

# **The ABLE Agent Platform User's Guide and Reference**

Version 2.3.0

Last Update: 6/15/2005 1:40 PM

**com.ibm.able.platform**

---

Licensed Materials - Property of IBM

Package: com.ibm.able.platform

(C) Copyright IBM Corporation 1999, 2005. All Rights Reserved.

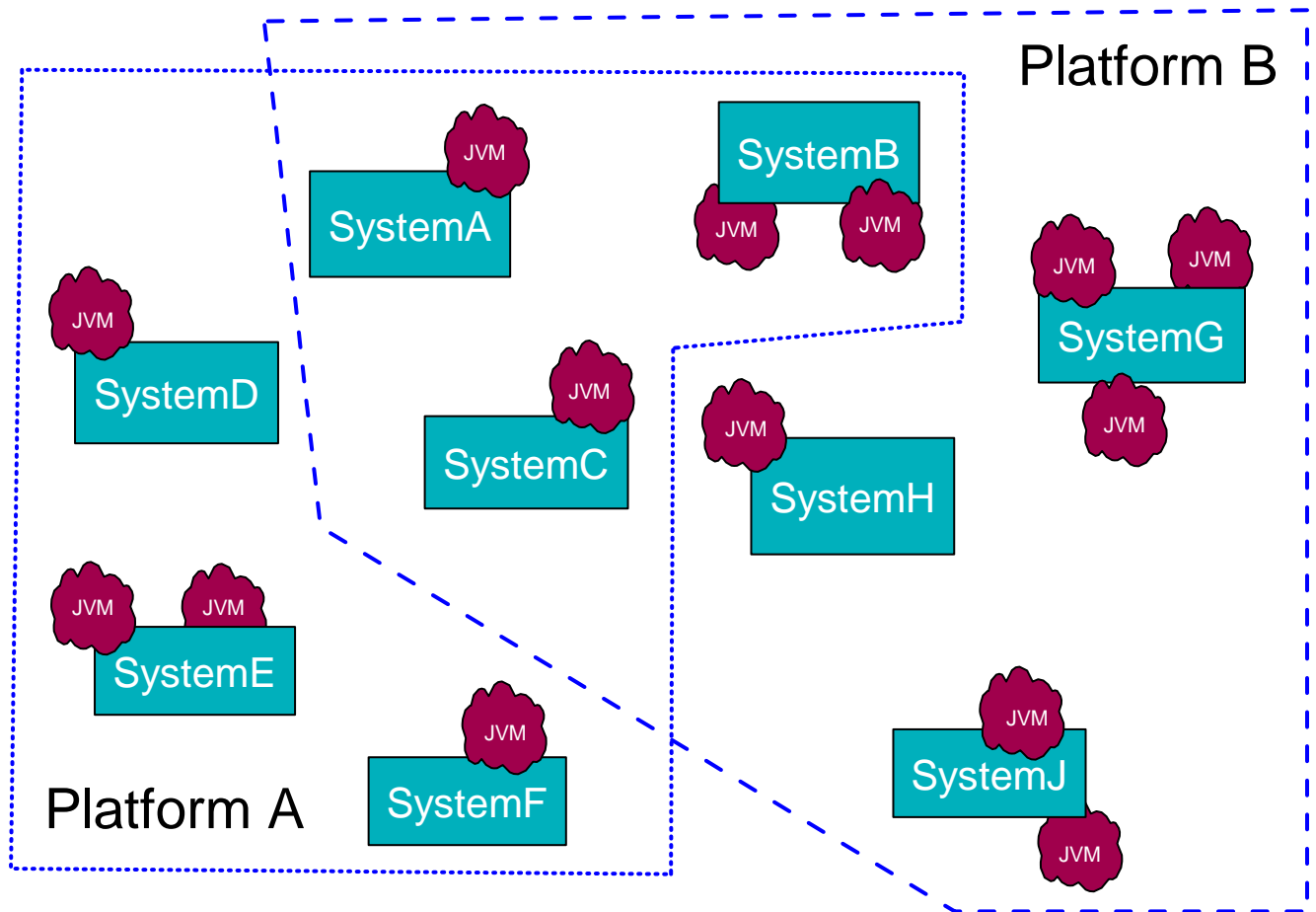
US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP  
Schedule Contract with IBM Corp.

## Table of Contents

Introduction.....	4
The ABLE Platform Console.....	5
Platform Services .....	6
Platform Agents .....	7
Platform Security .....	8
Creating a Platform Agent .....	8
Configuring the Platform .....	9
Configuration Basics.....	10
Platform Name .....	10
Cryptography Parameters.....	10
Principals.....	10
Trust Levels .....	11
Agent Pools .....	11
Services .....	13
Permitted Agents.....	14
Setting Java Permissions.....	17
Starting the ABLE Platform.....	18
Using the ABLE Platform Console to Run Platform Agents .....	18
An Example Platform Agent.....	19

## Introduction

The `com.ibm.able.platform` package makes it easy to set up an environment wherein Agents and Agent Services can be created and distributed across many disparate computer systems. Of course, the systems involved must be able to run Java and be connected to a network. This loose collection of computer systems, with each system running one or more Java Virtual Machines (JVMs) each at a different port, and each JVM containing zero or more agent services and zero or more agents, comprise an agent platform. **Figure 1** shows two overlapping ABLE platforms, Platform A and Platform B:



**Figure 1.**

In the figure, Platform A consists of Systems A, B, C, D, E, and F. Platform B consists of Systems A, B, C, G, H, and J. Systems A, B, and C belong to both platforms. As shown, each system can have one or more JVMs and each JVM is addressable by its system name and unique port number through a Java RMI Registry. In ABLE terminology, a JVM used to run platform services and agents is called an Agent Pool. Not shown in the figure are agents and agent services running within the JVMs or agent pools.

Within a platform, agents can freely communicate with other agents in several ways: by direct remote method invocation, by sending and receiving JAS Transport Messages, by sending and

receiving synchronous and asynchronous events, and by carrying on “conversations”. Agents can also interact with agent services, and this is always done by direct method invocation.

## The ABLE Platform Console

As of ABLE 2.3.0, ABLE introduces the ABLE Platform Console, a tool that lets you manage a *running* ABLE platform. To define and run your platform, you still need to follow the instructions in this guide. Documentation for the console is shipped with the console itself, which can be run as an Eclipse plug-in or a standalone Eclipse Rich Client Platform (RCP) application. Please be aware that there are two changes to the `ableplatform.preferences` file from previous versions of ABLE:

1. Agent.N.ConstructorArgs

If the first argument to an agent’s constructor is a `String`, the console, when used to create a running agent, assumes that the argument is the agent’s display name and will pass a user-entered name in that argument.

2. Agent.N.Type

Previously, anything entered here was simply for documentation and ignored by ABLE. Now, however, the ABLE Platform Console uses type information to distinguish between agents, and every agent must have a type that is unique. If you fail to give each agent definition its own unique type, the console may not behave as expected when you try to create a new agent.

## Platform Services

The platform package does not manage agents itself, *per se*, but it does provide services that agents can make use of. Services provided with the platform package include:

- Verifiable Agent Naming Service

The verifiable agent naming service acts as a certificate authority, handing out a globally unique, verifiable Agent Name to each agent. Agents can use the naming service to verify that agent names sent to them have not been tampered with.

- Verifiable Agent Directory Service

An agent can use the directory service to obtain an empty Agent Description, fill out the description with pertinent information about itself, and then register the description with the directory service. All agents can search the directory to find other agents with which they might want to communicate. An agent can update its own description at any time.

- Agent Lifecycle Service

The agent lifecycle service can be used to create and initialize instances of agents anywhere within the platform. Agents can be suspended, resumed, and terminated, but only by other agents with the proper authority. Agent Lifecycle can create only those agents that have been predefined to the platform, so that rogue agents cannot be easily introduced into the platform.

- Platform Persistence Service

The persistence service allows other services and agents to persist information to an external database so that vital information can be recovered in the event of a crash and restart.

- Agent Logging Service

The agent logging service can be used by agents to log `AbleAgentSituationReports`. These reports can be simple history logs of what situations an autonomous agent has encountered and what actions the agent has taken, or, for non-autonomous agents, the reports can contain details of situations agents have encountered, along with possible responses (actions that the agent can perform) to the situations. These situations can then be reviewed and responded to by other agents.

- Platform Support Service

The platform support service provides methods to control aspects of the platform; for example, the entire platform (all agent pools) can be shut down using this service.

---

All of the above services implement an ABLE Platform Service Event Generator interface, so that other Java entities can register and deregister as service event listeners with the services. Listeners must implement the `AblePlatformServiceEventListener` interface.

- Message Transport System

The message transport system lets agents send Java Agent Services (JAS) Transport Messages to, and receive Transport Messages from, other agents.

## Platform Agents

In addition to all the above services, the platform package provides a base class, an `AblePlatformDefaultAgent`, from which other platform agents can be derived. The `AblePlatformDefaultAgent` contains no special logic to perform any unique task (that must be added in derived agents), but it does make it easy to perform the following functions:

- obtain access to all the platform services
- obtain a unique, verifiable agent name from the agent naming service
- register a description with the Agent Directory Service so that other agents can find it

All of the above can happen automatically when the ABLE platform agent is initialized.

The `AblePlatformDefaultAgent` also provides methods to make it easy to send JAS Transport Messages to and receive JAS Transport Messages from other platform agents. At least one JAS Locator is placed into each agent's description, and this Locator can be used by other agents to route messages to the Locator's owning agent. An `AblePlatformDefaultAgent`, if it configures itself so, can have many Locators in its agent description, allowing it to receive Transport Messages at many different access points.

Note that an `AblePlatformDefaultAgent` can also place another type of Locator into its agent description. This special, ABLE-defined Locator contains an RMI reference to the agent so that other agents can directly invoke remote methods on the agent.

As of ABLE 2.2.0, agents can advertise their roles or behaviors by utilizing `AbleAgentCapability` objects. For example, an agent might advertise that it is a "DASD Monitor" agent. Along with each capability, an agent can specify the types of situations that it expects to encounter using `AbleAgentSituation` objects. For example, a "DASD Monitor" agent might expect to encounter "DASD Thrashing" situations. Each capability that an agent advertises can be set (and dynamically managed) to allow the agent to:

- respond to expected situations autonomously,
- ask permission before for taking an action by logging an `AbleAgentSituationReport` to the Agent Logging Service where it can be viewed and answered by other agents (or people, if provided with a console), or

- not take any action at all.

Through dynamic management of an agent's capabilities, an agent can be running, asking for permission to carry out actions at one point in time, or be fully automated, taking action on its own, at another point in time.

## Platform Security

The ABLE platform can operate with security on or off; off is the default. When security is off, the platform behaves as in the JAS reference implementation, and any agent can do anything to any other agent or service, including spoofing other agents' names and changing other agents registered agent descriptions.

When security is on, the Verifiable Naming Service takes on the role of a certificate authority, and ABLE uses Verifiable Agent Names (or VANS), `AbleSecureKeys`, defined Kerberos principals, defined trust levels, and other mechanisms to authenticate and authorize one agent to another.

To control the security setting for the platform, use a text editor to edit the file `able.preferences` and change the `security=` entry.

Note that when security is on, ABLE requires access to the **Java Generic Security Services (JGSS)** API, which must be installed in the Java environment. JGSS in turn requires Kerberos. JGSS is not shipped with ABLE.

To make it easy to add security checking to Java entities that want to interact with the ABLE platform, ABLE has created an `AbleSecuritySupport` object that can be plugged into any Java object. `AbleSecuritySupport` knows how to create and decode `AbleSecureKeys` used on secure method calls, and use the naming service to verify agent names. When security is on, an `AblePlatformDefaultAgent` will automatically acquire an `AbleSecuritySupport` object for itself.

Because `AblePlatformDefaultAgent` is derived from `AbleDefaultAgent`, many of the old methods can be used just as before. However, many of the methods will no longer work in a secure environment. Instead, new methods, that take an `AbleSecureKey`, must be used in their place.

## Creating a Platform Agent

Creating a platform agent is simple; as already mentioned, just derive the new agent from `AblePlatformDefaultAgent`. Everything the agent needs to operate in the platform is already provided in this base class. For example, when the agent is constructed, it will automatically obtain addressability to the available agent services and then obtain a globally unique verifiable Agent Name from the Agent Naming Service. When the agent is initialized, it will create an Agent Description for itself and then, by default, register the description with the Agent Directory Service, but the agent can be customized so that it doesn't automatically register with this service.

In the usual ABLE paradigm, an agent is first constructed, then it is customized by calling the agent's methods, and then the agent is initialized so that it begins running. In the platform world, agents are typically self-customizing (from within their constructor methods) and may also be self-initializing (also from within the constructor methods). This is because it is not wise to let "outside," untrusted code configure an agent. Note that the Agent Lifecycle Service, when used to create an instance of an agent, can pass predefined customization parameters to an agent's constructor method, and that it can also pass predefined initialization parameters to an agent's initialization method.

Of course, it is possible to define new secure methods for an agent that may only be used by other Java objects with the proper authority. Because all ABLE platform agents are also RMI remote objects, you can add secure remote methods to your agent by first defining an interface that your agent can implement; then implement those methods in your agent; and finally compile the agent with **rmic**. To be secure, these methods should take an **AbleSecureKey** object as a parameter.

You can customize your agent so that it will register with an RMI registry, but by default this will not happen since the default agent is setup to register with the Agent Directory Service instead.

By default, an `AblePlatformDefaultAgent` receives JAS Transport Messages at its `receiveMessage(TransportMessage)` method. This method, in turn, simply calls the old standby `process(Object)` method, which does nothing but simply log the message. Either of these methods, as desired, can be overridden to do any special processing that is required.

Review the API documentation of `AblePlatformDefaultAgent` for further details on any of these and other agent behaviors.

## Configuring the Platform

Before you can start any piece of the platform, or run any agents in the platform, the platform must be configured. Configuring the platform means deciding

- What computer systems you want to use in the agent platform
- How many Java Virtual Machines (hereafter called **Agent Pools**) are available on each system
- Which agent services run in which Agent Pools
- Which agents are allowed on the platform
- Which agents are permitted to run in which Agent Pools

Furthermore, if security is on, you will need to define

- Which Kerberos principals are allowed in the platform
- Trust levels for each principal and Agent Pool

To begin, you need to find the **ableplatform.preferences** file in your ABLE installation's **examples** directory and copy it to the directory specified in the startup script's `able.prefdir` VM argument or, if that variable is not set, to your ABLE installation's **bin** directory. The value provided for `able.prefdir` in the startup scripts as delivered assumes that preferences will be read from the

directory above **ABLE\_HOME**. Then, using any text editor, edit the file using the comments in the file and the instructions here to guide you.

### Configuration Basics

In the preference file, lines that begin with the ‘#’ character are comments. You may freely add comments to the file. Except for comment lines and blank lines, all other lines are configuration parameters. **Do not try to add a comment at the end of a configuration line.**

Some parameters are specified in numbered groups. When adding, changing, or deleting groups, **you must make sure that the numbers begin with 1 and are consecutive.** ABLE configuration code will not “see” any groups after a gap in the numbering sequence. For example, if you are configuring AgentPools and you have AgentPool.1, AgentPool.2, AgentPool.3, and AgentPool.5, ABLE will not know about AgentPool.5 since AgentPool.4 isn’t specified.

Some of the configuration parameters might be for a future enhancement, and even though they are documented here, they may have no effect on the operation of the platform.

The following tables describe the configuration parameters.

#### Platform Name

PlatformName	Any string that is meaningful to you; used by the ABLE Platform Console.
--------------	--

#### Cryptography Parameters

CryptographyAlgorithm	The name of a cryptography algorithm available in your Java environment. The default is DSA.
CryptographyProvider	The provider of the cryptography algorithm specified above.

### Principals

In a secure platform, principals are used to tie agents, services, and requests to actual computer users. Principals are made up of two parts: an Alias and an actual KERBEROS principal. You may define as many Principals as you wish. Just remember to number each pair of statements consecutively.

When security is off, you may omit Principal statements entirely, or you may include them. If you do include them, they will not play a part in platform security. Because of this, you can code anything you wish in the Principal.N.Principal entry, or simply leave it blank.

Principal.N.Alias	Any string that you will use to tie this principal to other configuration entries. For example,  <code>Principal.1.Alias=johnDoe</code>  Each Principal alias must be unique.
Principal.N.Principal	An actual KERBEROS principal. For example,  <code>Principal.1.Principal=jd/binford.com@KERBDOMAIN</code>

## Trust Levels

In a secure platform, trust levels can be established for each specified principal. Trust levels are completely arbitrary and you can define as many as you wish, giving them whatever names you wish. `TrustLevel.1` is the most trusted; `TrustLevel.2` is less trusted than `TrustLevel.1`, and so on. When security is on, trust levels are examined in order to determine whether requests from one agent or service to another are to be honored. This can help prevent less trustworthy agents from doing bad things to agents running in your critical Agent Pools.

When security is off, you may omit `TrustLevel` statements entirely, or you may include them. If you do include them, they will not play a part in platform security.

TrustLevel.N.Alias	Any string that you will use to tie this trust level definition to other configuration entries. For example,  <code>TrustLevel.1.Alias=MostTrusted</code>  Each TrustLevel alias must be unique.
TrustLevel.N.Principals	A comma delimited list of principal aliases that you defined with <code>Principal.N.Alias</code> entries. For example,  <code>TrustLevel.1.Principals=johnDoe,maryDoe</code>

## Agent Pools

You must define the Agent Pools (Java Virtual Machines, or JVMs) that will contain your agents and agent services. The Agent Pools can be scattered across a network, and each Agent Pool, through association to a Principal, when used, can be running with a different authorization and security level. Later on, you can allocate agents and services to these Agent Pools, perhaps placing less trustworthy agents in Agent Pools that are restricted in some way, while trusted agents are placed

into Agent Pools that are completely unrestricted. (You may want to become familiar with the `java.policy` file and how it can be used to restrict Java code.)

AgentPool.N.Alias	<p>Any string that you will use to tie this Agent Pool definition to other configuration entries. For example,</p> <pre>AgentPool.1.Alias=host1</pre> <p>Each AgentPool alias must be unique.</p>
AgentPool.N.IPAddress	<p>The IP address of the system where this Agent Pool is running. The address may be given as a name (e.g. <code>tools.binford.com</code>) or as a numeric address (e.g. <code>192.169.1.100</code>). <code>localhost</code> is <b>not</b> permitted. Example:</p> <pre>AgentPool.1.IPAddress=tools.binford.com</pre>
AgentPool.N.Port	<p>The port on which this Agent Pool will be listening. See also <a href="#">Setting Java Permissions</a>. Example:</p> <pre>AgentPool.1.Port=55551</pre>
AgentPool.N.Principal	<p>When security is on, the alias of a principal that you defined as described in the section <a href="#">Principals</a>. In a secure platform, the specified principal must be the one who starts the Agent Pool. Example:</p> <pre>AgentPool.1.Principal=johnDoe</pre> <p>When security is off, you may omit this statement entirely, or you may leave the alias entry blank. If you do code an alias, the alias <b>must</b> be defined in the Principal section as described above. Example:</p> <pre>AgentPool.1.Principal=</pre>

## Services

The ABLE platform requires that certain agent services always be available whenever the platform is available. The `Services` statement identifies all services that you want to run when your platform is started. You can add your own services to the list, and then, for each service you add, enter another statement that specifies where (in which Agent Pool) the service is to run, and what Java class knows how to create the service (this is known as the *factory*). If you write your own service, you should derive it from `AbleBasicService`.

Services	<p><code>Services</code> and the following statements specify the services that are available to the platform and where the services reside. You do not need to change anything on the <code>Services</code> statement.</p> <p>However, if you are creative, you can replace any service with one of your own (specify the fully qualified class name of the service's factory) or add additional services to the platform.</p> <p>The following services are required:</p> <pre> Agent-Directory-Service Agent-Naming-Service Message-Transport-System Agent-Life-Cycle-Service Agent-Logging-Service Platform-Support-Service </pre> <p>The following service is optional:</p> <pre> Persistence-Service </pre>
Agent-Directory-Service Agent-Naming-Service Message-Transport-System Agent-Life-Cycle-Service Agent-Logging-Service Platform-Support-Service	<p>You do not need to change anything on these statements except to <b>specify in which Agent Pool each service runs</b>. You can place all services in one Agent Pool or place each service in its own Agent Pool.</p> <p>You can specify whether persistence is on or off, but if on, the <code>Persistence-Service</code> must be running.</p> <p>You can replace an IBM supplied service with one of your own, but this is not recommended. If security is on and you replace a service, your platform will not operate correctly.</p>

## Permitted Agents

Agent Lifecycle Service will create only those agents that have been identified to it. You identify agents to the Agent Lifecycle Service by entering information about them into your preferences file. For each agent, you specify where (in which Agent Pools) the agent is allowed to run, and who is allowed to create and run it.

Agent.N.Alias	<p>Any string that you will use to tie this agent definition to other configuration entries. For example,</p> <pre>Agent.1.Alias=spiffyAgent</pre> <p>Each Agent alias must be unique.</p>
Agent.N.AutonomyLevel	<p>Any string that is meaningful to you; not used by ABLE code; for documentation only.</p>
Agent.N.ClassName	<p>The fully qualified Java class name of the agent. For example, if you work for the Binford Tool Company and you have an agent implemented by <code>SpiffyAgent.java</code>, the class name might be:</p> <pre>com.binford.tools.SpiffyAgent</pre> <p>This class must be in the Java class path whenever the Agent Lifecycle Service is asked to create an agent of this type.</p>

Agent.N.ConstructorArgs	<p>Either empty or a comma delimited list of arguments to pass to the agent's constructor method when the agent is created by the Agent Lifecycle Service.</p> <p>Each argument is of the form</p> <pre>type:defaultValue</pre> <p>where <code>type</code> is one of <code>String</code>, <code>Boolean</code>, <code>Byte</code>, <code>Character</code>, <code>Double</code>, <code>Float</code>, <code>Integer</code>, <code>Long</code>, or <code>Short</code> and <code>defaultValue</code> is a value of the specified type. For example, if your agent's constructor takes a <code>boolean</code> and a <code>double</code> as arguments, you might code:</p> <pre>Agent.1.ConstructorArgs=Boolean:false,Double:12.34</pre> <p><b>NOTE:</b> When using the ABLE Platform Console to create agents within a platform, agents must be defined with a constructor that takes at least one argument, the first of which is a <code>String</code>. This first argument is taken to be the agent's display name as typed in by a user at the console, and the console will pass the name to the agent. If an agent is defined with a no-argument constructor, or if the agent's first constructor argument is not a <code>String</code>, the Console cannot set the name in the agent.</p>
Agent.N.EligiblePrincipals	<p>When security is on, a comma delimited list of Principal aliases that you have defined as described in the section <a href="#">Principals</a> and specifies those principals that are allowed to run the agent. The agent will inherit the trust level (see <a href="#">Trust Levels</a>) associated with whichever principal starts the agent. Example:</p> <pre>Agent.1.EligiblePrincipals=johnDoe,maryDoe</pre> <p>When security is off, you may omit this statement entirely, or you may leave the alias entry blank. If you do code any aliases, all of them <b>must</b> be defined in the Principal section as described above. Example:</p> <pre>Agent.1.EligiblePrincipals=</pre>

Agent.N.EligibleAgentPools	<p>A comma delimited list of Agent Pool aliases that you defined (see <a href="#">Agent Pools</a>) and specifies which of those Agent Pools the agent is allowed to run in. For example, if you defined Pool-1, Pool-2, and Pool-3 as Agent Pool aliases, then you might code</p> <pre>Agent.1.EligibleAgentPools=Pool-1,Pool-3</pre> <p>which indicates that the Agent Lifecycle Service is allowed to create instances of this agent in Agent Pools Pool-1 and Pool-3, but not in Pool-2.</p>
Agent.N.InitArgs	<p>Either empty or a comma delimited list of arguments to pass to the agent's initialization method when the agent is initialized by the Agent Lifecycle Service. The format of this statement is exactly the same as for the ConstructorArgs statement above.</p>
Agent.N.LastChangedDate	<p>Any string that is meaningful to you; not used by ABLE code; for documentation only.</p>
Agent.N.Type	<p>Any string that is meaningful to you; each agent must have a unique type. For example,</p> <pre>Agent.1.Type=DASD Capacity Monitor A</pre> <p><b>NOTE:</b> The ABLE Platform Console uses type to allow users to distinguish between kinds of agents that can be created.</p>
Agent.N.Vendor	<p>Any string that is meaningful to you; not used by ABLE code; for documentation only.</p>
Agent.N.Version	<p>Any string that is meaningful to you; not used by ABLE code; for documentation only.</p>

Agent.N.Properties	<p>Either empty or a comma delimited list of properties to set within the agent immediately after the agent is created by the Agent Lifecycle Service.</p> <p>Each property entry is of the form</p> <pre>propertyName:propertyValue</pre> <p>where <code>propertyName</code> is a simple string containing the name of the property to set and <code>propertyValue</code> is a simple string that is the value of the property. For example, if you might code something like:</p> <pre>Agent.1.Properties=foo:bar, prop2:any ol' string</pre> <p>Note that only simple strings are allowed; there is no escaping or quoting. If <code>""</code> are used, for example, they are taken to be an actual part of the name or value. If an agent needs a number as a property, the agent must get the property, parse the number out of the string, and reset the property as a number object.</p> <p>Obviously, a <code>propertyName</code> cannot contain the <code>'.'</code> character, nor can a <code>propertyValue</code> contain the <code>','</code> character, as these are used as parsing delimiters.</p>
--------------------	---

## Setting Java Permissions

Before you can start up the platform you may need to modify your `java.policy` file to allow communications on the ports you configured in your `ableplatform.preferences` file. To do this, find the `java.policy` file that is supplied with your Java installation and, using any text editor, add lines similar to the following to it.

```
permission java.net.SocketPermission "tools.binford.com:55551-55553",
"connect,accept,listen,resolve";
permission java.net.SocketPermission "9.10.84.20:55551-55553", "connect,accept,listen,resolve";
permission java.net.SocketPermission "localhost:55551-55553", "listen,resolve";
```

If you need to, add a statement for every system that you have specified in your `ableplatform.preferences` file, and the port numbers you enter here must also match the ports you specified there.

You can use the `java.policy` file together with the `java.security` file to restrict the capabilities of each Java VM (Agent Pool) you have defined in your preference file.

## Starting the ABLE Platform

Before starting the platform, check that, *for every system identified in your `ableplatform.preferences` file*, you have

- Installed ABLE
- Made all necessary JARs and classes available to the Java runtime environment
- Placed a copy of your updated `ableplatform.preferences` on the system. The default location is in the directory above that specified by the **ABLE\_HOME** environment variable. (You can actually put this file in any arbitrary place, but you will need to modify the start commands so that the Java system property `-Dable.prefdir=` points to the proper place. If you want to run with security on, you should treat this file as a secure resource and move it to a secure place.)
- Modified your `java.policy` file, if needed
- If security is on, obtained a valid Kerberos ticket for the principal you log in as

Now you are ready to start the platform. Again, for every system that will participate in the platform, you will need to open a command prompt, change to ABLE's `bin` directory and run the **startPlatform** command (think of it as `startAgentPool`), supplying as a parameter the port number configured to be used on that particular system. If a system will run more than one Agent Pool (JVM), each with a separate port number, you will need to open additional command prompts and run the `startPlatform` command specifying that Agent Pool's configured port number each time. For example, if you want to run two Agent Pools on one system, you will open up two separate command windows, and in each window you will run one of the example commands below (your port numbers might be different):

```
startPlatform 55551
startPlatform 55552
```

**NOTE that before you start any other JVM** you must first start the JVM that will contain the **Agent Naming Service**.

Once all bits of the platform are up and running, you can start your agents running.

## Using the ABLE Platform Console to Run Platform Agents

As of ABLE 2.3.0, ABLE provides a platform console that you can use to create, run and otherwise manage agents running in your distributed agent platform. Once you have defined your platform and started your Agent Pools running as described in this document, you can fire up the ABLE Platform Console, connect it to your platform, and then use it to manage the platform and agents running in it. Of course, for the console to successfully interact with your agents, you need to package them up in a JAR file, install that JAR file in the console's `lib` directory, and modify the console's `plugin.xml` file accordingly. The console is separately downloadable and documented in its own package.

## An Example Platform Agent

It is a simple matter to create an agent (derived from `AblePlatformDefaultAgent`) that connects to the Agent Lifecycle Service and creates other agents through that service. An agent can also initialize, suspend, resume, and quit agents through the Agent Lifecycle Service. Using the Agent Directory Service, an agent can find out which other agents are registered and running in the platform.

The following example code shows how an agent derived from `AblePlatformDefaultAgent` can interact with the platform's agent services to perform the tasks described above. Note that this agent is not designed to be created and run from the ABLE Platform Console; its purpose is only to show useful code snippets.

```
//=====
// Imports
//=====
import java.rmi.RemoteException;
import java.util.Vector;

import javax.agent.AgentName;
import javax.agent.JasConstants;
import javax.agent.Locator;
import javax.agent.service.directory.AgentDescription;

import com.ibm.able.*;
import com.ibm.able.platform.*;

/**
 * ExampleAgent is a self-initializing agent that interacts with Agent
 * Lifecycle Service to create, in various agent pools, instances of
 * agents defined in the <code>ableplatform.preferences</code> file.
 *
 */
public class ExampleAgent extends AblePlatformDefaultAgent
    implements AblePlatformServiceEventListener {
```

```
/**
 * Creates, configures, and initializes an ExampleAgent agent that
 * shows how to connect to the Agent Lifecycle Service and create
 * and initialize an instance of another agent in some other agent
 * pool.
 */
public ExampleAgent() throws RemoteException {

    // Standalone agents must specify which agent pool to associate with
    AblePlatform.Preferences.setLocalAgentPool("AgentPool1");

    // -----
    // IF SECURITY IS ON (and because this is an agent that runs from
    // the command line outside of a platform Agent Pool) this agent
    // must set the principal that is running this agent/agent pool.
    // -----
    if ( Able.Preferences.isSecure() ) {
        myPrincipal = "al/borland.binford.com@KERB5REALM.BINFORD.COM";
    }

    // Configure this instance. Only a few agent attributes are shown,
    // but many more can be changed to control the agent's behavior.
    setName          ("ExampleAgent");
    setComment        ("Example agent program.");
    setAgentSummary   (new AbleMessageContainer("Just getting initialized"));
    setAgentType      ("Binford_Agent");
    addJasAgentAttribute("some-arbitrary-user-attribute",    "attribute 1 value");
    addJasAgentAttribute("another-arbitrary-user-attribute", "attribute two value");

    // Initialize this instance. Note that when this method completes,
    // the agent will have obtained a unique AgentName from the Agent
    // Naming Service and will have registered an AgentDescription
    // with the Agent Directory Service. The agent is ready to receive
    // JAS Transport Messages, ABLE events, and do whatever else it is
    // configured to do.
    _init();
}

// This agent registers as a platform Service Event Listener with
// several agent services. When those services send out Service
// Events, this method is where they will appear. This method is
// from the AblePlatformServiceEventListener interface.
public void handleAblePlatformServiceEvent(AblePlatformServiceEvent theServiceEvent)
throws RemoteException {
    System.out.println("\n ExampleAgent received a platform service event: <" +
theServiceEvent.getEventDescription() + ">. \n\n");
}

/**
 * If security is on, this local helper method generates an
 * AbleSecureKey to be used on a subsequent secure method
 * call. Otherwise, null is returned. "mySecSppt" is an
 * AbleSecuritySupport object built-in to the
 * AblePlatformDefaultAgent, but the object is null unless security
 * is on.
 */
private AbleSecureKey secureKeyOrNull() throws Exception {
    return ((Able.Preferences.isSecure())? mySecSppt.generateKey() : null);
}
```

```

//=====
//
// Mainline logic
//
// This method uses Agent Lifecycle Service to create an instance of
// an agent defined in the ableplatform.preferences file. The agent
// is created in its first eligible agent pool, and with overridden
// constructor parameters.
//
// The Agent Directory Service is used to locate the agent, and the
// Message Transport System is used to send the agent a message.
//
// Remote methods are invoked on the agent, and the Agent Lifecycle
// Service is used to terminate the agent.
//
//=====
private void runExampleAgent() throws Exception {

    AgentLifecycleService      agentLifecycleService      = null;
    AbleAgentClassDescription[] agentsDefinedInPreferences = null;

    // AblePlatformDefaultAgent has built-in access to the Agent
    // Naming Service, the Agent Directory Service, and the Message
    // Transport System, but not the Agent Lifecycle Service, so
    // acquire the Agent Lifecycle Service from the JAS Service Root.
    agentLifecycleService = (AgentLifecycleService)myJasServiceRoot.getService
(com.ibm.able.platform.AgentLifecycleService.SERVICE_TYPE);
    if (agentLifecycleService==null) return; // Simply return on any error

    // Register as a service event listener with some of the agent services.
    VerifiableAgentDirectoryService directoryService =
(VerifiableAgentDirectoryService)myJasAgentDirectoryService;
    VerifiableAgentNamingService    namingService    =
(VerifiableAgentNamingService)myJasAgentNamingService;

    agentLifecycleService.addAblePlatformServiceEventListener(this);
    directoryService.addAblePlatformServiceEventListener(this);
    namingService.addAblePlatformServiceEventListener(this);

    // Ask Agent Lifecycle Service for a list of agents that have been
    // defined in the ableplatform.preferences file. These are the
    // agents that Lifecycle is permitted to create.
    agentsDefinedInPreferences = agentLifecycleService.getPermittedAgents();
    if (agentsDefinedInPreferences==null) return;

    // In all the agents defined in preferences, look for an agent
    // defined with the alias of "TimAgent". This is the agent to
    // create later.
    String agentAlias = "TimAgent";
    AbleAgentClassDescription foundAgentClassDescription = null;
    for (int i=0; i<agentsDefinedInPreferences.length; i++) {
        AbleAgentClassDescription testAgentClassDescription =
agentsDefinedInPreferences[i];
        String testAgentAlias = testAgentClassDescription.getAgentAlias();
        if ( agentAlias.equals(testAgentAlias) ) {
            foundAgentClassDescription = testAgentClassDescription;
            break;
        }
    }
    if (foundAgentClassDescription==null) return;

```

```
// A preference description of the "TimAgent" was found. Save
// information about the agent; for example, find what Agent Pools
// the agent is allowed to run in, and select the first Agent Pool.
Vector eligibleAgentPools =
foundAgentClassDescription.getEligibleAgentPools();
AblePlatformPreferences.AgentPoolEntry_ agentPoolInfo =
(AblePlatformPreferences.AgentPoolEntry_)eligibleAgentPools.elementAt(0);
String agentPoolAlias =
agentPoolInfo.getAgentPoolAlias();

// The constructor of the (fictitious) TimAgent takes 3
// parameters: a String (which will become the agent's display
// name), a Long, and a Double. The ableplatform.preferences file
// supplies default values for these parameters, but they are
// overridden here with new values.
String agentDisplayName = "DisplayNameOfTimAgent";
Vector newCtorArgs = new Vector();
newCtorArgs.addElement(new String (agentDisplayName));
newCtorArgs.addElement(new Long (3000L));
newCtorArgs.addElement(new Integer(Able.ProcessingEnabled_PostingEnabled));
foundAgentClassDescription.setConstructorArgs(newCtorArgs);

// Have the Agent Lifecycle Service create and initialize an
// instance of the TimAgent.
AblePlatformAgent timAgent =
agentLifecycleService.createAgentInstanceAndInitialize(foundAgentClassDescription,
agentPoolAlias, secureKeyOrNull());

// Use the Agent Directory Service to look up the agent just
// created. ("lookupAgent" is a built-in method in
// AblePlatformDefaultAgent that uses the Agent Directory
// Service.) Save the newly created agent's unique AgentName and
// also all its JAS Locators.
AgentDescription[] timAgent_AgentDescription =
lookupAgent(JasConstants.AGENT_DISPLAY_NAME, agentDisplayName);
AgentName timAgent_AgentName =
timAgent_AgentDescription[0].getAgentName();
Locator[] timAgent_LocatorList =
timAgent_AgentDescription[0].getLocators();

// Get the RMI Locator for the agent. This Locator, and the
// AgentName saved above, will be used to send the agent a JAS
// Transport Message.
Locator timAgent_RmiLocator = null;
for (int i=0; i<timAgent_LocatorList.length; i++) {
    if ( timAgent_LocatorList[i].getType().equalsIgnoreCase("rmi") ) {
        timAgent_RmiLocator = timAgent_LocatorList[i];
        break;
    }
}

// Send the agent a JAS Transport Message. Again, the method is a
// built-in of AblePlatformDefaultAgent. Note that instead of a
// String, any Java object can be sent as a "message".
sendTransportMessage(timAgent_RmiLocator, timAgent_AgentName, "Hello, Tim Agent");

// Because AblePlatformDefaultAgents are remote objects, their
// methods can be accessed directly. Here, events are sent to the
// agent, and the agent is suspended and resumed.
timAgent.handleAbleEvent( new AbleEvent(this, null) ); // Send the agent an event
timAgent.suspendAgent( secureKeyOrNull() ); // Suspend the agent
```

```
        for (int i=0; i<20; i++) {timAgent.handleAbleEvent( new AbleEvent(this, null)
    );} // Queue up many events for the agent
        timAgent.resumeAgent( secureKeyOrNull() ); // Resume the agent

        // Use the Agent Directory Service to find the agent again, and
        // once found, use the Agent Lifecycle Service to terminate the
        // agent. Note that the agent's "quit" method can be called
        // directly, just like the suspend and resume methods above, but
        // for demonstration purposes Lifecycle is used.
        AgentDescription foundAgentDescription[] =
lookUpAgent(JasConstants.AGENT_DISPLAY_NAME, agentDisplayName);
        agentLifecycleService.quitAgent(foundAgentDescription[0], secureKeyOrNull());

        // Deregister as a service event listener with the same services
        agentLifecycleService.removeAblePlatformServiceEventListener(this);
        directoryService.removeAblePlatformServiceEventListener(this);
        namingService.removeAblePlatformServiceEventListener(this);
    }
```

```
//=====
//
// MAIN Method
//
//=====
/**
 * Creates an instance of an ExampleAgent agent, and calls its
 * mainline routine, runExampleAgent().
 *
 * @param args
 *       are all ignored....
 */
public static void main(String args[]) {

    // Start ABLE's message and trace logging
    Able.startMessageAndTraceLogging();

    // Kickoff an instance of this agent. ExampleAgents are
    // self-customizing and self-initializing, which means that when
    // the constructor is finished, the agent instance will have a
    // unique AgentName and will have registered its AgentDescription
    // with the Agent Directory Service.
    try {
        ExampleAgent anExampleAgent = new ExampleAgent();
        anExampleAgent.runExampleAgent();
    }
    catch(Exception theException) {
        theException.printStackTrace();
        System.out.println("\n++ExampleAgent.main(): Caught exception: " +
theException);
    }

    System.exit(0);
}
}
```