CORMAT

A Component Oriented Mobile Agent Toolkit

Author: Robert Young, 100292143
Supervisor: Dr. Anthony White
Department of Computer Science, Carleton University
April 11th, 2005
Abstract

This report gives an overview of the Component Oriented Mobile Agent Toolkit (CORMAT) mobile agent system (MAS). CORMAT is designed to run on java based OSGi platforms. The main goals of the project were to investigate agent technologies, service based designs, and the OSGi framework. The project resulted in the development of a number of components that formed a runtime configurable mobile agent system. The system is intended to be used in a small to medium scale setting. Agent security was also addressed through the development of a secure agent framework and pluggable security mechanism.
Acknowledgements

I would like to thank Dr. Tony White for his help, guidance and suggestions during this project. I would like to acknowledge Anne, Greg, John, and Donald Young, whom without I would not be in the field of computer science today.
# Table of Contents

Table of Figures .................................................................................................................. 1  
Introduction ......................................................................................................................... 2  
  Background ..................................................................................................................... 3  
  Agents ............................................................................................................................ 3  
  OSGi Platform .............................................................................................................. 5  
Motivation ............................................................................................................................. 7  
Project Description .............................................................................................................. 8  
  System Components ...................................................................................................... 8  
  System Architecture .................................................................................................... 9  
Mobility ............................................................................................................................... 11  
System Components .......................................................................................................... 14  
  Agent Class, Agents, Agent Services ........................................................................... 14  
  Agent Communication Services ................................................................................... 15  
  Agent Location and Service Discovery Facilities ....................................................... 16  
Security ............................................................................................................................... 19  
  Agent Isolation ............................................................................................................ 19  
    OSGi Isolation ......................................................................................................... 21  
    Reflection Based Attacks ......................................................................................... 22  
    Bundle Based Attacks ............................................................................................. 22  
    CORMAT Permissions ............................................................................................ 23  
  Security Manager Service ............................................................................................. 23  
    Security Manager Implementation & Behaviour .................................................... 28  
    Environment to Environment Security .................................................................... 28  
Demonstration .................................................................................................................... 29  
Conclusion ........................................................................................................................ 32  
Appendix A: Glossary ....................................................................................................... 33  
Appendix B: System Configuration Parameters ............................................................. 34  
Appendix C: References ................................................................................................... 35  
Appendix D: OSGi Programming Tutorials ....................................................................... 37  
Appendix E: OSGi Platforms ............................................................................................. 38  
Appendix F: Classpath loading issues ............................................................................. 39  
Appendix G: Sample Component Manifest files .............................................................. 40  
Appendix H: System Component Listing ......................................................................... 41
## Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bundle State Diagram [OSGi R3 Specification pg 58]</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>CORMAT Component Diagram</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>Active agent state</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>Agent Class Interfaces</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>AgentEnvironmentContext interface</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>Overview of the CORMAT Security Interfaces</td>
<td>24</td>
</tr>
<tr>
<td>7</td>
<td>Agent Thread Group structure</td>
<td>26</td>
</tr>
<tr>
<td>8</td>
<td>Manager Agent GUI</td>
<td>30</td>
</tr>
<tr>
<td>9</td>
<td>Logger Agent GUI</td>
<td>31</td>
</tr>
</tbody>
</table>
Introduction

Most software systems deal with many different components, each with specific roles and responsibilities. The deployment of these components may be set at compile time or dynamically assembled at runtime. As component based systems grow in size and complexity, ensuring consistency and proper coordination between components becomes difficult to achieve. Communication between components is often component specific, leading to difficulty in interchange of components. This binding of communication between components also leads to increased code maintenance and poor maintainability: refactoring of communication often breaks compatibility between components until all of the components involved in the communication are updated.

Agent oriented systems have many features for overcoming these problems. Communication between agents is performed using standard message protocols called Agent Communication Languages (ACL). This standardized ACL based communication ensures that communication can occur between any two agents in an agent system. Agents are dynamic: they often perform communication asynchronously, and are loosely coupled. Agents often prove to perform well in fault tolerant and disconnected communication environments. An agent environment is a software environment in which agents can be created, executed, and managed. The agent environment manages all aspects of agent lifecycle, interaction, and security.

Service Oriented Architecture (SOA) is another approach for dealing with complexity and binding between components in software systems. SOA attempts to decompose a system into services which can be interchanged. The exact use of services is often not set at design time, allowing them to be reused in a generic manner.

This paper details the design and implementation of a Component Based Mobile Agent Toolkit (CORMAT) using the OSGi component platform. This system deals with issues of agents, mobility, and service oriented architectures. CORMAT is a toolkit for developing on OSGi platforms.
Background

Agents

An agent is an autonomous process that acts on its perceptions of its environment, based on the goals and beliefs of the agent. There are a number of proposed definitions to the term software agent, but no one definition has been accepted as a standard definition. Agents are generally structured similarly to objects, in that they are comprised of state and behaviour. Agent behaviour is often expressed in terms of a general purpose language as opposed to through statically defined method calls. Similarly to objects, agents have the properties of encapsulation and inheritance. Comparisons can also be made between software components and software agents: agents are often thought of as reusable units of deployment, similarly to components. Furthermore, agents, like components, are usually deployed into a middleware environment within which it functions. The middleware that the agent(s) operate within is referred to as the agent environment.

Most definitions of software agents agree that agents have the following properties:

- Autonomy: agents are often implemented as active objects. An agent is run in a separate process or thread of execution. Agents have control over their internal state, and on how it handles requests from other agents.
- Goals: an agent is often programmed with goals that it will attempt to accomplish, leveraging the use of its environment if possible. This is also referred to as the agent's desires.
- Beliefs: agents have a view of the agent environment based on their perceptions of the environment. These beliefs are subject to change, as the agents only have a partial view of the system.
- Intentions: based on the agents goals and beliefs an Agent will often formulate plans to complete its goals in the form of steps it intends to take to accomplish its goals.
- Mobility: some agent systems allow for agents to move from one
  agent environment to another in order to accomplish goals.
- Intelligence: some definitions of software agents insist that agents
  be intelligent, and be able to perform intelligent reasoning.

Communication between agents is often accomplished using a high level
agent communication language (ACL) such as the Knowledge Query and Markup
Language (KQML) or through Smalltalk style dynamic method invocation. The
recipient agent can handle the message in any means it feels fit, including
disregarding it completely.

KQML is a general purpose ACL that was developed to allow agents to
communicate with one another and to collaborate on tasks. Each message in
the language is given a performative, which is essentially a type for that
message. Some performatives include ‘inform’, ‘request’, ‘error’, and ‘decline’.
One of the criticisms of KQML was that different systems implemented slightly
different sets of performatives.

Agent systems grew in popularity in the 90s, along with interest in mobile
code and artificial intelligence technology. A number of standards and agent
systems were developed during this time. The most active agent standards and
technology group to come out of this period of time is the Foundation for
Intelligent Physical Agents (FIPA). FIPA was created in 1996 with the intent of
creating standards and technologies for agent systems. A number of standards
have been developed by FIPA, including the FIPA-ACL. FIPA-ACL is a
standardized agent communication language similar to KQML. FIPA-ACL was
developed with a much stricter specification than KQML, whose informal
specification had lead to agent systems which all spoke KQML, but could not
speak to each other because of differences in dialects of KQML.

During this time the Mobile Agent System Interoperability Facility (MASIF)
standard was developed by the Object Management Group (OMG). The MASIF
specification defined a standard CORBA interface for agent system to
communicate together. MASIF included migrating agents from one system to
another, agent naming services, and naming services for agent environments. It
did not include details on how to pass agent messages from one framework to another.

During my research for the project I investigated three existing java based mobile agent systems: Aglets, FIPA-OS and the Java Agent Development Framework (JADE). Aglets was one of the first java based MAS to be developed: it was developed by IBM, but has since been released as an open source project. FIPA-OS was developed by Bell Northern Research, and is a showcase for a FIPA compliant mobile agent system. FIPA-OS is a component based system, with a number of closed and open source components: the core of the system is now open source. JADE is another open source FIPA compliant MAS.

**OSGi Platform**

The Open Service Gateway Service Initiative (OSGi) service platform is a standardized java component framework developed by the OSGi consortium. Bundles of code are the base components of an OSGi platform deployment. Each bundle can query the platform for services and register services to the framework. Lifecycle management of bundles is heavily defined and specified in the OSGi platform. Services and bundles developed for OSGi platforms have nondeterministic, dynamic behaviour. Bundles can be upgraded, installed, stopped, started and uninstalled at runtime. Code executing within a bundle can register and deregister services at runtime via a handle to the framework they are given when started. Bundles have six states that can be seen from the state chart below.
Services are defined through interfaces\(^1\) and registered with the OSGi Service Registry. Each service can have properties associated with it, which allows for filtering of services. Bundles can query the Service Registry for services, and attach listeners to react to changes to the state of services such as the registration of new services, or deregistration of a registered service.

Services provide strict boundaries between active bundles in the system. OSGi platform strictly separates and hides packages defined in one bundle from being visible in another. Bundles must explicitly declare which packages to allow other bundles to access: likewise, a bundle must explicitly declare the packages it wishes to import. This allows for bundles to declare interfaces for services in packages and keep the implementation strictly separated and inaccessible to any bundles using the service.

There are a number of available OSGi platform implementations: a list of implementations encountered during the project can be found in Appendix F. The Oscar and Knopflerfish open source OSGi platforms were used as deployment platforms for the agent environment bundles developed. Oscar was

---

\(^1\) Services may also be defined using classes. The preferred method is to define services as interfaces, so implementation details are left to classes that implement these interfaces.
chosen for primary development due to its clean design and low overhead compared to the desktop oriented Knopflerfish.

Interestingly, behaviour differences between Oscar and Knopflerfish were encountered during the project and are discussed in Appendix G.

Motivation

CORMAT is a java based mobile agent system that has been developed to run on an OSGi platform. The main objects of the project were to develop a mobile agent system that leveraged the capabilities of the OSGi platform when possible. Primary research interests were those of agent technology, mobile code and service based architectures with a focus on the OSGi platform. Through the project I wanted to gain experience in the design of service oriented systems, and concepts related to agent oriented systems. The primary aspects of the system were to accomplish code mobility, SOA, and agent security. Agent intelligence was not part of the scope of the project.

Agent mobility was to be accomplished by leveraging the mobility facilities built into the OSGi platform. A problem with many existing building mobile agent systems in java has been how to distribute the class files to remote hosts in a safe, reliable manner. The OSGi platform provides mechanisms for loading code into a running JVM: a goal of CORMAT was to take advantage of this functionality for agent mobility.

The design of systems using Service Oriented Architectures was to be investigated during the design of the project.
Project Description

CORMAT is a mobile agent system built to run within an OGSi platform or as a standalone MAS with an embedded OSGi component for service management. CORMAT is intended for small to medium scale agent based systems. Large scale agent system issues are not handled by CORMAT. Specifically, issues such as agent principles, host to host security, signed agents, and principals\(^2\) associated with agents are not addressed by the system. These features could be incorporated into the system at a later date. A security assertion language such as the Security Assertion Markup Language (SAML) could be used for agent environments to communicate assertions and requests for agents and services. Principles could be associated to agents, leveraging tools such as the JAAS.

The system is an open agent system: agents are not the only entities that reside within the system. Other components can (and most like will) run on the same OSGi platform that CORMAT will be installed to. These components may interact with CORMAT: the system components were developed to address this wherever possible.

System Components

Services were key in the development of the system. Services were used to allow for loose binding of ACL message types, the transport mechanism for messages from environment to environment, agent classes, and the security manager. Many agent systems are bound to one design, such as choice of ACL, or transport protocol to use for message and agent transportation over a network. CORMAT was developed to support many different agent communication languages and transport protocols through the use of services. The system was developed to work with five types of components: agent environment, agent class, agent communication protocol, agent transport protocol driver and security

\(^2\) “A principal is an entity whose identity can be authenticated by any system that the principal may try to access. A principal may be an individual, an organization, or a corporation.” [Lange and Oshima. Pg 19] Associating a principle to an agent is important in large systems, to ensure that the legal and moral responsibility of actions taken by an agent can be traced to an individual, organization or corporation.
manager. An agent environment control component was also developed for performing testing on the environment.

Figure 2 CORMAT Component Diagram

During the course of the project implementations of each type of component were developed. Below is an overview of the system architecture, behaviour, and details on the specification and implementation of the system components.

**System Architecture**

The CORMAT toolkit is focused around the maintenance and safe execution of agents within the agent environment. Each agent is run within a separate thread within the system. All agent threads are associated with a thread group. This provides the security system with a way to determine whether a security check is associated with an agent in the system. The components of the system make heavy use of the whiteboard service model described in the Knopflerfish OSGi Service tutorial.

The agent environment defines the core of the system. The agent environment listens for relevant services to be registered with the OSGi platform: when such services are registered, the agent environment will incorporate them into it’s framework, changing it’s runtime behaviour. The system behaviour,
communication and security depend on what ACL driver, transport protocol
driver, agent class and security manager services are registered with the system.

Agents are created using an AgentClass object. The AgentClass object is
a factory for creating agent objects. Every implementation of agents for a
CORMAT environment includes the creation of an AgentClass object. The
AgentClass class is responsible for creating agent objects, and decoding agent
objects from byte arrays. Each AgentClass has a string identifier associated with
it that is used to differentiate it from other AgentClass objects.

Agents are developed as normal OSGi bundles. To register an agent with
the system, the bundle is loaded into the OSGi platform and started. The
activation of the bundle will result in the agent class for the bundle to be
registered with the OSGi platform. The registered service reference is then used
by the agent environment (or agents running in the environment) to create agents
from the registered agent class object.

Agent behaviour is implemented as Behaviour objects. This Behaviour
object is very similar to the Behaviour objects used to describe agent behaviour
in the JADE agent system. The behaviour interface has two methods: action and
done. The agent environment continually executes the action method of the
agent’s behaviour until done indicates the agents behaviour has reached
completion. Mobility events occur between calls to action, ensuring the state of
the agent is consistent. A context object for the agent environment is passed to
the action and behaviour method each call: this context is the agent’s only way of
interacting with the agent environment. The context object allow the agent to
perform mobility actions on the environment such as creating new agents,
cloning agents, pushing agents to a remote agent host, pulling them from a
remote host and destroying agents. The context is also allows agents to retrieve
ACL drivers from the environment, and query/request security permissions from
the environment.
Figure 3 Active agent state

**Mobility**

Mobility adds many useful features to an agent environment. Mobile agents allow for an agent to transfer its code and execution state to a remote agent environment during the lifetime of its execution. This has many advantages when developing distributed systems, which will be discussed later in the document. In the case of CORMAT, the state of the agent refers to the local fields associated with the instances its agent and behaviour objects. Code refers to the byte code that is associated with the agent being transferred.

Agent mobility provides many benefits to the design of distributed systems. By transferring their state and execution from one environment to another, agents are able to deal with such issues as load balancing, network delay, and fault tolerance. The following is a list of several benefits gained from agents mobility has been compiled from [Wooldridge pg 236-239], [Carzaniga, Picco and Vigna pg 27-28] and [Lange and Oshima pg 3-5].
1. **Reduction of network load.** Components within distributed systems may enter states where high volumes of data are required to pass between the components. If the components are on different machines, the communication between the components may place a significant load on the system's network. This can be avoided by transferring the remote component (agent) so that both components reside on one machine where communication can take place with significantly less overhead.

2. **Overcome network latency.** It may not be feasible to reliably send messages to distributed components in a timely fashion. By sending agents to perform this messaging on a local host, network latency delays can be avoided.

3. **Encapsulate protocols and code.** Agents can be used to encapsulate code and protocols in use within a system. This simplifies the life cycle of code and protocols existing within the system. When system code or protocols are changed, the existing agents can be recalled from the system to be replaced by new agents encapsulating the new code and protocols.

4. **Asynchronous and autonomous execution.** Once created, a mobile agent is independent of its creating process and execute asynchronously. This promotes asynchronous communication and coordination between agents.

5. **Robust and fault tolerant.** Agents can often react dynamically to adverse situations in their environment, allowing them to better handle adverse faults and adverse situations. For example, an agent could move another agent environment upon receiving evidence that its current environment is shutting down.

Representing the agent’s code is done in one of three ways: as object code, source code or as a platform independent byte code. The code is
packaged by the sender and either sent automatically by the sender, or when requested by the receiver.

The entire execution state of the agent (stack frame, object instances,) can be transferred to the remote host and the agent restarted with the same frame of execution. This is referred to as strong mobility. The Telescript agent language is an example of an agent programming language that supports strong mobility. Weak mobility refers to the transmitting of state and code that is used on the receiving host to create a new process that mimics the original process. CORMAT implements weak mobility, as it is not feasible to capture the execution state of java threads using current java virtual machines.

**Mobility and CORMAT**

Mobility is a key part of the CORMAT system. Agents running within a CORMAT system are able to perform the following basic mobility actions characteristic of mobile agent systems: cloning, pushing, pulling, creation and destruction of agents. All of these actions have been designed in a location transparent manner. Agent state is captured using the java object serialization API. Agent state is transferred from host to host as binary data.

Thread safety was ensured for all of the mobility events in the system through the careful use of java synchronization. Each agent is run in a separate thread, with each action method call synchronizing the system’s access to the thread. Mobility events can only occur on the agent when the agent’s synchronization lock is not held. This makes a large assumption that the agent will periodically return from call to action. If this never occurs, the lock for the agent can never be obtained, and other agents and components of the system will be in a state of deadlock, waiting for the agent. A reimplementation that of the mobility synchronization that failed if it could not obtain the lock after a certain period of time would be an excellent way to overcome this problem.

The system maintains a count of hops and clones that has been performed on an agent. Currently agents have a closing event they can act upon: mobility events were not completed, and would be an excellent extension for the toolkit.
System Components

Agent Class, Agents, Agent Services

A key part of this project was creating agents in such a way that the OSGi platforms support for code lifecycle management could be leveraged in the delivery and mobility of agents within CORMAT. The OSGi platform specifies a Bundle as the unit of code that can be managed by the framework. These bundles can be dynamically loaded and unloaded from a system using the OSGi facilities. Based on this, it was decided that bundles would be used to enclose the code of an agent or classes of agents.

The agent environment was required to be able to create agents by identifier: the ability to pass data to the agents during creation was also required. A factory [Gamma, Helm, Johnson and Vlissides (1995)] was required so that agent objects could be created and handled by the agent environment. The AgentClass interface was developed to meet these requirements: AgentClass objects are responsible for creating new agents and reconstituting agent objects from byte arrays. Each AgentClass class has a string identifier that is used to uniquely identify a given class of agent.

![Agent Class Interfaces](image)

Each AgentClass bundle is stored as a jar file that stores the code that represents the agent, as well as any libraries or native code the agent requires.
An activator object needs to be defined for each AgentClass bundle: this activator handles the registration of an AgentClass object with the OSGi framework. This registration is done when the bundle is started, and automatically deregistered when the bundle is stopped or uninstalled. The Activator rjyoung.carleton.honors.util.AgentClassActivator was developed as a generic Activator for CORMAT agent class bundles. When an agent class service is deregistered from the OSGi platform, the agent environment will begin the shutdown sequence of every agent that was created using that service. Under normal operation agent class services are deregistered when the bundle that registered the agent class leaves the ACTIVE state (ie, is stopped, uninstalled, updated, etc.)

It is worth noting that when the bundle representing an agent class object is updated, all of the agents of this class will be shutdown. An alternate behaviour would be for the system to persist the agent state of all of the agents for the given agent class and reconstitute the agents when the agent class is reregistered with the system. Currently the agent environment listens for service events on the OSGi platform, and handles agent managed based on these events.

**Agent Communication Services**

Agent communication is performed asynchronously between agents and is accomplished using messages whose format is specified by an agent communication language (ACL) such as KQML or FIPA-ACL. That agents communication with ACL messages is a key requirement for the system. The standardization of communication format between agents allows for communication between any two agents to be possible. This allows for coordination and collaboration between agents in ways not defined in the initial design.

Messages and agent data are transferred using a transport protocol. The transport protocol is responsible for the transfer of ACL messages, agents data, and other communication between agent environments. The functions for this transportation was generalized, and encapsulated as a service. This allowed the
transport of information between environments to be configurable: messages can be transported over a network using different underlying networking technologies, such as SOAP, XML-RPC or CORBA. A Transport Service component that provides this service was defined. The transport protocol used by the agent environment is determined by which services are currently loaded by the system.

An CORBA implementation of the transport service was developed. Instead of binding CORMAT to one particular ACL specification or implementation, the system was designed to support multiple ACL. This was done by defining an ACL driver service: each ACL Driver acts as a factory for creating ACL messages. The ACL driver and linked ACL driver implementation are responsible for their serialization and deserialization. This design allowed for multiple agent communication languages to be used in the system.

Agent Location and Service Discovery Facilities

The location of agents is important in mobile agent systems. Agents often need to collaborate to accomplish their goals. In order to collaborate, agents need to be able to find other agents that can perform the services they need. Agents also need to be able to broadcast services they are willing to provide so that other agents can decide whether the publishing agent can help them accomplish their goals. Often, as an agent moves through hosts on a system other agents need to be able to locate and communicate with the agent regardless of the agent’s current position. To accomplish this, facilities for allowing agents to find other agents residing within the system needs to be provided by the MAS. These facilities should allow for agents to post services they provide and search for agents that provide services they require. The aspects of agent service publishing and querying is a problem of service discovery, and very similar to problems encountered in many other systems: OSGi service management and CORBA Trading objects are two such examples.

Agent finding services for mobile agent systems are usually required to track agents existing on multiple platforms and as a result architectures for finding agents are often distributed across the network of agent systems. Ideally,
as agents move from host to host the location service will track the agent’s location. The dispatching of requests is often optimized to allow for scalability in the number of agents connected to the network of agent systems, and the number of services and agents registered with the finder service. I found the papers “A Comparison of Mechanisms for Locating Mobile Agents” [Baumann, 1999], “A Taxonomy of Middle-agents for the Internet” [Sycara and Wong, 2000] and “Scalability Issues for Query Routing Service Discovery” [Gibbins and Hall, 2001] to be excellent references for understanding the different ways to design such a service/agent location facility.

Middle agents are one solution to the service delivery problem. “In an open multi-agent system, there are two types of agents: end-agents and middle agents, henceforth denoted by MAs. End-agents act as providers when they offer services; and requesters when they need them. MAs exist to enable interactions among end agents.” [Sycara and Wong pg 465] This approach is used in the JADE mobile agent system: there exists a directory facilitator (DF) agent whose goal is to maintain a directory of agent profiles for agents running within a JADE agent container. Wong and Sycara discuss two types of middle agents, the Matchmaker and the Facilitator. A Matchmaker middle agent will allow end agents to query the middle agent for references to the provider agent. A Facilitator middle agent acts as an intermediary between agent requests. This level of indirection is used for a number of reasons, including “to implement anonymity of the parties involved in a transaction; to guarantee fairness; or to collect affidavits for possible future disputes.” The JADE DF agent is an example of a Matchmaker middle agent.

Another key issue in maintaining a consistent agent location directory is tracking the agent’s location during migration between hosts. The responsibility for maintaining these links can either be placed on agents or on the framework supporting the agents.

If the responsibility for maintaining the agent location directory integrity is placed on the agent, each time it moves to a new location the agent will need to update the system of its new position. Some ways this can be done include
notifying the directory service of its new position, by leaving a pointer to its new host on the current system or by following a predetermined path. This is does not provide completely reliable reporting of an agent’s location in a network, as the agent may forget to update its current position, pointers may become stale/corrupt, or it may have to stray from its planned path.

If the responsibility for maintaining the agent location directory integrity is placed on the agent environment, the environment will have to send updates to the directory service whenever an agent is moved from one environment to another.

Many agent systems create unique names for agents across the entire agent platform. CORMAT was developed to support unique agent names only in the context of their own environment: an agent named “Logger” may exist on two separate hosts that communicate with each other. This was done as hosts were loosely coupled, and hosts could be added or removed at run time. Instead of trying to ensure the uniqueness of agent names across hosts, each host ensured the uniqueness of agent names only on the local system. The intent was that agent location would be performed by a higher level agent location or service facilitation service that would either be built into the toolkit, extensible as an OSGi service, or as a specialized agent.

Unfortunately, the development of an agent location service was not completed, and agent location and service discovery is not part of the CORMAT project deliverables. The usefulness of a distributed agent location and service discovery service became quickly apparent when implementing such a small problem as the demonstration application. The ability to locate agents independent of their location within the system would have allowed manager agents to discover workers in the systems, and for all of the agents to transparently access the Logging agent. Extending the system to include an agent location service would require either:

**Solution A**

a) Extending the agent context interface with a method such as `getFinder()` that returned a reference to the agent finding service.
b) Implementing a middle agent or middleware solution built into the agent framework. This would involve adding functionality to the sections of code responsible for agent lifecycle and mobility to maintain the state of the agent directory. Ideally the agent location service would be extracted to an OSGi service so many different agent locations could be implemented, and the runtime behaviour of the system could vary based on the type of agent location service loaded into the system. This could prove difficult, as algorithms for maintaining agent directory state vary greatly and finding a common interface for the algorithms would likely be very difficult to find.

**Solution B**

a) Creating a special class of agent responsible for agent location. One such agent implementation is given in [OSHIMA page 126-130].

**Security**

There are a number of areas of the system that have been designed to ensure security between the agents running within the system and between the interactions with the agents and the system. Steps were taken in the project design to strictly separate agent objects, threads of control, and access to system resources. A security manager interface was also added to the system to allow for the restriction of the actions of agents within the system.

**Agent Isolation**

A key part of the security for the system was separating each agent from the other agents in the system, and the underlying components of the system. The agents had to be autonomous, run independently of one another, and isolated from other agents. This was important for a number of reasons: agents shouldn’t be able to directly replace, modify, destroy, or corrupt the state of another agent in the system. The only way agents can affect other agents in the system should be through controlled access via the framework. The framework should also predictably handle the results of these controlled actions.
Access to the framework, properties of the framework, and objects within the framework should also be limited to controlled access. This helps ensure that agents cannot disrupt the consistency of the host they are executing within, and restricts their interaction with the system to well defined operations.

The CORMAT agent environment ensures that agents are isolated from each other, and the system. Each agent ran in a separate container object, an instance of the AgentThread class. The agent is run within a thread of control that continually executes the agent’s current behaviour. The only reference to the agent framework, an AgentEnvironmentContext object, is given to the agent during each execution. The AgentEnvironment defines a number of operations that the agent can attempt to perform on the system, as shown in the figure below.

![AgentEnvironmentContext Interface](image)

**Figure 5 AgentEnvironmentContext Interface**

These methods had to be designed carefully so that the underlying state of the agent system would not be exposed. Methods that send commands to the framework were designed with extensive error checking to ensure that they could not result in the system being compromised. This included ensuring that proper thread synchronization of the underlying system was performed, and that agents could not gain references to objects they should not be directly accessing. During the development of the demonstration a deadlock was found within the framework, stemming from improper message queue synchronization that allowed two agents to deadlock each other by sending a message to the one
while the second attempted to pull on the first agent. Through the use of careful thread synchronization this kind of deadlock situation dealt with in this, and other similar problems that could arise from agents calling methods of the agent environment context were averted.

Care had to be used in the agentInformation method. This method returned an AgentInfo object: this object is used to encapsulate all of the metadata related to an agent. This object is also used by the underlying AgentThread container: a deep copy of the object is made each time the agent requests it through the agent context. This is done so the agent cannot modify its metadata directly using the object reference (ex: modifying its hop count, or name.)

**OSGi Isolation**

A further separation of agent and agent environment framework code is accomplished by using the OSGi service framework. In a normal scenario (when agents are deployed in separate bundles,) the agents and agent environment framework code exist in different bundles within the OSGi platform. The OSGi framework provides strict package separation bundles: packages defined in the agent bundle will not be visible in the agent environment bundle unless they are exported by the agent bundle and imported by the agent environment bundle. This is usually not the case, and therefore the only classes that are accessible to both bundles are the packages that are exported by the one bundle and imported by the other. A scenario that demonstrates how this makes the framework more secure is given below:

Agent A’s action method is called by the toolkit, and an AgentEnvironmentContextImplementation object is passed into the action method, which is cast to an AgentEnvironmentContext object automatically. Agent A wants to get a reference to an AgentEnvironmentContextImplementation object, as this object contains hooks into a number of implementation objects in the system that A could then modify. A attempts to cast the AgentEnvironmentContext object to a AgentEnvironmentContextImplementation object, and a ClassNotFoundException is thrown!
This occurs because the bundle’s class loaders are strictly separated by the OSGi platform, the class AgentEnvironmentContextImplementation is not available to A, only to the agent environment implementation.

Reflection Based Attacks
Java reflection allows an object to determine the public methods of an object at runtime. The current method of hiding the implementation of the underlying system objects in the framework revolves around the agent not being able to access the methods of the object's implementation object directly. I did not have time to properly determine if reflection could be used to access the methods of implementation object methods on a OSGi service, such as the AgentEnvironmentContext in the scenario above. If this is the case, then a security manager would have to be developed to deny agent threads from performing reflection on agent context objects.

Bundle Based Attacks
Another way to compromise the system is to allow one bundle to provide multiple services all used by the framework. Since all of the services are executed in the same bundle context, it is entirely possible for the services to share state. One such situation that could compromise the system involves an agent class bundle that also defines an ACL driver. This could allow the agents to snoop on messages created by the ACL driver, or messages decoded by the driver.

One solution to this problem would be to allow bundles to only provide one type of service relevant to the agent framework. The OSGi framework cannot enforce this: one solution would be for the service listeners of the agent environment to listen to which bundles they receive services from. Once a service of one type is received, the listeners would not accept/register services of other types from a given bundle, until the bundle leaves the active state, at which point will accept the first service of any type from the bundle.
CORMAT Permissions
Currently any agent can perform lifecycle and mobility actions on any other agent in the system. It would be useful for these behaviours to be restricted: creating Java permission objects for these actions is an obvious way to accomplish this requirement. To extend the framework to include permission checking for this type of action, permissions would have to be added to the agent environment interface package (that is exported to agents), and checks would have to be put in place in all of the appropriate places in the environment implementation.

Security Manager Service
A requirement for the project was to investigate agent security, and methods of enforcing and granting permissions to agents within the framework. This allows for agents to be run within a sandbox environment, where they will only be allowed to access a system resource (files, sockets,) if they are given permission to do so. A requirement was also that agents should be able to query the system about what permissions they have, and be able to request new permissions. (The ability to remove features would have also been useful, but was not implemented.)

The security manager was implemented as an OSGi service. It is responsible for providing security handler objects to the environment and for installing security management code into the underlying JVM. It expects to be given an instance of the AgentSecurityAccessHandle class as a call back object. This call back allows the security manager service to get information about the agents in the system through a controlled interface.
The ResourceProvider is the object used by agents to request permissions and check to see if they have permissions. Its implementation queries the installed security manager (if a security manager is in use by the agent environment.) If a security manager is not installed, it returns true.

The AgentEnvironmentSecurityService is the interface the security service registers with the OSGi platform. If an environment wants to use the service, it attaches itself to it, and then activates the service. When it attaches itself to the service, it passes an AgentSecurityAccessHandle to the security service that can be used to get information about environment, and about agents running within the environment. The AgentEnvironmentSecurityService may not be able to perform the activation or attaching actions: a security policy may already be in place in the system, or an environment may already be attached to the security service. For this reason, both of these methods return a boolean value representing their success. In retrospect, an Exception would have been a better way of representing failure of these methods, as Exceptions require a more explicit handling than status variables.

The AgentSecurityAccessHandle is a call back method provided by the agent environment to the security manager. It provides three methods that the security manager may use to query the environment: getAgent, getAgentThreadGroup, and getEnvironmentName. The getAgent method returns
the metadata of the agent that is responsible for a given thread. The getAgentThreadGroup method returns the thread group object that all agent threads are created with. The final method, getEnvironmentName, returns the host name of the environment.

It was required that permissions should be assignable to individual agents. A programmer or administrator should be able to assign permission to an agent either through policy or dynamically at runtime. The agent should be able to query the manager regarding its permissions and request new permissions. In order to be able to do this, the system had to know if a thread that required permissions was an agent in the system, or normal system code that the special agent permissions did not apply to. All agents were run in a global thread group, and each agent in a separate thread group. A thread group is a set of threads that can also contain sub thread groups. When a thread creates a new thread, it will be added to the thread group of the creating thread. A method exists for thread group that checks if a given thread group is the ancestor of another thread group. This allowed the security system to check if a given thread was an agent or normal thread, as it could perform the following check:

```java
if (agentThreadGroup.parentOf(Thread.currentThread().getThreadGroup())) {
    // thread is an agent thread
} else {
    // thread is not an agent thread
}
```
A mapping between agent threads and agent identifiers had to be maintained, so the security manager could maintain a mapping of agents to permissions. When a call from an agent needs to be checked by the security manager the entire list of agents is scanned until the agent whose thread group matches (is equal to or a child of) the executing threads thread group is found. This is a $O(nm)$ operation, where $n$ is the number of agents in the system, and $m$ is the time needed to compute the ThreadGroup::isParent function. A more efficient algorithm to this problem is worth evaluating in future releases of the system as these security checks could become costly if there are too many threads and agents in the system.

The initial design for the security manager service revolved around creating a subclass of the java Security Manager. Prior to the Java 2 security model, the java.lang.SecurityManager class was responsible for checking the permission of possibly restricted method calls (such as a process attempting to
open a socket connection, or write to a file.) If an application wanted to implement custom security, subclassing the SecurityManager object was all that needed to be done. The security manager consisted of a number of calls that were called by underlying objects when a possibly unsafe call was being performed. As the java API evolved, it was found that this solution was not extensible: the methods checked explicit types of security permissions, such as checkRead, or checkSocket. Software engineers could not extend the system with new permission types. A new design for security management was built into the Java 2 environment: the java.lang.Policy and java.security.Permission classes were defined, and the role of security checking was shifted from the SecurityManager to the Policy object. Each JVM has a singleton Policy object that maintains collections of Permissions. The role of the security manager is to route checks performed on it to the permissions collection maintained by the policy object. The Permission object can be subclassed to model specific security permissions. An excellent discussion of this topic is given by [Neward 2001].

When the initial SecurityManager object was developed for the system it quickly became apparent that extending the SecurityManager was not the correct way to implement security management. When the manager was installed, the Oscar framework the test began to throw security permission exceptions: after investigation, the framework has a number of permission objects defined: installing a security manager activated the checking of these permissions that resulted in the exceptions being thrown. Therefore the security manager component was redesigned with a policy object based solution.

The security manager service implementation involved the use of two objects: AgentEnvironmentSecurityServiceImplementation, and AgentEnvironmentPolicy. AgentEnvironmentSecurityServiceImplementation is responsible for maintaining the list of agent permissions, and providing a ResourceProvider to the agent environment. The list of maintained permissions in queried by agents in the system and the system itself when authorizing method calls.
AgentEnvironmentPolicy is an extension of the existing Policy object. When created, it is passed the existing system Policy object, which it maintains a reference to. When an authorization call is received, the policy object checks if the call has been made by an agent using the thread group method discussed above. If it was, the policy object checks the existing system policy regarding permissions as well as the permissions of the agent. If either of these permission sets implies the action can occur, the agent is allowed to perform the action. If the call was not made by an agent, the AgentEnvironmentPolicy uses the existing policy object to determine whether a call should be allowed.

Security Manager Implementation & Behaviour

A distribution of CORMAT that demonstrates a use of the security manager has been provided on the deliverable CD under the 'secure-cormat' directory. The underlying policy is set with a static policy file that gives privileges to the necessary jars in the system, and denies the rest of the jars access.

The implementation of AgentEnvironmentSecurityServiceImplementation uses a very simple logic for agent permissions: if an agent requests a permission, it will be given to the agent. All of the agents in the system are initially given no permissions, but are given any permission they request using the ResourceProvider object. While this behaviour is very simple (and suggested for use in a deployment), the manager could be easily extended to incorporate much more complex methods of security tracking and authorization for agents. A good way to accomplish this would be by extending or modifying the hasPermission, getPermissionCollection, and requestPermission methods of the AgentEnvironmentSecurityServiceImplementation object.

Environment to Environment Security

Environment to environment is not addressed by the system. A certain amount of benevolence by the remote hosts interacting with the local system is assumed. An agent received from a remote host is given the same treatment as agents that originated from the local environment. Ideally environments could communicate between one another regarding what events they would allow each
to perform on the other, and request the permission to execute permissions on a remote environment. Such events include agent transfer, agent class transfer, agent location/service query requests, etc. The Security Markup and Assertion Language (SAML) was researched for modeling security related communication between agent environments. As stated on the SAML website, “SAML is an XML-based framework for communicating user authentication, entitlement, and attribute information. As its name suggests, SAML allows business entities to make assertions regarding the identity, attributes, and entitlements of a subject (an entity that is often a human user) to other entities, such as a partner company or another enterprise application.”

SAML has been developed by the Organization for the Advancement of Structured Information Standards (OASIS) group as a language for systems to communicate security assertions and requests. Agent environments could build elaborate policies to restrict other systems from performing certain operations on the system, and communicate and resolve these permissions using a language such as SAML.

**Demonstration**

The demonstration for the project was developed to showcase the mobility, thread safety, and messaging capabilities of the system. The demonstration is a simple job dispatching system. The demonstration included three classes of agents: manager, worker and logging agents. The goal of the demonstration is to show the key features of CORMAT in a clear and simple manner.

A agent control language called “DemoACL” was developed for the demonstration. It is a small subset of FIPA-ACL: each message has a performative, action, and data. The performative represents the overall class, or type of the message. There are three performatives for DemoACL messages: inform, request, and error performatives. The action acts as a header for the specific type of message. Data is an object, and can hold any serializable data object.

---

4 http://www.oasis-open.org/committees/security/faq.php
A manager agent can create worker agents, and perform a number of mobility related operations on the workers, and request tasks from the workers. The manager can request the worker to start a job, stop a job, and shutdown. These messages are sent asynchronously to the worker. Each manager has a GUI the demonstrator can use to send the messages and perform the mobility commands. The manager updates its GUI display as it receives status updates from workers. The manager logs updates to the Logging agent.

The worker agent performs a very simple task: it counts from 1 to 1000, waiting for a period of time after each increment to simulate the execution of a task. This delay is set randomly between 0 and 1000 ms for each worker, to make the updates of the manager seem more realistic (otherwise all of the agents would update at the same time.) The worker agent has two states: working and waiting. Either the agent is working on a job for a manager, or waiting to receive a command. When the agent is working, it will only accept messages from the manager who sent it the request. If it receives a message from another agent, it will reply with an error message, indicating that it is currently busy. If it receives a stop or shutdown message from its current manager, it will comply. When waiting, it performs a blocking call to receive a
message, thus waiting until it receives a message (i.e.: it’s message queue is not empty.)

The Logger agent simply displays a log of all of the messages it has received. Upon creation it creates a simple GUI that displays a table of all of the messages it has received. The manager and worker agents in the system attempt to send log messages to a local agent called Logger, and an agent called Logger on the ‘Main’ agent environment. These calls are not critical: if either of the logging agents are not available, the logging calls return without any warning.

![Logger Agent GUI](image)

**Figure 9 Logger Agent GUI**
Conclusion

I believe I have achieved the goals I had set out in the development of the CORMAT project. It has been an interesting project that has challenged my abilities as a software designer. The threading and concurrency issues involved were the most difficult aspect of the project: my one regret is that the system relies on the agents correctly returning from their behaviour loops. I believe that reimplementing the agent threading with a higher level concurrency model would allow for agent mobility events to avoid the deadlock they currently are susceptible to.

The use of the OSGi platform as an enabling technology for a mobile agent technology has worked out very well. The dynamic bundle loading has allowed me to develop a pluggable system that allows for upgrading of components dynamically at runtime. The explicit package boundaries have proved to be an excellent method for ensuring component security, if also a source of grief at times in the project (just because a package complied, did not mean it would be able to resolve classes at runtime.)

Future work for the system include revisiting the concurrency handling for agents, the addition of agent location/service facilities, and adding FIPA compliant ACL and transport components.

I look forward to investigating the use of CORMAT and the OSGi platform in my personal projects. The CORMAT system components run on a very small profile, and as a result I am interested in investigating use of the system on low profile devices. I have found the OSGi platform to be an excellent platform for developing loosely coupled software components: by explicitly separating components into specified services, the system quickly either becomes very loosely coupled. The OSGi platform would make an excellent system for an plugin architecture.
## Appendix A: Glossary

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACL</td>
<td>Agent Communication Language</td>
</tr>
<tr>
<td>FIPA</td>
<td>Foundation for Intelligent Physical Agents</td>
</tr>
<tr>
<td>JAAS</td>
<td>Java Authentication and Authorization Service</td>
</tr>
<tr>
<td>JADE</td>
<td>Java Agent Development Framework</td>
</tr>
<tr>
<td>KQML</td>
<td>Knowledge Query and Markup Language</td>
</tr>
<tr>
<td>Knopflerfish</td>
<td>An open source OSGi framework</td>
</tr>
<tr>
<td>MAS</td>
<td>Mobile Agent System</td>
</tr>
<tr>
<td>MASIF</td>
<td>Mobile Agent System Interoperability Framework</td>
</tr>
<tr>
<td>OSCAR</td>
<td>An open source OSGi framework</td>
</tr>
<tr>
<td>SAML</td>
<td>Security Assertion Markup Language</td>
</tr>
</tbody>
</table>
Appendix B: System Configuration Parameters

Bundles in the system are configured using parameters passed to them through the OSGi configuration file. On the OSCAR OSGi platform, this is the system.properties file found in the same directory as the main OSGi library. On the Knopflerfish OSGi platform, this is the props.xargs file. These parameters are used by CORMAT to configure component behaviour. Another way to configure bundles is through the OSGi configuration service. This is a much more expressive way to handle bundle configuration, but for the purposes of CORMAT, the level of detail of the configuration service was not necessary.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Parameter Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent Environment Name</td>
<td>agent.environment.host.name</td>
<td>The name of the agent environment. If this is not set, the agent environment will not initialize.</td>
</tr>
<tr>
<td>CORBA Name Server Port</td>
<td>ORBInitialPort</td>
<td>This sets the port the corba environment driver attempts to connect to.</td>
</tr>
<tr>
<td>CORMAT debug</td>
<td>cormat.debug</td>
<td>A debug flag. If set to true, a number of the components will print debug messages to the console.</td>
</tr>
<tr>
<td>CORBA name server host</td>
<td>cormat.corba.host</td>
<td>This specifies the host the corba driver will attempt to resolve a name server to.</td>
</tr>
</tbody>
</table>

Note: in the Knopflerfish parameter file, the key names have to be entered with the prefix `-D`.
Appendix C: References


Appendix D: OSGi Programming Tutorials

Bundle Development Tutorial
http://www.knopflerfish.org/programming.html

OSGi Service Tutorial
http://www.knopflerfish.org/osgi_service_tutorial.html

Gravity Service Binder: a OSGi framework for automated service dependency management
http://gravity.sourceforge.net/servicebinder/

OSGi and Gravity Service Binder Tutorial
http://oscar-osgi.sourceforge.net/tutorial/
Appendix E: OSGi Platforms

A number of OSGi platforms were researched during the course of the CORMAT project. Below is a brief list of those researched during the project.

Oscar
http://oscar.objectweb.org/

Knopflerfish
http://www.knopflerfish.org/

IBM Service Management Framework

Prosyst
http://www.prosyst.com/osgi.html

Java Embedded Server
http://java.sun.com/docs/books/jes/

Connected Systems
http://www.connectedsys.com/

Eschelon Corporation LonWorks Bundle Deployment Kit
http://www.echelon.com/products/development/osgi/default.htm

Espial Devicetop OSGi compliant Customizable Graphical Operating Environment

Ubiserve OSGi platform
http://www.gatespacetelematics.com/userarea/login/ubiserv.shtml

AveLink Embedded (OSGi) gateway
Appendix F: Classpath loading issues
During development, it was noticed that there were perceived differences between the way Oscar and Knopflerfish handled user defined classpaths. Running Oscar inside of eclipse, classes were being resolved when not explicitly imported by the bundle they were in. This was due to Eclipse resolving the classes to classes on the user class path when they could not be resolved to the bundle classpath. More information can be found about this issue with the following links.

Knopflerfish mailing list
https://sourceforge.net/forum/message.php?msg_id=3069304

Oscar mailing list
Appendix G: Sample Component Manifest files

Agent Environment
Manifest-Version: 1.1
Bundle-Activator: rjyoung.carleton.honors.components.agentenvironment.Activator
Export-Package:
  rjyoung.carleton.honors.interfaces.agentenvironment,rjyoung.carleton.honors.interfaces.acl,rjyoung.carleton.honors.interfaces.agentenvironment.control,rjyoung.carleton.honors.interfaces.agentclass
Bundle-Name: Cormat Agent Environment
Bundle-Description: Agent environment
Bundle-Vendor: Robert Young
Bundle-Version: 1.0.12

Agent Environment Control
Manifest-Version: 1.1
Bundle-Activator: rjyoung.carleton.honors.components.agentenvironmentcontrol.Activator
Import-Package:
  rjyoung.carleton.honors.interfaces.agentenvironment.control,rjyoung.carleton.honors.interfaces.agentenvironment,rjyoung.carleton.honors.interfaces.agentclass
Bundle-Name: Cormat Agent Environment Control
Bundle-Description: Agent environment controller GUI
Bundle-Vendor: Robert Young
Bundle-Version: 1.0.2

Agent Class
Manifest-Version: 1.1
Bundle-Activator: rjyoung.carleton.honors.components.agentsendingagentclass.Activator
Bundle-Name: Simple Agent Sending Agent Class
Bundle-Description: This bundle provides a simple agent that tries to send itself to another agent host
Bundle-Vendor: Robert Young
Import-Package:
  rjyoung.carleton.honors.interfaces.acl,rjyoung.carleton.honors.interfaces.simpleacl,rjyoung.carleton.honors.interfaces.agentclass,rjyoung.carleton.honors.interfaces.agentenvironment
Bundle-Version: 1.0.1

Agent Transport Protocol Driver
Manifest-Version: 1.1
Bundle-Activator: rjyoung.carleton.honors.components.environmentdriver.Activator
Bundle-Name: Cormat CORBA Protocol Driver
Bundle-Description: CORBA CORMAT protocol driver.
Bundle-Vendor: Robert Young
Import-Package: rjyoung.carleton.honors.interfaces.agentenvironment,rjyoung.carleton.honors.interfaces.acl
Bundle-Version: 1.0.1

ACL Driver
Manifest-Version: 1.1
Bundle-Activator: rjyoung.carleton.honors.components.demoacl.Activator
Export-Package: rjyoung.carleton.honors.interfaces.acl,rjyoung.carleton.honors.interfaces.demoacl
Bundle-Name: Demo ACL driver
Bundle-Description: This bundle provides a demo ACL that consists of a performative, action, and associated data.
Bundle-Vendor: Robert Young
Bundle-Version: 1.0.0

Security Manager
Manifest-Version: 1.1
Bundle-Activator: rjyoung.carleton.honors.components.securitymanager.Activator
Bundle-Name: Cormat Security Manager
Bundle-Description: Basic agent environment security manager
Bundle-Vendor: Robert Young
Import-Package: rjyoung.carleton.honors.interfaces.agentenvironment,rjyoung.carleton.honors.interfaces.agentclass
Bundle-Version: 1.0.0
# Appendix H: System Component Listing

The following is a listing of the components in the bundle directory of the attached CD.

<table>
<thead>
<tr>
<th>Bundle Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AgentEnvironment.jar</td>
<td>The main cormat agent environment.</td>
</tr>
<tr>
<td>AgentEnvironmentControl.jar</td>
<td>A simple agent environment controller that can be used to send commands to the environment.</td>
</tr>
<tr>
<td>AgentSendingAgent.jar</td>
<td>An test agent that sends itself to the AETesting2 remote host. This is a test for agent sending mobility.</td>
</tr>
<tr>
<td>CountingWorkerAgent.jar</td>
<td>Used in the demonstration as a worker agent.</td>
</tr>
<tr>
<td>DemoACL.jar</td>
<td>Demonstration ACL.</td>
</tr>
<tr>
<td>EnvironmentDriver.jar</td>
<td>CORBA message transport driver.</td>
</tr>
<tr>
<td>FinderAgent.jar</td>
<td>Location agent (not complete).</td>
</tr>
<tr>
<td>LoggingAgent.jar</td>
<td>Logging agent.</td>
</tr>
<tr>
<td>ManagerAgent.jar</td>
<td>Manager agent.</td>
</tr>
<tr>
<td>MessageCatchingAgent.jar</td>
<td>A simple agent that catches SimpleACL messages it is sent and displays them on the console.</td>
</tr>
<tr>
<td>MessageSendingAgent.jar</td>
<td>A simple agent that continually sends messages to agents.</td>
</tr>
<tr>
<td>SecurityManager.jar</td>
<td>The implementation security manager component.</td>
</tr>
<tr>
<td>SimpleACL.jar</td>
<td>Simple ACL.</td>
</tr>
<tr>
<td>SimpleAgent.jar</td>
<td>This agent counts from one to 22, and also attempts to perform a write operation to a file when initially created to test the security manager class.</td>
</tr>
</tbody>
</table>