

Security and Reliability in Concordia™

Tom Walsh, Noemi Paciorek, David Wong
Mitsubishi Electric ITA
Horizon Systems Laboratory
300 Third Avenue
Waltham, MA 02154, USA
{walsh, noemi, wong}@meitca.com

Abstract

Concordia provides a robust and highly reliable framework for the development and execution of secure, mobile agent applications. Concordia incorporates many advanced security and reliability features beyond the basic functionality found in other mobile agent systems.

Concordia provides a rich security model that can be used to allow or deny access to system resources down to a very fine level of granularity and that protects agents and the information they carry from tampering or unauthorized access. The system utilizes transactional message queuing to provide reliable network transmissions. Further, Concordia uses proxy objects and a persistent object store to insulate applications from system or network failures. This paper discusses the design and implementation of these features.

1. Introduction

The popularity of the Internet and the World Wide Web has brought much attention to the world of distributed applications development. Now, more than ever, the network is being viewed as a platform for the development of cost-effective, mission-critical applications. While distributed applications represent a great potential future, today's reality of distributed applications development contains many pitfalls. The task of writing distributed applications which run over the Internet is wrought with problems of scalability, reliability, and security. Much interest exists in technologies that aid in the development of these applications.

Mobile agent technologies present an attractive option for distributed application development. The mobility of an agent hides the specifics of the network from the application developer and provides a familiar

development paradigm. Further, this mobility allows an agent to inter-operate with existing systems in a manner not possible with other technologies. Also, an agent's mobility and autonomy enable it to perform its task without regards to the quality or reliability of the underlying network, allowing it to provide a solution applicable to WAN or wireless environments.

The scale of applications now being considered for network environments will require a level of security and reliability previously only available in large transaction processing systems. In order for agent systems to present a realistic development alternative for such applications, these systems must evolve to provide the highly secure and reliable environments needed.

Concordia is a framework for developing and executing mobile agents. It has been designed to provide flexible agent mobility within a highly robust and secure environment. Concordia provides a rich security model that allows access to server resources to be controlled based on the identity of the user on whose behalf the agent is executing. Concordia protects agents from tampering when travelling on the network or when stored on disk. Concordia uses proxies and a transactional message queuing system to shield applications from the reliability problems inherent in a distributed environment.

The remainder of this paper discusses the general architecture of Concordia as well as the architecture of the specific components that provide security and reliability.

2. System Overview

The Concordia infrastructure toolkit consists of a set of Java class libraries for server execution, agent application development, and agent activation. Each

node in a Concordia system consists of a *Concordia Server* that executes on top of a Java virtual machine. The Concordia Server consists of a number of components as shown in Fig. 1.

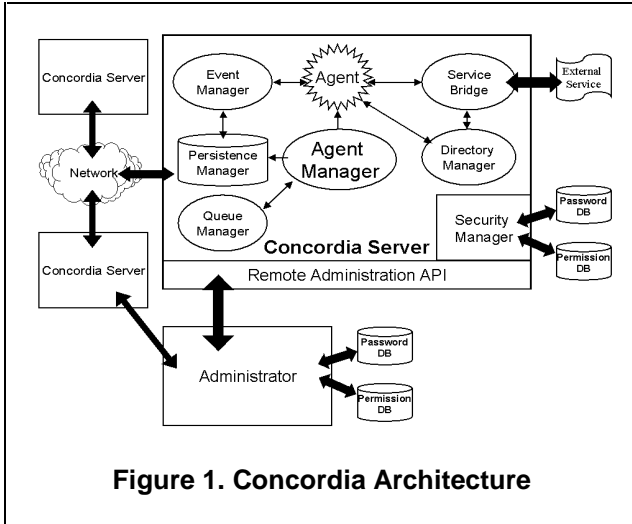


Figure 1. Concordia Architecture

Concordia, like a number of existing agent infrastructures and toolkits, supports the basic communication plumbing needed for agent mobility [19]. Within a Concordia Server, this propagation server is called the *Agent Manager* (formerly known as the Conduit Server in prior Concordia documents).

The Agent Manager provides the basic mobility support needed for agent transfer. It also provides the basic execution context for agents. 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. Details of this flexible scheme for agent mobility can be found in [19].

Concordia provides two forms of inter-agent communication: distributed events and agent collaboration. Distributed events are scheduled and managed by the *Event Manager*. Agent collaboration, which makes use of this event mechanism, allows agents to interact, modify external states (e.g., a database), as well as internal agent states. Details of these two forms of inter-agent communications can be found in [19].

Concordia system administration is handled by the *Administrator*. The Administrator, which resides on a separate Java virtual machine, starts up and shuts down

the components in a Concordia Server. It also manages changes in the security profile of both agents and servers in a Concordia system and makes requests to the Security Manager on behalf of the agent or server. The Administrator also monitors the progress of agents throughout the network and maintains agent and system statistics.

The Concordia *Service Bridge* component makes it possible for an application/agent developer to add services to a Concordia Server to support agents when they travel to the server. Service Bridges may be managed remotely using the remote administration API. One major use of Service Bridges is to provide controlled access to native system services.

The *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.

Concordia's security model provides support for three types of protection: (1) agent storage protection, (2) agent transmission protection, (3) server resource protection. Details of Concordia security features are discussed in Section 3.

Concordia agent mobility can extend to a number of local as well as wide area networks. To prevent potential performance and reliability problems associated with the transmission of agents across networks with different characteristics, Concordia also provides support for transactional queuing of agents between Agent Managers residing on different networks. The *Queue Manager* is discussed in more detail in Section 4.

Agent persistence is required to ensure that agents can recover successfully from system crashes. The *Persistent Store Manager* allows the internal state of agent objects to be persisted on disk. The Persistent Store Manager is discussed in more detail in Section 5.

Proxy objects increase the reliability of the Concordia system by shielding agents and other objects from the effects of server and system failures and this feature is discussed in Section 6.

3. Security

Security in mobile agent systems is generally accepted to encompass four distinct problems: (1) the secure network transfer of agents, (2) the protection of a host from attack or misuse by malicious agents, (3) the protection of an agent from attack by a another malicious agent, and (4) the protection of an agent from attack from a malicious host [3][6]. The Concordia security model provides the first three types of protection. There has been considerable debate as to whether the problem of protecting an agent from a malicious server is solvable [5]. The current Concordia security implementation does not support this type 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.

Within the Concordia system, the term *agent protection* is used to refer to the protection of agents from tampering during transmission or when stored in an on-disk representation. Protection of an agent during transmission is a generally recognized problem in agent systems. For added 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 (See Section 5 for details on Concordia persistence). 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.

3.1. Agent Protection

As previously mentioned, 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: (1) *transmission protection*, which deals with

the network protection of an agent and (2) *storage protection* which protects an agent when represented on disk.

3.1.1. Transmission Protection

Concordia provides transmission protection using Secure Sockets Layer version 3 (SSLv3). SSL is a general-purpose network security protocol, which can provide authentication and encryption services for TCP connections. We chose SSL as the basis for Concordia transmission protection for several reasons. First, SSL itself is emerging as a standard for secure communication over the Internet. Due to its popularity, several implementations are available, including pure Java implementations.

In addition, since SSL plugs into the application at the socket layer, it can conveniently sit underneath other protocols such as HTTP, RPC, IOP and RMI, in essence securing those protocols. The current implementation of Concordia makes use of Java RMI middleware [12] to provide network communications. RMI provides a Java specific distributed object layer on top of standard TCP/IP sockets. Providing for secure communications over RMI is simply a matter of replacing RMI's usage of standard sockets with secure sockets. SSL's ability to plug into our existing network infrastructure made it a very attractive solution.

Further, since SSL plugs in at the socket level, it doesn't require any special security knowledge or security related code at higher levels of the application, making it somewhat easier to implement than other solutions. Network security is encapsulated at a very low level and the details of that security remain hidden at that level.

SSL also allows for the mutual authentication of both sides of a network connection. This feature allows a server to protect itself from spoofing if an untrusted machine tries to masquerade itself as a Concordia server.

We investigated the possibility of ensuring transmission protection using a public-private key encryption scheme independent of the socket communications level. Similar mechanisms are used by the Agent/TCL and Itinerant Agents systems [2][6]. 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.

3.1.2. 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, the agent's bytecodes, internal state and travel status are all written to disk. This data is referred to collectively as the *agent's information*. 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 filesystem or on removable media which can be physically secured by the administrator.

For further protection, the bytecodes for uninstantiated agents can optionally be stored in signed Java Archive (JAR) files [8]. Concordia can then use the signature on the JAR file to guarantee that the agent was in fact written by a *trusted* author and has not been tampered with since it was released by that author. This is similar to more traditional code signing done with web applets.

3.2. Resource Protection

With the popularity of the World Wide Web, Java applets, and other forms of downloadable executable content, the need to protect machine resources from unauthorized, potentially harmful, access has become apparent. This problem is no less evident in the world of mobile agents where downloadable executable content is replaced by autonomous mobile executable content. Ensuring protection of sensitive machine resources is one of the most important features of Concordia security. Concordia's approach to resource protection builds upon existing Java security mechanisms but greatly extends them to provide a useful solution for mobile agent systems.

Within the world of Java, traditional approaches to resource protection have focused either on the *sandbox*

model or on the use of signed classes. Systems that make use of the sandbox approach classify Java objects into two categories: trusted and untrusted. Trusted objects are granted full access to machine resources while untrusted objects are given very limited access. In general, untrusted objects are not allowed to access the local file system and have very limited access to the network. Java makes the decision as to whether an object is trusted or untrusted by examining the way in which its class was loaded into the Java virtual machine. If a class is loaded from a local disk then it is assumed to be trusted, whereas a class loaded from a network source via a *ClassLoader* object [9], is assumed to be untrusted. Thus, classes that arrive from a network source receive very limited access to machine resources.

The sandbox approach, while providing reliable security, does have some drawbacks, which are especially evident in the mobile agent world. In general, the sandbox is too restrictive. The limits it places on Java classes makes it very difficult to write full-scale applications. Further, since any implementation of a mobile agent system involves the mobility of code around a network, virtually all agent and agent related classes executing on a particular machine are loaded from a network source and thus would be treated as untrusted. This fact would severely limit the usefulness of agents, since their access to machine resources would be so restricted.

The restrictions of the Java sandbox can be relaxed through the use of signed code. The Java Development Kit version 1.1 introduced code-signing support, which allows the author of a Java class to sign that class using a digital certificate. Users can then configure their systems to trust certain authors and can configure differing levels of trust for different authors. In essence, the user of the code trusts that the author of the code will not write malicious or damaging code or at least partially trusts the author not to do so.

While code signing can play a role in mobile agent systems, we feel it does not supply a suitable basis for the security of a mobile agent environment. In most cases, an owner of a machine resource wants to restrict access based on the identity of the user on whose behalf the agent is acting. For example, if you send a mobile agent to your desktop machine, you most likely would want to give it access to your personal files stored on that machine, whereas you would want to restrict access to agents sent by other users. The identity of the author of that code is not the deciding factor in the decision. Thus, in the Concordia philosophy, there is a level of trust between the system owner and the agent owner and not

just between the system owner and the agent author. This approach is clearly superior to the one adopted by other mobile agent systems such as IBM's Aglets and General Magic's Odyssey [1][11]. These systems rely solely on the standard Java security model, which does not provide the proper degree of flexibility and extensibility for mobile agent systems, as the basis for their security.

3.2.1. 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: (1) a user name, (2) a user group, and (3) 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 inclusion of user groups is primarily intended to ease administration, by allowing the security properties of multiple users to be administered simultaneously. In the current implementation, a user can only be a member of a single group. Allowing multiple group membership is being considered as a future enhancement.

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`.

As the agent travels around the network, it carries its identity with it. During an agent's travels, its identity is protected by the fact that the password is only stored in a hashed form and by the fact that during network travel or on disk storage, the entire agent is protected by Concordia's Agent Protection. Thus it would be very difficult for a third party to even obtain the hashed password moreover to discover the clear text password.

At each stop in the agent's travels, its identity is verified against a list of valid users of the system. Each server is configured with a list of users as well as corresponding resource access permissions allowed for the user. This approach is actually quite different than that adopted by the Agent/TCL system. Within Agent/TCL, an agent's identity is not re-verified at each stop in its travels [6]. Each server passes the agent's identity to the next server

in an unverifiable form and this form is accepted because of an implied trust between servers. We decided to perform verification at each stop because of the added level of security this introduced as well as the further scalability of naming that is possible. In a large intranet or particularly in a large extranet, it is not unimaginable that two users could have the same user name and group name. Due to the expected uniqueness of their password, Concordia would identify these two users as different.

The server stores its list of valid users in a file referred to as the *password file*. This password file can either be stored on a local disk of the server, in a location that can be accessed through standard protocols via a URL, or in a JDBC database. In a single server setup or an installation of a small number of servers, the password file most likely would be stored locally on each server. In larger installations, the password file would be stored and administered centrally and accessed remotely by the servers using either JDBC or URLs for access. The network accessibility of the password file was introduced primarily in the expectation of such central administration of large installations.

The password file contains the usernames and hashed password of all the users of the system. Validation of an incoming agent is performed by comparing the hashed password value travelling with the agent to the value stored in the password file.

To prevent unauthorized modification, the contents of the password file are hashed and the hash value is stored with the file. Any modification to the file requires recomputing the hash value for the modified file. If the server discovers that the hash value is incorrect, it considers the password file to be corrupt, an error is logged, and all security requests are rejected. For further security, the hash value can then be signed using the DSA digital signature algorithm and the server's private key. Signing only of the hash value is a performance enhancement used commonly in the signing of documents [13].

We are investigating the use of personal digital certificates as a future mechanism for detecting the identity of agents. In such a system, an agent could be reliably detected to have originated from a particular user because it would have been signed using that user's personal certificate. This is very much like the approach used by the Itinerant Agents system [2].

Another option for user identity would be to map an agent into the identity of an operating system account.

For example a particular agent would be mapped to a particular UNIX or Windows NT user name and then restricted according to the operating system's security policy. This approach is attractive because it does not require the agent system's administrators to maintain multiple databases of user account information. We did not adopt this approach for Concordia because current implementations of Java do not provide a platform independent way to access operating system account information. Platform independence was one of the goals of the Concordia system and we did not want to introduce OS specific code unless absolutely necessary.

Once the server has identified and validated the user, it examines a list of *resource permissions* to see what level of system access is allowed to that user. Resource permissions are analogous to access control lists (ACLs) supported by operating systems. They can be assigned either to individual users or to groups of users.

3.2.2. Resource Permissions

Resource permissions can be used to allow or deny access to machine resources at a very low level of granularity. For example, a resource can be constructed to allow read access to the filesystem of the machine. Another could be constructed to deny such access. A third could specify read access only to a particular file on the machine. There is roughly a one-to-one mapping between Concordia resource permissions and the standard security checks built into Java. For example, resource permissions can be used to control access to files or network resources, the ability to create new threads or processes, the ability to access or change the Java virtual machine's operating properties, the ability to load non-Java code, and the ability to access to the system's console or graphical user interface. Concordia's resource permission mechanism is built on top of the standard Java security classes.

The standard Java security model makes use of a class called a `SecurityManager` [9]. The `SecurityManager` class contains methods, which are invoked internally by the Java runtime libraries whenever a potentially dangerous call has been made into the library. For example, if a Java class attempts to read from a file on the local filesystem, the *checkRead* method of the `SecurityManager` is called. The standard `SecurityManager` class attempts to determine if the class making the call is trusted or untrusted and then permits or denies access based on that fact.

Concordia security provides a subclass of the Java `SecurityManager` class. When a call is made into the

Concordia Security Manager, it first determines whether the call was made by a class that originated from a network source or by a class loaded from the local machine. If a local class (also referred to as a *native class*) made the call, the Security Manager allows the access to be made. Local classes are always considered trusted and have full access to machine resources.

If the Security Manager determines that the call was made by network code, it then determines if an agent made the call. If this is the case, the manager retrieves that user identity associated with the agent from the agent's execution context. Then the manager validates the identity against the password file and then checks the resource permissions of the user to determine if it should allow the access.

If the Security Manager determines that the class making the call was loaded from a network source but is not an agent, the call is considered to have been made by an untrusted source and is only allowed to execute in a sandbox. Further, if the Security Manager determines an agent made the call, but is unable to verify the identity of the agent, the call also executes in the sandbox. The Concordia security system defines a special user identity known as *untrusted* which is used to control resource access given to these types of unknown or unverifiable agents. The resource permissions of the untrusted identity can be configured just like any other identity. Since the untrusted account defines Concordia's sandbox, this sandbox is configurable to any level of access desired.

The resource permissions for a server are stored in a file called the *permissions file*. The permissions file is hashed and signed in the same manner as the password file and also can be accessed either from the local filesystem, an URL or a JDBC database.

As mentioned before, Concordia's resource permissions build upon the standard Java `SecurityManager` class. The `SecurityManager` provides method callbacks for such things as file access, network access, and thread and process management. Concordia provides resource permissions for all of the resources protected by the standard Java `SecurityManager`. Concordia also provides resource permissions for higher-level procedures such as starting, suspending or stopping a server or an agent. These high level procedures are generally accessed remotely using the Concordia Administrator, a GUI administration tool. A request to administer a server is validated and regulated in the same way as a resource request by an agent.

Through the application of resource permissions, it is possible to protect an agent from attack by other agents. Since any request to administer an agent requires an authorized user identification, it is not possible for an unauthorized agent to stop, suspend, or alter the travel plans of another agent. Concordia provides no facilities that allow agents to obtain direct references to other agents nor make direct method calls on each other as is seen in Telescript meetings [17]. Agents usually communicate via messages or Concordia's group-oriented communications mechanisms. However, if an agent were to retrieve a reference to another agent through other means, it would be allowed to make public method calls on the other agent, but would not be allowed to interfere with its travels. When agents arrive from the network, their bytecodes are verified using the standard Java bytecode verifier. This guarantees that an agent cannot make an illegal call into the non-public interface of another agent. Thus, the only method of entry into an agent is its public interface. It is the responsibility of the agent developer to guarantee that the public interface is secure. For example, the public interface should not publish confidential information.

Concordia also takes steps to guarantee the integrity of the system. In the next release, Concordia's bytecodes will be shipped in a signed JAR file. During the startup procedure, Concordia will inspect the JAR file to guarantee that it has not been altered in any way. While this provides additional security, it does not completely secure an agent against the possibility of harm from a malicious server. Any code loaded from the local disk of a server will be considered trusted and will be given full access to the system. Further, the bytecodes of locally loaded classes are not verified on loading for performance reasons. Finally, given that a server manages the marshalling and unmarshalling of agents from a network, handles their writing to and reading from persistent storage, and controls their thread of execution, a high level of trust in the server is needed. In general, this is not a problem since the user decides where an agent is to travel, and can send highly sensitive agents only through trustworthy machines.

4. Queuing

The Concordia infrastructure provides support for reliable transmission of agents across the network via use of an underlying message queuing subsystem. Concordia's queuing support allows the Agent Manager to submit an agent enqueue request to the Queue Manager in an asynchronous manner. The agent message queue serves as a transmission buffer for the

request in this manner. This feature of the message queuing subsystem is a natural fit to the disconnected operational mode of the mobile agent paradigm because it provides a "store and forward" mechanism. Agents can be stored on the message queue of a local server while a remote server is undergoing repair or is simply moved to a different physical location. When the remote server comes back online, the local server would then forward the agent to the returned server.

A message queuing subsystem provides additional reliability by maintaining a copy of the agent to be transmitted in an on-disk queue until the recipient of this agent transmission has acknowledged its receipt via the well-known handshaking protocol called the *Two Phase Commit* protocol. This is known in the industry as *transactional message queuing*. The Queue Manager communicates with its local Agent Manager and performs handshaking with other remote Queue Managers for reliable agent transmission.

The Queue Manager manages both an inbound and an outbound queue. The Agent Manager submits an agent enqueue request for the local Queue Manager's outbound queue. The local Queue Manager then proceeds to dequeue agents off its outbound queue and submits an enqueue request with a remote Queue Manager's inbound queue. The remote Queue Manager then proceeds to dequeue agents off its inbound queue and transmits the agent object to its local Agent Manager.

The Queue Manager is implemented as a multi-threaded Java thread object to allow for concurrent access to the underlying queue storage. The preservation of an object's class specification on disk is handled by the Java object serialization facilities while Queue Manager communication relies on the Java RMI package [12]. Reliable queuing of agent transmission appears to be an important feature lacking in other commercial mobile agent system offerings.

The Queue Manager's design goals included achieving: optimal disk space utilization, fast write operations, fast recovery from server failure, and reliable management of pending agent transmissions. Its on-disk queue storage architecture implementation borrows some ideas from the log-structured file systems research area [14][15] to employ a unique data architecture that ensures better overall performance over traditional message queuing system architectures [4][10].

Traditional message queuing architectures are generally not optimized for write operations without requiring

extra hardware to work efficiently (e.g., utilization of low level RAID devices to cluster data blocks). Such systems appear to require special performance optimization in both hardware and software in order to handle workloads with high throughput and low message residence time, important characteristics of mobile agent systems.

Furthermore, traditional message queuing architectures employ separate queue data and log files, which introduce an extra level of *unreliability* since there are two points of potential file corruption and media failure. There is also usually no means for the message queuing systems administrator to predefine the amount of work needed to do recovery a priori. The utilization of automatic system checkpointing by the Queue Manager handles this problem efficiently.

In Concordia, the data architecture of the on-disk queue storage design consists of a combined on-disk file structure for the message queue data and log records. This architecture utilizes a circular queue and consists of a single flat file that is created when a Queue Manager is first initialized. Each entry in the queue data file contains the agent object in contiguous blocks on the disk. The queue data file is partitioned into a predefined number of logical segments. Each segment contains a predefined number of control blocks at the beginning of each segment. These control blocks contain control information for the queue entries and log record information. Queued agent objects and their log records are stored after the control blocks with potential mixing of agent objects and log records. When a new segment is reached in the queue data file, a new set of control blocks is written to disk at the beginning of the new segment. In this manner, the logical segments serve as forced *checkpoint* intervals on the state of the entire queuing system and the data that is stored in the queue file.

The Queue Manager design adheres to the Atomicity, Consistency, and Isolation properties of the well-known ACID properties. The Durability property to guard against hardware failures can be achieved by duplicating the combined queue data/log file on a separate disk. We can utilize existing RAID technology to do duplicate writes transparently. We made this architectural design decision in order to provide flexibility to Concordia application developers with respect to the possible tradeoffs between cost and levels of reliability required. For example, for deployment on mobile hand-held devices, one may wish to limit the hardware requirements on such systems at the expense of lack of media durability.

5. Persistence

Concordia's infrastructure incorporates highly-reliable servers that save their internal state to persistent storage and, during failure recovery, retrieve and reconstruct any data required to continue servicing their clients. Concordia also transparently saves agents' states to persistent storage, thereby enabling agents to recover from system and server failures and to continue execution unaffected. Applications and agents running on Concordia systems may also utilize persistent storage to checkpoint themselves and, after failure, restart from their latest checkpoint.

The *Persistent Store Manager* (PSM) is a general-purpose facility utilized by Concordia's servers to save their state and also that of mobile agents. Typically, each server or application owns an instance of the PSM and each instance utilizes a different file for persistent storage. The PSM exports methods to create, delete, update, and fetch objects from persistent storage.

The PSM manages saved objects by utilizing a random access facility built upon Java's object serialization package. The object serialization code ensures that when an object is stored, all objects reachable from it (i.e., all objects it refers to and all objects its references refer to, etc.) are also stored. Similarly, when an object is retrieved, all the objects reachable from it must also be retrieved.

The Agent Manager on each host is responsible for sending agents to remote hosts, receiving them from remote nodes, and providing an execution environment for agents. When the Agent Manager receives an agent, it writes its state to persistent storage before creating its main thread of execution. (For more details on the Agent Manager's operation, see [19].) After the agent finishes executing, the Agent Manager updates the agent's state in persistent storage before transmitting it to the Agent Manager at the agent's next destination. The agent remains in persistent storage until it has been successfully transferred to the next Agent Manager. When an agent finishes executing at its final destination, the Agent Manager deletes it from persistent storage.

After a system or server failure, the Agent Manager retrieves each agent's state from persistent storage and restarts it. A restarted agent may potentially repeat work it has already completed. Therefore this persistence scheme only guarantees correctness for agents performing idempotent operations. This criterion is sufficient for many mobile agent applications, which can often be classified as information retrieval and filtering.

However, the need for idempotency can be eliminated if agents utilize the PSM to checkpoint their internal state and restart their execution from their last checkpoint.

If the Agent Manager's recovery process fetches an agent that has already finished executing, it transfers the agent to its next destination (if one exists) and deletes it from the persistent store once the transfer is complete. This algorithm may result in duplicate agent transfers if a system or server failure occurs during or immediately after an agent transfer. This problem is detected and handled by the Queue Manager, which ensures that agent transfers complete reliably and that each time an agent travels to a new host, the Agent Manager receives exactly one copy of that agent. (See Section 4 for details on the Queue Manager.)

Many of Concordia's servers, in addition to the Agent Manager, employ the PSM. Both the Event Manager and Directory Manager save their registrations in persistent storage. Objects (e.g., agents) that receive distributed events, register their interest in specific events with the Event Manager. Each time the Event Manager receives a registration, it caches it and writes its updated registration information to persistent storage. If the Event Manager is restarted, its initialization process retrieves the registrations from its persistent store file. While the Event Manager is unavailable, some of its registrants may terminate. If so, the Event Manager will be unable to notify the defunct registrants of new events and will delete their registrations.

The Directory Manager utilizes persistent storage in a similar manner. As described in Section 2, Application Servers may export their services to mobile agents by registering with the Directory Manager. The Directory Manager caches registration information, saves it in persistent storage, and retrieves it when it restarts.

System administrators may optionally enable persistence for the Agent Manager, Event Manager, and Directory Manager. The Concordia approach is flexible – it allows administrators and developers to weigh the costs of persistence against the reliability requirements of the applications. This is in contrast to the Telescript paradigm, in which all objects are persistent [7]. Because the Telescript engine provides a garbage-collected persistent object store, it incurs a large cost in terms of resource utilization and performance¹. Other recent mobile agent systems [18] do not provide a high

¹ General Magic's Telescript User's Guide recommends that systems running the Telescript engine have 96MB of memory.

degree of reliability, and therefore do not include support for persistence.

6. Proxies

Proxies increase the reliability of the Concordia system by shielding agents and other objects from the effects of server and system failures [16]. Server proxies transparently attempt to re-establish connections when they are unable to communicate with their counterparts. Concordia provides proxies for servers that support potentially long-lived connections (currently only the Event Manager).

A distributed system with multiple nodes executing Concordia servers may be configured with one or more Event Managers. Agents may choose to maintain a reliable connection to an Event Manager by communicating via a proxy instead of directly with the server. (In applications that are more concerned about performance than reliability, agents may interact directly with the Event Manager.) An Event Manager Proxy is essentially an agent's local representative for a potentially remote Event Manager. An agent may communicate with the Event Manager on its local host, on the node where it was launched, or on any other host. Hence, as an agent travels, it often interacts with a remote Event Manager.

Each Event Manager registers itself with Java's RMI Registry and, potentially, the Directory Manager. The Event Manager Proxy is responsible for obtaining a reference to the Event Manager, via the Registry, and establishing a connection to it. If communication fails, the proxy also re-establishes a connection to the Event Manager. (Concordia's Administration Manager is responsible for restarting the Event Manager.)

If a mobile agent attempts to interact with a failed Event Manager, its proxy will be unable to communicate with the defunct server. After retrying its operation, the proxy requests a new reference to the Event Manager from the Registry. If the Event Manager has been restarted by this time, the proxy obtains a reference to the new instance; otherwise, the proxy throws an exception. This combination of persistent Event Manager state and server proxies comprises a highly reliable form of communication between mobile agents and a potentially remote Event Manager.

7. Conclusion

Concordia provides an environment for the development of highly secure and highly robust mobile agent

applications. The infrastructure extends the standard security mechanisms of the Java language to provide an identity-based system where the rights 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.

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. As described in this paper, Concordia provides many features vital to the development of complex distributed applications.

Information on obtaining Concordia is available at the Mitsubishi Electric ITA Web site (URL=<http://www.meitca.com/HSL/Projects/Concordia>). Future extensions to the existing functionality may include support for transactional multi-agent applications and knowledge discovery for collaborating agents.

8. Acknowledgments

The authors wish to thank the members of the Concordia team for their contributions to this paper. In particular we would like to thank Joe DiCeglie for his input into the section on Concordia security.

9. References

- [1] *Aglets: Mobile Java Agents*, IBM Tokyo Research Lab, URL=<http://www.trl.ibm.co.jp/aglets>
- [2] D. Chess, B. Grosz, C. Harrison, D. Levine, C. Parris, "Itinerant Agents for Mobile Computing", *IEEE Personal Communications Magazine*, 2(5), October 1995.
- [3] M. Condit, D. Milojicic, F. Reynolds, D. Bolinger, "Towards a World-Wide Civilization of Objects", In *Seventh ACM SIGOPS European Workshop*, 1997.
- [4] *Encina RQS Programmer's Guide*, Transarc Corporation, Pittsburgh, Pennsylvania, 1994.
- [5] W. M. Farmer, J. D. Guttman, and V. Swarup, "Security for Mobile Agents: Issues and Requirements", In *19th National Information Systems Security Conference (NISSC 96)*, 1996.
- [6] R. S. Gray, "Agent Tcl: A flexible and secure mobile-agent system", In *Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL 96)*, Monterey, California, July 1996.
- [7] "An Introduction to Safety and Security in Telescript", General Magic White Paper, 1996.
- [8] "JAR - Java Archive", Javasoft Corporation, URL=<http://www.javasoft.com/products/jdk/1.1/docs/guide/jar/index.html>
- [9] "Java™ Platform 1.1.4 Core API", Javasoft Corporation, URL=<http://www.javasoft.com/products/jdk/1.1/docs/api/packages.html>
- [10] *MQSeries: Message Queuing Interface Technical Reference*, IBM Corporation, Armonk, New York, 1994.
- [11] *Odyssey*, General Magic, URL=<http://www.genmagic.com/agents/odyssey.html>
- [12] "Remote Method Invocation for Java", Javasoft Corporation, URL=<http://chatsubo.javasoft.com/current/rmi/index.html>
- [13] B Schneier, *Applied Cryptography*, p. 38, John Wiley & Sons, Inc., New York, NY, 1994.
- [14] M. Seltzer, "Transaction Support in a Log-Structured File System", In *Proceedings of the Ninth International Conference on Data Engineering*, February, 1993.
- [15] M. Seltzer, K. Bostic, M. McKusick, C. Staelin, "A Log-Structured File System for UNIX", In *Proceedings of the 1993 Winter Usenix Conference*.
- [16] M. Shapiro, "Structure and Encapsulation in Distributed Systems: The Proxy Principle", *Proceedings of the 6th International Conference on Distributed Systems (ICDCS)*, IEEE Computer Society, 1986, pp. 198-205.
- [17] J. E. White, "Telescript Technology: Mobile Agents", General Magic White Paper, 1996.
- [18] J. E. White, "Mobile Agents – A Presentation by Jim White", General Magic, 1997.
- [19] D. Wong, N. Paciorek, T. Walsh, J. DiCeglie, 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.