11-28-2007

Peer-to-Peer Distributed SyD Directory Synchronization in a Proximity-based Environment

Sunetri Priyanka Dasari

PEER-TO-PEER DISTRIBUTED SYD DIRECTORY SYNCHRONIZATION IN A PROXIMITY-BASED ENVIRONMENT

by

SUNETRI PRIYANKA DASARI

Under the Direction of Dr. Sushil K. Prasad

ABSTRACT

Distributed directory services are an evolving paradigm in the distributed computing arena. They are a shift from the centralized directory that causes delay and does not scale well to widespread peer-to-peer networks. With networking becoming more pervasive, there is a need to integrate the heterogeneity of device, data and network with the applications that are built on them. SyD or System on Mobile Devices is a middleware that is being used to implement such a distributed directory service. To provide a persistent global view of data, we serialize and synchronize the distributed directories. The SyD APIs provide a high-level environment to rapidly develop collaborative applications for such networks in a systematic manner. An inter-vehicle communication application that notifies the driver of a vehicle of the available parking spots in the vicinity, allows us to see the practical working and benefits of the distributed directory paradigm.

INDEX WORDS: Distributed Directory, SyD, Synchronization, Peer-to-Peer, Transient, Proximity Network

PEER-TO-PEER DISTRIBUTED SYD DIRECTORY SYNCHRONIZATION IN A PROXIMITY-BASED ENVIRONMENT

by

SUNETRI PRIYANKA DASARI

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

in the College of Arts and Sciences

Georgia State University

2007

PEER-TO-PEER DISTRIBUTED SYD DIRECTORY SYNCHRONIZATION IN A PROXIMITY-BASED ENVIRONMENT

by

SUNETRI PRIYANKA DASARI

Committee Chair: Dr. Sushil Prasad

Committee: Dr. Raj Sunderraman
Dr. Yingshu Li

Electronic Version Approved:

Office of Graduate Studies
College of Arts and Sciences
Georgia State University
December 2007

ACKNOWLEDGEMENTS

I would like to sincerely thank Dr. Sushil Prasad for his patient and able guidance throughout my research work. His advice and insight have been invaluable towards the completion of this thesis. As a teacher, he has always tried to bring out the best in me and put my efforts to good use.

I am grateful for Dr. Raj Sunderraman's expertise and sound advice during my Masters program. His persistence to make sure I submit my Master's Thesis has finally paid off.

A note of thanks to Dr. Yingshu Li for graciously accepting my request to be a member of my Thesis Committee.

A special thank you goes to Srilaxmi Malladi for getting me started off on Syd and patiently going back and forth on all my SyD related questions.

Chad – Thank you for saying "Just Rewrite It …"

Akshaye – Thank you for giving a real-world need and context to the parking lot application.

Charuka, Saara, Joseph – Your interest, feedback and questions really helped clear my thoughts.

Milan – Thank you for patiently going through all the endless discussion on middleware.

Sankari, Krishna – Thank you for taking time out of your busy schedule to debug Java programs before I had any idea about what Java does.

I would also like to thank my husband, Sridhar, immensely, for taking complete care of our daughter and bearing with me while I did my research. Without his unwavering support and constant encouragement, I would not have been able to get all my work done on time.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

Mobile wireless technology usage has mushroomed tremendously in recent times. To keep up with the rapid hardware evolution of devices like cell phones, laptops, gaming devices and PDAs, there is a need to integrate greater amount of functionality and performance into the applications that are available on them. We are greatly interested in leveraging off this widespread usage so as to provide group collaborative applications in a peer-to-peer (P2P) mobile computing environment. Such applications would require co-operation between peers to provide useful, up-to-date information in a quick and efficient manner.

A particularly good example would be of users travelling in cars communicating with other cars about current information on traffic speed, accident awareness or resource availability in the vicinity. In the near future, every vehicle will be equipped with wireless connectivity devices that enable communication with roadside objects and also with other vehicles. Drivers in these vehicles are interested in information relevant to their trip. For example, a driver would like his/her vehicle to continuously display on a map, at any time, the available parking spaces around the current location of the vehicle. Or the driver may be interested in the traffic condition one mile ahead wherein the application may suggest an alternate route based on certain parameters. Such information is important for drivers to optimize their travel, alleviate traffic congestion or avoid wasteful driving.

More specifically, the ability of geographically close devices to participate in transient groups and communicate with each other by forming a personal area network is an active area of research called proximity networking. At the forefront of this research, is the necessity to have a high-level programming environment that would allow us to concentrate on collaborative application development without being held back by the heterogeneity of devices on which the applications will be hosted. This transparency is achieved through the middleware layer that acts as an interface between the application layer and network layer. This is an active area of research with enormous potential and in this thesis work we present a way to enhance the System on Mobile Devices (SyD) [2, 3] middleware by incorporating a distributed directory to achieve this objective.

1.1 Motivation

There is a need to have a framework that allows rapid development and deployment of collaborative applications in peer-to-peer networks. The sheer number and variety of devices and services available make it necessary for providing a transparent layer that can provide a simple, uniform view of the devices and services available by modeling them as generic objects. This is where the System on Mobile Devices (SyD) middleware comes into the picture. Keeping this in mind, the work done in this thesis focuses on employing a distributed directory of these application objects, which would periodically synchronize with proximal directories and update its application objects with relevant data. Using SyD, we show how to develop and deploy such an application rapidly so as to always provide up-to-date information to the co-operating peers.

The applications are modeled as SyD Application Objects that have server and client components providing services (server) or requesting services (client) from a particular user's device. Therefore, each peer acts as both a client and a server. The SyD Directory is where all the data regarding these Application Objects is published, so that the participating peers know where, how and what can be accessed. The centralized directory version did not translate well to our peer-to-peer environment which was the main motivation behind developing a distributed directory version of SyD that uses synchronization between the peer directories to mimic the presence of a central directory for proximal peers.

1.2 Aim of Thesis

The main aim of this thesis work is to extend the SyD middleware so as to allow co-operating SyD-enabled peers to synchronize their directory information with neighboring peers regarding SyD Application Server Objects. As stated earlier, the earlier version of SyD makes use of a centralized directory in a relational database to maintain SyD Application Object information. A distributed directory version of SyD is developed to support the transient nature of the peers in our proximity-based environment. Each peer maintains its own SyD Directory as opposed to a central SyD Directory that was accessible to all users earlier.

A parking lot application that tells the user traveling in a vehicle about the available number of spaces in nearby parking lots is implemented using the distributed directory version of SyD. This is possible by modeling the SyD Directory as an object and eliminating the usage of a central database. This application helps demonstrate the ease with which a peer can access local and neighboring Application Server Objects to get relevant up-to-date data, after synchronization between their respective SyD Directory objects.

1.3 Thesis Layout

The document layout aims at presenting the research and development work done in a clear and comprehensive manner. The chapter details are as follows.

- Chapter 1 provides the introduction to what is being done in this thesis work by way of motivation and aim of the thesis.

- Chapter 2 discusses the related work being done in this area by leading researchers.
- Chapter 3 provides an overview on the SyD middleware describing the functionality of the various modules in the central directory version of SyD.
- Chapter 4 presents the theory and reasoning behind developing a distributed directory version of SyD and how it is suitable to our peer-to-peer proximity network.
- Chapter 5 presents a case study on the Parking Lot application that has been developed using the distributed directory version of SyD.
- Chapter 6 concludes the thesis work by presenting open problems and future research work on extending SyD.

2 RELATED AREAS OF RESEARCH

The evolution of distributed systems and mobile computing has led to the vision of pervasive computing. We are looking into the creation of environments [7] saturated with computing and communication capability, yet gracefully integrated with human users. Such profound technologies weave themselves into the fabric of everyday life so as to become indistinguishable from it. With pervasive computing becoming more of a norm than the exception, middleware [1] appears as a major building block for the development of future software systems. The work done in this thesis is thus significant as it is a step towards developing a pervasive computing environment. This chapter examines related research work in the areas of distributed directory middleware frameworks, software development using such middleware, distributed databases in mobile environments and resource discovery protocols like MobidiC [21] and Carmen [9] that were a starting inspiration to develop a parking lot application using the SyD middleware.

2.1 Mobile P2P Databases

Mobile P2P databases [23, 24] are a very closely related area of research where a database is stored in the peers of a mobile P2P network. The network is composed of a finite set of mobile peers that communicate with each other via short range wireless protocols, such as IEEE 802.11, Bluetooth or Ultra Wide Band (UWB). These protocols provide broadband (typically tens of Mbps) but short-range (typically 10-100 meters) wireless communication. On each mobile peer there is a local database that stores and manages a collection of data items or reports. A report is a set of values sensed or entered by the user at a particular time or otherwise obtained by a mobile peer. Often a report describes a physical resource such as an available parking slot. All the local databases maintained by the mobile peers form the mobile P2P database. The peers communicate reports and queries to neighbors directly, and the reports and queries propagate by transitive multi-hop transmissions. A peer may not know the identities of other peers in the network and the data they store.

Mobile P2P databases enable matchmaking or resource discovery services in many application domains, including social networks, transportation, mobile electronic commerce, emergency response and homeland security. Communication is often restricted by bandwidth and power constraints on the mobile peers. Furthermore, often reports need to be stored and later forwarded, thus memory constraints on the mobile devices constitute a problem as well. Thus, careful and efficient utilization of scarce peer resources (specifically bandwidth, power, and memory) are an important challenge for mobile P2P databases. Despite these challenges, the main advantages of mobile P2P databases are:

(i) As short-range wireless networks (eg: Bluetooth) utilize the unlicensed spectrum, communication to the mobile P2P database is free.

(ii) They can be used for search in emergency, disaster, and other situations where the infrastructure is destroyed or unavailable.

(iii) They are more reliable in the sense that failure of the central site will not render the system unavailable.

These factors have been used in the design of a distributed directory for SyD that works on similar principles.

In [24, 25], the authors present a problem where the challenge is processing queries in a highly mobile environment of hundreds of vehicles, with an acceptable delay, overhead and accuracy. One approach to solving this problem is maintaining a distributed database stored at fixed sites that is updated and queried by the moving vehicles via the infrastructure wireless networks. Potential drawbacks of this approach are

- responses to queries may be outdated
- response time may not meet the real-time requirements
- access to infrastructure communication service is costly

The Mobidic or Mobile Discovery of Local Resources project [21] aims at creating an environment that will enable a new class of local search-and-discover applications that are independent of an infrastructure or a database server. Each vehicle is assumed to have the capability of communicating with its neighbors. A mobile user discovers the desired information from its neighboring vehicles or from remote vehicles by multi-hop transmission relayed by intermediate moving vehicles. Thus, resource discovery is performed in an inter-vehicle ad hoc network.

A novel rank-based broadcast algorithm that makes use of the relevance of spatio-temporal data [8] to provide latest data to users in real-time is presented. The major work done here is on algorithms for opportunistic resource dissemination in mobile peer-to-peer wireless networks but there is no actual framework that helps rapidly develop high-level distributed applications. Also, all the research work here concentrates on issues specific to VANETs (Vehicular Ad-hoc Networks).

2.2 Dynamic Service Discovery

The growth of ubiquitous computing will see the spread of autonomous agents and distributed systems in the devices surrounding us. To be effective, these devices will need to interact and access each other's services in order to provide the user with a productive experience. But in order to work together there needs to be a mechanism for devices to find each other based on the services they require. Service discovery protocols are network protocols which allow automatic detection of devices and services offered by these devices on a computer network. When the location of the requested service (typically the address of the service provider) is determined, the user may then access and use it. The authors in [10] present a survey on various kinds of service discovery protocols for use in mobile ad-hoc networks. This is an extremely informative paper that can help in the design of discovery protocols for the distributed SyD Directory.

The Carmen [9] project is an effort to design a powerful service discovery protocol for "smart" access points capable of delivering a variety of services to mobile hosts. It is used to enable a very large number of heterogeneous nodes, both dynamic and static, to offer and search for an unlimited number of services and information. This makes use of the static nature of the core network to facilitate search and service discovery for dynamic users. The authors claim that Carmen can be deployed in a variety of environments to bridge the gap between local dynamic service discovery and global static search. It can scale to global proportions, maintain up-to-date information on short-lived services and allow any provider to input its own service descriptions.

Such a discovery protocol is interesting from our point of view as the usage of a distributed directory is based on dynamic discovery. Though actual device discovery is outside the scope of this research work, a protocol like Carmen would work as an excellent precursor to help implement a similar architecture for the distributed SyD Directory.

2.3 Middleware-based Software Engineering

As the focus of this research is on the development of a high-level programming environment for creating collaborative applications using middleware, an excellent paper on this topic [1] showcases the various kinds of middleware and the need to come up with a generic software engineering process for developing applications for them. Middleware establishes a new software layer that homogenizes the infrastructure's diversities by means of a well-defined and structured distributed programming model. Over the years, the role of middleware has proven central to addressing the ever increasing complexity of distributed systems in a reusable way. Further, methods and related tools are required for middleware-based software engineering. This need becomes even more demanding if we consider the diversity and scale of today's networking environments and application domains, which makes middleware and its association with applications highly complex.

Middleware contributes to relieve software developers from low-level implementation details, by abstracting socket-level network programming in terms of high-level network abstractions matching the application computational model and managing networked resources through reusable middleware-level services. However, software development accounting for middleware support is an emerging area of research. Mature engineering methodologies to comprehensively assist the development of middleware-based software systems from requirements analysis to deployment and maintenance are lagging behind.

Figure 2.1 taken from [1] depicts the phases (represented by squared boxes) of the overall middleware-based software development process and their productions (represented by ovals).

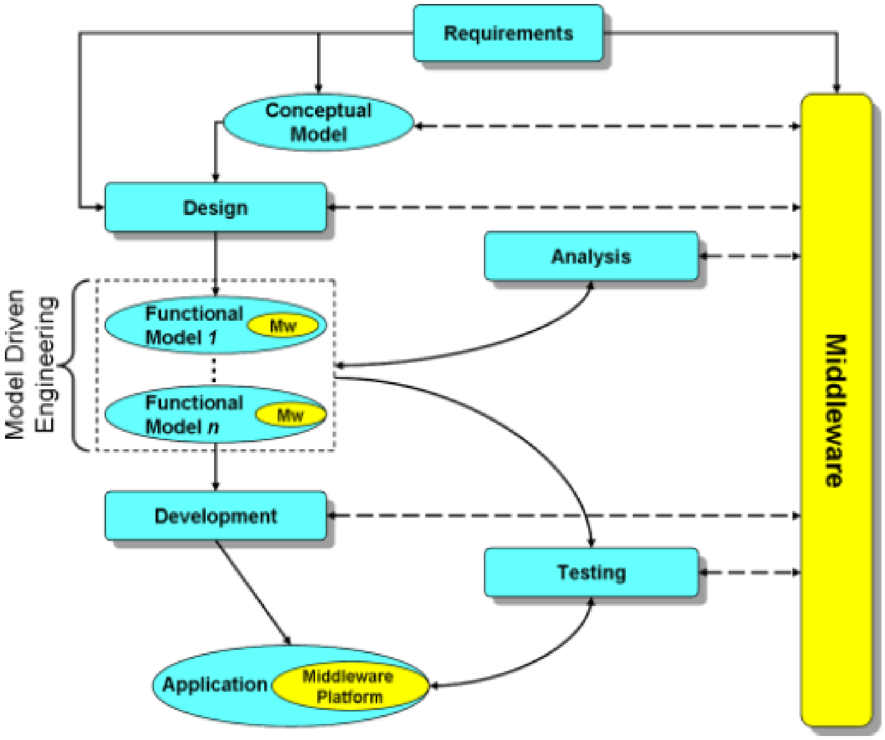


Figure 2.1: Middleware-based Software Process

SyD provides a perfect platform for delivering solutions for such kind of application development. In the next chapter, we look at the modular architecture of SyD that facilitates such a high-level programming environment.

3 SYSTEM ON MOBILE DEVICES (SYD): AN OVERVIEW

System on Mobile Devices (SyD) middleware was designed to enable rapid development and deployment of collaborative applications by masking the heterogeneity of the distributed infrastructure. This has been achieved by modeling the heterogeneous devices and applications as objects that help in providing a uniform and persistent view, independent of devices, data and underlying networks. SyD combines ease of application development, mobility of code, application, data and users, independence from network and geographical location, and the scalability required of large enterprise applications concurrently with the small footprint required by handheld devices.

This chapter presents a description of the SyD modules, SyD methodology that facilitates rapid application development and deployment, and the various layers in the SyD Runtime Environment.

3.1 SyD Modules

Every application developed using SyD is modeled as an Application Object with two components, namely the Server Object and the Client Object. The SyD Directory is responsible for maintaining a list of SyD Application Server Objects that provide remote services. The core SyD modules [2, 27] and their functions are described as follows.

1. **SyDDirectory** provides user/application publishing, management, and lookup services to SyD users and Application Objects. It keeps track of SyD Application Objects and their associated devices via location information (IP plus port number). Since connectivity cannot

be maintained all the time especially for mobile devices, server applications can register their proxies too. The SyDDirectory actively maintains the availability of applications hosted on mobile devices vs. their proxies via a "livebit" in the directory entries; this bit can be set/reset by an application on power-off/power-on or by the SyDEngine on timing out on a method invocation on a server application.

2. **SyDRegistrar** registers the SyD Application Server Objects as remote services so that they are accessible for remote method invocations. Registration is done locally to the local object repository (RMI registry) and globally to the SyDDirectory. This is a sub-module of the SyD Listener.
3. **SyDListener** is instantiated at the application server side and performs listening, parsing and invoking of local methods in response to remote invocations from the application client objects. This kind of dynamic method invocation capability is achieved in the SyDListener by integrating Java reflection mechanism with RMI.
4. **SyDListenerDelegate** is at the application client side and performs communication with the SyDListener, including transmission of request messages and receiving of results. The SyDListenerDelegate acts as an adaptor for the SyDListener and hides all the communication details from clients. This is a sub-module of the SyD Listener.
5. **SyDDispatcher** is the major component in the **SyDEngine** that allows users to execute services remotely. It is responsible for providing method invocation capabilities for applications to access data either on the local device or on remote devices in a transparent manner. At runtime, the SyDDispatcher looks up the SyDDirectory for current device location and makes calls to the methods accordingly. The SyDDispatcher class has an invoke

method that can be used to invoke a single method on multiple application objects located remotely.

6. **SyDDoc** utility provides a uniform data exchange capability throughout the SyD middleware and SyD-enabled modules by making use of XML-like document strings. The data to be transmitted from one module to another is compacted into an XML document string. Methods within SyDDoc are then used to retrieve this information at the other module that is located (possibly) at another device. SyDDoc is used by the SyD Dispatcher component of the SyD Engine to compose method calls into services to be invoked on devices. The method details such as parameter types and values and also return types are packaged into a single XML string and then extracted by the user services at the remote end. SyDDoc is also used by server applications hosted on devices to publish information to the SyD Directory Service.

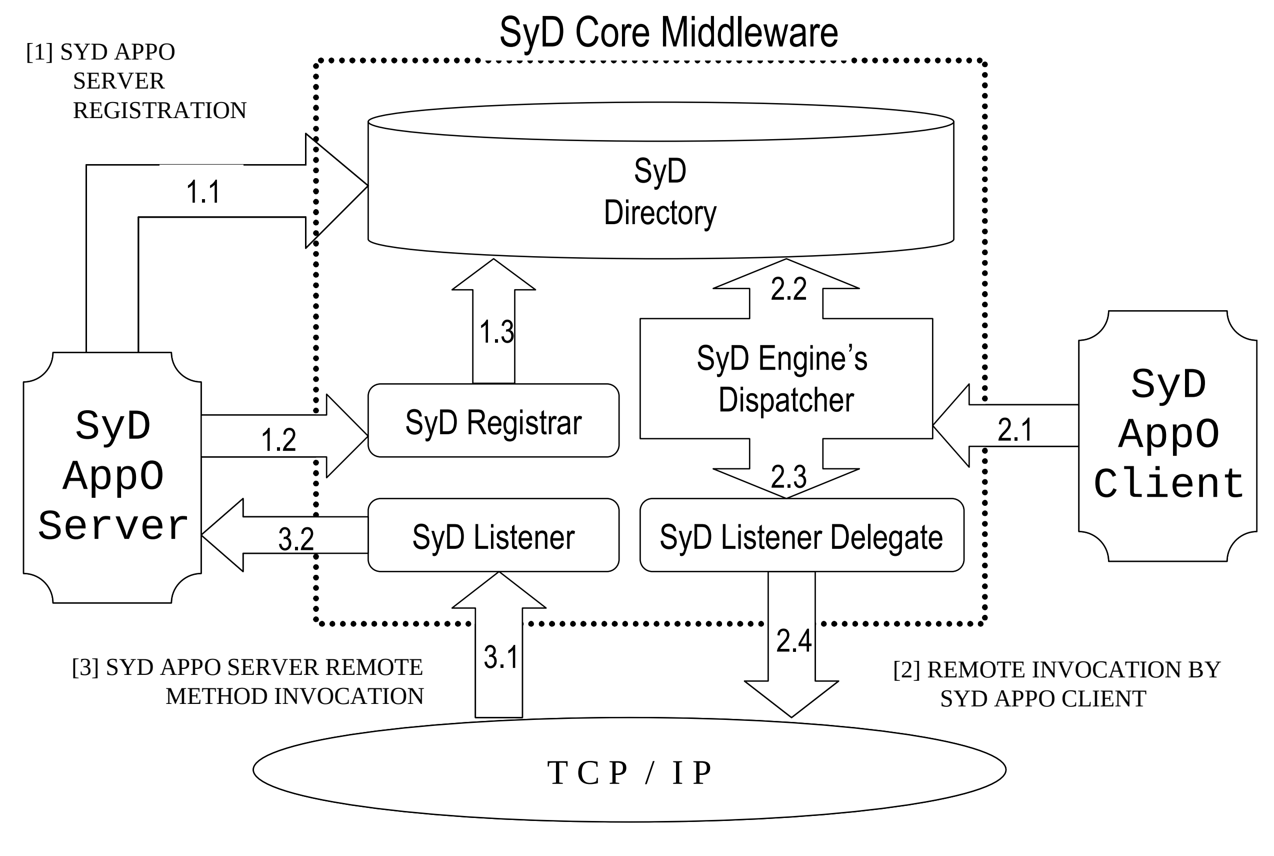
3.2 SyD Methodology

Now that we have a clear idea of the working of each of the core SyD modules, the SyD middleware simplifies the efforts of an application developer by providing APIs and guidelines for application development. We now describe the steps involved in application development.

Step1: Model the application using SyD objects (SyDAppO Server and Client) using the rich set of SyD APIs provided.

Step2: Incorporate SyD Directory functionality to publish objects along with their data and methods in the SyD Directory Service.

Step 3: Develop high-level application server code by employing the SyD Application Objects in Step1. Develop client code that typically includes the user interface.



(1) SyD Application Object (SyDAppO) Server Registration

1.1 Lookup Directory for SyDAppOServer Object ID

1.2 IF 1.1 does not succeed, send SyDAppOServer details to SyDRegistrar

1.3 Globally register with SyDDirectory and obtain Object ID

(2) Remote Method Invocation by SyD Application Object Client

2.1 Invoke Method Call

2.2 Lookup SyDAppO in SyDDirectory

2.3 SyDAppO location sent to SyDListenerDelegate

2.4 Make a remote call to the SyDAppO location from 2.3

(3) SyDAppOServer Remote Method Invocation

3.1 Listener receives client request

3.2 Listener invokes local method call on SyDAppOServer

Figure 3.1: SyD Middleware Architecture

Figure 3.1 illustrates the interactions between the core SyD modules and the application objects. An explanation of the sequence of interactions follows.

Once an application has been developed using SyD Application Objects and APIs, the next step is the deployment of the application in the SyD environment. The steps involved in deploying a SyD application and the interaction among various SyD modules are as follows:

Step 1: SyD Application Server Object registers its service with the SyD Registrar and SyD Directory Service. Please refer (1.1, 1.2 and 1.3) in Figure 3.1.

Step 2: SyD Application Client Object calls the SyD Dispatcher to utilize the service of other users. Please refer (2.1, 2.2) in Figure 3.1.

Step 3: SyD Dispatcher provides the SyD Listener Delegate with details of the SyD Application Client Object's location. Please refer (2.3) in Figure 3.1.

Step 4: SyD Listener Delegate calls the SyD Listener in other devices over TCP sockets. Please refer (2.4, 3.1) in Figure 3.1.

Step 5: SyD Listener in the remote device calls the related service. Please refer (3.2) in Figure 3.1. The results are finally returned back to the calling SyD Dispatcher.

Step 6: SyD Dispatcher returns the results to the user.

To run an application, the application server is launched on a device A's SyD middleware and registered with the SyD Directory Service. A SyD Client Application is launched on a remote device B's SyD middleware and can now access the corresponding SyD Application Server Object being hosted on device A. As described, the process of developing SyD applications takes on a highly distributed approach that creates flexibility and ease of programming, allowing rapid development of applications by effectively using the SyD APIs.

3.3 SyD Runtime Environment

The various layers in the SyD Runtime Environment [4] are illustrated in Figure 3.2 below. The clear distinction between the application, SyD middleware and communication layers shows how the middleware layer provides distribution transparency which helps with the high-level application development. It is located between applications and the communication services provided by primitive distribution middleware (TCP Sockets, RMI). Each layer depends on the services provided by a lower layer and each layer hides complexities of the tasks provided in that layer from upper layers. In this framework, applications are developed rapidly using SyD Kernel modules without any knowledge about the lower layers.

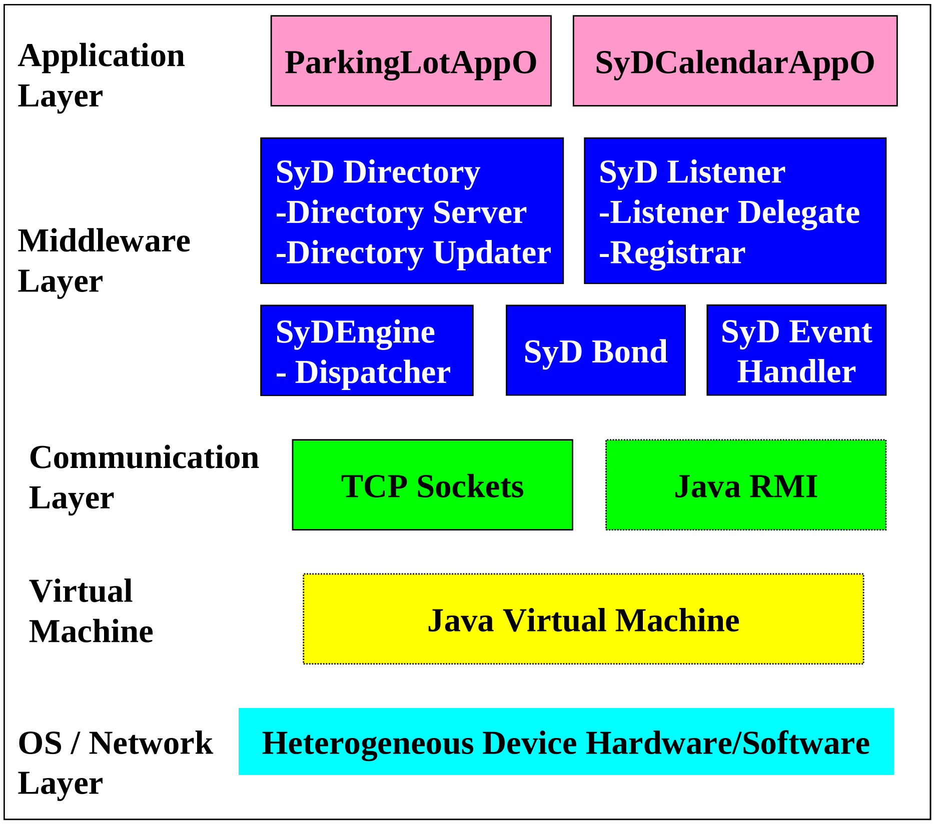


Figure 3.2: SyD Runtime Environment

The different layers at which the SyD middleware and the application objects operate provide a structured, easy to understand way to implement our requirements.

3.4 Chapter Summary

To summarize the concepts presented in this chapter, we have

(i) Publishing on SyD Directory Service

Application Objects register and publish their information including location and service availability on SyD Directory Service for other users to lookup and execute via SyD Engine.

(ii) Registering services as listeners using SyD Registrar and SyD Listener

SyD Registrar registers application methods locally in the RMI registry and globally in the SyD Directory. SyD Listener provides associated listeners for remote invocations.

(iii) Execution via SyDEngine

Users can execute individual server objects remotely using the SyDEngine that executes remote services by invoking the SyDListener on the remote device and returns results.

This completes the entire cycle of the SyD Application process from development to deployment. There is a central SyD Directory that is accessible to all users and centrally hosts the list of all available SyD Application Server Objects. In the next chapter, we present the motivation behind the development of a distributed directory version of SyD and its design methodology.

4 DISTRIBUTING THE SYD DIRECTORY

In this chapter, we present the rationale for developing a Distributed Directory version of SyD. Starting off with a discussion on peer-to-peer computing, we also look into proximity as an attribute of the network for which the application is being developed. These elements were factored into the actual design of the directory after which the implementation details are presented.

4.1 Peer-to-Peer Computing

Peer-to-peer computing [14] exploits the diverse connectivity between participants in a network and is typically used for connecting nodes via largely ad hoc connections. The term "ad hoc" is important as it specifies the transient nature of the peers in the network. The earliest peer-to-peer network in widespread use was the Usenet news server system, in which peers communicated with one another to propagate Usenet news articles over the entire Usenet network. The most popular example of P2P applications is content-sharing, besides which such networks are also useful for providing real-time data wherein it is not feasible for a central server to maintain and update it self continuously in addition to serving client requests.

4.1.1 Peer-to-peer Model vs. Client-server Model

A pure peer-to-peer network does not have the notion of clients or servers, but only equal peer nodes that simultaneously function as both "clients" and "servers" to the other nodes on the network. This model of network arrangement differs from the client-server model where communication is usually to and from a central server. A typical example for a non peer-to-peer file transfer is an FTP server where the client and server programs are quite distinct, and the clients initiate the download/uploads and the servers react to and satisfy these requests. An example of a pure peer-to-peer application layer file-sharing network is Gnutella that uses a fully peer-to-peer structure and is greatly facilitated by directory servers that inform peers of the network addresses of other peers.

Figure 4.1 [14] clearly illustrates the difference between the peer-to-peer and client-server models of computing.

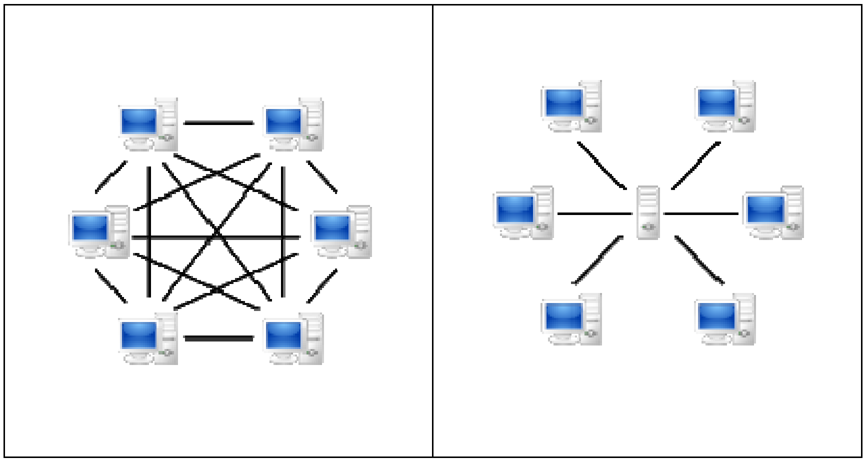


Figure 4.1: Peer-to-Peer Model (L) vs. Client-Server Model (R)

In pure peer-to-peer networks:

- Peers act as equals, merging the roles of clients and server.
- There is no central server managing the network.
- There is no central router.

4.1.2 Advantages of Peer-to-Peer Networks

An important advantage of peer-to-peer networks is that all the peers provide resources, either hardware or software, such that as the number of peer nodes in the system increase, the capacity of the system also increases. This is not true in a centralized architecture, wherein an increase in the number of clients would inevitably result in a decrease in overall performance. The distributed nature of peer-to-peer networks provides fault-tolerance by replicating data over multiple peers, and by enabling peers to find the data without relying on a centralized index server. These features were a primary design consideration behind the development of the distributed directory for SyD.

4.2 Proximity Networks

The ability of devices or systems to move into a networked environment and immediately gain access to all relevant services will become a requirement for future networks. Connecting portable devices with the underlying computing and communications infrastructure should becomes as simple as turning these devices on. Networks in which the physical proximity of a user to the network allows interactions with the local network service environment are called proximity networks. [13]

4.2.1 Properties of Proximity Networks

Proximity networks have the following unique properties that differentiate them from wired networks.

(i) They are ad-hoc and dynamic in nature. So, devices have no prior knowledge of other devices they will be connecting to.

(ii) They are independent of any infrastructure support (e.g. Access-point, Base-stations).

(iii) Their existence is only for a short period of time compared to wired networks and they usually disappear without any trace. This makes them extremely transient.

(iv) They are not bound to a specific location and can be formed anywhere and anytime.

(v) They are self-organizing and once connected, devices can automatically configure a channel for communication.

These unique properties of proximity networks give end-users the freedom to connect anywhere, anytime to any device without any external assistance. Once a proximity network is established, it creates a local service environment for end-users who can use several proximity services.

4.2.2 Requirements of Proximity Networks

In order to establish a proximity network, the following requirements need to be met.

(i) A fast and efficient device discovery protocol to discover neighboring devices.

(ii) A simple mechanism to authenticate and connect devices called device pairing.

(iii) A fast and efficient hand-off mechanism to hand-over data sessions to other devices, independent of infrastructure (e.g. Access Point).

(iv) A middleware framework to run high-level distributed applications.

Now that we have a clear picture of what peer-to-peer and proximity networks are , we present the features of our parking lot application that urge the need to have a distributed directory.

4.3 SyD Directory: Central vs. Distributed

The properties and requirements of proximity networks specified in the earlier sections uniquely suit the specifications of the parking lot application mentioned in section 1.2. To reiterate our objective, we would like to develop an application that would allow users travelling in vehicles to get the latest updated number of available parking spaces from parking lots in the vicinity. This information needs to be derived from peers that are in close proximity of the requesting vehicle. We will look more into the application details and any assumptions that we’ve taken in chapter 5. In this section, using a high-level view of the application, we develop the reasoning behind why we need a distributed directory for such a scenario.

We treat this as a transient proximity network as the exchange of information is supposed to occur when two vehicles, that have no prior knowledge of each other, come close enough to exchange their respective parking lot server data about available parking spaces. This kind of data is relevant to the devices/vehicles that are within the vicinity of a particular parking lot. It is irrelevant to other vehicles that are in some other location or geographically far away from the requesting vehicle. It makes sense therefore to maintain each device’s data on the device itself instead of on a central directory. The device would then get updated with the necessary, requested and latest data from its neighboring devices and not be concerned with other irrelevant data about other devices or parking lots. Had there been a central directory, such data would not be useful in a global context at all, which creates the need for a distributed directory.

The final result is that the application server objects get updated with the latest number of available spaces based on a timestamp (assuming a global timestamp). This is taken care of at the SyD middleware layer instead of the application layer. There is no explicit instruction from the user to perform the exchange, but occurs implicitly upon discovering a neighboring peer device. This satisfies the requirement of automatic device discovery for proximity networks. Also, a point to note here is that this kind of vicinity search is irrelevant to and is not supported had we been using a central directory. All devices would publish themselves in the central directory and there would be no incentive for discovering neighboring devices, as a simple lookup would return the currently available devices.

As the vehicles keep moving, these proximity networks can be formed anywhere and at any time. Once the vehicle goes out of range, which happens quite soon, the vehicle gets disconnected from the proximity network. These frequent connections and disconnections reaffirm the transient nature of proximity networks. Again this would put a whole lot of load on a central directory to publish services at such a rapid pace, as a device may no longer offer its services or the data it offers may become stale after it moves away from a particular location.

Besides the requirements of the parking lot application, the centralized directory has performance drawbacks that make us think in a distributed fashion. The obvious issues here are those of network delay and non-scalability. In a mobile wireless environment, there is no guarantee of maintaining a stable connection. So, any kind of disruption may end up in data loss or inconsistent data being published to the directory. Typically, a central directory is placed in a fixed physical location that may lead to delayed responses while servicing the requests of mobile clients that are rapidly connecting and disconnecting from the network. Such a centralized directory would also create a bottleneck when servicing the requests of several clients, each requesting data on a different parking lot. This raises issues of non-scalability as the performance of the entire system is affected.

The parking lot application that we've chosen to develop demonstrates the need to have a distributed directory that would offer the services that we've described above. The central directory version of SyD would simply not allow us to have a pure peer-to-peer network as indicated in section 4.1 above. This urged us to shift to a distributed directory paradigm that would make it easy to satisfy all the properties of a peer-to-peer proximity network.

4.4 Distributed SyD Directory Design

Now that we've gained an idea on why we need to have a distributed directory, we proceed to clearly define what the requirements of a distributed directory are, pertaining to applications like the parking lot application. This section details the elements that were taken into consideration in the design of a distributed directory version of SyD. Before we proceed, we need to also have a high-level view of how SyD devices connect to the central SyD Directory to help differentiate it from the distributed directory version of SyD.

In the centralized directory version, every SyD device would contain the core SyD modules and access the central SyD Directory that is available at some fixed location. Figure 4.2 shows the way in which SyD devices connect to a central directory.

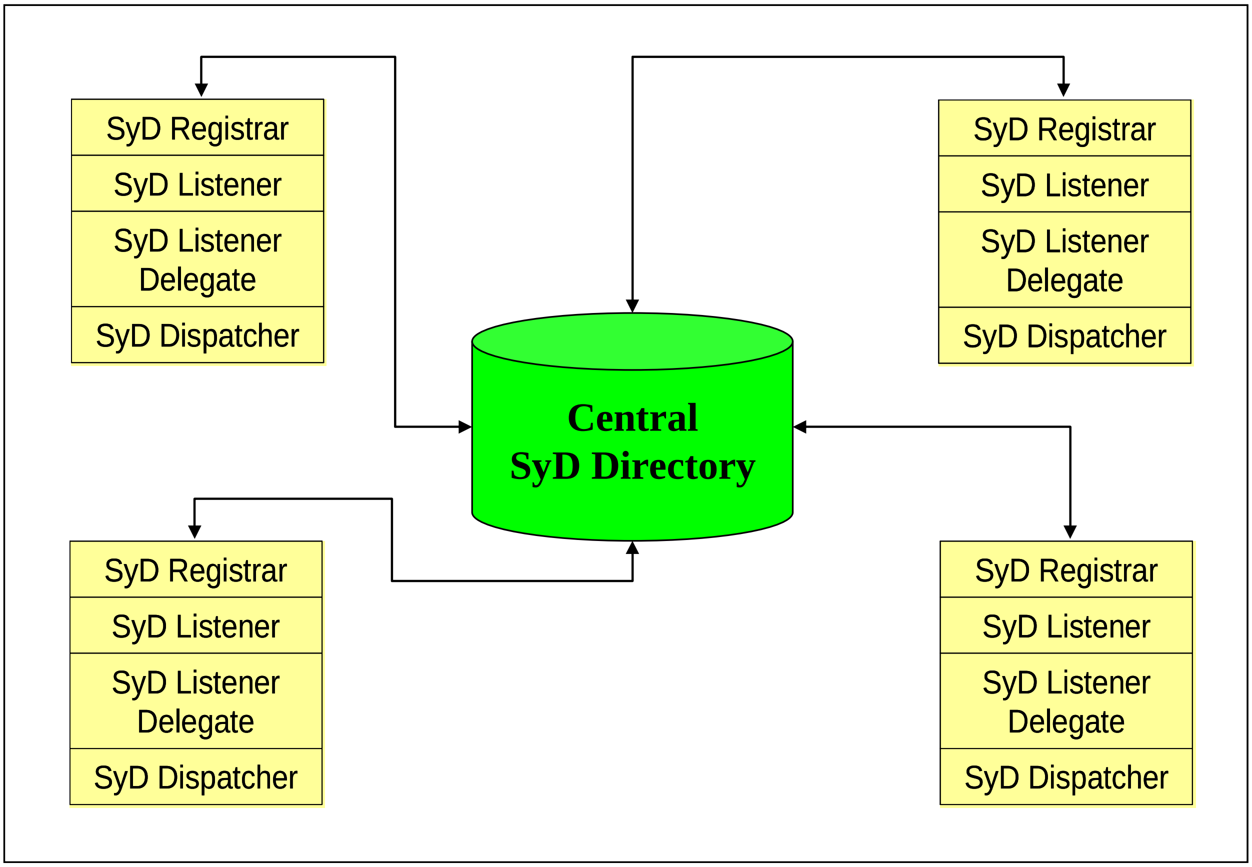


Figure 4.2: SyD Devices using a Central SyD Directory

To satisfy the requirements of our application, we need to make the SyD Directory device-resident along with the core SyD modules. There should also be a Directory Updater module that is responsible for device discovery, synchronization between directories and updating the services available on a device. This leads to a scenario as illustrated in Figure 4.3 where SyD devices with a distributed directory connect to each other.

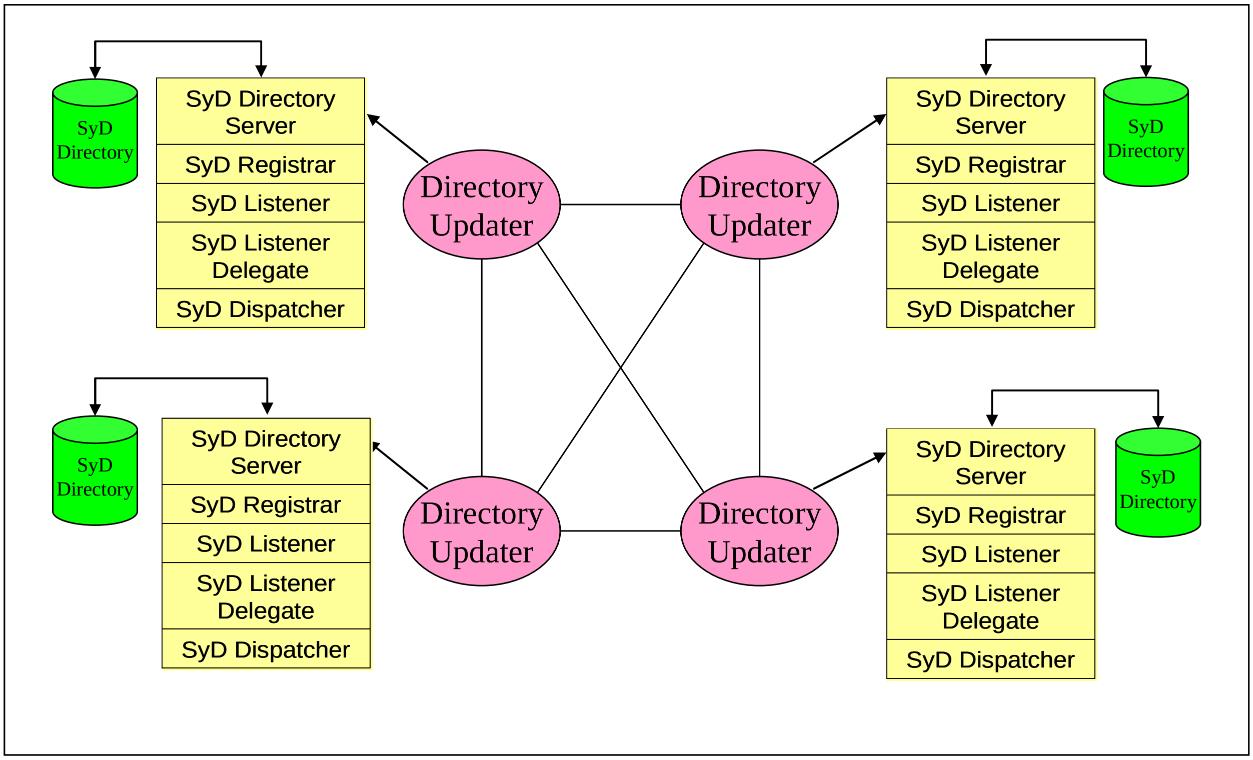


Figure 4.3: SyD Devices with Distributed SyD Directory

The figure above depicts a pure peer-to-peer computing environment wherein the devices communicate with each other directly without any kind of lookup in a central directory. Every SyD device now has its own SyD Directory. Notice the placement of the Directory Updater module whose sole purpose is to discover neighboring devices, synchronize SyD Directory data with them and update the local Application Server Objects with latest data available.

4.4.1 Support for Device Discovery

The Directory Updater module is responsible for device discovery. The network layer monitors the availability of SyD devices in the vicinity. Short-range wireless technologies like Bluetooth and Ultra Wide Band (UWB) can be used for this purpose. This module receives a trigger from the network layer whenever a neighboring SyD device is found. This trigger initiates the synchronization of the SyD Directory data between the two devices.

4.4.2 Support for accessing SyD Directory data

Once device discovery occurs, the discovered device’s details like location information are obtained so that a remote connection can be made to its Directory Server. Once connected, the discovered device’s SyD Directory object is received for synchronization purposes. Any new devices registered here are passed on to the local SyD Directory Server to be published in the local SyD Directory. This makes it possible for other devices that are searching for SyD devices to lookup in the peer’s directory and directly access the services published without actually needing to discover all the devices available. The premise here is that whenever a device is discovered, a quick update is done by comparing its Application Server Object with the local Application Server Object. Thus, a client accesses only the local application server to obtain the latest results. This can be tailored to meet an application’s specific needs.

4.4.3 Support for high-level distributed applications

Our high-level parking lot application’s server is updated with the latest data after comparing its data with that of the discovered device’s application server object. This guarantees that even after the discovered device moves away, any new client device joining the network is guaranteed to get the latest available data from our local device.

4.4.4 Effect of Frequent connections/disconnections

There is no bottleneck caused at any single point and no single point of failure. If any device fails, the features of the environment are such that there are other devices ready to co-operate in providing the necessary services as they would have been updated with the latest data available. So, the design is well suited to the transient nature of the environment.

By incorporating the Directory Updater module in the design of our SyD Distributed Directory, we satisfy the requirements of a simple proximity network that allows peer-to-peer networking with transient nodes.

4.5 Distributed SyD Directory Implementation

The SyD Distributed Directory design has two main components, namely the local SyD Directory and the Directory Updater module. Java is the programming language of choice for the implementation and we make use of the Generic Collections Framework available from Java 1.5. The SyD Directory is modeled as an object with its own methods. This is in stark contrast to the centralized version of the SyD Directory that was hosted on a relational database and was accessed using Java Database Connectivity (JDBC). In this section, we present the implementation details of the SyD Directory, SyD Directory Server, SyD Directory Updater and the way in which synchronization is achieved between two SyD Directory objects. Finally we also explain a subscription-based method for updating the local application server objects.

4.5.1 SyD Directory

The SyD Directory is modeled as an object that implements the Serializable class. This was an important design decision that was based on the need to have a persistent view of the data in the directory. Object serialization [12] is the process of saving an object's state to a sequence of bytes, as well as the process of rebuilding those bytes into a live object at some future time. The Java Serialization API provides a standard mechanism for developers to handle object serialization. Using this, each of the SyD tables that were earlier a part of the database in the centralized SyD Directory, are now a part of the Directory object. As they are Serializable, they are saved to disk in a file named “syd.data” and whenever the Directory Server is stopped and started again, the directory data is loaded from this file “syd.data” which mimics persistence. There is no lapse while loading this file as it is device resident and completely depends upon the amount of available storage space on the device.

Please refer to the source code of Directory.java in Appendix B and its methods load(filename) and save(obj,filename) for a clear depiction of how the Serialization API works. The following table details the data values or fields that are stored in each of the tables of the SyD Directory.

Table 4.1: SyD Directory Tables and Fields

SYD_USER	***SYD_USER_PROPERTIES***	**USER_APPO_MAPPING**
userID userName userPassword userURL userLiveBit userPublishTime	*userID* *userName* *directoryame* *devicehost* *directoryPort* *listenerPort* *appServerPort*	userID appID SydObjectID *localBit*
SYD_APPO	**SYD_METHOD**	**APPO_METHOD_MAPPING**
appID appName	methodID methodName methodReturnType methodParam	appID methodID
SYD_GROUP	**USER_GROUP_MAPPING**	**SYD_PROXY**
groupID groupName appID	groupID userID	proxyID userID appID proxyHost proxyPort

- SYD_USER

This table stores details about the user/device/peer that is hosting a particular service on the network. Every device is assumed to host a single instance of an application server object. This instance is then registered with a unique userName value across all devices.

- SYD_USER_PROPERTIES

This is a new table that has been introduced in the Distributed Directory version of SyD. It mainly provides a way for the Directory Updater module to obtain a device's details accurately in order to make remote method calls to services running on the ports specified in the fields.

- SYD_APPO

This table stores the name of the SyD application that can be used for the lookup of corresponding SyD server objects.

- SYD_METHOD

This table again stores the methods that are available with a particular SyD application and are accessible remotely.

- USER_APPO_MAPPING

Provides the list of applications associated with a particular user device. It also stores a SydObjectID that is used to register the object in the RMI registry. A new field called the localBit indicates whether the object is hosted on the local device or not. It is used with Subscription-based Updates to obtain a reference to the local Application Server Object.

- APPO_METHOD_MAPPING

Provides the list of methods associated with a particular SyD-enabled application.

- SYD_GROUP

This table provides support for the formation of user groups for a particular application.

- USER_GROUP_MAPPING

This table stores the list of users belonging to a group.

- SYD_PROXY

This table keeps track of proxy information for a particular SyD application.

The SYD_GROUP, USER_GROUP_MAPPING and SYD_PROXY are carried forward from the central directory version of SyD. The group tables have not been used in the distributed directory implementation due to the transient nature of the nodes in the proximity network that inhibits the maintenance of long-term groups. The proxy table is included so as to provide support for a future implementation of the proxy module.

The SyD Directory objects can now be manipulated just like any other object using their own methods, or can also be passed as parameters to other methods. This becomes important when synchronization needs to be done between two SyD Directory objects. The best argument here for a distributed directory is that there is no more lapse while accessing data from the centralized SyD Directory and there is no possibility of the Directory never being accessible as it is already device resident. This was one of the major drawbacks of the central directory version wherein the system hosting the SyD Directory should always be up and running. The SyD Directory Server provides methods for manipulating the SyD Directory that are presented in the following section.

4.5.2 SyD Directory Server

The SyD Directory Server initializes the Directory Server by creating an RMI registry at the port specified ("directoryPort" field from SYD_USER_PROPERTIES table). This Directory Server is then bound to a name so that it can be accessed remotely by other SyD devices. It also provides methods for adding data to the tables in the SyD Directory, publishing data to the directory, lookup of data from the directory and synchronization. This prevents the SyD Directory from being changed by any of the other SyD modules and control is restricted to the SyD Directory Server that a SyD Directory is attached to. The following private methods add the data provided in the method parameters to the corresponding table in the SyD Directory.

(i) SydUser (userID, userName, userPassword, userURL, userLiveBit, publishTime)

(ii) SydUserProperties (userID, userName, deviceHost, directoryPort, listenerPort, appServerPort)

(iii) SydAppo (appID, appName)

(iv) SydMethod (methodID, methodName)

(v) UserAppoMapping (userID, appID, SydObjectID, localBit)

(vi) AppoMethodMapping (appID, methodID)

Please refer to the source code in Appendix B for a complete listing of these methods.

The getDirectory( ) method is useful to obtain the device's SyD Directory as an object that can be used as a parameter to other methods. This is especially useful during SyD Directory Synchronization. The getSydObjectID (appname) returns the local application server object's ID which is useful during application update.

The Lookup(SydTable, …) method is similar in functionality to the SQL SELECT statement. It returns the value of a particular field for the condition specified in the method parameters. This function is used while publishing data to the directory and also to check whether an application object has already been published in the SyD Directory.

The publish(info) method publishes information about the SyD application server object whose details have been passed as a parameter to this method. Additions are made to the SYD_USER,SYD_APPO,USER_APPO_MAPPING,SYD_METHOD,APPO_METHOD_MAP

PING and SYD_USER_PROPERTIES tables after which the SyD Object ID associated with this application is returned. The SyD Object ID is also registered with the local RMI registry so that the application server object can be accessed remotely.

The other important methods here are Sync(directory) and Update(Application Server Object). Sync( ) performs synchronization between the local directory object and the one specified as the method parameter. Update( ) performs the subscription-based update of the application server object with the details obtained from the object in the method's parameter. As they are called by the Directory Updater, a more detailed explanation is given in the next few sections.

4.5.3 SyD Directory Updater

The SyD Director Updater is a major module that was developed to add functionality to the distributed SyD Directory. The main functions performed here are:

(i) SyD Device Discovery

(ii) SyD Distributed Directory Synchronization

(iii) Update of local server objects

From the perspective of the middleware, these Directory Updater modules of several SyD devices seem to be talking to each other about their availability and location information. As can be seen in Figure 4.3, externally these modules obtain peer device information and SyD Directory data which gets internally sent to the SyD Directory Server for Synchronization and Application Server updates. This keeps the SyD Directory safe from being arbitrarily changed by other device. Let us look at these three functions in detail in the following sections.

4.6 SyD Device Discovery

The device discovery module searches for SyD devices in the vicinity. This discovery is currently represented as an abstraction at the network layer which is responsible for activating a trigger that causes SyD Directory Synchronization upon discovery.

The process of device discovery is a matter of finding the closest neighboring SyD device. Upon device discovery, the searching device gets a copy of the discovered device's sydprop.properties file which contains the directory service name, device host, directory service port, listener port and the application server port. A connection is made to the remote directory server using the values specified in devicehost, directoryport, directoryname and its directory object is obtained. Taking this into consideration, we can come up with a set of methods that will be used during device discovery.

- device = ifDeviceFound ( ) – As the name suggests, this device waits on a trigger from the network layer that gets activated whenever a nearby device is found. This may or may not be a SyD device. The method should return the device's identification information like IP address and port so that the Directory Updater can obtain the device's sydprop.properties file using getProperties( ).
- getProperties (device) – This method should be responsible for a quick, efficient and easy way to transfer the neighboring device's sydprop.properties file to the local device by setting up a communications channel especially for this purpose. If an appropriate sydprop.properties file is found, it means that a SyD device has been discovered and it is renamed as device2.properties locally for further synchronization.

- sendProperties (device) – Corresponding to the above method, this method is on the neighboring device and after establishing the channel, sends out its sydprop.properties file to the device looking for it.

These simple high-level abstractions hide the actual networking details and can be changed to suit the type of wireless technology being used for discovery (Bluetooth, UWB or CALM etc.).

Consider two devices A and B which are in close proximity to each other. The Directory Updater modules of both the devices have been started. Let us look at the steps that take place during device discovery from device A's perspective. Also indicated are the high-level API that SyD would provide to accomplish device discovery.

(i) A trigger from the network layer tells device A about neighboring device B – B=A.ifDeviceFound ( )

(ii) A rerequests B's properties file - A.getProperties (B)

(iii) B sends out its sydprop.properties file – B.sendProperties (A)

(iv) Rename B's sydprop.properties file as "device2.properties" on A, using whose details A remotely connects to B's directory server. Device Discovery ends here, followed by synchronization.

(v) A obtains B's directory object and passes it on as a parameter to be synchronized with A's directory – Sync(B.getDirectory( )) defined in the SyD Directory Server

(vi) Every remote object that is synchronized is also updated with the local application object – Update(Remote Object Details)

By the time synchronization and update have occurred, device A and device B would have moved farther apart from each other. The same sequence of steps has occurred on device B's side too. The following figure shows the above sequence of steps.

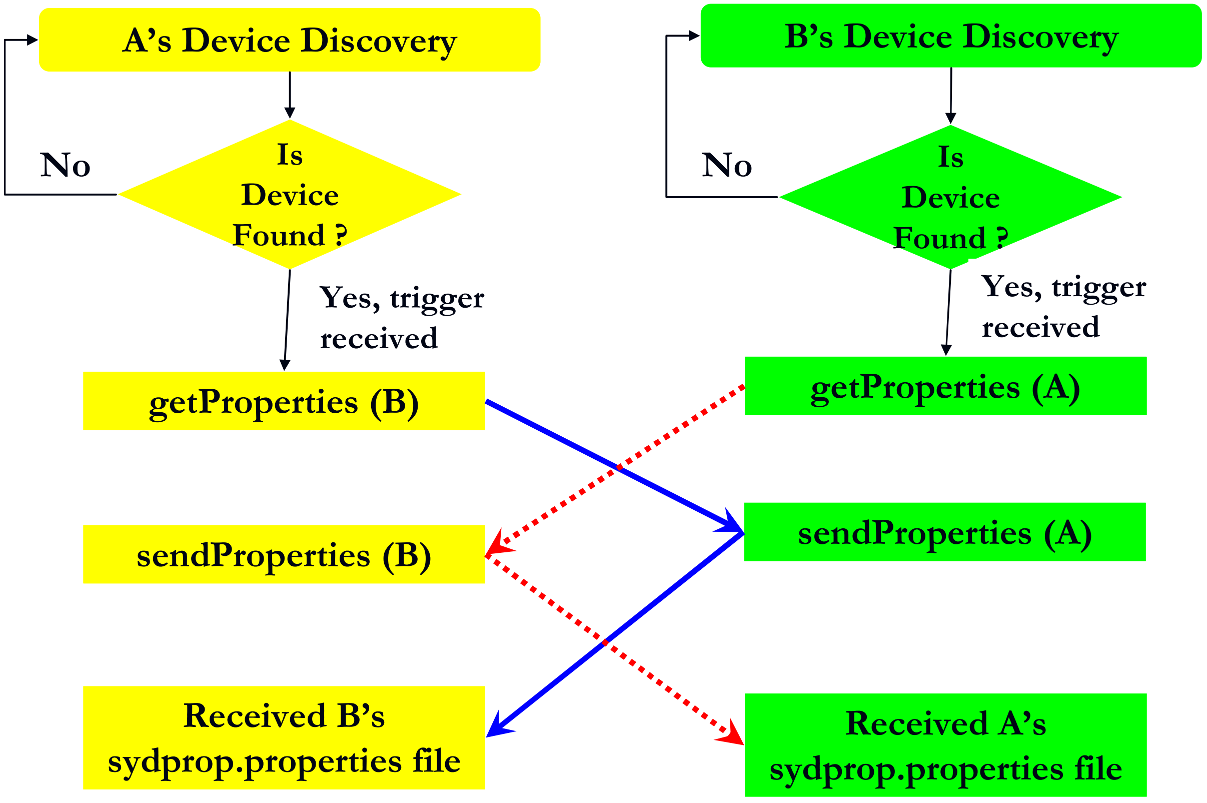


Figure 4.4: Device Discovery Process

The final result of device discovery is that device A obtains a copy of device B's sydprop.properties file which is used to remotely obtain device B's SyD Directory object used during synchronization.

The next section presents in detail the purpose and logic behind synchronization of the SyD Distributed Directory data

4.7 SyD Distributed Directory Synchronization

The SyD Directory Server (section 4.5.2) is responsible for the actual synchronization between SyD Directory data. There is no need for synchronization in a centralized directory as it is a single entity providing a complete, persistent, global view of the data to every SyD device.

We aim to achieve this view at least partially for the neighboring devices in consideration and for whom such information would make sense in a local context than a global one.

Continuing with the devices A and B from section 4.5.3, after A receives B's directory object, the Sync ( ) method provided by the SyD Directory Server is called. The synchronization that takes place from A's perspective is as follows:

(i) For every entry in B's USER_APPO_MAPPING table get the userID and appID.

(ii) Using userID from (i), obtain corresponding userName from the B's SYD_USER table.

(iii) Check if userName from (ii) is published in device A's directory.

If already published, obtain userID from directory A, else add the new user B to SYD_USER and SYD_USER_PROPERTIES tables in directory A.

(iv) Using appID from (i), obtain corresponding appName from SYD_APPO table.

(v) Check if appName from (iv) is published in device A's directory.

If already published, obtain appID from directory A, else add the new application and its methods from B to SYD_APPO, SYD_METHOD and APPO_METHOD_MAPPING tables in directory A.

(vi) Using the current values for userID and appID from steps (i) through (v) above, add a new entry to the USER_APPO_MAPPING table.

The following is a simple example of how two devices A and B synchronize their distributed SyD Directory data.

SyD Directory data for devices A and B before Synchronization

SYD_USER: (userID, userName, userPassword, userURL, userLiveBit, publishTime)

Device A => (1001, A, ***, 192.168.15.2, 1, 11/15/2007 11:30)

Device B => (1001, B, ***, 192.168.15.2, 1, 11/15/2007 12:30)

SYD_USER_PROPERTIES:

(userID, userName, directoryName, directoryHost, directoryPort, listenerHost, listenerPort)

Device A => (1001, A, DirectoryA, 192.168.15.2, 1099, 192.168.15.2, 8888)

Device B => (1001, B, DirectoryB, 192.168.15.2, 1100, 192.168.15.2, 8889)

SYD_APPO: (appID, appName)

Device A => (2001, ParkingLot)

Device B => (2001, ParkingLot)

SYD_METHOD: (methodID, methodName, returnType, methodParams)

Device A => (3001, getparkingLot, java.lang.String, String)

(3002, setparkingLot, java.lang.String, String Integer Long)

Device B => (3001, getparkingLot, java.lang.String, String)

(3002, setparkingLot, java.lang.String, String Integer Long)

USER_APPO_MAPPING: (userID, appID, SydObjectID)

Device A => (1001, 2001, 4001)

Device B => (1001, 2001, 4001)

APPO_METHOD_MAPPING: (appID, methodID)

Device A => (2001, 3001) (2001, 3002)

Device B => (2001, 3001) (2001, 3002)

SyD Directory data for devices A and B after Synchronization

SYD_USER: (userID, userName, userPassword, userURL, userLiveBit, publishTime)

Device A => (1001, A, ***, 192.168.15.2, 1, 11/15/2007 11:30)

(1002, B, ***, 192.168.15.2, 1, 11/15/2007 12:45) +

Device B => (1001, B, ***, 192.168.15.2, 1, 11/15/2007 12:30)

(1002, A, ***, 192.168.15.2, 1, 11/15/2007 12:45) +

SYD_USER_PROPERTIES:

(userID, userName, directoryName, directoryHost, directoryPort, listenerHost, listenerPort)

Device A => (1001, A, DirectoryA, 192.168.15.2, 1099, 192.168.15.2, 8888)

(1002, B, DirectoryB, 192.168.15.2, 1100, 192.168.15.2, 8889) +

Device B => (1001, B, DirectoryB, 192.168.15.2, 1100, 192.168.15.2, 8889)

(1002, A, DirectoryA, 192.168.15.2, 1099, 192.168.15.2, 8888) +

SYD_APPO: (appID, appName)

Device A => (2001, ParkingLot)

Device B => (2001, ParkingLot)

SYD_METHOD: (methodID, methodName, returnType, methodParams)

Device A => (3001, getparkingLot, java.lang.String, String)

(3002, setparkingLot, java.lang.String, String Integer Long)

Device B => (3001, getparkingLot, java.lang.String, String)

(3002, setparkingLot, java.lang.String, String Integer Long)

USER_APPO_MAPPING: (userID, appID, SydObjectID)

Device A => (1001, 2001, 4001) (1002, 2001, 4001) +

Device B => (1001, 2001, 4001) (1002, 2001, 4001) +

APPO_METHOD_MAPPING:	(appID, methodID)	
Device A =>	(2001, 3001)	(2001, 3002)
Device B =>	(2001, 3001)	(2001, 3002)

Note: Typically A and B are devices having two different IP addresses. For the purpose of this example, we have shown otherwise. Also a '+' sign next to a record indicates that it has been added newly after synchronization.

Additions have been made to the SYD_USER, SYD_USER_PROPERTIES and the USER_APPO_MAPPING tables only. As the same Parking Lot application occurs in both devices, no additions have been made in either device's directory to the SYD_APPO, SYD_METHOD or APPO_METHOD_MAPPING tables

The sample scenario shows that at the end of the synchronization process, each directory has a uniform view of applications and users. This proves that the distributed directory synchronization was effective in capturing the essence of a centralized directory without any of its drawbacks. Using this data, we now have to update the data in our application server objects which is dealt with in the following section.

4.8 Subscription-based Service Updates

An important benefit of synchronization is that having information about all the neighboring devices in one place, namely the local SyD Directory enables us to make method calls to them and receive their results. A standard method called syncAppServer( ) is made available to the application server object and is invoked using the Java Reflection API by the SyD Directory Server module after synchronization. This method would take the recently published remote Application Server Object as an argument. Data can then be compared between the local and remote application server objects. After this, whatever is the latest data based on a higher timestamp value, is updated to the local application server object. In this way, the client need only access the local server object and be assured of getting the latest data.

The Update (remote Application Server Object details) method is provided by the SyD Directory Server that is responsible for invoking the local Application Server Object's syncAppServer( ) method. Now the rules for updating the server object may vary between different applications, but having a generic name helps call the same method for the appropriate local server object. Java Reflection helps us achieve this kind of dynamic method invocation, where all we need to know is the method name, type of the parameter and parameter value for the syncAppServer( ) method. Any Application Server Object should be designed to support a method called syncAppServer(Remote Stub), which takes a single remote stub as its parameter. This makes it easy for the Update method to make a call to the syncAppServer( ) method on the corresponding local Application Server object after providing the remote stub as a parameter.

An important point to note here is that the Class object corresponding to the remote stub is of the form _Stub. For example, in the Parking Lot application, the remote stub being accessed has the following Class associated with it:

```
syd.sydapp.ParkingLot.ParkingLotImpl_Stub
```

So, the method signature of the syncAppServer( ) method defined in the ParkingLotImpl.java is

```
void syncAppServer( ParkingLotImpl_Stub p)
```

Though not implemented, we can also have the notion of a subscription bit attached to the local Application Server Object. By default, every remote object that gets published in the SyD

Directory is updated. However, the subscription bit provides a way to update only specified objects to save time and improve performance. When this bit is set to 1, it would mean that the server object is subscribed for updates and if it is not set, then no updates need be performed. This avoids wasteful processing when memory, power, storage and other resources are at a premium in mobile devices.

4.9 Chapter Summary

In this chapter, after clearly defining peer-to-peer and proximity networks, we presented a detailed description of why we need the distributed directory version of SyD, its advantages over a centralized directory and how it is well-suited for the parking lot application. A comprehensive description of the SyD Directory, SyD Directory Server and the SyD Directory Updater built the background to clearly explain the distributed directory synchronization process. Subscription-based updates of the local Application Server Objects allow results to be delivered much faster to the client as opposed to the client making remote calls to all the remote objects published in the directory.

The next chapter presents a case study of the parking lot application that enables us to clearly see the distributed directory, its synchronization and subscription-based service update in effect.

5 CASE STUDY: A PARKING LOT APPLICATION

In this chapter, we present the details of the Parking Lot application that has been developed for showcasing the working of the distributed SyD Directory and its synchronization. The preceding chapters have given us some idea about the parking lot application that highlights the need for a distributed SyD Directory for use in a peer-to-peer transient proximity-based environment.

5.1 Application Description

We proceed to substantiate the working details of the parking lot application by taking into consideration all the assumptions involved and its actual relevance to the real world. Consider a real-time traffic scenario wherein multiple vehicles are searching for available spaces in parking lots in their vicinity. A point to note here is that such information in not useful in a global context or to all vehicles that may not be looking for a parking lot or are far away from the parking lot. This fact clearly justifies the need for a distributed directory as opposed to a central directory in such an environment.

Every vehicle stores information about one or more parking lots, the number of available spaces in that lot and a timestamp that indicates when this information was last updated. At any given time, a vehicle near a parking lot would like to know which parking lot has free spaces so that it can directly go there and not waste time and fuel searching for one. This is a realistic scenario in the downtown area of major cities, where parking space is limited and expensive. There is a demand for such an application support at both the hardware and software level that provides current, up-to-date information on the fly.

Before we can proceed with the actual details of how this is accomplished, we enumerate the assumptions that were taken while designing this application.

(i) Each vehicle is equipped with a wireless device that is SyD enabled and runs the SyD middleware.

(ii) Vehicles leaving a parking lot know that the number of available spaces has increased by one. This information should typically be sensed and updated to the parking lot server object. But for now, this data is manually entered.

(iii) Vehicle discovery occurs when vehicles come into each other's transmission range that starts the synchronization process, basically making it appear as if information is exchanged as two vehicles physically cross each other.

(iv) Such SyD enabled devices cooperate with each other via the middleware to synchronize their distributed SyD Directory data.

(v) Cooperation is necessary as there is currently no secure authentication module in SyD that can take care of such details.

(vi) All cooperating devices make use of a "global" clock to generate the timestamp values.

(vii) SyD device discovery is done using some kind of powerful, robust, wireless technology like Bluetooth that provides short-term "device pairing" for simple data exchange.

Keeping these assumptions in mind, we detail the design of the application elements that will be integrated to smoothly operate with the SyD middleware.

5.2 Application Design

In a peer-to-peer system, every node behaves like both a client and a server. Therefore, we have to design both the client and server objects for our parking lot application. Each module would be responsible for performing the designated role. This clear demarcation helps to assign a set of functions to each object that confirms with the modular architecture of the SyD middleware. This high-level application design should be unaffected by the workings of the SyD middleware itself. So, details like distributed SyD Directory synchronization, SyD Listener and SyD Listener Delegate communication, SyD Registrar details should be made transparent to the application.

5.2.1 Design of Parking Lot Server

The Parking Lot server side maintains a list of parking lots, their available parking spaces and associated timestamp values. It has the following important functions:

(i) Register itself as a service with its local SyD Directory.

(ii) Provide necessary data to local or remote client requests.

(iii) Provide a generic method that allows the SyD Director Updater to update local application server objects after SyD Directory synchronization.

In order to register itself with the SyD Directory, the application server needs to know the register method's signature from the SyD Registrar module for registration to the local SyD Directory.

To service remote clients, there is a need to have a remote object with methods that service these remote requests. A remote interface that specifies the methods that can be accessed remotely, its corresponding implementation for these methods and the server itself that hosts these remote application objects must be developed for the application server. At the least, our parking lot application has methods to send parking lot information from clients and also receive new parking lot information to be added or updated to the server object's data.

A simple method called syncAppServer( ) should be made part of every kind of Application Server Object. This method would take care of updating the local server object based upon rules pertinent to the application currently in context. So, in the case of the parking lot application, this method would be responsible for updating the local server object with the latest data.

5.2.2 Design of Parking Lot Client

The Parking Lot client side requests the available spaces in a specified parking lot and assumes that it is the most recent data available. All the client needs to know is the invoke method's signature from the dispatcher module and also the names of the methods from the Parking Lot interface that are accessible by it. It requires no lower layer information like IP addresses or port numbers. Everything is taken care of by the core modules of the SyD middleware.

This shows how quick and easy it is to formulate the design for any SyD application following which the implementation becomes a matter of putting the methods together and making the necessary calls.

5.3 Application Implementation

The implementation details of the Parking Lot Application are described in this section. There are four main files that will be used for this purpose:

(i) ParkingLot.java

This is the main interface file for creating remote objects of the parking lot application. It provides three methods that can be called by remote objects, namely, getParkingLot( ) that takes a parking lot name as its method parameter and setParkingLot( ) that takes three arguments, namely the parking lot name, the number of available spaces and a timestamp value. Finally, we have the syncAppServer ( ) method that provides subscription-based application updates.

(ii) ParkingLotImpl.java

This is the implementation file for the ParkingLot interface specified in (i). The important methods implemented here are

- getParkingLot( ) to retrieve the parking lot information
- setParkingLot( ) to add or update new parking lot information.
- syncAppServer( ) is the generic method that takes as its argument another remote ParkingLot object. The local and remote objects are compared and the local server object is updated with the latest data based on the maximum timestamp value.

(iii)ParkingLotServer.java

The Server is responsible for registering itself with the SyD Directory. It accepts the username and password for accessing the server object. Using these details, an XML-like string containing the user, application and method details is published. This data is parsed and used to register the application server object in the SyD Directory. The generated SydObjectID associated with this application server object is registered with the local RMI registry at the application server's port.

A quick mention of the sydprop.properties file is necessary here. This file keeps track of the following values:

<u>directoryname</u> = The name with which the SyD Directory Server is registered

<u>devicehost</u> = IP address of SyD Device on which SyD Directory, Application Server and its Listener are running

<u>directoryport</u> = Port at which the SyD Directory Server is registered

listenerport = Port at which the SyD Listener receives invocation details from the SyD Listener Delegate

appServerPort = Port at which the Application Server Object is registered with the RMI registry.

Using these values from the sydprop.properties file which is unique for every SyD device, the Server Object is bound to a URL of the form [rmi://devicehost:appserverport/SydObjectID] which is used by the SyD middleware to access the Application Server Object.

(iv) ParkingLotClient.java

The client needs to know the name of the parking lot where the driver of a vehicle is looking for free space. This parameter, its data type and method, namely the getParkingLot, are sent to the SyD Dispatcher which makes sure they reach the correct destination sever object to obtain results.

5.4 Execution Flow

Let us look at the sequence of steps that occur in the execution of the Parking Lot application using the SyD Distributed Directory.

5.4.1 Start SyD Directory Server

The SyD Directory Server is started and ready to receive requests at the directoryport, whose value is present in the sydprop.properties file.

5.4.2 Start Application Server Object

The ParkingLotServer is started which registers itself with the SyD Directory Server, binds itself to its appserverport, whose value is specified in the sydprop.properties file, and waits for ParkingLotClient invocations.

5.4.3 Start SyD Directory Updater

The SyD Directory Updater is started and it begins the process of “discover and publish” that is discovery of neighboring SyD devices and publishing them in the local SyD Directory, so that synchronization occurs continuously independent of the application.

5.4.4 Start SyD Listener for Application Server

The SyD Listener is started and listens for client requests from the SyD Listener Delegate. It processes these requests and sends the results back over a TCP socket connection back to the SyD Listener Delegate on the client side.

5.4.5 Start SyD Application Client

The client’s user interface accepts details about the parameter values for the methods that need to be sent to the server. The SyD Dispatcher sends these requests to the SyD Listener Delegate which communicates with the SyD Listener to invoke the remote methods and obtain results. The SyD Dispatcher again receives these results from the SyD Listener Delegate and conveys them back to the client’s user interface.

This process is very similar to that of the SyD Middleware Architecture shown in Figure 3.1, the only difference being that there is also a SyD Directory Updater module now that constantly keeps updating the distributed SyD Directory.

5.5 Experimental Results

In this section, we will look at simple timing measurements that were done to find out how long it would take to actually synchronize two SyD devices. These results are important from a practical point of view as we only have very limited time available to us when two vehicles cross each other. When two Directory Services are run on the same machine, we obtain an average timing value of 300ms. This indicates the minimum time required to perform synchronization and application update.

The tables below show a sample scenario that occurs after the subscription-based update of two Application Server objects. What's important is how the data is updated between the two objects based on their timestamp values, keeping only that data which is most recent. This is indicated by the rows highlighted in red in the bottom two tables.

Table 5.1: Subscription-based Application Update on Devices A and B

BEFORE UPDATE
ParkingLotServer Data on Device A

Name	Available Spaces	Timestamp
GSU	10	11:30
TSRB	15	12:45

BEFORE UPDATE
ParkingLotServer Data on Device B

Name	Available Spaces	Timestamp
GSU	12	12:00
TSRB	8	12:30

AFTER UPDATE
ParkingLotServer Data on Device A

Name	Available Spaces	Timestamp
GSU	***12***	***12:00***
TSRB	15	12:45

AFTER UPDATE
ParkingLotServer Data on Device B

Name	Available Spaces	Timestamp
GSU	12	12:00
TSRB	***15***	***12:45***

A quick note on the java.rmi.ConnectException is necessary here. While trying to run two directory servers on two different machines, we encountered this exception that prevented us from successfully testing out the application realistically. In [19 p.202], there is a clear explanation for this kind of RMI behavior that is dependent upon router firewall settings that prevent RMI access by default. By reconfiguring these settings, it may be possible to avoid these exceptions.

5.6 Extending the Parking Lot Application

This simple setup provides an easy to understand, easy to use template that allows us to extend this application or create new applications that include more features. For example, if a positioning system is integrated with the vehicle, a distance parameter can be added that also finds the closest parking lot.

Another useful application in this category would be traffic rerouting. Instead of available parking spaces, we would maintain the average speed of traffic near certain exits or predefined landmarks. If a vehicle knows this information beforehand, slow moving traffic wherein the average speed falls below a certain threshold value, can alert the user to take an alternate route to avoid traffic.

In this way, distributing the SyD Directory has helped us develop applications that allow SyD devices to directly communicate with each other. The rapid development of any application using SyD shows its powerful simplicity that can be molded to perform more complex tasks.

6 CONCLUSION AND FUTURE RESEARCH

The work presented in this thesis focuses on providing a high-level programming environment for the rapid development of peer-to-peer applications using the distributed SyD Directory. Applications like the SyD Calendar [26], Fleet [27] and a SyD Travel Application [28] have been developed using the central directory version of SyD. But the requirements of the parking lot application urge the need for a distributed SyD Directory. With the help of this extension to SyD, SyD devices behave like smoothly conducted musical instruments in an orchestra, to produce up-to-date results to the users. We have successfully shown the methodology that paves the way for developing similar applications in the future and opens up the following research issues that need to be explored further.

The next step in this research would logically be to port the code developed here to mobile handheld devices like iPAQs with hardware support for wireless communication using IEEE 802.11 or Bluetooth. Testing on these devices would paint a realistic picture of the problems presented by the environment and the amount of time required to perform synchronization and updates.

In a peer-to-peer network, instead of providing subscription-based service updates, a mobile agent paradigm can be designed that would launch a “virtual knight” kind of program that has the ability to ‘jump’ from one peer to the next to collect service related information. This virtual knight can keep sending results back to the source peer and continue on its way to collect results from some more peers.

There is immense scope for developing a SyD Security module that encapsulates the core SyD modules from being tampered by any outside entity. In the real world, with increasing concerns over security and privacy, such a module is imperative in the final delivery of a working SyD application. Also, as described in an earlier chapter about SyD device discovery, Bluetooth is seen as an effective way to accomplish this. Bluetooth is a short-range wireless technology that is emerging as a de-facto technology for wireless network. Its attractive features like support for ad-hoc connections, power efficiency and robustness to interference makes it suitable for creating proximity networks. The advantage of using Bluetooth is that it has a well defined protocol to discover, authenticate and connect to other devices.

This thesis reiterates the fact that middleware is a major building block for the development of future software systems. Future applications will need to cope with advanced properties such as context-awareness and mobility, for which adequate middleware support must be devised, together with accompanying software development notations, methods and tools. There is a need for closer integration of middleware with application development that ultimately suggests middleware-based software processes.

The vision of future computing infrastructure is towards a global, virtually loosely connected world with invisible computers everywhere that are embedded in the environment. Exploiting both mobility and availability of a potentially infinite number of heterogeneous resources at the same time requires scalability and high performance resource-discovery. With all the technology that is available to us at our fingertips, it is up to us to develop frameworks to make this technology work for us in pro-active ways for our benefit.

REFERENCES

[1] Valerie Issarny, Mauro Caporuscio, Nikolaos Georgantas, A Perspective on the Future of Middleware-Based Software Engineering, in 2007 Future of Software Engineering. 2007, IEEE Computer Society.

[2] S. K. Prasad, A. Hariharan, W. Xie, P. Madiraju, S. Malladi, R. Sivakumar, A. Zelikovsky, Y. Zhang, Y. Pan, S. Belkasim, Syd: A Middleware Testbed for Collaborative Applications over Small Heterogeneous Devices and Data Stores, in Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware. 2004. p. 352-371.

[3] S. K. Prasad, V. Madisetti, R. Sunderraman, System on Mobile Devices (Syd): Kernel Design and Implementation, in First International Conference on Mobile Systems, Applications, and Services (MobiSys), Poster and Demo Presentation, May. 2003. p. 5-8.

[4] S. K. Prasad, E. Dogdu, R. Sunderraman, B. Liu, V. Madisetti, Design and Implementation of a Listener Module for Handheld Mobile Devices, in Proceedings of 41st Annual ACM Southeast Conference. 2003. p. 7-8.

[5] S. Asthana, D. N. Kalofonos, P. Shah, Qiao Chunming, An Experimental End-Node Architecture and Communications Middleware for Dynamic Proximity Networks, in 2nd International Conference on Broadband Networks. 2005. p. 787-796 Vol. 2.

[6] Mohammad Malli, Chadi Barakat, Walid Dabbous, Application-Level Versus Network-Level Proximity, in Lecture Notes in Computer Science. 2005.

[7] M. Satyanarayanan, Pervasive Computing: Vision and Challenges, in IEEE Personal Communications. 2001.

[8] Y. Luo, O. Wolfson, B. Xu, A Spatio-Temporal Approach to Selective Data Dissemination in Mobile Peer-to-Peer Networks, in Proceedings of the Third International Conference on Wireless and Mobile Communications. 2007.

[9] Sergio Marti, Venky Krishnan, Carmen: A Dynamic Service Discovery Architecture, in Hewlett-Packard Labs. 2002.

[10] Seyed Amin Hosseini Seno, Rahmat Budiarto, Tat-Chee Wan, Survey and New Approach in Service Discovery and Advertisement for Mobile Ad Hoc Networks, in IJCSNS International Journal of Computer Science and Network Security. 2007. p. 275 - 284.

[11] W. Xie, Supporting Distributed Transaction Processing over Mobile and Heterogeneous Platforms. Ph.D. Dissertation 2005, Georgia Institute of Technology.

[12] Todd Greanier. Discover the Secrets of the Java Serialization API. 2000 http://java.sun.com/developer/technicalArticles/Programming/serialization/

[13] Dr.Chunming Qiao, Somil Asthana. Dynamic Proximity Networking. 2005 http://www.cse.buffalo.edu/~qiao/proximity/

[14] Wikipedia. Peer-to-Peer Computing. http://en.wikipedia.org/wiki/Peer-to-peer

[15] Ouri Wolfson. Mobile Peer-to-Peer Computing (Mobi-Dik). http://www.cs.uic.edu/~wolfson/html/p2p.html

[16] Doug Lowe, Java All-in-One Desk Reference for Dummies. 2005.

[17] Alfred Wai-Sing Loo, Peer-to-Peer Computing: Building Supercomputers with Web Technologies. 2007.

[18] Merlin Hughes, Michael Shoffner, Derek Hamner, Java Network Programming: A Complete Guide to Networking, Streams, and Distributed Computing 1999.

[19] Esmond Pitt, Kathy McNiff, Java.RMI: The Remote Method Invocation Guide 2001.

[20] Bill McCarty, Luke Cassady-Dorion, Java Distributed Objects. 2001.

[21] Ouri Wolfson et al., MOBI-DIC: MOBIle DIscovery of loCal Resources in Peer-to-Peer Wireless Network, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, Vol. 28, No. 3, Special Issue on Database Issues for Location Data Management, Sept. 2005, pp. 11-18.

[22] Ouri Wolfson et al., A Feasibility Study on Disseminating Spatio-temporal Information via Vehicular Ad-hoc Networks, Proc. of the Third International Workshop on Vehicle-to-Vehicle Communications (V2VCOM), Istanbul, Turkey, June 2007.

[23] Ouri Wolfson et al., Mobile Peer-to-Peer Databases, in the Encyclopedia of Geographic Information Science, to be published by Springer, July 2007.

[24] Ouri Wolfson et al., Search-and-Discover in Mobile P2P Network Databases, in The 26th International Conference on Distributed Computing Systems, Lisboa, Portugal, July 4-7, 2006, pp. 1-9

[25] Ouri Wolfson et al., Opportunistic Data Dissemination in Mobile Peer-to-Peer Networks, Springer Verlag Lecture Notes in Computer Science vol. 3633, Proceedings of the 9th International Symposium on Spatial and Temporal Databases, Angra dos Reis, Brazil, Aug., 2005, pp. 346-363.

[26] S. K. Prasad, A. G. Bourgeois, E. Dogdu, R. Sunderraman, Y, Pan, S. Navathe, V. Madisetti. 2003. Implementation of a Calendar Application Based on SyD Coordination

Links, Proceedings of The Third International Workshop on Internet Computing and E-Commerce in conjunction with the 17th Annual International Parallel & Distributed Processing Symposium (IPDPS 2003), IEEE, 22-26 April, Nice, France.

[27] S. K. Prasad, M. Weeks, Y. Zhang, A. Zelikovsky, S. Belkasim, R. Sunderraman, and V. Madisetti. 2003. Toward an Easy Programming Environment for Implementing Mobile Applications: A Fleet Application Case Study using SyD Middleware, IEEE Intl Workshop on Web Based Systems and Applications (WEBSA), at 27th Ann Intl Comp. Softw and Applns Conf (COMPSAC 2003), Dallas, Nov 3-6.

[28] A. Hariharan, S. K. Prasad, A. G. Bourgeois, E. Dogdu, S. Navathe, R. Sunderraman, and Y. Pan. 2004. A Framework for Constraint-based Collaborative Web Service Applications and a Travel Application Case Study. In Proc. International Symposium on Web Services and Applications (ISWS'04), June 21-24, Las Vegas, pp. 866-872.

[29] Arthi Hariharan, Constraint-based Collaborative Web Services, Master’s Thesis 2003, Georgia State University.

[30] W. Xie, S. B. Navathe, S. K. Prasad. 2005. Filter Indexing: A Scalable Solution to Large Subscription Based Systems. The 10th International Conference on Database Systems for Advanced Applications (DASFAA 2005), April 18-20, 2005, Beijing, China.

APPENDIX A: SYD SETUP DETAILS

In order to set up SyD and run SyD applications on a device, the details on the following checklist must be taken care of.

- Install the Java Development Kit and Java Runtime Environment 1.5 or higher.
- Make sure a period (.) is added to your CLASSPATH. Add the following files too. They can be downloaded from the Internet if not available.
    - JDBC Drivers: classes12.jar
    - XML Parsing: xercesImpl.jar
- Check the values of the host and port in sydprop.properties
- Always compile files from the directory or folder in which you place them due to directory structure dependency in 0-SyDCompiler.bat.

The contents of the compilation file 0-SyDCompiler.bat are as follows:

```
javac syd\sydutil\SyDDoc.java
javac syd\sydutil\Publisher.java
javac syd\sydutil\SyDPropertyFile.java

javac syd\syddirectory\Directory.java
javac syd\syddirectory\DirectoryServer.java
javac syd\syddirectory\DirectoryServerImpl.java
rmic -vcompat syd.syddirectory.DirectoryServerImpl
javac syd\syddirectory\DirectoryUpdater.java

javac syd\sydlistener\SyDListener.java
javac syd\sydlistener\SyDListenerDelegate.java
javac syd\sydlistener\SyDRegistrar.java

javac syd\sydengine\SyDDispatcher.java

javac syd\sydapp\ParkingLot\ParkingLot.java
javac syd\sydapp\ParkingLot\ParkingLotImpl.java
rmic -vcompat syd.sydapp.ParkingLot.ParkingLotImpl
javac syd\sydapp\ParkingLot\ParkingLotServer.java

javac ParkingLotClient.java
```

The directory structure for all the SyD files is shown in the figure below. A + indicates a folder and – indicates a file. Java 1.5 or later is preferred for the JDK and JRE. The batch files are to be executed in the naming order starting from 1 onwards.

```
-----------------------------------------------------------------------------

+ Folder in Home Directory (eg: C:\syddemo)
      + syd
            + sydapp
                  + ParkingLot
                        - ParkingLot.java
                        - ParkingLotImpl.java
                        - ParkingLotServer.java
            + syddirectory
                  - Directory.java
                  - DirectoryServer.java
                  - DirectoryServerImpl.java
                  - DirectoryUpdater.java
            + sydengine
                  - SyDDispatcher.java
            + sydlistener
                  - SyDListener.java
                  - SyDListenerDelegate.java
                  - SyDRegistrar.java
            + sydutil
                  - Publisher.java
                  - SyDDoc.java
                  - SyDPropertyFile.java
      - 0-SyDCompiler.bat
      - 1-SyDDirectoryServer.bat
      - 2-ParkingLotServer.bat
      - 3-ParkingLotListener.bat
      - 4-SyDDirectoryUpdater.bat
      - 5-ParkingLotClient.java
      - sydprop.properties

-----------------------------------------------------------------------------
```

APPENDIX B: SOURCE CODE

The source code for the SyD middleware using the distributed SyD Directory and the Parking Lot Application are included here. The code is organized as follows:

- syddirectory - Major contribution of this thesis work
    - Directory.java
    - DirectoryServer.java
    - DirectoryServerImpl.java
    - DirectoryUpdater.java
- sydengine
    - SyDDispatcher.java
- sydlistener
    - SyDListener.java
    - SyDListenerDelegate.java
    - SyDRegistrar.java
- sydutil.java
    - Publisher.java
    - SyDDoc.java
    - SyDPropertyFile.java
- sydapp.ParkingLot - Case Study developed for this Thesis work
    - ParkingLot.java
    - ParkingLotImpl.java
    - ParkingLotServer.java
- ParkingLotClient.java

Directory.java

```
package syd.syddirectory;

import java.util.*;
import java.io.*;

public class Directory implements Serializable
{
      private static final long serialVersionUID = 42L;
      Vector<Integer> ID;
      Vector<HashMap<String,String>> SYD_USER;
      Vector<HashMap<String,String>> SYD_APPO;
      Vector<HashMap<String,String>> SYD_METHOD;
      Vector<HashMap<String,String>> USER_APPO_MAPPING;
      Vector<HashMap<String,String>> APPO_METHOD_MAPPING;
      Vector<HashMap<String,String>> SYD_USER_PROPERTIES;
//    Vector<HashMap<String,String>> SYD_GROUP;
//    Vector<HashMap<String,String>> USER_GROUP_MAPPING;

      Directory()
      {
            ID = new Vector<Integer>(5);
            SYD_USER = new Vector<HashMap<String,String>>(1);
            SYD_APPO = new Vector<HashMap<String,String>>(1);
            SYD_METHOD = new Vector<HashMap<String,String>>(1);
            USER_APPO_MAPPING = new Vector<HashMap<String,String>>(1);
            APPO_METHOD_MAPPING = new Vector<HashMap<String,String>>(1);
            SYD_USER_PROPERTIES = new Vector<HashMap<String,String>>(1);
//          SYD_GROUP = new Vector<HashMap<String,String>>(1);
//          USER_GROUP_MAPPING = new Vector<HashMap<String,String>>(1);
      }

      String nextID(int n)
      {
            Integer i = (Integer)ID.get(n);
            i++;
            ID.setElementAt((Integer) i,n);
            return Integer.toString(i);
      }

      //Returns record from SydTable matching the key and value pair
specified
      public HashMap<String,String> getRecord(String SydTable,String
key,String value)
      {
            Vector<HashMap<String,String>> table = new
Vector<HashMap<String,String>>();
            HashMap<String,String> record = null, t = null;
            table = getTable(SydTable);
            for(int i=0;i<table.size();i++)
            {
                  record = table.get(i);
                  if(record.get(key).equals(value))
                        t = record;
            }
            return t;
      }

      /**
      * @param filename String - The filename for the file to be loaded
      */
      Object load(String filename) throws Exception
      {
         ObjectInputStream objstream = new ObjectInputStream(new
FileInputStream(filename));
         Object obj = objstream.readObject();
         objstream.close();
         return obj;
      }

   /**
    * @param obj Object - The object that is saved.
    * @param filename String - The filename of the file it is saved to.
    */
      void save(Object obj,String filename) throws IOException
      {
            ObjectOutputStream objstream = new ObjectOutputStream(new
FileOutputStream(filename));
            objstream.writeObject(obj);
            objstream.close();
      }

      void DisplayDirectory()
      {
            System.out.println("\nSYD_USER: " + show(SYD_USER));
            System.out.println("\nSYD_APPO: " + show(SYD_APPO));
            System.out.println("\nSYD_METHOD: "+ show(SYD_METHOD));
            System.out.println("\nUSER_APPO_MAPPING: " +
show(USER_APPO_MAPPING));
            System.out.println("\nAPPO_METHOD_MAPPING: " +
show(APPO_METHOD_MAPPING));
            System.out.println("\nSYD_USER_PROPERTIES: " +
show(SYD_USER_PROPERTIES));
            System.out.println("\nID = " + ID);
//          System.out.print("SYD_GROUP");
//          System.out.print("USER_GROUP_MAPPING");
      }
      String show(Vector<HashMap<String,String>> v)
      {
            String s = "\n";
            HashMap<String,String> h = new HashMap<String,String>();

            for(int i = 0;i < v.size(); i++)
            {
                  h = v.get(i);
                  s = s + h.toString() + "\n";
            }
            return s;
      }

      Vector<HashMap<String,String>> getTable(String SydTable)
      {
            Vector<HashMap<String,String>> table = null;

            if(SydTable.equalsIgnoreCase("SYD_USER"))
                  table = new Vector<HashMap<String,String>>(SYD_USER);
            else if(SydTable.equalsIgnoreCase("SYD_APPO"))
                  table = new Vector<HashMap<String,String>>(SYD_APPO);
            else if(SydTable.equalsIgnoreCase("SYD_METHOD"))
                  table = new Vector<HashMap<String,String>>(SYD_METHOD);
            else if(SydTable.equalsIgnoreCase("USER_APPO_MAPPING"))
                  table = new
Vector<HashMap<String,String>>(USER_APPO_MAPPING);
            else if(SydTable.equalsIgnoreCase("APPO_METHOD_MAPPING"))
                  table = new
Vector<HashMap<String,String>>(APPO_METHOD_MAPPING);
            else if(SydTable.equalsIgnoreCase("SYD_USER_PROPERTIES"))
                  table = new
Vector<HashMap<String,String>>(SYD_USER_PROPERTIES);
/*          else if(SydTable.equalsIgnoreCase("SYD_GROUP"))
                  table = SYD_GROUP;
            else if(SydTable.equalsIgnoreCase("USER_GROUP_MAPPING"))
                  table = USER_GROUP_MAPPING;
*/          else
                  System.out.println("Invalid SydTable: " + SydTable);

            return table;
      }
}

-----------------------------------------------------------------------------
```

DirectoryServer.java

```
package syd.syddirectory;

import java.io.*;
import java.util.*;
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public interface DirectoryServer extends Remote
{
      public Directory getDirectory() throws RemoteException;

      public String getSydObjectID(String appname) throws RemoteException;

      public void Sync(Directory d) throws RemoteException;

      public String Lookup(String SydTable,String 
LookupKey,HashMap<String,String> map) throws RemoteException;

      public String publish(String info) throws RemoteException;

      //SyD Group Methods
//    public void addUser(String groupname,String username) throws 
RemoteException;

//    public void deleteUser(String groupName,String username) throws 
RemoteException;

//    public String listUsers(String groupname) throws RemoteException;
}
```

DirectoryServerImpl.java

```
package syd.syddirectory;

import syd.sydutil.*;
import syd.sydengine.*;

import java.io.*;
import java.util.*;
import java.text.*;
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.lang.reflect.*;

public class DirectoryServerImpl extends UnicastRemoteObject
                        implements DirectoryServer
{
      private static final long serialVersionUID = 1L;
      private String DSName;
      private Integer DSPort;
      public static Directory d;

      private DirectoryServerImpl(String dsname,Integer port) throws
RemoteException,Exception
      {
            DSName = dsname;
            DSPort = port;
            Directory temp = new Directory();

            try
            {
                  d = (Directory) temp.load("syd.data");
                  System.out.println("SyD Directory Loaded");
                  d.DisplayDirectory();
            }
            catch(FileNotFoundException e)
            {
                  System.out.println("Creating new SyD Directory ...");
                  d = new Directory();
                  d.ID.add((Integer) 0);
                  d.ID.add((Integer) 1000);
                  d.ID.add((Integer) 2000);
                  d.ID.add((Integer) 3000);
                  d.ID.add((Integer) 4000);
            }
      }

      private static Registry initialize(int port) throws RemoteException
      {
            try
            {
                  return LocateRegistry.createRegistry(port);
          }
            catch (Exception noRegistry)
            {
                  return LocateRegistry.getRegistry(port);
            }
      }

      private void SAVE() throws RemoteException
      {
            try
            {
                  d.save(d,"syd.data");
                  System.out.println("\nSyD Directory Updated\n");
            }
            catch(Exception e)
            {
                  System.out.println("Error updating SyD Directory: " + e);
            }
      }

      public Directory getDirectory() throws RemoteException
      {
            return d;
      }

      private String sysdate() throws RemoteException
      {
            Date date = new Date();
            SimpleDateFormat sdf = new SimpleDateFormat("MMMM dd,yyyy hh:mm
aa");
            return sdf.format(date);
      }

      private void SydUser(String id,String user,String pass,String
url,String bit,String ptime) throws RemoteException
      {
            HashMap<String,String> v = new HashMap<String,String>();
            v.put("userID",id);
            v.put("userName",user);
            v.put("userPasswd",pass);
            v.put("userURL",url);
            v.put("userLiveBit",bit);
            v.put("userPublishTime",ptime);

            d.SYD_USER.add(v);
      }
      private void SydAppo(String id,String app) throws RemoteException
      {
            HashMap<String,String> v = new HashMap<String,String>();
            v.put("appID",id);
            v.put("appName",app);

            d.SYD_APPO.add(v);
      }

      private void SydMethod(String id,String method,String returnType,String
paramTypes) throws RemoteException
      {
            HashMap<String,String> v = new HashMap<String,String>();
            v.put("methodID",id);
            v.put("methodName",method);
            v.put("methodReturnType",returnType);
            v.put("methodParamTypes",paramTypes);

            d.SYD_METHOD.add(v);
      }

      private void UserAppoMapping(String uid,String appid,String
SydObjectID,String localBit) throws RemoteException
      {
            HashMap<String,String> v = new HashMap<String,String>();
            v.put("userID",uid);
            v.put("appID",appid);
            v.put("SydObjectID",SydObjectID);
            v.put("localBit",localBit);   //indicates whether object is local
or remote

            d.USER_APPO_MAPPING.add(v);
      }

      private void AppoMethodMapping(String appID,String methodID) throws
RemoteException
      {
            HashMap<String,String> v = new HashMap<String,String>();
            v.put("appID",appID);
            v.put("methodID",methodID);

            d.APPO_METHOD_MAPPING.add(v);
      }

      private void SydUserProperties(String userID,String userName,String
filename) throws RemoteException
      {
            SyDPropertyFile p = new SyDPropertyFile(filename);
            HashMap<String,String> v = new HashMap<String,String>();
            v.put("userID",userID);
            v.put("userName",userName);
            v.put("directoryname",p.getValue("directoryname"));
            v.put("devicehost",p.getValue("devicehost"));
            v.put("directoryport",p.getValue("directoryport"));
            v.put("listenerport",p.getValue("listenerport"));
            v.put("appserverport",p.getValue("appserverport"));

            d.SYD_USER_PROPERTIES.add(v);
      }

      //Returns the SydObjectID for the local Application Server Object for
the appname specified
      public String getSydObjectID(String appname) throws RemoteException
      {
            HashMap<String,String> record = new HashMap<String,String>();
            HashMap<String,String> h = new HashMap<String,String>();
            String objID = "";

            h.put("appName",appname);
            String appID = Lookup("SYD_APPO","appID",h);
            h.clear();

            for(int i = 0; i< d.USER_APPO_MAPPING.size(); i++)
            {
                  record = d.USER_APPO_MAPPING.get(i);
                  if(record.get("localBit").equals("1") &&
record.get("appID").equals(appID))
                        objID = record.get("SydObjectID");
            }
            return objID;
      }


      //Returns true or false depending on whether record exists or not
      private Boolean ifPublished(String SydTable,String key,String value)
throws RemoteException
      {
            Vector<HashMap<String,String>> table = new
Vector<HashMap<String,String>>();
            HashMap<String,String> record= new HashMap<String,String>();

            table = d.getTable(SydTable);
            for(int i = 0;i<table.size();i++)
            {
                  record = table.get(i);
                  if(record.get(key).equalsIgnoreCase(value))
                        return true;
            }
            return false;
      }

      //synchronize local directory d with remote directory d2
      public void Sync(Directory d2) throws RemoteException
      {
            Boolean userPublished = new Boolean(false);
            Boolean appPublished = new Boolean(false);
            String userID = "",appID = "",methodID = "";

            //Get all of d2's server objects
            for(int i = 0; i < d2.USER_APPO_MAPPING.size(); i++)
            {
                  HashMap<String,String> h = new HashMap<String,String>();
                  h = d2.USER_APPO_MAPPING.get(i);
                  String d2userID = h.get("userID");
                  String d2appID = h.get("appID");
                  String d2objID = h.get("SydObjectID");

                  //Begin publishing to local SyD Directory
                  //Add user to SydUser table
                  h = d2.getRecord("SYD_USER","userID",d2userID);
                  String username = h.get("userName");
                  String userURL = h.get("userURL");

                  userPublished = 
ifPublished("SYD_USER","userName",username);
                  if(!userPublished)
                  {
                        //change the userID to be consistent with local 
directory
                        userID = d.nextID(1);

      SydUser(userID,h.get("userName"),h.get("userPasswd"),h.get("userURL"),h
.get("userLiveBit"),h.get("userPublishTime"));

                        //add user to SydUserProperties table
                        h = 
d2.getRecord("SYD_USER_PROPERTIES","userID",d2userID);

      SydUserProperties(userID,h.get("userName"),"device2");
                  }
                  else
                  {
                        h = d.getRecord("SYD_USER","userName",username);
                        userID = h.get("userID");
                  }

                  //add app details
                  h = d2.getRecord("SYD_APPO","appID",d2appID);
                  String appname = h.get("appName");
            appPublished = ifPublished("SYD_APPO","appName",appname);
            if(!appPublished)
            {
                appID = d.nextID(2);
                SydAppo(appID,h.get("appName"));

                //Also add methods
                for(int j = 0;j<d2.APPO_METHOD_MAPPING.size();j++)
                {
                    h = d2.APPO_METHOD_MAPPING.get(j);
                    String d2methodID = h.get("methodID");
                    methodID = d.nextID(3);
                    AppoMethodMapping(appID,methodID);

                    h = 
d2.getRecord("SYD_METHOD","methodID",d2methodID);

    SydMethod(methodID,h.get("methodName"),h.get("methodReturnType"),h.get(
"methodParamTypes"));
                }
            }
            else
            {
                h = d.getRecord("SYD_APPO","appName",appname);
                appID = h.get("appID");
            }

            if(!userPublished)
                UserAppoMapping(userID,appID,d2objID,"0");

            Update(userID,appID,d2objID);
            System.out.println("Application Server Object Updated for 
userID = "+userID+" appID = "+appID+" objID = " + d2objID);
        }

        SAVE();
        d.DisplayDirectory();
    }

    void Update(String userID, String appID, String objID) throws 
RemoteException
    {
        HashMap<String,String> h = new HashMap<String,String>();

        h = d.getRecord("SYD_USER_PROPERTIES","userID",userID);
        String appHost = h.get("devicehost");    //listener is for the 
appServer, so same Host
        String appPort = h.get("appserverport");
        String url = "rmi://" + appHost + ":" + appPort + "/" + objID;
```

```
            h = d.getRecord("SYD_APPO","appID",appID);
            String appname = h.get("appName");

            String localObjID = getSydObjectID(appname);    //Get local
Application Server Object's ID
            SyDPropertyFile p = new SyDPropertyFile("sydprop");
            String localAppHost = p.getValue("devicehost");
            String localAppPort = p.getValue("appserverport");
            String localURL = "rmi://" + localAppHost + ":" + localAppPort +
"/" + localObjID;

            //Get Remote Object & Local Object
            Object localRef = null,stubRef = null;
            try
            {
                  localRef = Naming.lookup(localURL);
            }
            catch(RemoteException ex)
            {
                  System.err.println("Couldn't contact rmiregistry for" +
localURL);
            }
            catch(NotBoundException ex)
            {
                  System.err.println("There is no object bound to " +
localURL);
            }
            catch(MalformedURLException ex)
            {
                  System.err.println("Invalid URL " + localURL);
            }

            try
            {
                  stubRef = Naming.lookup(url);
            }
            catch(RemoteException ex)
            {
                  System.err.println("Couldn't contact rmiregistry for " +
url);
            }
            catch(NotBoundException ex)
            {
                  System.err.println("There is no object bound to "+url);
            }
            catch(MalformedURLException ex)
            {
                  System.err.println("Invalid URL: "+ url);
            }
            try
            {
                  //syd.sydapp.ParkingLot.ParkingLotImpl
                  Class<?> appClass = 
Class.forName("syd.sydapp."+appname+"."+appname+"Impl_Stub");

                  Method m = null;
                  Object result = null;
                  Class[] paramTypes = {stubRef.getClass()};
                  Object[] paramValues = {stubRef};
                  m = appClass.getMethod("syncAppServer",paramTypes);
                  result = m.invoke(localRef, paramValues);

                  //getAllData() method can be coded for all server objects
                  //Here it is used to return server data - not mandatory
                  paramTypes = null;
                  paramValues = null;
                  m = appClass.getMethod("getAllData",paramTypes);
                  result = m.invoke(stubRef, paramValues);
                  System.out.println("getAllData remote = " + 
result.toString());
                  result = m.invoke(localRef, paramValues);
                  System.out.println("getAllData local = " + 
result.toString());
            }
            catch(Exception e)
            {
                  System.out.println("Error in Application Server Update in 
DirectoryServerImpl: " + e);
                  e.printStackTrace();
            }

      }

      //Searching Directory
      public String Lookup(String SydTable,String 
LookupKey,HashMap<String,String> map) throws RemoteException
      {
            Vector<HashMap<String,String>> table = new 
Vector<HashMap<String,String>>();
            HashMap<String,String> record = new HashMap<String,String>();
            Set<Map.Entry<String,String>> condition = map.entrySet();
            String t = null;
            int recordNum = -1;
            int i = 0,j = 0,k = 0;

            //Search the table
            table = d.getTable(SydTable);
            for(i=0; i < table.size(); i++)
            {
```

```
            record = table.get(i);

            j = 0;
            k = 0;
            //AND Condition Check within record
            for (Map.Entry<String,String> x : condition)
            {
                  j = j + 1;
                  String key = x.getKey();
                  String val = x.getValue();

                  if(record.get(key).equalsIgnoreCase(val))
                  {
                        k = k + 1;
                        recordNum = i;
                  }
            }
            if(j == k)  break;
      }

      if(recordNum != -1)
            t = table.get(recordNum).get(LookupKey);

      return t;
}

public String publish(String info) throws RemoteException
{
      //Used to store key-value pairs for searching in DirectoryLookup
      HashMap<String,String> map = new HashMap<String,String>();

      Publisher p = new Publisher(info);
      String userName = (p.getSimpleValue("userName"));
      String appName = (p.getSimpleValue("appName"));
      String userPass = (p.getSimpleValue("userPasswd"));
      String url = (p.getSimpleValue("userURL"));
    String bit = "1";
      String proxyID = (p.getSimpleValue("proxyID"));

      //Check if SyDAppO has already been published
      String userID = null,appID = null, objID = null,methodID = null;
      map.put("userName",userName);
      userID = Lookup("SYD_USER","userID",map);
      map.clear();
      map.put("appName",appName);
      appID = Lookup("SYD_APPO","appID",map);
      map.clear();

      if(userID != null && appID != null)
      {
            map.put("userID",userID);
            map.put("appID",appID);
            objID = Lookup("USER_APPO_MAPPING","SydObjectID",map);
            map.clear();
      }

      if(objID != null) return objID;

      //IF this is a new SyDAppO, publish it in the Directory
      if(userID == null)
      {
            userID = d.nextID(1);
            SydUser(userID,userName,userPass,url,bit,sysdate());
      }
      if(appID == null)
      {
            appID = d.nextID(2);
            SydAppo(appID,appName);

            p.getMethodDetails();
            Vector methodNames = p.getMethodNames();
            Vector returnTypes = p.getReturnTypes();
            Vector paramTypes = p.getParamTypes();

            String temp3="", temp4="";
            for(int i=0;i<methodNames.size();i++)
            {
                  String temp1 = (String)methodNames.elementAt(i);
                  String temp2 =(String)returnTypes.elementAt(i);
                  Vector params = (Vector)paramTypes.elementAt(i);
                  for (int j=0;j<params.size() ;j++ )
                  {
                        temp3 = (String)params.elementAt(j);
                        temp4 = temp4 +" "+temp3;
                  }

                  SydMethod(d.nextID(3),temp1,temp2,temp4);
                  temp4="";
            }

            for(int k=0;k<methodNames.size();k++)
            {
                  String temp = (String)methodNames.elementAt(k);
                  map.put("methodName",temp);
                  methodID = Lookup("SYD_METHOD","methodID",map);
//                System.out.println("methodID = "+ methodID);
                  AppoMethodMapping(appID,methodID);
            }
      }
```

```
            objID = d.nextID(4);
            UserAppoMapping(userID,appID,objID,"1");

            //Also add syd user properties file
            SydUserProperties(userID,userName,"sydprop");

            //Save Directory Object
            SAVE();
            d.DisplayDirectory();
            return objID;
      }

      public static void main(String args[]) throws RemoteException
      {
            SyDPropertyFile p = new SyDPropertyFile("sydprop");
            int port = Integer.parseInt(p.getValue("directoryport"));
            String dsname = p.getValue("directoryname");

            try
            {
                  System.out.println("\nSyD Directory Service Initiated\n");
                  DirectoryServerImpl ds = new 
DirectoryServerImpl(dsname,port);
                  Registry r = initialize(port);
                  r.rebind(dsname, ds);
                  System.out.println(ds);
                  System.out.println("\nWaiting for invocations from 
clients...\n");
            }
            catch(Exception e)
            {
                  System.out.println("SyD Directory Service Error: " + e);
                  e.printStackTrace();
            }
      }
}

-----------------------------------------------------------------------------
```

DirectoryUpdater.java

```
package syd.syddirectory;

//import syd packages
import syd.syddirectory.*;
import syd.sydengine.*;
import syd.sydlistener.*;
import syd.sydutil.*;

import java.util.*;
import java.io.*;
import java.rmi.*;

public class DirectoryUpdater
{
      HashMap<String,String> h;

      public DirectoryUpdater()
      {
            h = new HashMap<String,String>();
      }

      public void startDeviceDiscovery()
      {
            //1. Start device discovery by network layer
            //2. While device is not discovered,keep searching
            //3. If device discovered, obtain device's sydprop.propertiesfile
            //4. Rename file as device2.properties and copy to C:\OSyD

            //The new device has been discovered (eg: Bluetooth)
            //and its sydprop.properties file has been received
            //and copied as device2.properties in C:\OsyD

            //Code for steps 1-4 is triggered by the network layer

            setHashMap("device2");
      }

      public void setHashMap(String filename)
      {
            SyDPropertyFile p = new SyDPropertyFile("device2");
            h.clear();
            h.put("directoryname",p.getValue("directoryname"));
            h.put("devicehost",p.getValue("devicehost"));
            h.put("directoryport",p.getValue("directoryport"));
            h.put("listenerport",p.getValue("listenerport"));
            h.put("appserverport",p.getValue("appserverport"));
      }
      //Assumption: All SyD userNames are UNIQUE between different devices
      public void sync()
      {
            DirectoryServer localDS=null,remoteDS=null;
            String dirhost,dirport,dsname,url="";

            try
            {
                  //Compare local and neighbor directory data
                  SyDPropertyFile p = new SyDPropertyFile("sydprop");
                  dirhost = p.getValue("devicehost");
                  dirport = p.getValue("directoryport");
                  dsname = p.getValue("directoryname");
                  url = "rmi://"+dirhost+":"+dirport+"/"+dsname;
                  localDS = (DirectoryServer)Naming.lookup(url);

                  dirhost = h.get("devicehost");
                  dirport = h.get("directoryport");
                  dsname = h.get("directoryname");
                  url = "rmi://"+dirhost+":"+dirport+"/"+dsname;
                  remoteDS = (DirectoryServer)Naming.lookup(url);
            }
            catch(Exception e)
            {
                  System.out.println("Error in DirectoryUpdater.sync(): " +
url);
                  e.printStackTrace();
            }

            try
            {
                  Directory d = remoteDS.getDirectory();
                  localDS.Sync(d);
            }
            catch(Exception e)
            {
                  System.out.println("Error in DirectoryUpdater.sync: " + e);
                  e.printStackTrace();
            }
      }

      public static void main(String[] args)
      {
            DirectoryUpdater x = new DirectoryUpdater();
//          while(true)
//          {
                  //Discover closest SyD device
                  x.startDeviceDiscovery();

                  //synchronize with neighbor's directory
                  Long start = new Long(System.currentTimeMillis());
                  x.sync();
                  Long end = new Long(System.currentTimeMillis());
                  Long timeTaken = end - start;
                  System.out.println("Time(ms) taken for synchronization and 
update = "+ timeTaken);

//          }
      }

}

-----------------------------------------------------------------------------
```

SyDDispatcher.java

```
package syd.sydengine;

import java.util.*;
import java.lang.*;
import java.awt.*;
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

import syd.sydutil.*;
import syd.sydlistener.*;
import syd.syddirectory.*;

/**
 * Generic class for dispatching method calls remotely
 */

public class SyDDispatcher
{
      Vector<Object> doclist = new Vector<Object>();

      /**
       *  invoke method is used to invoke a methodname using the list of 
parameters specified
       *  @param appname: application name
       *  @param methodname : method to be executed
       *  @param paramlist : parameter list
       *  @return Vector
       */

      public Vector<Object> invoke(String appname,String methodname,Vector 
paramtype,Vector paramvalue) throws IOException
      {
            SyDPropertyFile p = new SyDPropertyFile("sydprop");
            String dsname = p.getValue("directoryname");
            String dirhost = p.getValue("devicehost");
            Integer dirport = Integer.parseInt(p.getValue("directoryport"));
            Integer listenerport =
Integer.parseInt(p.getValue("listenerport"));

            DirectoryServer DS = null;
            try
            {
                  Registry r = LocateRegistry.getRegistry(dirhost,dirport);
                  DS = (DirectoryServer)r.lookup(dsname);
            }
            catch(Exception e)
            {
                  System.out.println("Error in SyDDispatcher.local lookup: "
+ e);
            }

            //System.out.println("ACCESS LOCAL DIRECTORY SERVICE FOR
AVAILABLE USERS ...");

            String url = "";
            String teststring = "";
            SyDDoc doc = new SyDDoc();
            Vector<Object> resultstrings = new Vector<Object>();
            HashMap<String,String> h = new HashMap<String,String>();
            SyDListenerDelegate sld = null;

            //Returns the local Application Server Object's ID
            String objID = DS.getSydObjectID(appname);
            System.out.println("dispatcher.objID = "+ objID);

            sld = new SyDListenerDelegate(dirhost,listenerport);

            //System.out.println("CALLING SYDDOC FOR CREATING THE REQUEST
DOCUMENT \n");
            doc.createRequest(objID, methodname, paramtype, paramvalue);
            teststring = doc.getString().toString();

            // Engine to syd listener
            teststring = sld.invoke(teststring);
            resultstrings.addElement(teststring);
            return (resultstrings);
      }
}

-----------------------------------------------------------------------------
```

SyDListener.java

```
/**
 * Listener class for receiving invocation message on given port and make
method invocation using RMI.
 */

package syd.sydlistener;

import syd.sydutil.*;

import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.*;
import java.io.*;
import java.lang.reflect.*;
import java.util.*;

public class SyDListener
{
      String RMIHost;
      int RMIPort;
      String url;

      public SyDListener(String RMIHost, int RMIPort)
      {
            this.RMIHost = RMIHost;
            this.RMIPort = RMIPort;
            this.url  = "rmi://" + RMIHost + ":" + RMIPort + "/";
      }

      public SyDListener(int RMIPort)
      {
            this("localhost", RMIPort);
      }

      public SyDListener()
      {
            this("localhost", 1099);
      }

   /**
    *  Wrapper for actual invoke method.
    *  <BR><BR>
    *  @param info invocation information
    *  @return invocation result
    */
      private String invoke (String message) throws
java.rmi.NotBoundException
```

```
      {
            // parse xml document

            SyDDoc outDoc =null;

            try
            {
                  SyDDoc doc = new SyDDoc(new StringBuffer(message));
                  String objectName = doc.getObjectID();
                  String methodName = doc.getMethodName();
                  Vector parameterTypeStrings = doc.getParameterType();
                  Vector parameterValues = doc.getParameterValue();

                  for(int j=0;j<parameterTypeStrings.size();j++)
                  {
                        Object a = parameterTypeStrings.elementAt(j);
                  }

                  for(int k=0;k<parameterValues.size();k++)
                  {
                        Object c = parameterValues.elementAt(k);
                  }

                  Vector<Object> parameterTypes = new Vector<Object>();
                  try
                  {
                        for(int i=0; i<parameterTypeStrings.size(); i++)
                        {
                              String type = 
(String)parameterTypeStrings.elementAt(i);
                              parameterTypes.addElement(Class.forName(type));
                        }
                  }
                  catch(Exception e) {
                        System.out.println("Error in Listener.invoke(String): 
" + e);
                        return null;
                  }

                  String result = invoke(objectName, methodName, 
parameterTypes, parameterValues);
                  outDoc = new SyDDoc();
                  outDoc.createResponse(result);

            }
            catch(Exception e)
            {
                  e.printStackTrace();
            }
            return outDoc.getString().toString();
      }

      private String invoke (String objectName, String methodName,
                              Vector parameterTypes, Vector parameterValues)
throws java.rmi.NotBoundException,IllegalArgumentException
      {
            Object result = null;
            try
            {
                  System.out.println("listener url = " + url+objectName);
                  Object stubRef = Naming.lookup(url + objectName);

                  // use  java.lang.reflection package to make invocation
                  Class<?> c = stubRef.getClass();

                  Object[] tempArray = vectorToArray(parameterTypes);
                  Class[] parameterTypesArray = new Class[tempArray.length];
                  for(int i=0; i<tempArray.length; i++)
                        parameterTypesArray[i] = (Class)tempArray[i];

                  Object[] argumentsArray = vectorToArray(parameterValues);

                  Method method = c.getMethod(methodName,
parameterTypesArray);
                  result = method.invoke(stubRef, argumentsArray);
            }
            catch(Exception e)
            {
                  System.out.println("Error in Listener.invoke2(): " + e);
                  e.printStackTrace();
            }

            return result.toString();
      }

      private Object[] vectorToArray(Vector vector)
      {
            int size = vector.size();
            Object[] array = new Object[size];

            for(int i=0; i<size; i++)
                  array[i] = vector.elementAt(i);

            return array;
      }

      private void work(Socket clientSocket)
      {
            try
            {
                  PrintWriter out = new 
PrintWriter(clientSocket.getOutputStream(), true);
                  BufferedReader in = new BufferedReader(new 
InputStreamReader(clientSocket.getInputStream()));

                  String input = "", output = "";
                  String inputLine;

                  // Get request from client
                  while (!(inputLine = in.readLine()).equals("</REQUEST>"))
                  {
                      input += inputLine + "\n";
                  }
                input += inputLine + "\n";

                  try
                  {
                        // invoke method
                  output = invoke(input);
                  }
                  catch(Exception e)
                  {
                        System.out.println("In SyDListener, error at invoke: 
"+e);
                  }

                  // Send result to client.
                  out.println(output);
              out.println("&end&");

/*                while (!(inputLine = in.readLine()).equals("&end&"))
                  {
                      // System.out.println(inputLine);
                  }
*/
                  // Close input and output streams, the client socket.
                  out.close();
                  in.close();
                  clientSocket.close();
            }
            catch (IOException e)
            {
                  System.out.println("Error in SyDListener.work(): " + e);
            }
      }

      public static void main(String args[])
      {
            SyDPropertyFile p = new SyDPropertyFile("sydprop");
            String host = p.getValue("devicehost");
            int serverPort = Integer.parseInt(p.getValue("listenerport"));
            int RMIPort = Integer.parseInt(p.getValue("appserverport"));

            SyDListener listener = new SyDListener(host,RMIPort);

            ServerSocket serverSocket = null;

            try
            {
                  serverSocket = new ServerSocket(serverPort);
                  System.out.println("\nSyD Listener running ...\n");
            } catch (IOException e)
            {
                  System.err.println("Error in SyDListener.main() Could not
listen on port: " + serverPort + e.toString());
            System.exit(1);
        }

            Socket clientSocket = null;

            while(true)
            {
            try
                  {
                clientSocket = serverSocket.accept();
                System.out.println("Local endpoint = " +
clientSocket.getLocalSocketAddress());
                System.out.println("Remote client connection = " +
clientSocket.getRemoteSocketAddress());
                System.out.println();
            } catch (IOException e)
            {
                System.err.println("Accept failed.");
                System.exit(1);
            }
            listener.work(clientSocket);
            }
    }
}

-----------------------------------------------------------------------------
```

SyDListenerDelegate.java

```
/**
 * Yamacraw Embedded Software - SyD Listener
 * <PRE>
 * SyDListenerDelegate.java, client side delegate provided by listener
provider to perform communication with server side listener object
 *
 * Revisions:   1.0  September 8, 2002
 *                   Created class SyDListenerDelegate
 *
 * </PRE>
 *
 * @author <A HREF="mailto:bingaero@hotmail.com">Bing Liu</A>
 * @version 1.0, September 8, 2002
 *
 */

package syd.sydlistener;

import syd.sydutil.*;

import java.io.*;
import java.net.*;

public class SyDListenerDelegate
{
      public final static String END_TAG = "<SyDListenerDelegate.end>";

      private String serverName;
      private int serverPort;

      public SyDListenerDelegate(String serverName, int serverPort)
      {
            this.serverName = serverName;
            this.serverPort = serverPort;
      }

      /**
       * Transform invocation information into a string, open a socket to
listener and send the string.
       * Then receive result as a string and transform it into Doc.
       * <BR><BR>
       * @param inputDoc invocation information.
       * @return result
       */

      public String invoke(String inputString) throws IOException
      {
        Socket listenSocket = null;
        PrintWriter out = null;
        BufferedReader in = null;

      try
      {
            listenSocket = new Socket(serverName, serverPort);
            out = new PrintWriter(listenSocket.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(
                                        listenSocket.getInputStream()));
        }
        catch (UnknownHostException e)
        {
            System.err.println("SLD: Don't know about host: " + e);
            System.exit(1);
        }
        catch (IOException e)
        {
            System.err.println("SLD: Couldn't get I/O for the connection to
server; " + e );
            System.exit(1);
        }

            String serverOutput = "";
            String serverOutputLine = "";

          out.println(inputString);
          out.println(END_TAG);

      //Read and display server response
            while ((serverOutputLine = in.readLine()) != null)
            {
                  if (serverOutputLine.equals("&end&"))
                        break;
                serverOutput += serverOutputLine + "\n";
            }

        out.println("&end&");

            // Close all the streams and socket.
            out.close();
            in.close();
            listenSocket.close();

            return serverOutput;
    }

}

-----------------------------------------------------------------------------
```

SyDRegistrar.java

```
/**
 * Yamacraw Embedded Software - SyD Listener
 * <PRE>
 * SyDRegistrar.java
 *
 * Revisions:   1.0  September 8, 2002
 *                   Created class SyDRegistrar
 *
 * </PRE>
 *
 * @author <A HREF="mailto:bingaero@hotmail.com">Bing Liu</A>
 * @version 1.0, September 8, 2002
 *
 */

package syd.sydlistener;

import syd.syddirectory.*;
import syd.sydutil.*;

import java.rmi.*;
import java.util.*;
import java.net.*;

public class SyDRegistrar
{
      String rmiHost;
      int rmiPort;
      String directoryHost;
      int directoryPort;
      String objectName = "";

      public SyDRegistrar(String rmiHost, int rmiPort, String directoryHost, 
int directoryPort)
      {
            this.rmiHost = rmiHost;
            this.rmiPort = rmiPort;
            this.directoryHost = directoryHost;
            this.directoryPort = directoryPort;
      }

      public SyDRegistrar(int rmiPort, String directoryHost, int 
directoryPort)
      {
            this("localhost", rmiPort, directoryHost, directoryPort);
      }

      public SyDRegistrar(String directoryHost, int directoryPort)
      {
            this("localhost", 1099, directoryHost, directoryPort);
      }

      public void register (String dsname,Object appInstance, String 
publishDoc)
      {
            registerToDirectoryService (dsname,publishDoc);
            registerToRMIRegistry (appInstance, objectName);
      }

      public void registerToDirectoryService (String dsname, String 
publishDoc)
      {
            System.out.println("Registering to Directory Service " + dsname);
            System.out.println(publishDoc);
            try
            {
                  String url  = "rmi://" + directoryHost + ":" + 
directoryPort + "/";
                  DirectoryServer remoteDS = 
(DirectoryServer)Naming.lookup(url + dsname);
                  System.out.println("Lookup Directory Server ... Success.");
                  objectName = remoteDS.publish(publishDoc);
                  System.out.println("Publish to Directory Service ... 
Success");
//                System.out.println(objectName);
            }
            catch(Exception e)
            {
                  System.out.println("Error in 
SyDRegistrar.registerToDirectoryService(): " + e);
                  e.printStackTrace();
            }
      }

      public void registerToRMIRegistry (Object appInstance, String 
objectName)
      {
            try
            {
                  String url = "rmi://" + rmiHost + ":" + rmiPort + "/" + 
objectName;
                  Naming.rebind(url, (Remote)appInstance);
                  System.out.println("Object bound to "+url);
            }
            catch(Exception e)
            {
                  System.out.println("Error in 
SyDRegistrar.registerToRMIRegistry(): " + e);
```

```
                  e.printStackTrace();
            }
      }

      public String getObjectName()
      {
            return objectName;
      }

}
```

```
-----------------------------------------------------------------------------
```

Publisher.java

```
package syd.sydutil;

import java.io.*;
import java.util.*;

/**
 * Class definition for Publisher to publish Users, Methods, Services, and
 * Groups and their associations in Directory Service.
 */
public class Publisher
{
      private StringBuffer docString;
      private     Vector<Object> methods,paramTypes,returnTypes ;

      /* This Constructor initialises the XML document String docString */
      public Publisher()
      {
            docString = new StringBuffer("");
            docString.append("<publisher-doc>\n");
      }

      public Publisher(String newdoc)
      {
            docString = new StringBuffer("");
            docString.append(newdoc);
            methods = new Vector<Object>();
            paramTypes = new Vector<Object>();
            returnTypes =  new Vector<Object>();
            System.out.println("\nReceived PublisherDoc.\n" +
docString.toString());
      }

      /** Retrieves the XML document
        * @return String
        */
      public String getString()
      {
            return docString.toString();
      }

      public String getSimpleValue(String node_name)
      {
            String str = docString.toString();
            //System.out.println(str);
            str = str.substring(str.indexOf(node_name));
            //System.out.println(str);
            str = str.substring(str.indexOf(">")+1,str.indexOf("<"));
            return str.trim();
      }

      public Vector getMethodNames()
      {
      return methods;
      }

      public Vector getReturnTypes()
      {
      return returnTypes;
      }

      public Vector getParamTypes()
      {
      return paramTypes;
      }

      public void getMethodDetails()
      {
      String method_str = new String();
      String m_name, ret,ptype;
      Vector<Object> p;
            String str = docString.toString();

      //Extract # of methods to be published
            str = str.substring(str.indexOf("methodcount"),str.length()-1);
      str = str.substring(str.indexOf('"')+1);

            int method_count = Integer.parseInt(
str.substring(0,str.indexOf('"')).trim());
      System.out.println("Methods Published: " + method_count);

            for(int i = 0;i<method_count;i++)
            {
                  method_str = str.substring(str.indexOf("<method>"),
str.indexOf("</method>"));
                  str = str.substring(str.indexOf("</method>")+9);
                  m_name = new String();
                  m_name =
method_str.substring(method_str.indexOf("<methodName>")+12,
method_str.indexOf("</methodName>"));
                  methods.addElement(m_name);
                  System.out.println("Method Name: " + m_name);

                  ret = new String();
                  ret =
method_str.substring(method_str.indexOf("<returnType>")+12,
method_str.indexOf("</returnType>"));
                  returnTypes.addElement(ret);
                  System.out.println("Return Type: " + ret);

                  //Parameter Count
                  method_str =
method_str.substring(method_str.indexOf("count"),method_str.length()-1);
                  method_str =
method_str.substring(method_str.indexOf('"')+1);

                  int param_count = Integer.parseInt(
method_str.substring(0,method_str.indexOf('"')).trim());
                  System.out.println("Parameters: " + param_count);

                  p = new Vector<Object>();
                  for(int j = 0;j<param_count;j++)
                  {
                        method_str =
method_str.substring(method_str.indexOf("<param")+6,method_str.length());
                  ptype = new String();
                  ptype = method_str.substring(method_str.indexOf("type")+4);
                  ptype= ptype.substring(ptype.indexOf('"')+1);
                  ptype= ptype.substring(0,ptype.indexOf('"'));
                  p.addElement(ptype.trim());
                  //System.out.println("ptype:" + ptype + " " +j);
                  }
                  paramTypes.addElement(p);
            }
      }

      /** Creates Publish request envelope for SYD_USER
        *  @param userName : Name of User
        *  @param userPasswd : Password
        *  @param userURL : User Location
        *  @param liveBit : Device status
        */
      public void createPublishUserMethodsRequest(String userName,String
userPasswd,String userURL, String proxyID,String appName, Vector methodNames,
Vector returnTypes, Vector paramTypes)
      {
            docString.append("\t<userName>" + userName + "</userName>\n");
            docString.append("\t<userPasswd>" + userPasswd + 
"</userPasswd>\n");
            docString.append("\t<userURL>" + userURL + "</userURL>\n");
            docString.append("\t<appName>" + appName + "</appName>\n");
            docString.append("\t<proxyID>" + proxyID + "</proxyID>\n");
            docString.append("\t<method-details methodcount=\""+ 
methodNames.size()+"\">\n");

            for(int i=0;i<methodNames.size();i++)
            {
                  String temp = (String)methodNames.elementAt(i);
                  docString.append("\t\t<method>\n");
                  docString.append("\t\t<methodName>" + 
methodNames.elementAt(i).toString() + "</methodName>\n");
                  docString.append("\t\t<returnType>" + 
returnTypes.elementAt(i).toString() + "</returnType>\n");
                  Vector params = (Vector)paramTypes.elementAt(i);
                  docString.append("\t\t\t<parameters count = \""+ 
params.size() +"\">\n");
                  for (int j=0;j<params.size() ;j++ )
                        docString.append("\t\t\t\t<param"+(j+1)+" 
type=\""+params.elementAt(j).toString()+"\"></param"+(j+1)+">\n");
                  docString.append("\t\t\t</parameters>\n");
                  docString.append("\t\t</method>\n");
            }
            docString.append("\t</method-details>\n");
            docString.append("</publisher-doc>\n");
      }     /* End Of createPublishUserMethodsRequest */

}

-----------------------------------------------------------------------------
```

SyDDoc.java

```
/**  Static Model  */
package syd.sydutil;

import java.util.*;
import java.lang.*;

/**
 * SyDDoc.java
 * Generic class for creating and parsing XML documents
 */

public class SyDDoc
{
      private StringBuffer docString;

      public SyDDoc()
      {
            docString = new StringBuffer("");
      }

      public SyDDoc(StringBuffer xmlString)
      {
            docString = xmlString;
      }

      public StringBuffer getString()
      {
            return docString;
      }

      public void createRequest(String objectID,
                              String methodName,
                              Vector parameterTypeList,
                              Vector parameterValueList)
      {
                  docString .append("<REQUEST>\n");
                  docString.append("<OBJECT  id = 
\"").append(objectID).append("\" />\n");
                  docString.append("<METHOD  name = 
\"").append(methodName).append("\"/ >\n");
                  docString.append("<PARAMETERS>\n");
                  for(int j=0;j<parameterTypeList.size();j++)
                  {
                        docString.append("<PARAMETER  type = \"
").append(parameterTypeList.elementAt(j));
                        docString.append("\" value = \"
").append(parameterValueList.elementAt(j)).append("\" />\n");
                  }
                  docString.append("</PARAMETERS>\n");
                  docString.append("</REQUEST>\n");
      }

      public String getObjectID()
      {
            String string2Parser = new String(docString);
            int i = string2Parser.indexOf("<OBJECT");
            int j = string2Parser.indexOf("\"", i);
            int k = string2Parser.indexOf("\"", j+1);
            /*System.out.println(i);
            System.out.println(j);
            System.out.println(k); */
            String objectID = string2Parser.substring(j+1,k);
            return objectID;
      }

      public String getMethodName()
      {
            String string2Parser = new String(docString);
            int i = string2Parser.indexOf("<METHOD");
            int j = string2Parser.indexOf("\"", i);
            int k = string2Parser.indexOf("\"", j+1);
            /*System.out.println(i);
            System.out.println(j);
            System.out.println(k); */
            String methodName = string2Parser.substring(j+1,k);
            return methodName;
      }

      public Vector getParameterValue()
      {
            int i, j, l, m;
            int k=0;
            String  parameterType;
            String parameterValue;
            Vector<Object> parameterVector = new Vector<Object>();
            String string2Parser = new String(docString);
            int start = string2Parser.indexOf("<PARAMETERS>");
            int index = start +1;
            int end = string2Parser.indexOf("</PARAMETERS>");
            /*System.out.println(start);
            System.out.println(end);*/
            while(index<end)
            {
                  i = string2Parser.indexOf("<PARAMETER", k);
                  if (i == -1)
                  {
                        break;
                  }
```

```
                  j = string2Parser.indexOf("\"", i);
                  k = string2Parser.indexOf("\"", j+1);
                  parameterType = string2Parser.substring(j+1,k).trim();
                  //System.out.println(parameterType);
                  l = string2Parser.indexOf("\"", k+1);
                  m = string2Parser.indexOf("\"", l+1);
                  parameterValue = string2Parser.substring(l+1,m).trim();
                  if (parameterType.equalsIgnoreCase("java.lang.String"))
                  {
                        String parameterObject = new String(parameterValue);
                        parameterVector.addElement(parameterObject);
                  }

                  else if (parameterType.equalsIgnoreCase("String"))
                  {
                        String parameterObject = new String(parameterValue);
                        parameterVector.addElement(parameterObject);
                  }
                  else if (parameterType.equalsIgnoreCase("int"))
                  {
                        Integer parameterObject = new
Integer(parameterValue);
                        parameterVector.addElement(parameterObject);
                  }
                  else if
(parameterType.equalsIgnoreCase("java.lang.Integer"))
                  {
                        Integer parameterObject = new
Integer(parameterValue);
                        parameterVector.addElement(parameterObject);
                  }

                  else if (parameterType.equalsIgnoreCase("double"))
                  {
                        Double parameterObject = new Double(parameterValue);
                        parameterVector.addElement(parameterObject);
                  }
                  else if
(parameterType.equalsIgnoreCase("java.lang.Double"))
                  {
                        Double parameterObject = new Double(parameterValue);
                        parameterVector.addElement(parameterObject);
                  }

                  else if (parameterType.equalsIgnoreCase("long"))
                  {
                        Long parameterObject = new Long(parameterValue);
                        parameterVector.addElement(parameterObject);
                  }
                  else if (parameterType.equalsIgnoreCase("java.lang.Long"))
                  {
                        Long parameterObject = new Long(parameterValue);
                        parameterVector.addElement(parameterObject);
                  }
                  else if (parameterType.equalsIgnoreCase("boolean"))
                  {
                        Boolean parameterObject = new
Boolean(parameterValue);
                        parameterVector.addElement(parameterObject);
                  }

                  else if
(parameterType.equalsIgnoreCase("java.lang.Boolean"))
                  {
                        Boolean parameterObject = new
Boolean(parameterValue);
                        parameterVector.addElement(parameterObject);
                  }

                  else if (parameterType.equalsIgnoreCase("char"))
                  {
                        Character parameterObject = new
Character(parameterValue.charAt(0));
                        parameterVector.addElement(parameterObject);
                  }
                  else if
(parameterType.equalsIgnoreCase("java.lang.Character"))
                  {
                        Character parameterObject = new
Character(parameterValue.charAt(0));
                        parameterVector.addElement(parameterObject);
                  }

                  else if (parameterType.equalsIgnoreCase("StringBuffer"))
                  {
                        StringBuffer parameterObject = new
StringBuffer(parameterValue);
                        parameterVector.addElement(parameterObject);
                  }
                  else if
(parameterType.equalsIgnoreCase("java.lang.StringBuffer"))
                  {
                        StringBuffer parameterObject = new
StringBuffer(parameterValue);
                        parameterVector.addElement(parameterObject);
                  }
                  else  //Add extra parameter types here
                        System.out.println("Reached
SyDDoc.getParameterValue() --- to be written");
```

```
            }
            return parameterVector;
      }


      public Vector getParameterType()
      {
            int i, j, l, m;
            int k=0;
            String  parameterType;
            Vector<Object> parameterTypeVector = new Vector<Object>();
            String string2Parser = new String(docString);
            int start = string2Parser.indexOf("<PARAMETERS>");
            int index = start +1;
            int end = string2Parser.indexOf("</PARAMETERS>");

            while(index<end)
            {
                  i = string2Parser.indexOf("<PARAMETER", k);
                  if (i == -1)
                  {
                        break;
                  }

                  j = string2Parser.indexOf("\"", i);
                  k = string2Parser.indexOf("\"", j+1);
                  parameterType = string2Parser.substring(j+1,k).trim();
                  parameterTypeVector.addElement(parameterType);
            }
            return parameterTypeVector;
      }

      public void createResponse(Object ob)
      {
//          docString.append("<table>\n");
            String classname = ob.getClass().getName();

            if (classname.equalsIgnoreCase("java.lang.String"))
            {
                  String str = (String) ob;
                  docString.append(str);
//                docString.append("\t<row1>\n");
//                docString.append("\t\t<column type = \"string\" name =
\"\">"+str+"</column>\n");
//                docString.append("\t</row1>\n");
            }
            else if(classname.equalsIgnoreCase("long") ||
classname.equalsIgnoreCase("java.lang.Integer"))
            {
                  Long fl = (Long) ob;
                  docString.append(fl.toString());
//                docString.append("\t<row1>\n");
//                docString.append("\t\t<column type = \"long\" name =
\"\">"+fl.toString()+"</column>\n");
//                docString.append("\t</row1>\n");
            }
            else if(classname.equalsIgnoreCase("int") ||
classname.equalsIgnoreCase("java.lang.Integer"))
            {
                  Integer i = (Integer) ob;
                  docString.append(i.toString());
//                docString.append("\t<row1>\n");
//                docString.append("\t\t<column type = \"integer\" name =
\"\">"+i.toString()+"</column>\n");
//                docString.append("\t</row1>\n");
            }
            else if(classname.equalsIgnoreCase("boolean") ||
classname.equalsIgnoreCase("java.lang.Boolean"))
            {
                  Boolean bool = (Boolean) ob;
                  docString.append(bool.toString());
//                docString.append("\t<row1>\n");
//                docString.append("\t\t<column type = \"boolean\" name =
\"\">"+bool.toString()+"</column>\n");
//                docString.append("\t</row1>\n");
            }
            else if(classname.equalsIgnoreCase("char") ||
classname.equalsIgnoreCase("java.lang.Character"))
            {
                  Character c = (Character) ob;
                  docString.append(c.toString());
//                docString.append("\t<row1>\n");
//                docString.append("\t\t<column type = \"char\" name =
\"\">"+c.toString()+"</column>\n");
//                docString.append("\t</row1>\n");
            }
            else
                  System.out.println("Reached SyDDoc.createResponse() --- to
be written");

//          docString.append("</table>\n");
      }

}
/* END CLASS DEFINITION SyDDoc */

------------------------------------------------------------------------------
```

SyDPropertyFile.java

```
// author : praveen madiraju
// requirements : place the properties file(propName) under some directory
// make sure the class path points to this properties directory and the
program by default checks for `.properties

package syd.sydutil;

import java.io.*;
import java.util.*;
import java.util.Enumeration;
import java.util.PropertyResourceBundle;

public class SyDPropertyFile
{
      PropertyResourceBundle propBundle = null;

      public SyDPropertyFile(String filename)
      {
            try
            {
                  propBundle =
(PropertyResourceBundle)PropertyResourceBundle.getBundle(filename);
            }
            catch(Exception e)
            {
                  e.printStackTrace();
                  System.exit(1);
            }
      }

      public String getValue(String name)  throws MissingResourceException
      {
            return propBundle.getString(name);
      }
}

-----------------------------------------------------------------------------
```

ParkingLot.java

```
package syd.sydapp.ParkingLot;

import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

public interface ParkingLot extends Remote
{
      public Vector<String> getAllData() throws RemoteException;

      public String getParkingLot(String name) throws RemoteException;

      public void setParkingLot(String name,Integer spaces,Long timestamp)
throws RemoteException;

      public void syncAppServer(ParkingLotImpl_Stub o) throws
RemoteException;
}
```

ParkingLotImpl.java

```
package syd.sydapp.ParkingLot;

import java.lang.*;
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.*;
import java.util.*;

import syd.sydengine.*;
import syd.sydutil.*;
import syd.sydlistener.*;
import syd.syddirectory.*;
import syd.sydapp.ParkingLot.*;

public class ParkingLotImpl extends UnicastRemoteObject implements ParkingLot
{
      static private Vector<String> parkingLot = new Vector<String>();
      final String x = " : "; //delimiter

      public ParkingLotImpl(String name,Integer spaces) throws
RemoteException
      {
            Long ts = new Long(System.currentTimeMillis());
            addNewLot(name,spaces,ts);
        }

        public Vector<String> getAllData() throws RemoteException
        {
            return parkingLot;
        }

        //generic method that is present in all Application Server Objects
        //change this method to suit how to update your server data
        public void syncAppServer(ParkingLotImpl_Stub p) throws RemoteException
        {
            //We need to update all parking lot data with the latest
available spaces
            Vector<String> v = p.getAllData();
            StringTokenizer t = null;

            //Compare data here for all available parkinglot names in p
            for(int i = 0;i < v.size();i++)
            {
                //Get details of each parkingLot from remote stub
                String r = v.get(i);
                t = new StringTokenizer(r,x);
                String rName = t.nextToken();
                Integer rSpaces = new Integer(t.nextToken());
                Long rTS = new Long(t.nextToken());

                //Get corresponding details for the same name from local
server
                int n = getRecord(rName);
                if(n == -1)
                {
                    //parkinglot not found on local server
                    //add it
                    addNewLot(rName,rSpaces,rTS);
                }
                else
                {
                    //parkinglot found on local server
                    //update it with latest data
                    String s = getParkingLot(rName);
                    t = new StringTokenizer(s,x);
                    String name = t.nextToken();
                    Integer spaces = new Integer(t.nextToken());
                    Long ts = new Long(t.nextToken());

                    if(rTS > ts)
                    {
                        //update local data with remote values
                        setParkingLot(name, rSpaces, rTS);
                        }
                  }
            }
            System.out.println("Sync with Remote Done.parkingLot =  " +
parkingLot);
      }

      //Adds a new parking lot's info
      private void addNewLot(String name,Integer spaces,Long timestamp)
throws RemoteException
      {
            //Stores [name:spaces:timestamp:,...]
            String s = name + x + spaces.toString() + x +
timestamp.toString() + x;
            parkingLot.add(s);
            System.out.println("New data added.parkingLot = " + parkingLot);
      }

      //This method returns details for the Parking Lot name specified
      public String getParkingLot(String name) throws RemoteException
      {
            String s = "";
            int n = getRecord(name);
            if(n != -1)
            {
                  s = parkingLot.get(n);
                  System.out.println("Data retrieved.parkingLot = " +
parkingLot);
            }
            if(s.equals(""))
                  return "null";
            else
                  return s;
      }

      //This method updates details for the Parking Lot name specified
      public void setParkingLot(String name,Integer spaces,Long timestamp)
throws RemoteException
      {
            String s = name + x + spaces.toString() + x +
timestamp.toString()+x;
            int n = getRecord(name);
            if(n == -1)
                  addNewLot(name,spaces,timestamp);
            else
            {
                  parkingLot.setElementAt(s,n);
                  System.out.println("Updated.parkingLot = " + parkingLot);
            }
      }
```

```
	private int getRecord(String name) throws RemoteException
	{
		int found = -1;
		String s;
		for(int i = 0;i < parkingLot.size();i++)
		{
			s = parkingLot.get(i);
			StringTokenizer st = new StringTokenizer(s,x);
			if(st.nextToken().equals(name))
			{
				found = i;
				break;
			}
		}
//		System.out.println("record num = "+found);
		return found;
	}
}
```

ParkingLotServer.java

```
package syd.sydapp.ParkingLot;

import java.lang.*;
import java.util.*;
import java.net.*;
import java.rmi.registry.*;
import java.lang.reflect.*;

import syd.sydlistener.*;
import syd.sydutil.*;
import syd.sydapp.ParkingLot.*;

public class ParkingLotServer
{
	public static void main(String args[])
	{
		System.out.println("Parking Lot Server started ...\n");
		Scanner sc = new Scanner(System.in);
		System.out.print("Username: ");
		String sydusr = sc.nextLine();
		System.out.print("Password: ");
		String sydpwd = sc.nextLine();

		SyDPropertyFile p = new SyDPropertyFile("sydprop");
		String dsname = p.getValue("directoryname");
		String host = p.getValue("devicehost");
```

```
            int directoryServerPort = 
Integer.parseInt(p.getValue("directoryport"));
            int portNum = Integer.parseInt(p.getValue("appserverport"));

            int flag = 1;           // Change this value when using for proxy

            Vector<Object> methods = new Vector<Object>();
            Vector<Object> returnType = new Vector<Object>();
            Vector<Object> methodParams = new Vector<Object>();
            Vector<Object> paramTypes = new Vector<Object>();

            /** User will use all these details to publish his methods */
            //Method 1
            methods.addElement(new String("getParkingLot"));
            returnType.addElement(new String("java.lang.String"));
            methodParams.addElement(new String("java.lang.String"));
            paramTypes.addElement(methodParams);

            //Method 2
            methods.addElement(new String("setParkingLot"));
            returnType.addElement(new String("java.lang.String"));
            methodParams = new Vector<Object>();
            methodParams.addElement(new String("java.lang.String"));
            methodParams.addElement(new String("java.lang.Integer"));
            methodParams.addElement(new String("java.lang.Long"));
            paramTypes.addElement(methodParams);

/*          try
            {
                  myurl = InetAddress.getLocalHost().getHostAddress();
                  System.out.println("localhost URL = " + myurl);
            } catch(java.net.UnknownHostException e){ }
*/
            Publisher publisher = new Publisher();

      publisher.createPublishUserMethodsRequest(sydusr,sydpwd,host,"proxyID",
"ParkingLot",methods,returnType,paramTypes);
            String publisherstr = publisher.getString();

            try
            {
                  if(flag == 1)
                  {
                        System.out.println("Creating RMI registry at port " + 
portNum + "...");
                        LocateRegistry.createRegistry(portNum);
                        System.out.println("RMI Registry created at port " + 
portNum);
                  }
                  System.out.println("Constructing server 
implementations...");
                  System.out.print("Parking Lot Name: ");
                  String s = sc.nextLine();
                  System.out.print("Available Spaces: ");
                  Integer n = Integer.parseInt(sc.nextLine());
                  ParkingLotImpl ParkingLotObject = new ParkingLotImpl(s,n);
                  getClassInfo("syd.sydapp.ParkingLot.ParkingLot");

                  System.out.println("Binding server implementations to 
registry...");

                  SyDRegistrar registrar = new SyDRegistrar(host ,portNum, 
host, directoryServerPort);
                  registrar.register(dsname, ParkingLotObject, publisherstr);

                  System.out.println("objectName: " + 
registrar.getObjectName());
                  System.out.println("Waiting for invocations from 
clients...");
            }
            catch(Exception e)
            {
                  System.out.println("Error in Registrator.main(): " + e);
                  e.printStackTrace();
            }
      }

      private static void getClassInfo(String interfaceName)
      {
            Class c = null;
            try {
                        c = Class.forName(interfaceName);
                  }
            catch(Exception e)
            {
                  System.out.println("Error in getClassInfo(): " + e);
            }

      Method[] theMethods = c.getMethods();

            for (int i = 0; i < theMethods.length; i++)
            {
            String methodString = theMethods[i].getName();
            System.out.println("Name: " + methodString);

            String returnString = theMethods[i].getReturnType().getName();
            System.out.println("\tReturn Type: " + returnString);

            Class[] parameterTypes = theMethods[i].getParameterTypes();
            System.out.print("\tParameter Types:");
            for (int k = 0; k < parameterTypes.length; k ++)
            {
                        String parameterString = parameterTypes[k].getName();
                        System.out.print(" " + parameterString);
            }
            System.out.println();
            }
      }
}

-----------------------------------------------------------------------------
```

ParkingLotClient.java

```
//import any syd packages
import syd.sydengine.*;
import syd.sydutil.*;
import syd.sydlistener.*;
import syd.syddirectory.*;
import syd.sydapp.ParkingLot.*;

//import any java related packages
import java.lang.*;
import java.util.*;
import java.io.*;
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class ParkingLotClient
{
      static Scanner sc = new Scanner(System.in);
      String appname = "ParkingLot";

      // Instantiating a SyDDispatcher object
      SyDDispatcher dispatcher = new SyDDispatcher();

      public ParkingLotClient()
      {
            Integer c = -1;
            while(c != 0)
            {
                  System.out.println("\n\tParkingLotClient v1.2");
                  System.out.println("\t0. Quit");
                  System.out.println("\t1. Get Available Spaces");
                  System.out.println("\t2. Update Parking Lot Server");
                  System.out.print("\nEnter choice: ");
                  c = Integer.parseInt(sc.nextLine());
                  switch(c)
                  {
                        case 0: System.exit(1); break;
                        case 1: getData(); break;
                        case 2: setData(); break;
                  default: System.out.println("Invalid choice ! Enter again
...");
                  }
            }
      }

      //Returns latest parking lot data from local directory
      void getData()
      {
            Vector<Object> paramtype = new Vector<Object>();
            Vector<Object> paramvalue = new Vector<Object>();
            Vector<Object> doclist = new Vector<Object>();

            System.out.print("\nEnter parking lot name: ");
            String name = sc.nextLine();

            //Invoking Application Server Object's Registered Methods
            try
            {
                  System.out.println("\nAccessing Parking Lot Data ...");
                  // Add the parameter type list
                  paramtype.addElement("java.lang.String");
                  //Add the parameter value list
                  paramvalue.addElement(name);
                  doclist =
dispatcher.invoke(appname,"getParkingLot",paramtype,paramvalue);
                  System.out.print("Client Received Parking Lot Data -> " +
show(doclist));
            }
            catch(java.util.NoSuchElementException ne)
            {
                  System.out.println("Parking Lot Name Invalid !");
            }
            catch(Exception e)
            {
                  System.out.println("Error in ParkingLotClient.getData(): "
+ e);
                  e.printStackTrace();
            }
      }

      void setData()
      {
            Vector<Object> paramtype = new Vector<Object>();
            Vector<Object> paramvalue = new Vector<Object>();
            Vector<Object> doclist = new Vector<Object>();

            System.out.print("\nEnter parking lot name: ");
            String name = sc.nextLine();
            System.out.print("Enter available spaces: ");
            Integer spaces = Integer.parseInt(sc.nextLine());

            try
            {
                  System.out.println("\nUpdating Parking Lot Server Object 
...");
                  // Add the parameter type list
                  paramtype.addElement("java.lang.String");
                  paramtype.addElement("java.lang.Integer");
                  paramtype.addElement("java.lang.Long");
                  //Add the parameter value list
                  paramvalue.addElement(name);
                  paramvalue.addElement(spaces);
                  paramvalue.addElement(System.currentTimeMillis());
                  doclist = 
dispatcher.invoke(appname,"setParkingLot",paramtype,paramvalue);
                  System.out.print("Client Updated Parking Lot Data -> " + 
show(doclist));
            }
            catch(java.util.NoSuchElementException ne)
            {
                  System.out.println("Parking Lot Name Invalid !");
            }
            catch(Exception e)
            {
                  System.out.println("Error in ParkingLotClient.setData(): " 
+ e);
                  e.printStackTrace();
            }
      }

      String show(Vector<Object> doclist)
      {
            String s = "";
            for(int i=0;i<doclist.size();i++)
                  s = s + (String)doclist.elementAt(i);
            return s;
      }

      public static void main(String[] args) throws IOException
      {
            ParkingLotClient p = new ParkingLotClient();
      }
}
-----------------------------------------------------------------------------
```