properly as is, under the Unix environment. The problem is with the declaration of property ”environment”, inside the build.xml file that is used to declare the ”jboss.home” property. In order for the Build file to work, replace the value inside the brackets ($env.JBOSS_HOME), in line
¡property name=”jboss.home” value=”$env.JBOSS_HOME”/¿
of every build.xml file, with the absolute path the JBoss Server is installed.

A.3 Step 2: Extend your EJB

Now that we have the basic components properly working the next step is to start expanding the EJB code in order to be able to take advantage of our platform capabilities. At this point we have to mention that a motivating factor behind this particular work was the widely adoption of EJB technology for the development of Enterprise applications. However the extensive use results in greater complexity for the EJB's already deployed and as so the task of modifying the source code must be very cautious. Therefore whether our implementation was intended to be used in an already deployed application or an application built from scratch it was essential to design our architecture in such a way that would require minimum effort on behalf of the EJB developer/maintainer.

We succeeded in this by organizing large portions of code in functions that can be added without any modification in the newly or already developed EJB. Nevertheless, it is obvious that there are limitations in the flexibility

and integration the code can reach. The programmer will have to write code of his own in order to perform specific tasks, yet even there we believe the task doesn't entail complex programming operations.

From now on and for the rest of this manual in order to demonstrate the way an EJB can be expanded we will use an example bean we have created for this purpose. The bean is called *customer* and is a typical Entity CMP (Container -Managed -Persistence) EJB, where we have made the modifications required. The source code of this bean along with a client application, that can be used in order to test our Bean, are located inside the directory *exampleBean* of p2pindex file. An Ant Build file exists in order to build the customer Bean and run the client. The demonstration that follows assumes that an EJB already exists and describes what changes must be made. This is the process we followed in order to create our *customer* EJB, since we took an already developed version of it with standard entity CMP functionality and start adding to it. However someone can use our example Bean as a starting point, and by using the opposite logic, alter it in order to develop a new EJB. The source code from the original *customer* EJB that was used is also included inside the directory *exampleBean/original_src*

Notice: Our *customer* Bean uses the default database embedded in JBoss, Hypersonic SQL. In order to deploy our EJB, open a shell (or command prompt) go to where you have unzipped the p2pindex file and change to directory *exampleBean*. Then run the *ant* command. If everything worked fine the *ant* command will build the source codes and create file *customer.jar*, which then will be automatically copied into the

$JBOSS_HOME/server/default/deploy directory. After the EJB is hot-deployed and while you are at the same directory run again the *ant* command with argument *run.client*. This will execute the client application. Details about the client will be given in a later section.

Notice 2: The first time you will try to deploy the Bean a JXTA Configuration window will be thrown. Fill in the fields with names of your preference. For more information about the Configuration options download the JXTA ProgGuide pdf from the JXTA homepage(*http://www.jxta.org/*.

As mentioned earlier, large pieces of code can be copied as is. Therefore in this section we begin with some basic copy-paste operations that must be made inside and outside your Beans source code prior to start writing code of your own. In later sections we describe how you can write code that creates the attribute indexes and their corresponding finder methods that locate Remote References of EJB's, based on the search key.

The first thing to do is copy into your EJB the Session Bean *dbSync*. To do this, open the directory your EJB resides. Create the directory *tuc/p2pindex* and copy in it, directory *dbSync* as is from directory *exampleBean/src/main/tuc/p2pindex* of p2pindex file. The reason you have to copy the *dbSync* in your EJB, is because this way you don't have to change any of your EJB XML related files (ejb-jar.xml, jboss.xml).

Then you will have to copy certain pieces of code from the *customer's* Bean Class java file (*exampleBean/src/main/tuc/p2pindex/customer/CustomerBean.java* into your EJB's Class java file. Open both files and:

– Copy all the import commands you don't have in your Bean.

– Copy the declaration of EntityContext *public EntityContext context;*

  along with the two functions that control it (*setEntityContext* and *unsetEntityContext*).

- Copy the body of function *ejbStore()* as is into the body of your *ejbStore()* function
- Copy the body of function *ejbRemove()* as is into the body of your *ejbRemove()* function

The above functions are not the only ones that are copied as is, however the rest of them are mentioned in a more relevant section.

A.4 Step 3: Define the indexed attributes

In this section we will describe how you can declare the attributes that you want to be indexed into the P2P network. As we have already mentioned our design dictates that each time *ejbStore* is called (when an EJB's setter, getter, create or remove methods is called) it invokes the indexing mechanism. This indexing mechanism has 2 parts.

- Invoke method *indexed_data* and collect the values of all chosen indexed attributes
- Send these values to *dbSync*

Therefore it becomes apparent that the whole indexed attributes selection is performed inside the *indexed_data* function. We will explain the semantics of this specific function by analyzing the *indexed_data* function we have created for our customer EJB. Its body can be seen below:

Listing A.1: "indexed_data()"

```
1
2 private  Object [] indexed_data(){
3
4       //The size of the array is based on the number
5   //of indexes the user wants(2 x Number of indexed
6       //attributes(odd number)
7       Object attributes[] =new Object [4] ;
8
9
10
11     //EXAMPLE SINGLE ATTRIBUTE
12      //first we declare the name
13   attributes [0] = "lname";
14
15   //modify if ,keep else as is
16   if (this.getLastName()!=null ) {
17
18     attributes [1] = this.getLastName();
19   }else{//attribute's value has not been set
20     attributes [1] = "Attribute not set";
21
22   }
23
24   //EXAMPLE  COMPOSITE ATTRIBUTE
25   //first we declare the name
26   attributes [2] = "fnamelname";
27
28   //modify if(make necessary concatenations),
29   //keep else as is
30   if (  (this.getFirstName()!=null ) &&
```

```
31        (this.getLastName() != null)   ) {
32        String temp =new String(this.getFirstName() );
33
34        temp =temp.concat(this.getLastName() );
35
36        attributes [3] = temp;
37
38     }else{//attribute's value has not been set
39        attributes [3] = "Attribute not set";
40     }
41     return attributes;
42  }
```

A first thing to notice is that the indexed attributes can be either single ones or combinations of them. We cover both of them, one from each category, however you may choose as many single ones you want and as many combinations of 2 or 3 attributes. (More can be declared however in order for a finder method to be able to locate them certain changes must be made). The first thing to do is declare an array of Objects called *attributes*, the size of which must be, 2 times the number of all indexed (single or combined) attributes. If for example you want 3 single and 3 combinations, then the declaration should be:
Object attributes[] =new Object [12] ;

The elements of this array are filled based on the following logic:

- Each index holds two consecutive places starting from position 0.
- The first place (odd elements: 0,2,4 etc) must contain a String value, of your choice, which should be relative to the names of the indexed

attributes, for simplicity. For example in our case values *lname* and *fnamelname* are relative to attribute fname and lname. This String value, which from now on we will call *name*, can be anything you want and it will be used to uniquely identify each of the indexes. An easy way to choose the value is for single attributes their name and for composite their names in one single String. For instance if you wanted to index the combination of attributes Date, Time, Year then declare it like
attributes [0 or 2 or 4...] = "datetimeyear";

– The second place (even elements: 1,3,5 etc) contains the String representation of the attribute(s) value(s) to be indexed or String value "Attribute not set" in case the attribute is NULL. We assume that, for each attribute, a get method exists that returns this attributes value. So you must declare an *if* statement that checks that the value of this attribute or in case of composite attributes all (logical AND) the required attributes don't have NULL values. If this is true then:
  1. For single attribute you set this position with the value of this attribute, by calling the getter for *this* EJB. If the value returned is not a String then function *toString()* must be invoked (for example attributes [1 or 3..] = this.getATTRIBUTE().toString();)
  2. For composite attributes a temporary String *temp* must be created and used in order to concatenate all the String representations of the participating attributes, in single String. Again if the returned value is not a String, function *toString()* must be used. Thereupon we set this element with the value of *temp*

attributes [1 or 3 ..] = temp;

The else part is always the same and sets the value of this index to "Attribute not set" that will prevent the indexing mechanism to be invoked, in *ejbStore* if we don't have the necessary values. Therefore it can be copied as is for each index, setting however every time the correct array position.

In our *indexed_data* function, index *lname* is an example of a single attribute and *fnamelname* of a composite one. However you can create as many indexes as you like, just by repeating one of the above procedures, each time.

A.5 Step 4: Create the Finder methods

Now let's suppose you have created the necessary indexes, through the *indexed_data* function, described in the previous section. The next step is to create the corresponding finder methods, because it would be pointless to create an index without having a method that would make use of him in order to answer queries related to him. Writing a finder method requires you copy and modify pieces of code from file *CustomerBean.java* and write some code of your own. We begin with some copy operations:

- Copy function *findp2p(Object par1)* as is into your code
- Copy function *findp2p(Object par1 ,Object par2)* as is into your code
- Copy function *findp2p(Object par1 ,Object par2 ,Object par3)* as is into your code

The reason we overload function *findp2p* will become apparent a little bit later.

The next copy operation is very important and requires we make certain modifications. First copy as is function *findP2P(Object[] keys)* into your code. This is the function that actually calls queryDHT, which is responsible for looking up the value we search over the P2P network. The value returned from this function is a Vector that contains Remote References (Remote Interfaces) to EJB's of the type our Bean is. In order to obtain them, we connect to the appropriate jnp server, call jndi's *lookup* function that returns an object we cast to the correct Home Interface and finally call *findByPrimaryKey* for a given id, that returns the Remote Reference we want. The above actions in the case of our *customer* Bean can be seen in the following piece of code.

Listing A.2: "findP2P "

```
1
2 Context jndiContext = new InitialContext (properties);
3
4 Object ref_c = jndiContext.lookup ("CustomerHomeRemote");
5
6 CustomerHomeRemote c_home = (CustomerHomeRemote)
7
8 PortableRemoteObject.narrow (ref_c ,CustomerHomeRemote.class);
9
10 Integer idd=new Integer(id);
11
12 CustomerRemote cust =c_home.findByPrimaryKey( idd );
```

It is therefore necessary for you to change everything that has to do with the type of the Home/Remote Interface and declare the correct class for your Bean. Based on our code values *CustomerHomeRemote*, *CustomerHomeRemote.class* and *CustomerRemote* must be replaced by the corresponding values of your EJB. **Notice**: We assumed that *findByPrimaryKey* takes an Integer parameter, something very common for Entity Beans. If however this is not the case in your Bean then line:
Integer idd=new Integer(id);
must be replaced by a declaration relevant to the type of your Primary key and the value that must be passed as a parameter in your *findByPrimaryKey* function.

Now in order to create a finder method you must declare two methods: the actual finder method, and a check method that is called from inside the finder. At first we will create the actual finder method. The name can be anything you like, however it is preferable it reflects the actual attributes involved. If for example you want to crate a finder method based on Date and time, findByDateTime, is a good name. The return type must always be a *Collection*. All finder methods are functions of the EJB's Home Interface and therefore must be declared there first. A finder method can have from one to three parameters of any type we want, based on the types of the attribute(s) that where used to create each of the existed indexes (the order must also be the same we used to create the index). For example
Collection findByDateTime(Date d,Time t); or
Collection findByPriceNameCode(Interger i, Name n,Code c);
are possible valid declarations. Eventually the String representations from

these values are used to create the search key (Remember in function *indexed_data* the String representations of each attribute were used to create the index). Based on the EJB specification, every Home method must be declared inside the Beans Class code with prefix ejbHome. The code for one of our finder methods can be seen below:

Listing A.3: "FindByName function"

```
1
2  public Collection ejbHomeFindByName(String fname,String lname)
3  {
4     Vector returned = new Vector ();
5     Vector result = new Vector ();
6
7     System.out.println("findByname calls findp2p")
8     returned =new Vector(findp2p(fname,lname) ) ;
9
10    if (returned.isEmpty()== false) {
11     for (int i=0;i<returned.size();i++){
12
13        CustomerRemote cust =(CustomerRemote)returned.elementAt(i++)
14        String key= (String ) returned.elementAt(i);
15         try{
16
17         if (cust.checkName(key ) == true) {
18              result.addElement(cust);
19         }//end if
20           } catch (java.rmi.RemoteException re)
21           {
```

```
22                  re.printStackTrace ();
23              }
24       }//end for
25     }
26     return result;
27
28 }
```

The first thing that must be done inside every finder is to call *findp2p* method with exactly the same parameters the finder method has and in the same order. For our imaginary *findByDateTime(Date d,Time t)* we should write :

returned =new Vector(findp2p(d,t)) ;

What actually happens is that each of the three overloaded functions *findp2p* creates a single array based on the number of parameters passed and calls *findP2P* function with this array as a parameter. Therefore the reason we created functions *findp2p*, was in order to make the call of function *findP2P* transparent to the EJB developer, no matter the number of finder method parameters. Moreover it is now obvious why there is a limitation to the number of finder parameters and as so the attribute(s) that create an index and how this can be extended.

Then for each Vector element we get as a result, we cast it to the appropriate Remote Reference, which obviously should be changed to the correct value for your EJB. This is a good point to describe how to declare the corresponding to this finder, *check** method that was mentioned earlier.

A *check** method must be declared for each finder. The name you choose can be anything you like, however it is preferable it reflects the actual

attributes involved. All *check** methods are functions of the EJB's Remote Interface and therefore must be declared there first. The return type must always be *boolean* and the parameter must be a String value. The code for the corresponding to our *FindByName* method, *checkName*, can be seen below:

Listing A.4: "checkName function"

```
1
2   public boolean checkName(String key) {
3
4     String temp =new String(this.getFirstName() );
5
6     temp =temp.concat(this.getLastName() );
7
8     if (temp.matches(key) ){
9       return true;
10    }
11    else{
12      return false;
13      }
14  }
```

Inside the body of each check method we create a single String by concatenating the values of the appropriate attribute(s) (the same attributes we concatenated inside indexed_data, for the index this finder method is relevant) and we return the result of the equality test with the key.

Going back now to the finder method we must call for each Remote Reference the corresponding *check** method, with the key we got as a result.

Only if the result is true, which means the value we got is still valid we add this Remote Reference to the returned Vector.

Copy both codes into your code the make the necessary changes explained above. Don't forget to declare each finder and check method to your Home and Remote Interface respectively. Remember that this procedure must be done for every index we have created.

A.6 Step 5: Deploy your Bean

Now that you have finished writing your bean's code, build it in order to produce a single .jar file with it and copy it into JBoss's deploy directory. Remember that the first time your bean is deployed a JXTA Configuration window will be thrown. Fill in the fields with names of your preference. For more information about the Configuration options download the JXTA ProgGuide pdf from the JXTA homepage(*http://www.jxta.org/*.

A.7 Step 6: Running a client application

The client application we have created covers all of the possible EJB operations. It creates five beans, sets their values ,re-set some of them, calls the finder method *FindByName* we have created and shows the results we get, calls some getters and finally removes some beans.

For our client application to be meaningful you must deploy the *customer* Bean in at least to different machines that run the JBoss Application Server. In each one of the machines set a different i variable value and run command

ant run.client. If everything worked fine you will see that the finder method returns the correct results, based on the indexes created.

In general every time a Client application creates an EJB and sets the appropriate attributes an index will be automatically created. Moreover the client can call a defined finder method, from the Home Interface, to get a Vector with Remote References to EJB's of the same type based on the values specified.

Appendix B

Javadoc

The following pages contain the *javadoc* produced from the source codes that we used in order to create the P2P indexing mechanism. The actual .html pages can be found in following directories of the p2pindex.zip file, respectively:

- p2pindex/DHTxmbean/javadoc
- p2pindex/dbSync/javadoc/
- p2pindex/exampleBean/javadoc/

Bibliography

[1] *Java Management Extensions Instrumentation and Agent Specification, v1.0, JSR003*, Tech. report.

[2] *PROST : A programmable structured peer-to-peer overlay network.*

[3] *Enterprise JavaBeans specification, version 2.1*, Specification, Sun Microsystems, November 2003.

[4] L. F. Cabrera, M. B. Jones, and M.Theimer, *Herald : Achieving a global event notification service*, Workshop on Hot Topics in Operating Systems (Elmau,Germany), May 2001.

[5] M. Castro, P. Drushel, A. M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, *Splitstream : High-bandwidth content distribution in a cooperative enviroment*, IPTPS, 2001.

[6] M. J. Freedman and D. Mazieres, *Sloppy hashing and self-organizing clusters*, In Proceedings of the IPTPS, 2003 (Berkeley), February 2003.

[7] L. Garces-Erice, P.A. Felber, E.W. Biersack, G. Urvoy-Keller, and K.W. Ross, *Data Indexing in Peer-to-Peer DHT Networks*, Tech. report, Institute EURECOM, 2002.

[8] Daishi Kaito, *GISP : Global information sharing protocol - a distributed index for peer-to-peer systems-*, Tech. report, Computer Science Department, Stanform University, November 2003.

[9] Nicholas Kassem and the Enterpsise Team, *Designing Enterprise Applications with the JavaTM 2 platform, Enterprise edition*, Tech. report.

[10] J. Kubiatowicz, *"oceanstore: An architecture for global-scale persistent storeage"*, ASPLOS 2000, 2000.

[11] A. Mislove and A. Post, *Post : A secure, resilient, cooperative messaging system*, HotOS IX, May 2003.

[12] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker, *A scalable content-addressable network*, In Proc. ACM SIGCOMM, 2001.

[13] A. Rowstron1 and P. Druschel, *Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems*, in Proc. of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001) (Heidelberg, Germany), November 2001.

[14] Scot Stark and the JBoss Group, *JBoss Administration and Development*, Tech. report.

[15] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana, *Internet indirection infastructure*, In Proceedings of ACM SIGCOMM'02, August 2002.

[16] I. Stoica, R. Morris, D. kanger, F. Kaanshoek, and H. Balakrishman, *Chord : A scalable peer-to-peer lookup service for internet applications*, In proc. ACM SIGCOMM (San Diego,California).

[17] Chunqiang Tang, Zhichen Xu, and Mallik Mahalingam, *pSearch: Information Retrieval in structured overlays*, Tech. report, HP Laboratories, 2002.

[18] B.Y. Zhao, J. Kubiatowicz, and A. D. Joseph, *Tapestry: An infrastructure for fault-tolerant wide-area location and routing*, Tech. report, Computer Science Division, University of California, Berkeley, April 2001.