Mitsubishi Electric ITA
Horizon Systems Laboratory

Mobile Agent Computing

A White Paper

Table of Contents

Mobile Agents ***1***

Introduction to Mobile Agents **1**
An Example of a Mobile Agent Application 1
Advantages of Mobile Agent Programming 3

Breaking the Client/Server Barrier **3**
The Limitations of Client/Server 3
Mobile Agents to the Rescue 4

Operation on Diverse Hardware **5**
The Desktop System 5
The Portable PC 5
The Personal Communicator 5
The Server 6

The Software Infrastructure **6**

The Advantages of Java **6**

The Mobile Agent System **7**

Concordia ***8***

Introduction to Concordia **8**
Concordia Overview 8
Concordia Components 9
Concordia Server 9
Agent Manager 9
Administrator 10
Security Manager 10
Persistence Manager 10
Event Manager 10
Queue Manager 11
Directory Manager 11
Service Bridge 11
Agent Tools Library 11
Advantages of Concordia 11
Uses of Concordia 12

Deploying Concordia **13**
Management 13
Administration 13
Security 14
Deployment 14

Programming Concordia Applications **15**
Writing Mobile Agents 15
Database Lookup 16
Agent-based Database Lookup 17
Writing Collaborating Agents 20
Collaborating Agent-based Database Lookup 21
Creating Services 24

Putting It All Together ***25***

Application Examples **25**

Example: Remote Database Access 26
Example: Smart Messaging 27
Example: Groupware Manager 28
Example: Workflow 29
Example: Information Retrieval 30

Figures

- **Figure 1 - Agents at Work** **2**
- **Figure 2 - Concordia Architecture** **8**
- **Figure 3 - Java SQL Lookup** **16**
- **Figure 4 - Agent SQL Lookup** **17**
- **Figure 5 - Collaborating Agent SQL Lookup** **20**
- **Figure 6 - Remote Database Query** **26**
- **Figure 7 - Smart Messaging** **27**
- **Figure 8 - Groupware Manager** **28**
- **Figure 9 - Workflow** **29**
- **Figure 10 - Information Retrieval** **30**

Mobile Agents

A new software paradigm for distributed application development

Introduction to Mobile Agents

The term **"agent"** is heard frequently today. While it means a variety of things to a variety of people, commonly it is defined as an independent software program which runs on behalf of a network user. An agent may run when the user is disconnected from the network, even if the user is disconnected involuntarily. Some agents run on specialized servers, others run on standard platforms. Many examples of agent systems exist, and they are receiving much attention on the World Wide Web ("WWW").

At Mitsubishi Electric Information Technology Center America[1], we have developed a framework for the deployment of specialized agents called **Mobile Agents.** A Mobile Agent is specialized in that in addition to being an independent program executing on behalf of a network user, it can travel to multiple locations in the network. As it travels, it performs work on behalf of the user, 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.

Our framework, called **Concordia**, allows the creation of Mobile Agent programs written in the Java language. These programs use Concordia services to move about a network of distributed machines and to access services available on them. Common examples are user GUIs, databases, and other agents. Administrators control which services are available to which agents and users, and full management features are provided. By using Concordia, a new class of simple, easy-to-write and easy-to-run programs is enabled.

An Example of a Mobile Agent Application

A good example of a Mobile Agent application is a database search. Let's imagine a user with access to a corporate database is at some remote location, say a sales person is at a customer site. The sales person needs a price quotation and availability information for a product. Being at a remote location, there is no direct access to the corporate database, and the communications links are problematic. Security is a concern for the corporation, of course. How can agents create the solution to this problem?

The corporation first deploys the agent server. This is a straightforward operation which requires selecting a platform within the corporate Intranet. Perhaps this platform already exists, if not then many systems can provide it, such as a UNIX or Windows NT system. It can be collocated on an existing server machine, such as the database server, which in

[1] MEITCA, Horizon Systems Laboratory, 300 Third Ave., Waltham, MA 02154
MEITCA, Horizon Systems Laboratory Technical Marketing, 1060 East Arques Ave., Sunnyvale, CA 94086
(408) 523-6843, (408) 735-0903 Fax

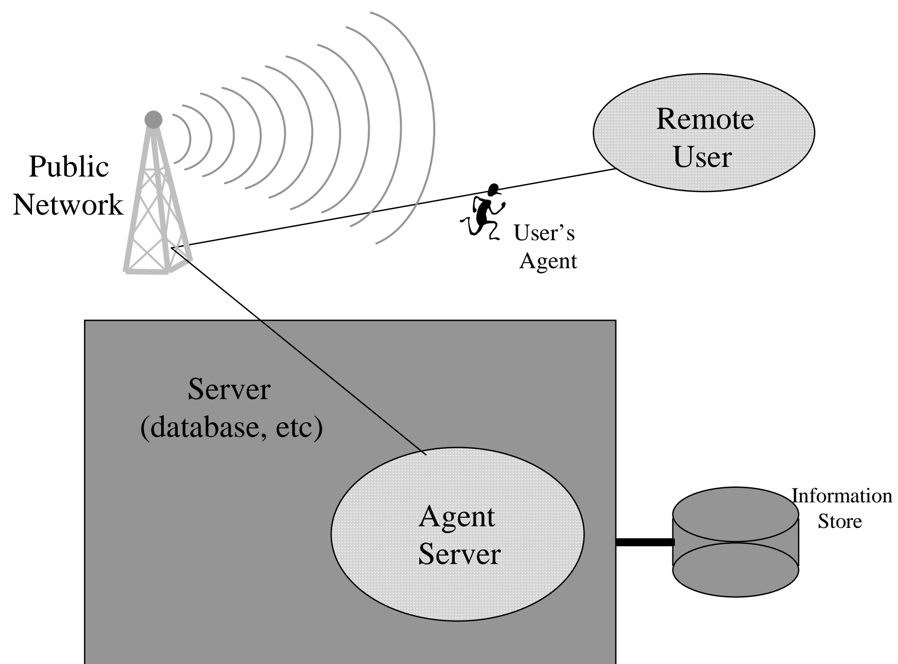
fact provides advantages of performance and management. The Mitsubishi Electric ITA product, Concordia, uses the Java virtual machine for its runtime environment, making installation particularly simple.

Once the agent server is installed, the sales people need portable computing platforms suitable for agents. In many cases, this platform will be a laptop or palmtop machine with either dialup or wireless communications devices. The software environment will be comprised of a suitable operating system combined with an agent runtime environment. In the case of Mitsubishi Electric ITA's product, Concordia, this again will be Java and may even be pre-installed.

The third step is to define security credentials and permissions which can be used to identify agents and their users to the existing database service, or whatever services are chosen for export by the corporation. An administrative process is then undertaken to manage and map these permissions, and the credentials are assigned to the users.

Finally, the corporation needs to create the agents which will actually perform the requests. This is not necessarily a major undertaking. For instance, Mitsubishi Electric ITA's Concordia system uses existing Java tools combined with a powerful Agent Tools Library to make easy the authoring of useful agents. In any case, the agent application is much easier to create and therefore is in production sooner. This is because it is the agent framework which transparently provides security, communications, and distribution so the programmer can focus on the job at hand.

Sales people then go to work productively, using the power of agents to perform their queries!



• Figure 1 - Agents at Work

Advantages of Mobile Agent Programming

The following are the primary advantages of Mobile Agents:

- They facilitate high quality, high performance, economical mobile applications.

  Applications employing Mobile Agents 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, allowing higher performance operation. They use the full spectrum of services available at each point in the network, such as GUI's at the user and database interface on servers. They make best use of the network as they travel.

- They enable use of portable, low-cost, personal communications devices.

  Network support, including security, is contained in a lightweight server which manages the movement of agents in the network. Coupled with the sophisticated, self-contained programming model afforded by agents, this permits a small footprint to be achieved on user devices, without sacrificing functionality for the application.

- 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 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.

- They efficiently and economically use low bandwidth, high latency, error prone communications channels.

  The agent network employs a store and forward mechanism to transfer agents between nodes. This is well suited to the problematic nature of many communications channels, especially in the mobile arena. Queuing and persistent checkpoints enhance this further, to the point that agents can use such channels with no degradation in reliability or response. Because the agent data processing takes place locally to the source, the network has no effect on the agent as it executes.

Breaking the Client/Server Barrier

The Limitations of Client/Server

Historically, distributed applications such as these are created with "client/server" programming. In this model, an operation is split into two parts across a network, with the client making requests from a user machine to a server which services the requests on a large, centralized system. 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 has the advantage of enabling the removal of the client to smaller, remote machines, and it 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. The use of two-tier systems or proxies only moves this problem to the network: it does not eliminate the basic problem.

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 and the end result is a special network version of an application programming interface. This interface is 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.

Mobile Agents to the Rescue

Mobile agents overcome all these inherent limitations in client/server.

First and foremost, the Mobile Agent shatters the very notion of client and server. With Mobile Agents, the flow of control actually moves across the network, instead of using the request/response architecture of client/server. In effect, every node is a server in the agent network, and the agent (program) moves to the location where it may find the services it needs to run at each point in its execution. For example, the same agent interacts with the user via a GUI to obtain request keys, then travels to a database server to make its request.

The scaling of servers and connections then becomes a straightforward capacity issue, without the complicated exponential scaling required between multiple servers. The relationship between users and servers is coded into each agent instead of being pieced out across clients and servers. It is the agent itself that creates the system, rather 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, in effect carrying a greater “payload” for each traversal. 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.

All in all, a significant advantage in Mobile Agents!

Operation on Diverse Hardware

To date, it has been very difficult, if not impossible, to provide user interfaces to systems from inexpensive, small, hand held user devices, nor in fact from mobile devices at all. Two things can change all that: the Java language and Mobile Agents.

We have seen how Mobile Agents can be used to create new, lightweight applications which move about the network to accomplish their jobs. Now consider what it means to deploy them on a range of devices from traditional desktop PC's to portables.

The Java language has created many new opportunities in the software world to create truly portable applications. The “skinny client” envisioned by the Java community consists of little more than the Java runtime, a GUI, and a communications path to a server. What better platform than this on which to consider agents? However, with so few local resources, how do we build powerful and useful applications which do not depend on expensive and local communications hardware?

The Desktop System

On the desktop, today the Web Browser is fast becoming the user interface of choice for many applications. Mobile agents are perfectly suited to this environment. With the powerful GUI tools, the integrated Java support, the security credentials and the rich communications across the LAN, all the pieces are in place for the Mobile Agent.

The Portable PC

The laptop or portable PC is basically identical to the desktop system, with the possible reduction in memory and disk resources, and of course the frequent disconnection from the LAN. This environment is where the advantages of Mobile Agents become apparent. While the machine may be able to function in a traditional client/server environment when docked in the office, it becomes much less useful as a remote client when used remotely. However, except for the network, all the software infrastructure is still available. Mobile agents can easily bridge this gap.

The Personal Communicator

The Personal Digital Assistant, or PDA, is not a new phenomenon, but the power of the hardware and software available on hand held, even pocketable devices, is. The Windows CE palmtop, the Apple Newton and the Palm Pilot communicator, are three excellent examples of powerful, portable user computing platforms. All three additionally can run Java, or will someday soon.

The Server

Finally, agents run on servers, such as databases, groupware servers, and virtually any other system of interest. The Java virtual machine is omnipresent on such systems and in many cases is already supporting local access to their services. To such a server, Mobile Agents are simply another standard client. When coupled to the power of the Mobile Agent network, an entirely different, more powerful system is created without impacting the server at all.

The Software Infrastructure

Creating a software infrastructure for the agents is the next step. Quite apart from the mechanisms of getting the agents to the various platforms in the network, verifying their identity and permissions, reconstituting their state and running them (all functions of Concordia), there is then the problem of making useful services available to them.

A number of possibilities exist:

> Use an existing legacy system. This requires exporting the legacy system's programming interface to the agent runtime. Given that Concordia uses Java, this is straightforward.
>
> Layer an existing legacy system under a standard agent API. This is similar to the first option, but more portable and possibly already provided by the software. A good example is JDBC, Java Data Base Connectivity, which is an open programming layer available for many databases.
>
> Code a new service as an agent. This is not so far-fetched as it may seem, given the power of agent programming. Agents are well suited to many dynamic tasks and can be the framework of choice for a wide variety of operations such as searching, directories, etc.
>
> Use a hybrid of all the above options.

When used as a "wrapper" for legacy systems, Mobile Agents can serve to provide numerous advantages not previously available. They can provide new clients for a fraction of the development cost. They can provide mobility to systems that were never designed with mobility in mind. They can provide management and security in systems over public networks, and a host of other advantages which we will cover when we discuss Concordia in detail.

The Advantages of Java

The Java language has a number of advantages that make it particularly appropriate for Mobile Agent technology. While Java is by no means the only language being employed by Mobile Agents, it is arguably the best choice. The reasons for this are many.

Java's main appeal for agents is its portability. Its use of bytecodes and its interpreted execution environment mean that any system with sufficient resources can host Java programs. There are even machines being built today that execute Java natively. For

Mobile Agents this is a tremendous opportunity. The more platforms capable of executing the agents' code, the better.

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. Application Programming Interfaces such as AWT, the Advanced Windowing Toolkit, JFC, the Java Foundation Classes, and JDBC, Java Data Base Connectivity, are leading toward even more deployment of Java. Additionally, this deployment exactly targets the sort of services that agents can best use.

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. Added to that is the migration of existing tools to Java and the creation of many more. The net result is an abundance of high quality, easy to use tools for both development and debugging.

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. We at Mitsubishi Electric ITA, as well as others, are committed to making Mobile Agents part of this progression.

The Mobile Agent System

We now come to see the characteristics of the systems that utilizes Mobile Agents. Starting with a legacy system, or simply the existence of a database, order entry, groupware, or other system, we add software interfaces to these existing services. The language bindings are in Java, perhaps to existing Java definitions such as JDBC. To these straightforward API extensions, we write agents, prototyping them in only a few lines of Java code, and these agents navigate the network transparently to perform the programmer's requests. Users are entered into a security database and under control of a central policy, are allowed to launch these agents. With truly a minimum of work, a secure, distributed, mobile system is up and running!

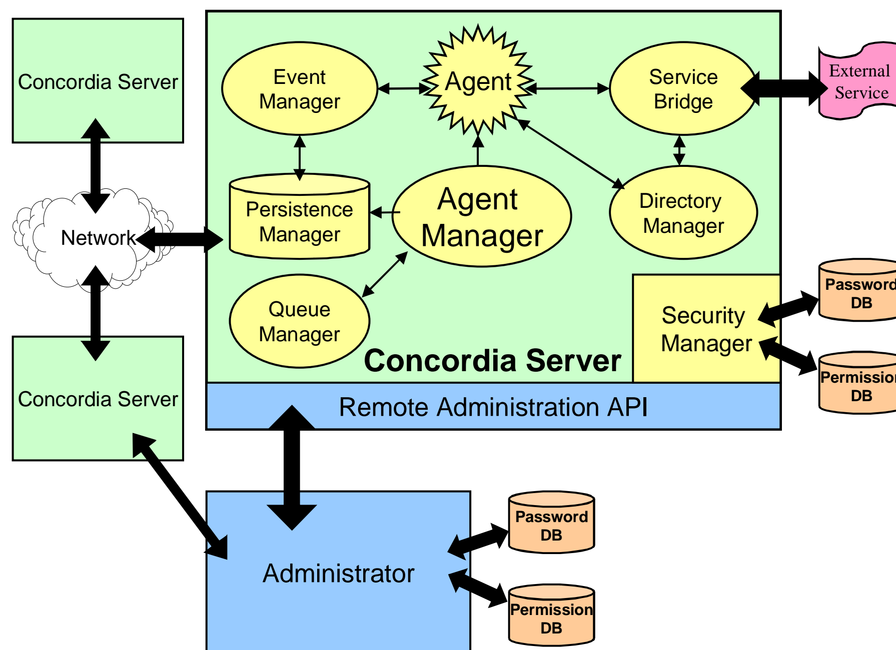
2

Concordia

A Framework for Mobile Agents

Introduction to Concordia

Concordia is a full-featured framework for the development and management of network-efficient Mobile Agent applications which extend to any device supporting Java. Concordia consists of multiple components, all written wholly in Java, which combine together to provide a complete, robust environment for applications.



• Figure 2 - Concordia Architecture

Concordia Overview

A Concordia system, at its simplest, is made up of a Java VM, a Concordia Server, and at least one Agent. The Java VM can be on any machine: it is a standard environment. 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.

Usually, there are many Concordia Servers, one on 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 agents in a secure and reliable fashion. The agent initiates the transfer by invoking the Concordia Server's methods. This signals the Concordia Server to suspend the agent and to create a persistent image of it to be transferred. 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 agent's 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. Or, the agent may simply be a kind of remote demon, such as a mailbox filter or notification sender. 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, 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.

The Concordia system is made up of numerous components, each of which integrates together to create the full Mobile Agent framework. 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 Components

All Concordia components are coded completely in the Java language.

Concordia Server

The Concordia Server is the name of the complete Concordia component installed and running on a machine in the Concordia network. It is comprised of manager components as follows.

Agent Manager

The Agent Manager provides the communications infrastructure that allows for agents to be transmitted from and received by nodes on the network. 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.

Administrator

Administration of the Concordia network is provided by the Administration Manager, in cooperation with Concordia services 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.

Security Manager

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 Verisign. All these security levels can be supported by Concordia's Security Manager.

Persistence Manager

The Persistence Manager 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.

Event Manager

The Event Manager 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.

Queue Manager

The Queue Manager is responsible for the 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.

Directory Manager

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.

Service Bridge

The 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 API's as well as interfacing these to the Directory Manager and Security Manager.

Agent Tools Library

The ATL 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.

Advantages of Concordia

Well, if you've read this far, we don't need to sell you again on the advantages of developing applications using Mobile Agents. What, however, are the advantages of Concordia itself?

Concordia is written in Java. Therefore it's portable, even ubiquitous. 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. Each agent carries the identity of the user that created it, and the operations the agent requests are subjected to the same user's permissions. Each agent is securely transmitted across the network, and no additional code is required to provide for secure, distributed operation.

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. The concept of collaboration is important and useful to the agent programmer. It can provide a number of benefits, such as enabling parallel operation over multiple servers or multiple networks. It can divide a task into suitable pieces, and these pieces can be carried out in the most appropriate places. The results of these sub-tasks are then assembled by collaboration. The collaboration framework then permits a decision to made based upon the results, which can be used to determine destination, action, or other appropriate behavior.

Uses of Concordia

Concordia:

- Enables mobilization of legacy applications
- Is a great way to program mobile devices as clients of applications
- Breaks client/server barriers
- Integrates with distributed objects e.g. CORBA
- Integrates with legacy systems e.g. databases.
- Easily piggybacks on Web
- Easily runs standalone

Concordia agents:

- Process data at the data source
- Pull data with them as they travel, i.e. they “learn”
- Can literally run anywhere: Web, desktop, palmtop, etc.
- Enable highly scaleable and parallel programming.
- Hide the network transport from application, developer, and user.
- Hide distribution, scale, parallelism from application.

And Concordia systems:

- Offer rapid prototyping with easy paths to production.
- Offer robust operation via persistent agents.
- Provide security and integrity.
- Support off-line and/or disconnected operation.

- Provide for heterogeneous database access
- Are a natural for software distribution - agents carry code to remote platforms

Deploying Concordia

Deploying Concordia is made substantially easier by Concordia's use of Java as its runtime framework. The portability of the Java virtual machine, coupled with Concordia's advanced administration makes the process an evolutionary one. This is because Java can be integrated into practically any system which runs the services that agents might need to access. In this way, no additional systems need be added to the network.

Next, Concordia provides its own advanced management functions, including administration and security. These are provided with easy to use GUI's and since they are coded completely in Java, they are immediately available wherever Concordia runs, without installation or porting effort.

Finally, Concordia makes use of available networks, it does not impose a protocol or distributed computing service of its own. Normally, Concordia employs existing TCP/IP communications services, widely available and compatible with local area network, dialup networks, and wireless public and private networks. Concordia provides its own security layer to protect the agents and their data as they pass across all such networks.

Management

Management of the many servers in a Concordia network is provided by the Concordia Administration Manager, which operates in conjunction with services available at each Concordia node. These services include the Concordia Servers themselves, along with the various Concordia Managers, including Agent, Security, Queue, etc. The Concordia Administration Manager provides a single graphical interface to the administrator for all the Concordia nodes in the network.

Administration

Concordia provides for administration of all its servers and services from any Concordia Administration Manager. The administration available falls into two major area, roughly divided along managing the Concordia Server and managing Concordia Agents.

The Concordia administrator can perform the following operations on Concordia Servers. This is not an exhaustive list but is intended to outline the major features of Concordia administration.

- Start and stop Concordia Servers and Managers.
- Upgrade and install Concordia Servers and installed software.
- Monitor Concordia Server performance.
- View Concordia Server logs.
- Manage the Concordia Persistent Store.

- Manage the Concordia Queues.

The Concordia administrator can also perform the following operations on individual Concordia agents. Again, this list is not exhaustive, but representative.

- Install and remove Agent code and libraries
- Manage agent itineraries
- Remotely launch agents
- Terminate, suspend, and resume agents.
- Monitor individual agent operations.

Security

The Concordia Administration Manager additionally manages Concordia security. Among the security aspects are the following.

- Management of trust relationships between Concordia Servers.
- User permission administration: user account, group and access to services.
- Encryption key administration.
- Monitor security logs.
- Monitor security statistics.

Deployment

Given these powerful tools, deploying Concordia is easy to achieve. Starting with an existing system or without, the requirements are few.

First, users must be assigned to devices, and these devices need the Java virtual machine, Concordia software, and a network connection such as TCP/IP.

Second, services must be provided and Java virtual machines must have access to them, either locally or remotely. Generally, the servers will have a local Java virtual machine, and will be connected to a LAN. User devices will be on the LAN or will connect via remote means. User devices could even be assigned accounts on the server itself, and both user and service will share the same Java virtual machine. Concordia software will be installed and configured.

Third, an administration node will be identified and configured. Again, this node can easily be the same as the major service node, or a different one. There can be multiple administration nodes in the network, for redundancy or for partitioning of tasks.

Fourth, agents are deployed in the new software infrastructure. These agents are individual and specially coded to perform their task, so there may be many or few, depending on need. Some agents will serve to execute tasks on user demand, others may

reside close to services to perform disconnected service enhancements. Agents are expected to travel as necessary to complete their tasks in the Concordia network.

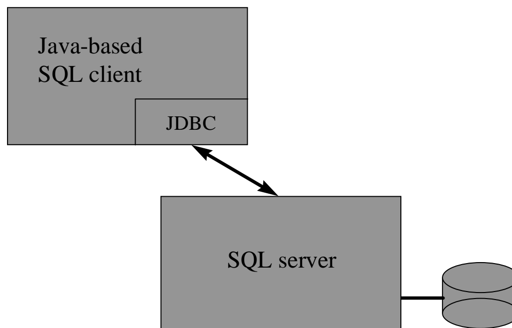
Programming Concordia Applications

Writing Mobile Agents

Writing a Concordia Mobile Agent is in many ways little different from developing a non-mobile Java program. The difference comes in structuring the application in such a way to take best advantage of the Concordia facilities. This section will give a brief introduction to the concept of writing a Concordia Mobile Agent. Mitsubishi Electric ITA has prepared a paper, ***"Concordia Agent Development Guide",*** which covers the issues of writing Mobile Agents in great detail. It is available upon request.

Database Lookup

Let's say we wish to perform a database lookup. Such an application will have three basic steps: determining the database to search and the keys, performing the lookup, and displaying the results.



• Figure 3 - Java SQL Lookup

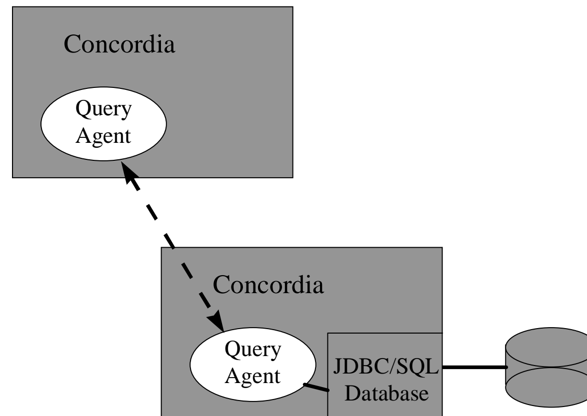
We'll start with this simple example, then see how our approach changes when we employ agents. A Java program invoking this remote SQL database might look like this:

```
class DBAccessProgram {

   public static void main(String args[ ]) {
      String url = "jdbc:odbc:corp";
      Connection con = DriverManager.getConnection(url);
      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(
                "SELECT CustomerID, CompanyName FROM Customers ");

      while (rs.next()) {
           System.out.println("CustomerID = " + rs.getString(1));
           System.out.println("CompanyName = " + rs.getString(2));
           System.out.print("\n");
      }
   }
}
```

In the above example, we start by specifying a JDBC database called "corp", which is presumably located on some accessible machine. We connect to the database and execute an SQL query, then print two fields in the result. It's useful to note how simple it is to code a relatively powerful example in a very few lines of Java.

Agent-based Database Lookup

Now, instead of using the client/server method of remotely querying the database, let's take advantage of Concordia to travel across the network to perform our database search locally, like this.



• Figure 4 - 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's the updated Java code:

```
class QueryResult {
   public String customerId;
   public String companyName;
}

public class DBAccessAgent extends Agent {

   Vector itsResults;
   public DBAccessAgent() {
      itsResults = new Vector();
   }

   public void queryDatabase() {
      String url = "jdbc:odbc:corp";
      Connection con = DriverManager.getConnection(url);
      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(
                  "SELECT CustomerID, CompanyName FROM Customers");

      while (rs.next()) {
            QueryResult result = new QueryResult();
            result.customerId = rs.getString(1);
            result.companyName = rs.getString(2);
            itsResults.addElement(result);
      }
   }

   public void reportResults() {
      Enumeration enum = itsResults.elements();
      while (enum.hasMoreElements()) {
            QueryResult result = (QueryResult)enum.nextElement();
            System.out.println("CustomerID = " + result.customerId);
            System.out.println("CompanyName = " + result.companyName);
            System.out.print("\n");
      }
   }
}
```

Note that we have coded our application in three distinct pieces, creating the object to store the results, actually generating the results and storing them in the object, and finally reporting them. In addition to being good coding practice, Concordia will use the individual pieces to schedule execution of our program at the appropriate locations in the network, as we see following.

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 it about it task. First, we create the agent, then create an itinerary to move it first to a machine called *dbserver*, then back to the *workstation*. The agentsCodebase and relatedClasses specify the objects containing the methods and data necessary to complete our task.

```
public class TestLaunch {

   public static void main(String args[ ]) {
      DBAccessAgent agent = new DBAccessAgent();
      Itinerary itinerary = new Itinerary();
      itinerary.addDestination(new Destination("dbserver", "queryDatabase"));
      itinerary.addDestination(new Destination("workstation", "reportResults"));
      String agentsCodebase = "file:C:\MyAgent";
      String relatedClasses[ ] = {"QueryResult"};

      BootStrap.launchAgent(agent, itinerary, agentsCodebase, relatedClasses);
   }
}
```

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. The itinerary may be dynamically modified in transit by the agent or by any Concordia administrator.

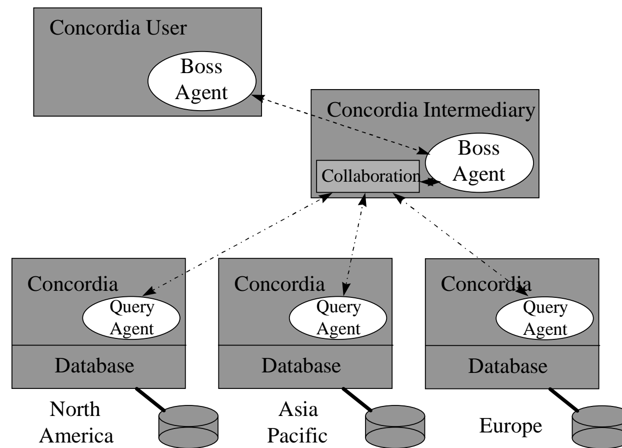
Here, the itinerary caused the agent to move to *dbserver* and execute the *queryDatabase* method, then to move back to *workstation* and execute *reportResults*. The additional arguments to the *launchAgent* method cause the codebase and the *QueryResult* class definitions to travel with the agent.

Our agent-based example performs exactly the same function as the stock client/server version in the previous example, but with the significant added features of agents. We achieved this simply by restructuring code and adding the "launcher". In return we have a distributed, mobile, secure and manageable solution.

Writing Collaborating Agents

The previous example shows us how to create a simple Concordia Mobile Agent. An even more powerful feature available with Concordia is collaboration between agents. In the previous example, we only needed to perform a single lookup in a single database. How might we restructure the application if we needed to query many databases?

One way would be to program the agent to travel to several servers or databases in the Concordia network, picking up information as it traveled. When the agent completed its itinerary, it would return with the results. Another, better way, would be for the original agent to spawn individual agents to perform the queries and to each return with a single result, then to assemble them in a process we call collaboration.



• Figure 5 - Collaborating Agent SQL Lookup

The advantages of collaboration are several. First, the operation is distributed, if a server is unavailable the other operations will not fail. Second, the queries happen in parallel, which increases throughput. Third, the agent could be programmed to make decisions based upon the results. For instance, if the agent were checking for the best price, it could then check availability and decide dynamically with which supplier to place an order.

Collaborating Agent-based Database Lookup

Here's an example of such an agent. In order to show the power of collaboration, let's consider a more complex SQL query, one which results in a report of the total sales for the region represented in each database. Then we will consider this for several regions, in this case "North America", "Asia Pacific", and "Europe", with the aim to be to determine the region with the highest sales. 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 region;
   public float sales;
}

public class DemoQueryAgent extends CollaboratorAgent {

   String itsRegion;
   public DemoQueryAgent(AgentGroup group, String region) {
      super(group);
      itsRegion = region;
   }

   public void queryDatabase() {
      String url = "jdbc:odbc:regional";
      Connection con = DriverManager.getConnection(url);
      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery("SELECT DISTINCTROW Regions.Region,
Sum([UnitPrice]*[Quantity]) AS Total " +
                              "FROM (Customers INNER JOIN (Orders INNER JOIN [Order
Details] ON Orders.OrderID = [Order Details].OrderID) ON Customers.CustomerID =
Orders.CustomerID) INNER JOIN Regions ON Customers.Country = Regions.Country " +
                              "GROUP BY Regions.Region " +
                              "HAVING (((Regions.Region)='" + itsRegion + "')) " +
                              "ORDER BY Regions.Region;");

      rs.next();
      QueryResult result = new QueryResult();
      result.region = rs.getString(1);
      result.sales = rs.getFloat(2);

     ((AgentGroup)getGroups().nextElement()).collaborate(newAgentResult(getAgentID(),
result));
     }
}
```

Now that we have coded the basic SQL queries, assembled their results, and provided them to the collaboration, we need to provide the collaboration routine, which will analyze the results. In this case, we will sort the results to determine the highest sales. 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.

This DemoAgentGroup is created on the intermediary server and exists there with the DemoBossAgent, but in fact it could be created anywhere in the Concordia network.

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

class DemoAgentGroup 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.sales > highest.sales) {
                      others[i++] = highest;
                      highest = query;
                } else {
                      others[i++] = query;
                }
          }
        }

        CollaborationResult result = new CollaborationResult();
        result.highest = highest;
        result.others = others;
        return result;
   }
}
```

We see that the Concordia collaboration resulted in an enumeration of the results of each *DemoQueryAgent*, which was passed to the *analyzeResults* method While iterating over all the results, we remember the region with the highest sales, as well as the individual results. This *CollaborationResult* is captured in the next code fragment.

Finally, we create the "*DemoBossAgent*", which initiates the collaboration. It will travel from an end-user device to an intermediary server. Once there, it constructs the three other Agents which travel to the database servers and perform the SQL queries. The *DemoBossAgent* shares in the collaboration, and once complete, takes the results back to the user for display.

```
public class DemoBossAgent extends CollaboratorAgent {
   CollaborationResult      itsResult;

   public void performCollaboration() {
      DemoAgentGroup    group = new DemoAgentGroup();
      addGroup(group);

      DemoQueryAgent    agent1 = new DemoQueryAgent(group, "North America");
      Itinerary      itinerary1 = new Itinerary();
      itinerary1.addDestination(new Destination("dbserver.NA.mycompany.com",
"queryDatabase"));

      DemoQueryAgent    agent2 = new DemoQueryAgent(group, "Asia Pacific";
      Itinerary      itinerary2 = new Itinerary();
      itinerary2.addDestination(new Destination("dbserver.AP.mycompany.com",
"queryDatabase"));

      DemoQueryAgent    agent3 = new DemoQueryAgent(group, "Europe");
      Itinerary      itinerary3 = new Itinerary();
      itinerary3.addDestination(new Destination("dbserver.EUR.mycompany.com",
"queryDatabase"));

      BootStrap.launchAgent(agent1, itinerary1, getHomeCodebaseURL(), null);
      BootStrap.launchAgent(agent2, itinerary2, getHomeCodebaseURL(), null);
      BootStrap.launchAgent(agent3, itinerary3, getHomeCodebaseURL(), null);

      itsResult = (CollaborationResult)group.collaborate(new AgentResult(getAgentID(),
null));
   }
}
```

In the example above, note that the programmer created an ***AgentGroup*** which permits collaboration among its member agents, wherever they may be. Then the original agent created copies of itself which each carried out as a separate task, and provided the results. When all the agents' tasks were complete, the *DemoAgentGroup* brought the results together via the *analyzeResults* method, and the original agent took subsequent action.

The only thing left to add is the launcher for the *DemoBossAgent*, which will create the master agent, send it to an appropriate node to start the three query agents and collaborate their results, then travel back to the user to report them. It is similar to the previous example and left as an exercise for the reader.

Mitsubishi Electric ITA has prepared a paper, ***"How to Write Collaborating Concordia Agents",*** which covers collaboration in more detail. It is available upon request.

Creating Services

Now that we know how to create Concordia Mobile Agents, all we need to know to take advantage of them is how to make services available to them, so that they may make themselves useful. In many instances, this is trivially easy: the services are already available in Java.

The best example of this is JDBC, the Java Data Base Connectivity layer. Most commercial databases are already available through this API. The examples above use it, in fact. Another example of established services is AWT, the Java Advanced Windowing Toolkit and its extension, JFC, the Java Foundation Classes. This set of interfaces allows Java programmers to present a graphical user interface.

Many other services are available in Java, and many more are arriving every day. The Web itself is available programmatically and with it the many services it provides. Interfaces to CORBA and IIOP are available for importing the significant systems available there.

In the case of private services, it becomes simply a matter of developing a Java API for them to be accessed by agents. This API is useful for any Java programming, but in the context of agents it becomes even more powerful. It is a simple matter to create such API bindings[2].

[2] Reference to Java server/servlet documentation

3

Putting It All Together

Creating Useful Mobile Agent Applications

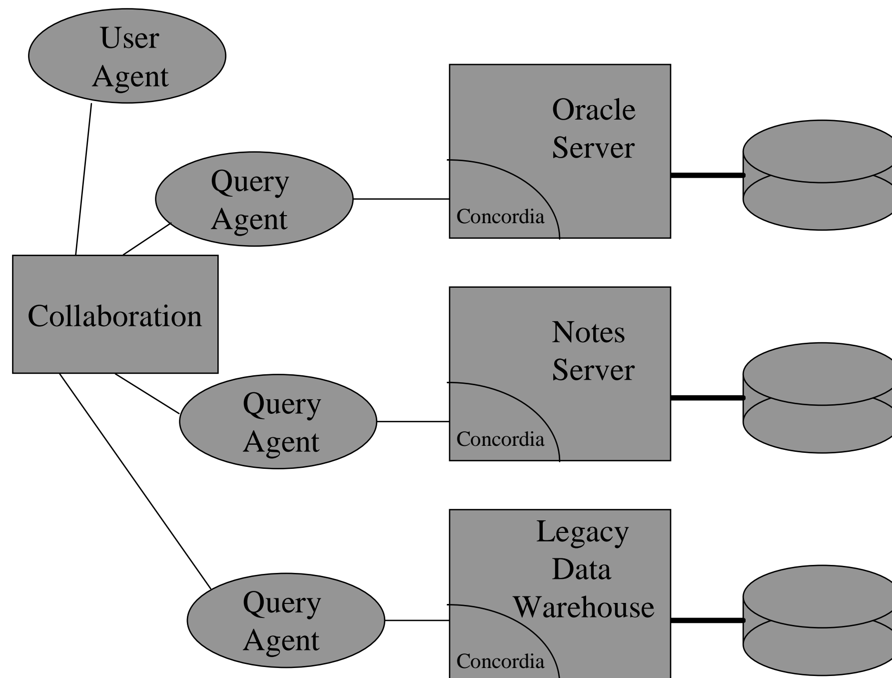
Application Examples

Many applications can be created using Mobile Agents and Concordia. Existing applications can be "mobilized" by integrating agents into their workings. In this way, agents can be integrated in an evolutionary way, existing alongside legacy applications and supporting key new features such as mobility.

Other applications may be written from the start with Mobile Agents. Messaging applications are a clear candidate, with "smart messages" that may find their user among a number of possible destinations, or messages which carry active content and must function properly on a wide variety of destination systems. Concordia has a companion Java-based email product which will integrate with agents to provide advanced messaging functionality.

Example: Remote Database Access

In the previous section, we saw an agent code example showing multiple agents traveling to multiple databases to solve a problem. While useful, the example can be taken much further. What if the various regions used different database formats or employed different software? Even without a generic database agent API, it would be possible to create Java drivers for the various databases for agents to employ. By using collaboration to pass the results, the agent programmer would not be required to understand the differences between the database formats. So in addition to the advantages of distribution, parallel operation and remote access, the agent approach gives advantages.

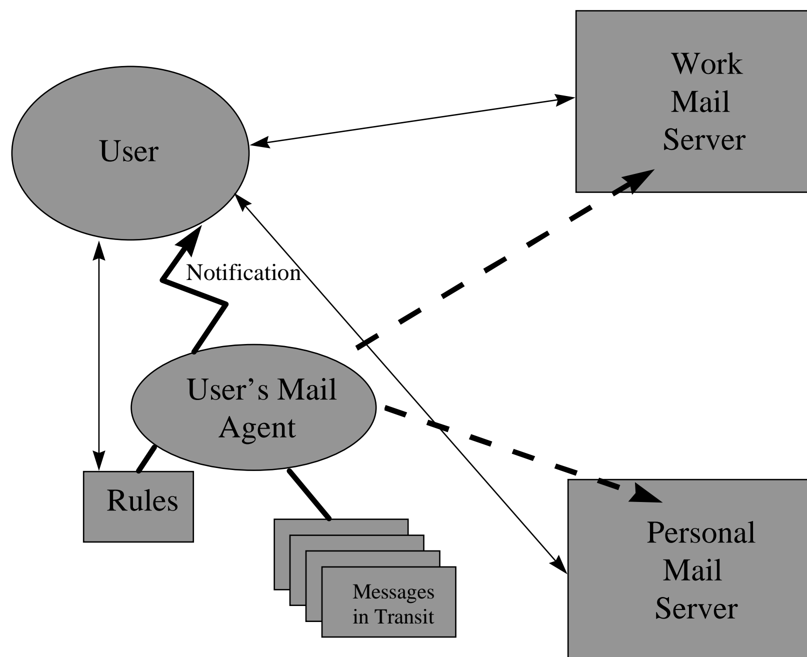


• Figure 6 - Remote Database Query

Example: Smart Messaging

Messages can employ agents to shepherd them to their destination under both delivery rules and user-defined rules. For example, users may specify their preferred mailboxes and mobile routing hints as well as specifying their location. During working hours, the user might be at their desk, while after hours at home. Or when on the road, the user's pager might be an appropriate destination. Perhaps all of the above in the case of an urgent message. As well, messages might be filtered to avoid such forwarding for routine or large messages. The capabilities are dynamic and completely under the control of the user in conjunction with the power of their agents.

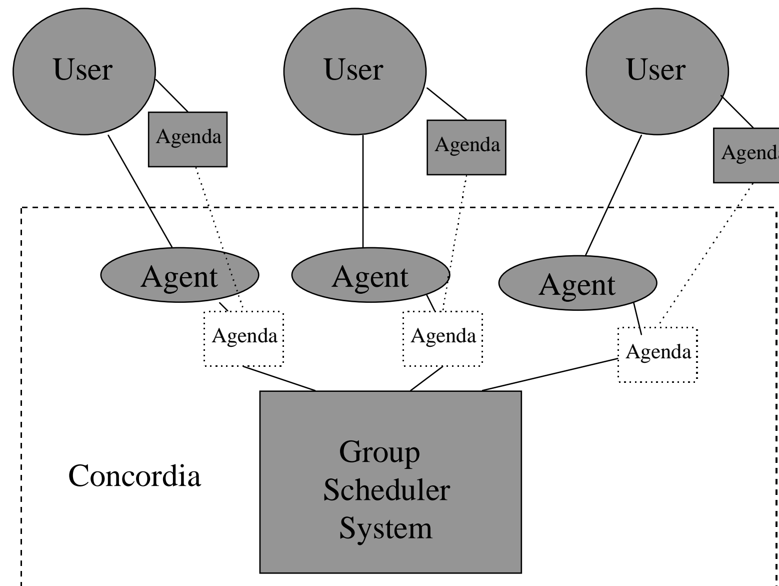
Another feature enabled by agents is the ability to provide "active content". Agents can enclose themselves in email or other messages, and given appropriate authentication and security, can execute on the destination device. This can provide for highly sophisticated applications, which can perform complex tasks at any point in the network. An agent could dynamically fetch information from the network once reaching the user, perhaps displaying complex data or interacting with the user. An agent could obtain billing or registration information and install software. In effect, the email system might become a transport service for agents.



• Figure 7 - Smart Messaging

Example: Groupware Manager

A contact manager or group scheduler type application is ideal for agents when the problems of wider distribution of users and frequent disconnection from the network are present. Agents can serve as proxies to solve the disconnection problem. Agents can collaborate to solve scheduling problems, to solve problems of scale and distribution. Agents can even notify users when critical events occur, such as priority meetings.

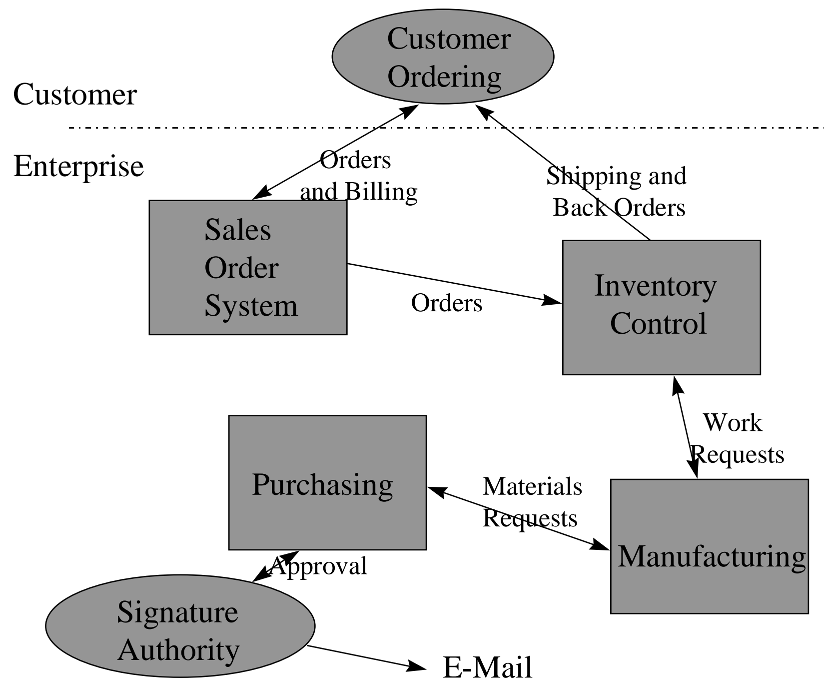


• Figure 8 - Groupware Manager

Example: Workflow

Agents carry workflow in distributed, scaleable fashion, Agents are well suited to watching and waiting, as well as dynamically making decisions. They can carry requests from user to system and from system to system. They can watch databases and carry out actions when critical events occur, such as inventory falling below certain levels. They can notify users and request information specific to the task, such as purchase approval from the appropriate individual in the promptest fashion.

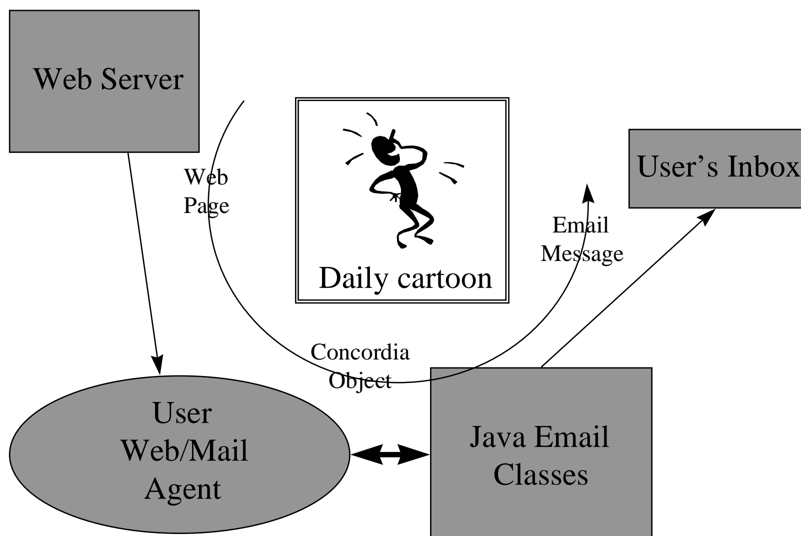
In such a system, it's interesting to notice that agents can carry dynamic workflow between systems. While it's not necessary that agents carry all such workflow, they are extremely well suited to the asynchronous, user-driven events such as order taking and signature authority, especially in conjunction with an agent-aware messaging system (as explored in a previous example). Even within an enterprise, agents can solve highly complex or large scale problems cleanly and efficiently.



• Figure 9 - Workflow

Example: Information Retrieval

A worthwhile but perhaps lighthearted application might be for agents to fetch the cartoon of the day, delivering it to user's mailbox via Mitsubishi Electric ITA's Java email classes. Of course, the cartoon could be useful news, from a news server, and such news could even be processed by the agent before delivery. Perhaps a stock market report could be summarized from the raw data, to highlight major indexes and portfolio items. The agent would do so dynamically, and under complete user control with no system overhead; if the user's interests changed, only the agent would need to be modified. This sort of agent application highlights both the powerful programmable capabilities of agents as well as the highly scaleable solution agents provide.



• Figure 10 - Information Retrieval