Mobile Agent in Concordia

Jian Peng, Binglin Li
Mathematics and Computer Science Department
Kent State University, Ohio, USA
email: {jpeng,binli}@mcs.kent.edu

Abstract

This paper first introduces the Concordia, including its principle, its components and its features. Then we focus our emphasis on agent collaboration and the facilities that Concordia provides us, which are the most important parts in concordia. In the end, the concordia’s merits and drawbacks are analyzed briefly.

1. Introduction

As the Internet and World Wide Web have become enormous popular in recent days, more and more attentions have been paid to the distributed applications development. Moreover, the network is being viewed as one of the most popular development platforms. However, there are still some shortcomings in this distributed development environment. The scalibility, reliability and security of distributed applications are still too far as desired. Some new software paradigms have been designed to enhance this distributed application development, such as IBM’s Aglets, General Magic’s Odyssey and etc.. One of which is Concordia.

Simply speaking, a Concordia system includes: a Java Virtual Machine, a Concordia Server, and Concordia Agents(at least one). The Java Virtual Machine, which is a standard environment, can be on any machine. The Concordia Server is a Java program which runs there, and at any other nodes on the network where agents may need to travel. The agent is also a Java program which the Concordia Server manages, including its code, data, and movement.

Generally, there can be many Concordia Servers and they are distributed each of the various nodes of a network, both user and server nodes. The Concordia Servers are aware of one another and connect on demand to transfer their agents in a secure and reliable fashion. The agent initiates the transfer by invoking the Concordia Servers methods. The Concordia Server inspects an object called the Itinerary, created and owned by each agent, to determine the appropriate destination. That destination is contacted and the agents image is transferred, where it is again stored persistently before being acknowledged. In this way the agent is given a reliable guarantee of transfer.

After being transferred, the agent is queued for execution on the receiving node. This happens promptly but possibly subject to certain administrative constraints. When the agent again begins executing, it is restarted on the new node according to the method specified in its itinerary, and it carries with it those objects which the programmer requested. Its security credentials are transferred with it automatically and its access to services is under local administrative control at all times.

The work that the agent performs depends on its purpose, that is, the code which it was programmed to execute. Generally, agents have several components, just as any program has. An agent might start interactively, by prompting the user for search information, then may travel to a server to perform the query. As its methods complete, the itinerary causes the agent to be moved to other Concordia nodes. Therefore, agents with different purposes will typically have different itineraries.

In all cases, Concordia provides a robust and highly reliable framework for the development and execution of secure, intelligent, mobile agent applications. Furthermore, Concordia's collaboration and Service Bridge features provide an ideal framework for enabling Cooperative Information Gathering, a multi-agent based approach for combining information from multiple, heterogeneous, information sources. The need for an efficient approach to information gathering is becoming increasingly important because of the changing and growing amount of information available in the World Wide Web and other complex information spaces. the Concordia agent is autonomous and self-determining in its operation. In this way, it is unique since it is in control of its own itinerary.

2. Architecture of Concordia

The Concordia system is made up of numerous components, each of which integrates together to create the full Mobile Agent framework. It is similar to a number of existing agent infrastructures and tool kits with respect to its support for the basic communication plumbing that is required for agent mobility. The Concordia Server is the major building block, inside which the various Concordia Managers reside. Certain Concordia components have a user interface, such as the Concordia Administrator. In any case, each Concordia component is responsible for a portion of the overall Concordia design, in a modular and extensible fashion. Concordia was developed based on java language. It consists of the followings:

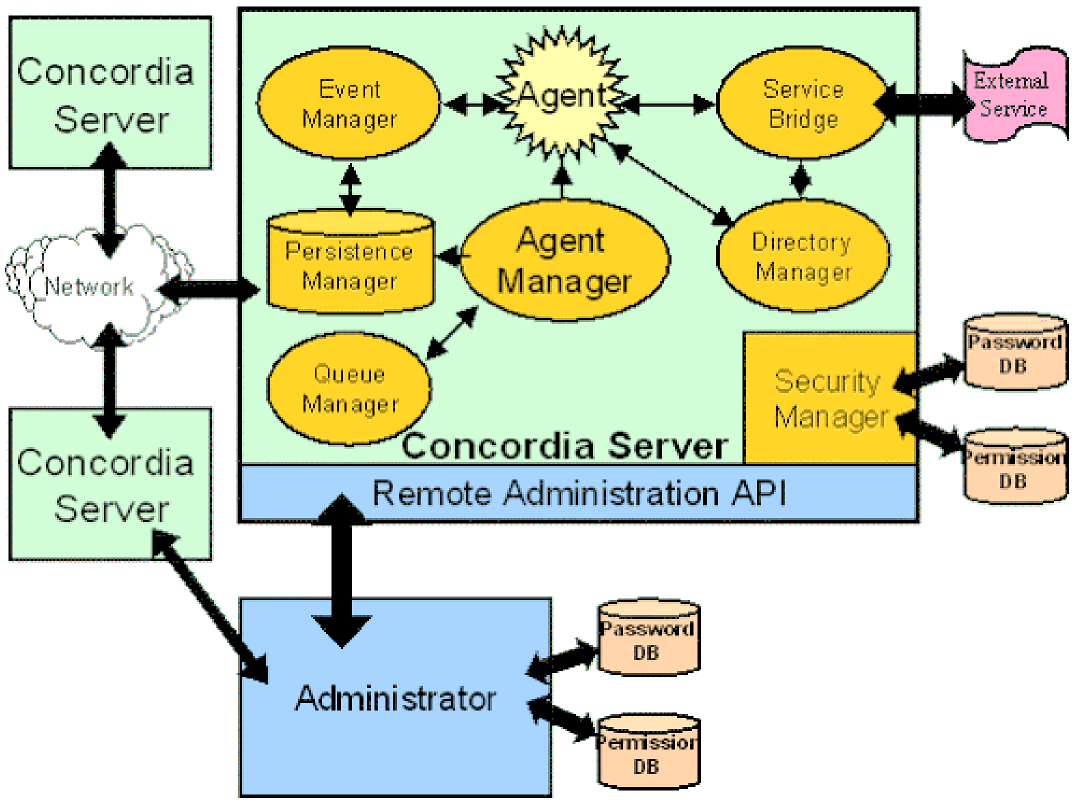

Figure 2.1. Concordia Architecture

(1) Concordia Server is the name of the complete Concordia component installed and running on a machine in the Concordia network. It serves as a communication server is responsible for concordia's agent transferring among the networks. At first, this transferring is activated by agent's invoking the server's methods. Then the agent traverses among the network until it reaches its destination. In the end, it brings the required data back.

(2) Agent Manager provides the communications infrastructure that allows for agents to be transmitted from and received by nodes on the network. Moreover, Concordia's agent mobility mechanism extends beyond the functionality provided in other Java-based agent systems. It offers a flexible scheme to dynamically invoke arbitrary method entry points within a common agent application. It abstracts the network interface in order that Agent programmers need not know any network specifics nor need to program any network interfaces. The Agent Manager Server also manages the life cycle of the agent. It provides for agent creation and destruction, and provides an environment in which the agents execute. It is highly mobile and its mobility can extend to a nimbler of local as well as wide area networks. to alleviate potential performance and reliability problems associated with the transmission of agents across networks with different characteristics in the underlying communication medium, the concordia infrastructure also provides support for transactional query of agents between conduit servers residing in different networks. And many applications can be achieved by users' programming their own agents, in the assistance of concordia's basic agents.

(3) Administrator is mainly responsible for the network administration of the Concordia. It resides on a separate Java virtual machine, and can cooperate with Concordia services which are running on the various nodes under administration. The Administration Manager manages all of the services provided by Concordia, including Agent Managers, Security Managers, Event Managers, etc. The Administration Manager supports remote administration from a central location, so only one Administration Manager is required in the Concordia network, although more can be employed as desired. The Administration Manager has a user interface component which is its primary means of use. The Administrator also monitors the progress of agents throughout the network and maintains agent and system statistics.

(4) Security Manager: Security is one of important criteria of programming languages. In concordia, the security manager is responsible for identifying users, authenticating their agents, protecting server resources and ensuring the security and integrity of agents and their accumulated data objects as the agent moves among systems. The Security Manager is also responsible for authorizing the use of dynamically loaded Java classes which satisfy the needs of agents. The Security Manager has a user interface component, in order to configure and monitor the security attributes of the various users and services known to Concordia. This user interface function is integrated into the Administration Manager interface. Security credentials used by the Security Manager may come from a number of sources. For secure, self-contained systems, it may be that no credentials are needed. For systems that traverse public or semi-public networks, encryption may be required but credentials may need only reflect user identity, such as user name or group id. For fully fledged agent systems deployed on the Internet, strong authentication and security can be provided from external authorities such as Version. All these security levels can be supported by Concordia's Security Manager.

(5) Persistence Manager is required to ensure that the concordia agents can recover from the system crash successfully. It maintains the state of agents in transit around the network. As a side benefit, it allows for the checkpoint and restart of agents in the event of system failure. Additionally, it can checkpoint objects upon request by agents, to provide finer granularity of reliability guarantees for critical procedures. The Persistence Manager is completely transparent in its operation, that is, neither the agents nor the administrator need control or monitor its operation. However, management access is available if needed.

(6) Event Manager: Concordia can provide two kinds of asynchronous distributed events: selected events and group oriented events and they are managed by the Event Manager. It handles the registration, posting and notification of events to and from agents. The Event Manager can pass event notification to agents on any node in the Concordia network. The Event Manager works in conjunction with the Concordia Server to distribute events as needed. An important function of the Event Manager is to support Concordia agent collaboration.

(7) Queue Manager: Queues are needed in the Concordia to store message produced by the concordia agents. The Queue Manager is responsible for scheduling and possibly retrying the movement of agents between Concordia systems. These features include the maintenance of agents as they await the opportunity to perform their work, maintaining their persistent state as they enter and leave a system, and retrying as necessary when Concordia systems are

disconnected from the network. The Queue Manager provides the mechanism for prioritizing and managing the execution of agents on entry to Concordia nodes.

(8) Directory Manager enables mobile agents to locate the application servers they wish to interact with on each host. It maintains a registry of application services available on each host it manages. A Concordia distributed system may be configured to include one or more Directory Managers (at most one per host). Application servers export their services to mobile agents by registering them with the Directory Manager. Mobile agents then obtain references to application servers via the Directory Manager's lookup operation.

The Directory Manager provides naming service in the Concordia network. The administrator may configure the name service in a number of ways, chosen according to the needs of the programmer and services. The Directory Manager may consult a local name service or may be set up to pass requests to other, existing name servers.

(9) Service Bridge provides the interface from Concordia agents to the services available at the various machines in the Concordia network. It comprises a set of programming extensions to provide access the native APIs as well as interfacing these to the Directory Manager and Security Manager.

(10) Agent Tools Library is a library which provides all the classes needed to develop Concordia Mobile Agents. This of course includes the Agent class, and others derived from Java base classes, with interfaces to the Concordia infrastructure.

3. Main Features of Concordia

3.1 Overview of main features

(1) Concordia employs existing TCP/IP communications services. Concordia does not impose a protocol or distributed computing service of its own.

(2) Advanced management function
This feature allows thousands of mobile agents to run on a single workstation. Concordia administration can start, stop, suspend, and resume Concordia Servers; view, stop, suspend, and resume agents at a Concordia Server; create, modify, delete users and/or permissions; upgrade and install Concordia Servers, monitor Concordia Server performance, and manage the components.

(3) Collaboration
It means that your agents can cooperate with each other to perform specific tasks. An application can be composed of several agents with specific sub-tasks. it can provide a number of benefits, such as enabling parallel operation over multiple servers or multiple networks. Using collaboration, an application can divide a task into subtask, the subtask can be carried out in the most appropriate places. The results of these sub-tasks are then assembled by the collaboration framework. A decision is made based upon the results, which can be used to determine destination, action, or other appropriate behavior.

(4) Service Bridge

It allows a developer to add services to a Concordia Server. Service Bridges may be managed remotely via the Concordia Administration Manager. For example, you can provide access to an application-specific service so the service does not need to travel with the mobile agent . The Service Bridge also provides a way out of the Virtual Machine to the outside world.

(5) Persistence and Queuing
Persistence and Queuing can provide for automatic retries of agent transmission and queue storage recovery in case of server and/or network failures. These 2 features also provides for load balancing when machines in a network provide different response time and the order of execution is important.

(6) Itinerary
It specifies where a mobile agent travels. It provides a method to allow destinations to be added or removed either by the application, mobile agent or the Concordia Administration.

(7) Service Naming
It is a name service for applications and agents. In an environment where information is dynamic (i.e. the internet), this provides an easy way to establish a list of locations where services reside.

(8) The Concordia Security Structure
Unlike most agent systems, it provides security based on the rights of the user of the applications - not the permissions given by the developer of the application. This provides for more control of which files, databases, resources, etc., are available to a specific end user. In addition , the security system protects resources from access by unauthorized mobile agents and protects mobile agents from being tampered with by unauthorized users.

(9) The Lightweight Agent Transporter API
It allows the developer to embed within a client application the ability to receive, execute, and launch Concordia Agents. The application can receive notifications from the mobile agent and can directly interact with mobile agent.

(10) Encryption
Concordia can provide users with the following two options: (i) Concordia provides Encryption as a security measure; (ii) the developers can plug in their own encryption scheme.

3.2 Concordia Security for Agent

There are general four types of security problems in mobile agent systems:
(1) secure network transfer of agents,
(2) protection of a host from attack or misuse by malicious agents,
(3) protection of an agent from attack by a another malicious agent, and
(4) the protection of an agent from attack from a malicious host.

The current Concordia security administrator only provides the first three types of protection. Within Concordia, once a server has been identified as a valid Concordia Server, it is considered to be a trusted environment and the Concordia security package does in fact take steps to ensure server integrity. It provides agent protection and resource protection.

Since the object store saves an agent and its state information to disk, it could also become a potential security risk. Concordia agent protection addresses securing this on-disk representation as well. Concordia does not attempt to protect an agent when it is present in memory, but relies on the protection offered by the operating system and the Java virtual machine. Concordia uses the term resource protection to refer to the process of protecting server resources from unauthorized access. This area of protection addresses the problem of securing a host from attack or misuse by agents. In addition, this protection is applied to protect agents from attack by other agents.

Agent protection refers to the process of protecting an agent's contents from tampering or inspection during transmission across a network connection or when stored on-disk. Such protection exists to ensure the privacy and integrity of the agent and the potentially sensitive information it carries. This problem can be thought of as being composed of two sub-problems: (i) transmission protection dealing with the network protection of an agent and (ii) storage protection protecting an agent when represented on disk.

(a) Transmission Protection

Protection of an agent during transmission is a generally recognized problem in agent systems. For more reliability, Concordia uses a persistent object store for periodically saving agent state. In case of system failure, this server uses the object store to reconstruct executing agents and resume agent travels.

Concordia provides transmission protection using Secure Sockets Layer version 3. SSL is a general-purpose network security protocol, which can provide authentication and encryption services for TCP connections.

Another choice for ensuring transmission protection is a public-private key encryption scheme independent of the socket communications level. However, SSL provides a more attractive solution primarily because it encapsulates security mechanisms below the application level and can be made to work with our existing RMI infrastructure.

b) Storage Protection

Storage protection guards an agent from access or modification when stored in Concordia's persistent store. This type of protection is particularly useful when Concordia is running on an operating system that does not have a secure file system.

When an agent is stored, its bytecodes, internal and travel status are all written to disk. This agent's information is encrypted using a symmetric key encryption algorithm and a generated agent-specific symmetric key. Concordia takes advantage of the Java Cryptography Extension to allow any of the following symmetric algorithms to be used: IDEA, DES, RC4, RC5, Misty, or Triple DES. The symmetric key used in the encryption is automatically generated by the Concordia Server when the Agent arrives. After the encrypted agent is written to disk, the symmetric key is then encrypted using a server-specific public key and then stored with the Agent. Decrypting an agent requires the server's private key. System administrators must ensure the security of this private key. This can be accomplished by storing the key on a secured file system or on removable media which can be physically secured by the administrator.

c) User Identities

In order to identify an agent with a particular user, Concordia associates a user identity with agents executing in the system. This identity is a Java object and is composed of three pieces of information: (i) a user name, (ii) a user group, and (iii) a password. These are roughly equivalent to the user names, groups and password found in secure operating systems. Within the user identity, the password is always stored in a secure hashed form and is never represented in clear text. Construction of an agent requires supplying the clear text password. Thus knowledge of the hashed password is not sufficient for assigning a user identity to an agent.

The user identity is usually represented in a shorthand form which looks like the following: username@group. So if a user named john were a member of the group accounting, this user's identity would be represented as john@accounting.

3.3 Mobile Agents in Concordia

Commonly Agent is defined as an independent software program which runs on behalf of a network user. It may run even if the user is disconnected from the network. Some agents run on specialized servers, others run on standard platforms. Concordia Mobile Agent can perform work on behalf of the users, such as collecting information or delivering requests. This mobility greatly enhances the productivity of each computing element in the network and creates a uniquely powerful computing environment well suited to a number of tasks.

The Concordia allows the creation of Mobile Agent programs written in the Java language. These programs use Concordia services to move about in a network of distributed machines and to access services available on them. Common examples are user GUIs, databases, and other agents. By using Concordia, a new class of simple, easy-to-write and easy-to-run programs is enabled.

Mobile Agent facilitates high quality, high performance, economical mobile applications. Therefore these applications can transparently use the network to accomplish their tasks, while taking full advantage of resources local to the many machines in the network. They process data at the data source, rather than fetching it remotely, therefore, allowing higher performance operation. They make best use of the network as they travel.

They permit secure Intranet-style communications on public networks. Security is an integral part of the Mobile Agent framework, and it provides for secure communications even over public networks. Agents carry user's credentials with them as they travel. And these credentials are authenticated during executing at every point in the network. Agents and their data are fully encrypted as they traverse the network. All this occurs with no programmer intervention.

Concordia agents are highly mobile and their mobility can extend to a number of local as well as wide area networks. To alleviate potential performance and reliability problems associated with the transmission of agents across networks with different characteristics in the underlying communication medium, the Concordia infrastructure also provides support for transactional queuing of agents between Conduit Servers residing on different networks.

(1) Agent Mobility

Since the agent objects are composed of a combination of code and data, object mobility means the network transportation of both code and data. Concordia also provides interfaces allowing agents to create other agents and to clone themselves. As stated earlier, agent mobility is accomplished by the Conduit Server. In the Mobile Agent, the following information should be provided: (i) destinations, (ii) itineraries and (iii) methods.

How does an agent travel? It is described by its Itinerary. The Itinerary is a data structure which is stored and maintained outside of the agent object itself. The Itinerary is composed of multiple Destinations. Each Destination describes a location to which an agent is to travel and the work the agent is to accomplish at that location. In the current implementation, location is defined by a hostname of a machine on the network and the work to accomplish is by a particular method of the agent class.

There are some important characteristics of this Itinerary model that is worth noting. The Itinerary is a completely separate data structure from the agent itself. Thus where the agent travels is maintained in a separate logical location than what the agent does. A design decision was better than the method of Telescript's. For flexibility reasons, the system allows agent's to modify their Itineraries at runtime. While this can introduce the same type of complexity as is seen with the Telescript's go, it was felt that this functionality will only be needed in very exceptional cases.

Mobility of an agent's data was accomplished using the Java Object Serialization facility. Transfer an agent state is a matter of serializing an agent's data down into a format suitable for network transmission, transmitting the data in this format, and then deserializing the data back into the original agent. This is very similar to the mechanism used by Java Remote Method Invocation (RMI) for passing an object by value between distributed objects. RMI itself does not provide for true object mobility as it provides for no mobility of an object's code and in fact requires that the code for any objects passed by value be pre-installed on both sides of the network connection. Java's Object Serialization features provide an almost transparent mechanism by which Java objects can be serialized into data streams and provided suitable technology for implementing agent mobility.

(2) Collaboration

Concordia's collaboration framework facilitates this type of interaction by enabling multiple agents to work together to solve complex problems. Suppose the agents visit local travel agencies and then share their intermediate results and collaborate before migrating to travel agency sites in other cities. If an agent determines that a particular resort does not have any available lodging meeting the user's criteria, the agents may determine to drop queries about trips to that destination. As more information is gathered, agents may also make other decisions. As this example demonstrates, agents can perform complex distributed computations more effectively if they correlate their results and alter their behavior based on the combined results. Concordia's collaboration framework facilitates this process.

The class of application described above divides a complex task into smaller pieces and delegates them to agents that migrate throughout the network to accomplish them. These agents perform computations, synchronously share results, and collaboratively determine any changes to future actions. Concordia employs a simple programming paradigm for this type of collaboration. The goals of the collaboration framework include: (i) A simple programming interface for synchronous collaboration. (ii) Asynchronous notification of exceptional conditions via events. (iii) Reliable and robust implementation utilizing proxy objects to shield agents from the effects of software failures within the collaboration framework. (iv) An infrastructure that enables location transparent inter-agent communication.

Agents within an application may form one or more collaboration units, known as agent groups. Concordia provides base classes for collaborating agents and agent groups. AgentGroups are implemented as distributed objects which export a simple interface to CollaboratorAgents. These agents hold remote references to AgentGroup distributed objects and access them via Java's Remote Method Invocation (RMI) facility.

AgentGroup collaboration is implemented as follows: The AgentGroup abstraction provides the distributed synchronization. Each application need only supply its own implementation of analyzeResults to analyze the collective results of the agents in the group and to allow each agent to adapt its behavior based on those results. Both the synchronization point and invocation of analyzeResults are encapsulated within the AgentGroup's collaborate method. This distributed synchronization scheme requires that each agent "arrive" at the collaboration point before collaboration may commence. Hence, it is ideally suited to applications that subdivide a complex problem into sub-tasks that correlate heir results. When each agent arrives at the collaboration point, it posts the results of its computation to the AgentGroup and blocks until all the agents in the group arrive. The AgentGroup collects the results of the agents' computations, and when all agents in the group arrive at a collaboration point, its collaborate method invokes analyzeResults on behalf of each agent, passing it the collective result set. The AgentGroup abstraction supports both parallel and serialized execution of the analysis stage of collaboration.

The AgentGroup also uses time-outs to detect potential deadlocks. Note that since AgentGroup collaboration is designed for closely coordinated agents, deadlocks are generally caused by programming errors. Hence, the AgentGroup does not need to use a more sophisticated scheme for deadlock detection or avoidance. An additional benefit of AgentGroup collaboration is that it enables location-transparent inter-agent communication. As each agent migrates, it carries a remote reference to an agentGroup distributed object and utilizes the AgentGroup as a gateway for communicating with the other members of the group.

As mentioned earlier, AgentGroups facilitate both synchronous collaboration and asynchronous notifications. This is possible because the AgentGroup object derives from the PersistentEventGroup object. AgentGroups forward any events they receive from their members to the remainder of the group. Occasionally, they may also initiate events that they deliver to the group.

As detailed above, Concordia's collaboration paradigm offers several benefits:
(1) simple programming interface for synchronous collaboration;
(2) asynchronous distributed event management;
(3) support for agent mobility;
(4) location-transparent inter-agent communication;
(5) reliability, persistence, and transparent recovery from failure;
(6) deadlock detection;
(7) a portable implementation.

4.Programming Methodology in Concordia

4.1 Writing Mobile Agents

Writing a Concordia Mobile Agent is in many ways little different from developing a non-mobile Java program. This section will give a brief introduction to the concept of writing a Concordia Mobile Agent.

4.1.1 Client/Server-Based Database Lookup

Assume we will perform a database lookup. Such an application will have three basic steps: searching the database, performing the lookup, and displaying the results.

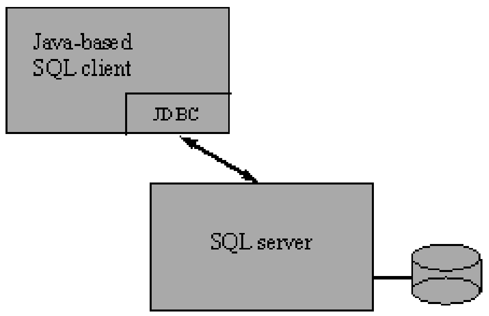

Figure 4.1. Java SQL Lookup

A Java program invoking this remote SQL database might look like this:

```
class DBQueryProgram {
    public static void main(String args[ ]) {
        String url = "jdbc:odbc:foxdatasource";
        String query = "SELECT sname, sid, average FROM FoxDatabase";
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        Connection con = DriverManager.getConnection(url);
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
            System.out.println("Student Name = " + rs.getString("sname"));
            System.out.println("Student ID   = " + rs.getString("sid"));
            System.out.println("Average Score= " + rs.getFloat("average"));
            System.out.print("\n");
        }
    }
}
```

In the above example, "foxdatasource" is a data source which can be accessed through jdbc:odbc. "FoxDatabase" is a FoxPro database which has at least three fields: sname,sid, average. The database is presumably located on some accessible machine. We connect to the database and execute an SQL query, then print three fields in the result. It is very simple to code a database lookup example in Java.

4.1.2 Agent-based Database Lookup

Now let’s take advantage of Query Agent to travel across the network to perform the database query.

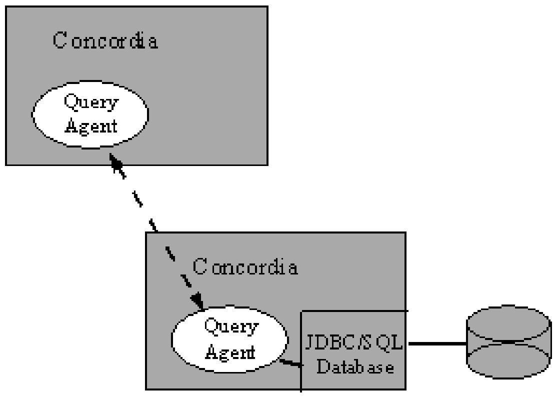

Figure 4.2. Agent SQL Lookup

First, we need to change our SQL query method to store the results into an object which we will move across the network. Our agent will travel to the server, execute the query and store the results, then return to the user and print them. We partition these tasks into individual methods, which we will instruct Concordia to invoke at the appropriate locations in the network.

Here is the updated Java code:

```
class QueryResult {
    public String sname;
    public String sid;
    public String average;
}

public class QueryAgent extends Agent {
    Vector itsResults;
    public QueryAgent() {
        itsResults = new Vector();
    }

    public void queryDatabase() {
        String url = "jdbc:odbc:foxdatabase";
        String query = "SELECT sname, sid, average FROM FoxDatabase";
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        Connection con = DriverManager.getConnection(url);
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
            QueryResult result = new QueryResult();
            result.sname = rs.getString("sname");
            result.sid = rs.getString("sid");
            result.average = rs.getFloat("average");
            itsResults.addElement(result);
        }
    }

    public void reportResults() {
        Enumeration enum = itsResults.elements();
        while (enum.hasMoreElements()) {
            QueryResult result = (QueryResult)enum.nextElement();
            System.out.println("Student Name  = " + result.sname);
            System.out.println("Student ID    = " + result.sid);
            System.out.println("Average Score = " + result.average);
            System.out.print("\n");
        }
    }
}
```

In agent-base database lookup, we coded our application in three distinct parts. First ,we create the object to store the results, then we generate the results and storing them in the object, and finally report them. Concordia will use the individual parts to schedule execution of our program at the appropriate locations in the network next.

Now that we have the *itsResults* object, the SQL query and the reporting, we need only to provide the "launcher" for the agent. Here we specify the itinerary, codebase, and classes to be used in constructing the agent, then set its task. First, we create the agent, then create an itinerary to move it first to a machine specified by *host1*, then back to another machine specified by *host2*. The agentsCodebase and relatedClasses specify the objects containing the methods and data necessary to complete our task.

```
public class QueryLauncher {
    public static void main(String args[ ]) {
        QueryAgent agent = new QueryAgent();
        Itinerary itinerary = new Itinerary();
        String host1 = "pc02.pcnet.mcs.kent.edu";
        String host2 = "vg20.vgnet.mcs.kent.edu";
        itinerary.addDestination(new Destination(host1, "queryDatabase"));
        itinerary.addDestination(new Destination(host2, "reportResults"));
        String agentsCodebase = "file:///C|/Query";
        String relatedClasses[ ] = {"QueryResult"};
        agent.setItinerary(itinerary);
        agent.setRelatedClasses(relatedClasses);
        agent.setHomeCodebaseURL(agentsCodebase);
        agent.launch();
    }
}
```

In the above example, the programmer created the *Itinerary*. When an agent is ready to travel in the network, it prepares a list of its intended destinations. The agent's itinerary is used by the Concordia Server to determine the network destination of the agent. As each method in the itinerary is completed, the local Concordia Server will move the agent and its objects to node specified in the next itinerary entry. When the itinerary is exhausted, the agent's journey is complete.

Here, the itinerary caused the agent to move to *host1* and execute the *queryDatabase* method, then to move back to *host2* and execute *reportResults*. The *setRelatedClasses* and *setHomeCodebaseURL* method of agent will cause the *QueryResult* class definitions and *agentsCodebase* to travel with the agent since the agent may use them when it travels.

The agent-based example performs exactly the same function as the client/server version in the previous example, but with the significant added features of agents.

4.2 Writing Collaborating Agents

The previous examples shows us how to create a simple Concordia Mobile Agent. An even more powerful feature available with Concordia is collaboration between agents.

4.2.1 Collaborating Agent-based Database Lookup

Let's consider an SQL query, one which results in a report of the average score for one class. Then we will consider this for several classes, in this case "class1", "class2", and "class3", with the aim to determine the class with the highest average score. We will use collaboration to return the results of the individual queries and also to analyze them by sorting the results.

```
class QueryResult implements Serializable {
  public String classname;
  public float  average;
}
```

```
public class QueryAgent extends CollaboratorAgent {
    private String className;
    private AgentGroup  itsGroup;
    private FinalResult combinedResult = null;
    public QueryAgent(AgentGroup group, String name) {
         super(group);
         itsGroup = group;
         className = name;
    }

public void queryDatabase() {
    String url = "jdbc:odbc:foxdatasource";
    String query = "SELECT AVG(average) From FoxDatabase"
            + " WHERE classname ='" + className + "'";
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    Connection con = DriverManager.getConnection(url);
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery(query);
    rs.next();
    QueryResult result = new QueryResult();
    result.classname = className;
    result.average = rs.getFloat(1);
    combinedResult = (FinalResult)(collaborate(itsGroup, (Object)result));
}
public void reportResults() {
     System.out.println("Other classes were");
        for(int i=0; i<combinedResult.others.length; i++) {
         System.out.println("Class Name: " +
                     combinedResult.others[i].classname+
                        "\tAverage: " + combinedResult.others[i].average);
            }
        System.out.println("\tClassname of Max average: " +
                     combinedResult.highest.classname+
                          "\tMaximum average: " + combinedResult.highest.average
    }
}
```

First, we code the basid SQL queries, assemble their results, then provide them to the collaboration. Second, we need to provide the collaboration routine, which will analyze the results. In this case, we will sort the results to determine the highest average. We do this by overriding the *analyzeResults* method of the *AgentGroupImpl*, which will be called only after all members of the *AgentGroup* have completed their tasks.

```
class CollaborationResult {
    public QueryResult highest;
    public QueryResult[ ] others;
}

class QueryAgentGroup extends AgentGroupImpl {
    protected synchronized Object analyzeResults(Enumeration results) {
        QueryResult highest = null;
        QueryResult[ ] others = new QueryResult[getGroupSize()-2];
        int i = 0;
        while (results.hasMoreElements()) {
            AgentResult result = (AgentResult)results.nextElement();
            QueryResult query = (QueryResult)result.getResult();
            if (query != null) {
                if (highest == null) {
                    highest = query;
                } else if (query.average > highest.average) {
                    others[i++] = highest;
                    highest = query;
                } else {
                    others[i++] = query;
                }
            }
        }
        CollaborationResult result = new CollaborationResult();
        result.highest = highest;
        result.others = others;
        return result;
    }
}
```

The Concordia collaboration resulted in an enumeration of the results of each *QueryAgent*, which was passed to the *analyzeResults* method While iterating over all the results. Next, we need a Launcher to launch each *QueryAgent*. First, we define a group, then we can define several *QueryAgent*, as following codes:

```
  QueryAgentGroup group;
  String host1 = "vg20.vgnet.mcs.kent.edu";
  String host2 = "pc16.pcnet.mcs.kent.edu";
  String relatedClasses[] = {"QueryAgentGroup", "FinalResult", "QueryResult"};
  String codebase = "file:///c|/collaborate";
  QueryAgent agent1 = new QueryAgent(group, "class1");
  Itinerary itinerary1 = new Itinerary();
  itinerary1.addDestination(new Destination(host1, "queryDatabase"));
  agent1.setItinerary(itinerary1);
  agent1.setRelatedClasses(relatedClasses);
  agent1.setHomeCodebaseURL(codebase);
  QueryAgent agent2 = new QueryAgent(group, "class2");
  Itinerary itinerary2 = new Itinerary();
  itinerary2.addDestination(new Destination(host2, "queryDatabase"));
  itinerary2.addDestination(new Destination(host2, "reportResults"));
  agent2.setItinerary(itinerary2);
  agent2.setRelatedClasses(relatedClasses);
  agent2.setHomeCodebaseURL(codebase);
  agent1.launch();
  agent2.launch();
```

After all QueryAgents are launched, they will travel to each host to perform their tasks specified in each agent's Itinerary. After all agents are exhausted, they will come to a collobaration point to collaborate their results. In this example, we have two agents. Agent1 will move to host1 to query the average score of "Class1". Agent2 will move to host2 to query the average score of "Class2". Then They will collaborate their results and the final results will be stored in *"combinedResult"*. Then agent2 will move to host2 to report the final results, just as defined in its Itinerary. Note that in this example, three classes "QueryAgentGroup", "FinalResult", "QueryResult" will travel with each agent since each agent will use them at each host.

5. Critical Analysis of features of Concordia

5.1 Compare Client/Server and Mobile Agents Programming

5.1.1 The Limitations of Client/Server

In distributed applications created with "client/server" programming, an operation is split into two parts: Client and Server. The Client making requests from a user machine to a Server which services the requests. A protocol is agreed upon and both the client and server are programmed to implement it. A network connection is established between them and the protocol is carried out.

The client/server model works well for certain applications. However it breaks down under other situations, including highly distributed systems, slow and/or poor quality network connections, and especially in the face of changing applications.

In a system with a single central server and numerous clients, there is only a problem of simple scaling. When multiple servers become involved, the scaling problems multiply rapidly, as each client must manage and maintain connections with the multiple servers.

With client/server comes a need for good quality network connections. First, the client needs to connect reliably to its server, because only by setting up and maintaining the connection may it be authenticated and secure. Second, the client needs to be assured of a predictable response, since its many requests of the server require full round trips to be completed. Third, it needs good bandwidth, since due to its very nature, client/server must copy data across the network.

Finally, the protocol which a client and server agree upon is by its very nature specialized and static. Often, specific procedures on the server are codified in the protocol and become a part of the interface. Certain classes of data types are bound to these procedures. These classes are extensible, but only at the high cost of recoding the application, providing for protocol version compatibility, software upgrade, etc. As the applications grow and the needs increase, client/server programming rapidly becomes an impediment to change.

5.1.2 Advantage of Mobile Agents Compared to Client/Server Model

Some limitations in Client/Server are overcomed by mobile agents.

With Mobile Agents, every node is a server in the agent network. The agent travels to the location specified by its Itinerary, then it will perform tasks at each point in its execution. So the flow of control actually moves across the network, instead of using the request/response architecture of client/server.

Since each agent moves with necessary information such as Itinerary, relatedClasses, HomeCodebaseURL, etc., the relationship between users and servers is coded into each agent instead of being pieced out across clients and servers. In fact, the agent creates the system, rathe than the network or the system administrators. Server administration becomes a matter simply of managing systems and monitoring local load.

The problem of robust networks is greatly diminished, for several reasons. The hold time for connections is reduced to only the time required to move the agent in or out of the machine. Because the agent carries its own credentials, the connection is simply a conduit, not tied to user authentication or spoofing. No requests flow across the connection, the agent itself moves only once. This allows for efficiency and optimization at several levels.

Last and most important, no application-level protocol is created by the use of agents. Therefore, compatibility is provided for any agent-based application. Complete upward compatibility becomes the norm rather than a problem to be tackled, and upgrading or reconfiguring an application may be done without regard to client deployment. Servers can be upgraded, services moved, load balancing interposed, security policy enforced, without interruptions or revisions to the network and clients.

5.2 Advantages of Concordia

Since Concordia is written in Java, it has all the advantages Java. It is portable, it runs on platforms large and small, and integrates easily with existing applications and frameworks.

Concordia agents provide for mobile applications. Agents support mobile computing as well as off-line processing and disconnected operation. These applications are in turn written with little or no knowledge of the underlying communications that they will employ. Concordia both hides the details from the programmer and user, as well as allows the agent to adapt to its environment and administration.

Concordia agents are secure. Security is an integral part of the Mobile Agent framework, and it provides for secure communications even over public networks. Agents carry user credentials with them as they travel, and these credentials are authenticated during execution at every point in the network. Agents and their data are fully encrypted as they traverse the network. All this occurs with no programmer intervention.

Concordia agents are reliable. All Concordia agents are checkpointed before execution by the Persistence Manager, and they may return to these checkpoints if necessary. Objects the agents may create are checkpointed as well. Coupled with the services of the Queue Manager while they are being exchanged across the network, Concordia agents are assured of reliability at every stage of their operation.

Concordia agents can collaborate. Collaboration provides a number of benefits, such as enabling parallel operation over multiple server or multiple networks. Using collaboration, an application can divide a task into subtasks, the subtasks can be carrried out by the colloboration framework. A decision is made upon the resuls, which can be used to determine destination, action, or other appropriate behavior.

5.3 Concordia and Java

5.3.1 The advantages of Java

The Java language has a number of advantages that make it particularly appropriatae for Mobile Agent technology. Java’s main appeal for agents is its portability. Its use of byecodes and its interpreted execution environment mean that any system with sufficient resources can host Java programs. A second advantage comes from the ubiquitous nature of Java on the Internet. Because it is embedded in many Web browsers, as well as application servers, there are many platforms deployed already. Another major advantage is the proliferation of tools that support Java programmers. Many programmers are already familiar with C++, which Java resembles in many ways. Finally, there is the movement of major segments of the software industry to Java. Not only will Java be here for many years to come, it will be employed in ever increasing applications.

5.3.1 The Disadvantages of Java

Java has brought about a lot advantages to Concordia, at the same time, Java has brought about some disadvantages to Concrodia too. Since Concordia uses Java Compiler to compile its application programs, it lost some flexibilities of its own. Let's see one simple example as follows :

```
    String host = "pc16.pcnet.mcs.kent.edu";
    String method1 = "queryDatabase";
    String method2 = "reportResults";
    itinerary.addDestination(new Destination(host, method1);
    itinerary.addDestination(new Destination(host, method2);
```

A very interesting thing here is: Concordia uses the String method1 to denote a member function of the Agent. If there is something wrong with the String such as the method1 is not a member function of the Agent , the compiler can not detect any compiling error! It is the programmar's responsibility to make sure that it is right. It is one main reason why Concordia codes can easily pass compiling, but it is prone to fail when they are executed. If Concordia uses compiler of its own, such kind of errors should be easily detected. Another situation happens when a SQL statement is executed. Only executing the program can the programmers know the SQL statement is right or wrong. That wastes a lot of time. Since querying a database and seting itinerary are back and bone of Concordia, when a large amount of complicated tasks need to be performed, it is very difficult for the programmers to make sure that the program can work or can not.

6. Conclusion

Concordia offers a complete frame work for the development and management of network-efficient mobile agent applications. The design goals of Concordia have centered on providing support for flexible agent mobility, agent collaboration, agent persistence, reliable agent transmission, and agent security.

Concordia's agent mobility mechanism extends beyond the functionality found in current Java-based agent systems by offering a flexible scheme for dynamic invocation of arbitrary method entry points within a common agent application. The Concordia framework offers support for agent interaction via the notion of agent collaboration, which allows agents to interact, modify external states, as well as internal agent states.

Through the use of proxies and object persistence, Concordia provides a highly robust environment where applications can gracefully recover from system or network failures. Using a transactional message queuing system, Concordia provides for reliable network transmission of agents even over unreliable network connections.

The infrastructure provided by Concordia extends the standard security mechanisms of the Java language to provide an identity-based system where the right given to an agent are determined from the identity of the user who launched the agent. Concordia also protects an agent and the information it carries from tampering when stored on disk or when transmitted over a network connection.

Mobile agents programming has many advantages. Mobile agents facilitates high quality, high performance, economical mobile applications. They enable use of portable, low-cost, personal communications devices. They permit secure Intranet-style communications on public networks. They efficiently and economically use low bandwidth, high latency, error prone communications channels

7. References

[1] Aglets: Mobile Java Agents, IBM Tokyo Research Lab, URL=http://www.ibm.co.jp/trl/projects/aglets
[2] D. T. Chang, D. B. Lange, "Programming Mobile Agents in Java" URL=http://www.trl.ibm.co.jp/aglets/
[3] Security and Reliability inConcordia, Mitsubishi Eletric ITA, URL=http://www.meitca.com/HSL/Projects/Concordia
[4] Concordia: An Infrastructure for Collaborating Mobile Agents, Mitsubishi Eletric ITA, URL=http://www.meitca.com/HSL/Projects/Concordia
[5] Concordia as Enabling Technology for Cooperative Information Gathering, Mitsubishi Eletric ITA, URL=http://www.meitca.com/HSL/Projects/Concordia
[6] D. T. Chang, D. B. Lange, "Mobile Agents: A New Paradigm for Distributed Object Computing on the WWW", In Proceedings of the OOPSLA96 Workshop: Toward the Integration of WWW and Distributed Object Technology, October 1996.
[7] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, "Itinerant Agents for Mobile Computing", IEEE Personal Communications Magazine, 2(5), October 1995.
[8] D.S. Milojicic, M. Condict, F. Reynolds, D. Bolinger, and P. Date, "Mobile Objects and Agents"
[9] "Object Serialization for Java", Javasoft Corporation, URL=http://chatsubo.javasoft.com/current/serial/index.html
[10] "Remote Method Invocation for Java", Javasoft Corporation, URL=http://chatsubo.javasoft.com/current/rmi/index.html
[11] T. Walsh, N. Paciorek, D. Wong, "Security and Reliability in Concordia&trade;", In Proceedings of the 31st Annual Hawaii International Conference on System Sciences (HICSS31), Kohala Coast, Big Island, Hawaii, January 1998.
[12] D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young, B. Peet, "Concordia: An Infrastructure for Collaborating Mobile Agents", In Mobile Agents: First International Workshop, Lecture Notes in Computer Science, Vol. 1219, Springer-Verlag, Berlin, Germany, 1997.