

|   |    |
|---|----|
| 1. LISA Agent Guide   | 2  |
| 1.1 LISA Agent Overview   | 2  |
| 1.2 LISA Agent Architecture                                       | 2  |
| 1.2.1 Agent Components and Data Flow                              | 2  |
| 1.2.2 Agent Database Schema                                       | 3  |
| 1.2.3 Open Wire Format  | 8  |
| 1.2.4 Required Files for Agent Deployment                         | 15 |
| 1.2.5 Agent Algorithms  | 15 |
| 1.3 LISA Agent Installation                                       | 16 |
| 1.3.1 Installing the Native Agent                                 | 16 |
| 1.3.2 Installing the Pure Java Agent                              | 17 |
| 1.3.3 Agent Downloads   | 17 |
| 1.4 LISA Agent Platforms  | 18 |
| 1.4.1 Configuring the Agent on BEA Oracle WebLogic                | 18 |
| 1.4.2 Configuring the Agent on IBM WebSphere                      | 19 |
| 1.4.3 Configuring the Agent on Sun GlassFish                      | 22 |
| 1.4.4 Configuring the Agent on TIBCO BusinessWorks                | 23 |
| 1.4.5 Configuring the Agent on Tomcat, JBoss, Geronimo, and Resin | 24 |
| 1.4.6 Configuring the Agent on WebMethods IS                      | 24 |
| 1.5 LISA Agent User Guide   | 24 |
| 1.5.1 Agent Getting Started                                       | 25 |
| 1.5.2 Agent Configuration Properties                              | 26 |
| 1.5.3 Agent Data Categories                                       | 28 |
| 1.5.4 Using the Dev Console                                       | 28 |
| 1.5.4.1 Viewing the Network Summary                               | 29 |
| 1.5.4.2 Viewing Transactions                                      | 30 |
| 1.5.4.3 Viewing the Call Tree                                     | 32 |
| 1.5.4.4 Viewing Agent Database Information                        | 33 |
| 1.5.4.5 Viewing Agent Logging                                     | 34 |
| 1.5.4.6 Viewing Agent Properties                                  | 34 |
| 1.5.4.7 Viewing Classes   | 35 |
| 1.5.4.8 Viewing Threads   | 36 |
| 1.5.4.9 Managing Files  | 37 |
| 1.5.4.10 Accessing a Remote Terminal                              | 37 |
| 1.5.4.11 Creating Agent Extensions                                | 38 |
| 1.5.4.12 Using the Virtual Service Playground                     | 39 |
| 1.5.5 Dev Console Videos  | 41 |
| 1.5.6 Developing Against the Agent                                | 42 |
| 1.5.6.1 Agent General APIs  | 42 |
| 1.5.6.2 Agent Discovery APIs                                      | 44 |
| 1.5.6.3 Agent Transaction APIs                                    | 48 |
| 1.5.6.4 Agent VSE APIs  | 52 |
| 1.5.6.5 Agent LEK APIs  | 55 |
| 1.5.6.6 Agent API Examples  | 58 |
| 1.5.7 Agent Custom Extensions                                     | 59 |
| 1.5.8 Agent Transaction Weight                                    | 62 |
| 1.5.9 Load Balancers and Native Web Servers                       | 63 |
| 1.6 LISA Agent Troubleshooting                                    | 63 |

# LISA Agent Guide

The LISA Agent is a piece of server-side technology that can be installed inside any Java process (including Java EE containers) and allows LISA to control and monitor server-side activities.

This guide contains the following chapters:

[LISA Agent Overview](#)  
[LISA Agent Architecture](#)  
[LISA Agent Installation](#)  
[LISA Agent Platforms](#)  
[LISA Agent User Guide](#)  
[LISA Agent Troubleshooting](#)  
  
[LISA Agent Client Javadoc](#)

## LISA Agent Overview

The LISA Agent is a piece of server-side technology that can be installed inside any Java process (including Java EE containers) and allows LISA to control and monitor server-side activities.

In a nutshell, it can do what most profilers do (monitor loaded classes/objects, CPU usage, memory usage, threads, track method calls, and so on) but works across multiple JVMs and is used with LISA to bring unique features to the testing game.

In particular, it can give you visibility into what each test or even test step causes the server(s) to do behind the scenes so as to identify bugs and bottlenecks. This capability is similar to what Pathfinder does, but works across all protocols used by Java applications without the need to instrument any code or even any configuration files.

Another target area for the Agent is supporting VSE. By enabling record and replay of traffic and/or method calls across protocols, it gives VSE complete control over any areas of the target application that users may want to virtualize, and provides a unified framework to accomplish this regardless of protocol.

## LISA Agent Architecture

This chapter contains the following sections:

[Agent Components and Data Flow](#)  
[Agent Database Schema](#)  
[Open Wire Format](#)  
[Required Files for Agent Deployment](#)  
[Agent Algorithms](#)

## Agent Components and Data Flow

The LISA Agent is composed of the following distinct components:

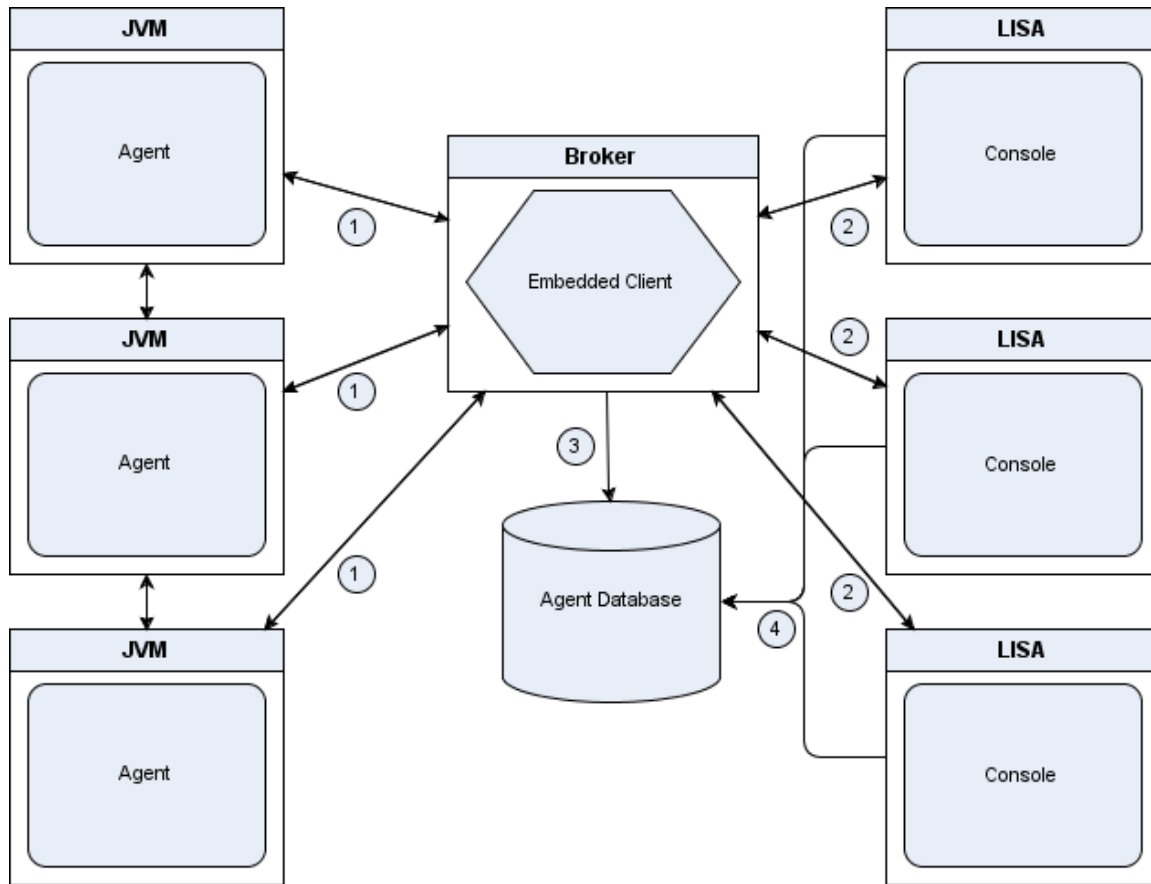
- The agent itself, which runs embedded in a Java process
- The JMS broker with an embedded JMS client
- The consoles (or clients)
- The database

The **Agent** captures data for LISA Pathfinder and LISA Virtualize. It also exposes APIs that can be invoked from the consoles.

The **Broker** is a hub that dispatches JMS messages between agents, consoles, and the embedded client. The embedded client keeps track of network wide/agent-spanning properties, such as the set of agents, their open queues or current network connections. As such it can assemble partial transaction data into full transactions for the benefit of consoles. After the data is assembled, it is sent to the consoles (short term) and persisted to the database for long-term storage. From this point on, we will ignore the distinction and refer to the embedded client as the broker.

The **Consoles** get their finalized data from the broker (if recent) and the database (if older) for display and user interaction.

The following diagram shows the components and how they interact.



The flow of data is fully specified by the following JMS destinations:

The **lisa.agent.info** topic is carried over connections 1 and 2, produced by the agents and consumed by the broker and the consoles. This gives the broker and the consoles a view of which agents are currently online and what their basic properties are.

The **lisa.agent.port** topic is carried over connection 1, produced by the agents and consumed by the broker. This gives the broker a view of the connections currently active between multiple agents.

The **lisa.agent.api** topic is carried over connections 1 and 2, produced by the consoles and consumed (and replied to) by the agents. This allows the consoles to invoke agent APIs over JMS.

The **lisa.broker.api** topic is carried over connection 2, produced by the consoles and consumed (and replied to) by the broker. This allows the consoles to invoke broker APIs over JMS.

The **lisa.stats** topic is carried over connections 1 and 2, produced by the agents and consumed by the broker and the consoles. This gives the consoles an idea of what kind of load the agents are currently under and lets the broker persist those to the database.

The **lisa.vse** topic is carried over connections 1 and 2, produced by the agents and consumed by the consoles. When VSE is turned on, the consoles receive VSE frames (and reply to them in playback mode).

The **lisa.tx.partial** queue is carried over connection 1, produced by the agents and consumed by the broker. When an agent has captured a partial transaction (that is, all the frames that happen in its JVM), the agent sends it to the broker for assembly.

The **lisa.tx.full** topic is carried over connection 2, produced by the broker and consumed by the consoles. When the broker is done assembling partial transactions received over **lisa.tx.partial**, the broker sends the full transactions to the consoles.

The **lisa.tx.incomplete** topic is carried over connection 2, produced by the broker and consumed by the consoles. This topic is similar to **lisa.tx.full**, but is used for transactions that could not be fully completed within the allowed timeout.

**JDBC** connection 3 is used when the broker saves **StatsFrame** objects or fully assembled **TransactionFrame** objects.

**JDBC** connection 4 is used by the consoles to perform their queries for transactions or statistics that are no longer held in memory.

## Agent Database Schema

The Agent database schema is created automatically by the broker. However, if you need to create it manually (for security or process reasons, or even migration or documentation), you can get the DDL statements by running the following command:

```
java -jar LisaAgent.jar -ddl
```

The output of this command is as follows:

```
create table LISA_AGENT_VERSION (  
    SCHEMA_VERSION int);  
  
create table LISA_AGENT (  
    AGENT_ID bigint not null,  
    AGENT_NAME varchar(128),  
    MACHINE varchar(256),  
    IP varchar(32),  
    WORKINGDIR varchar(256),  
    CLASSPATH varchar(1024),  
    LIBPATH varchar(1024),  
    MAINCLASS varchar(128),  
    VERSION varchar(32),  
    FLAGS int,  
    primary key (AGENT_ID));  
  
create table LISA_STATISTICS (  
    AGENT_ID bigint not null,  
    TIME bigint not null,  
    CPU int,  
    HEAP bigint,  
    NON_HEAP bigint,  
    IO_IN bigint,  
    IO_OUT bigint,  
    GC_COUNT bigint,  
    GC_TIME bigint,  
    foreign key (AGENT_ID) references LISA_AGENT(AGENT_ID));  
  
create table LISA_TRANSACTION_FRAME (  
    FRAME_ID varchar(64) not null,  
    PARENT_ID varchar(64),
```

```

TRANSACTION_ID varchar(64) not null,

AGENT_ID bigint not null,

THREAD_NAME varchar(256),

CLASSNAME varchar(128),

METHOD varchar(128),

SIGNATURE varchar(256),

MODIFIERS bigint,

SOURCE varchar(256),

ARGS varchar(1024),

RESULT varchar(256),

SESSION_ID varchar(128),

TIME bigint not null,

ORDINAL bigint not null,

CLOCK_DURATION bigint,

CPU_DURATION bigint,

LEK_INFO varchar(256),

CATEGORY int,

LOCAL_IP varchar(16),

LOCAL_PORT int,

REMOTE_IP varchar(16),

REMOTE_PORT int,

FLAGS bigint,

COMPLEXITY bigint,

primary key (FRAME_ID),

foreign key (AGENT_ID) references LISA_AGENT(AGENT_ID));

create table LISA_FRAME_DATA (

FRAME_ID varchar(64) not null,

STATE text,

REQUEST text,

RESPONSE text,

CDATA text,

foreign key (FRAME_ID) references LISA_TRANSACTION_FRAME(FRAME_ID) on delete cascade);

create table LISA_VSE_FRAME (

```

```

FRAME_ID varchar(64) not null,

AGENT_ID bigint not null,

SRC_ID varchar(64),

SESSION_ID varchar(128),

THREAD_NAME varchar(256),

CLASSNAME varchar(128),

METHOD varchar(128),

TIME bigint not null,

CLOCK_DURATION bigint,

FLAGS int,

primary key (FRAME_ID),

foreign key (AGENT_ID) references LISA_AGENT(AGENT_ID));


create table VSE_FRAME_DATA (

FRAME_ID varchar(64),

TYPE int,

ORDINAL int,

DATA text,

foreign key (FRAME_ID) references LISA_VSE_FRAME(FRAME_ID) on delete cascade)


create table LISA_TICKET(

TICKET_ID varchar(64) not null,

SESSION_ID varchar(128),

FRAME_ID varchar(64),

CAPTURE_TIME bigint not null,

DEFECT_ID varchar(64),

CUSTOMER_ID varchar(64),

REPORTER_ID varchar(64),

REPORTER_EMAIL varchar(128),

STATUS int,

SEVERITY int,

TITLE varchar(256),

COMPONENT varchar(128),

DESCRIPTION text,

SCREENSHOT text);

```

```
create index IDX_AGENT on LISA_TRANSACTION_FRAME (AGENT_ID);

create index IDX_TRANSACTION on LISA_TRANSACTION_FRAME (TRANSACTION_ID);

create index IDX_PARENT on LISA_TRANSACTION_FRAME (PARENT_ID);

create index IDX_TIME on LISA_TRANSACTION_FRAME (TIME, ORDINAL);

create index IDX_FLAGS on LISA_TRANSACTION_FRAME (FLAGS);

create index IDX_SESSION on LISA_TRANSACTION_FRAME (SESSION_ID);

create index IDX_ADDRESSES on LISA_TRANSACTION_FRAME(LOCAL_IP, LOCAL_PORT, REMOTE_IP,
REMOTE_PORT);

create index IDX_CATEGORY on LISA_TRANSACTION_FRAME (CATEGORY);

create index IDX_TDATA on LISA_FRAME_DATA (FRAME_ID);

create index IDX_SAGENT on LISA_STATISTICS (AGENT_ID);

create index IDX_STIME on LISA_STATISTICS (TIME);

create index IDX_VTIME on LISA_VSE_FRAME (TIME);

create index IDX_VAGENT on LISA_VSE_FRAME (AGENT_ID);

create index IDX_VMETHOD on LISA_VSE_FRAME (CLASSNAME, METHOD)

create index IDX_VDATA on VSE_FRAME_DATA (FRAME_ID);
```

```
insert into LISA_AGENT_VERSION (SCHEMA_VERSION) values (1);
```

## Open Wire Format

The open wire format used to communicate between agents and broker is [OpenWire](#) in combination with XML formats for all the objects mentioned previously, as specified in the following schemas.

### AGENT\_INFO



```

<?xml version="1.0" encoding="utf-8"?>

<xs:schema attributeFormDefault="unqualified"

    elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <!-- AgentInfo -->

    <xs:element name="ai">

        <xs:complexType>

            <!-- Agent Guid -->

            <xs:attribute name="id" type="xs:unsignedByte" use="required" />

            <!-- Agent Name -->

            <xs:attribute name="n" type="xs:string" use="required" />

            <!-- Agent Machine -->

            <xs:attribute name="m" type="xs:string" use="required" />

            <!-- Agent IP Address -->

            <xs:attribute name="ip" type="xs:string" use="required" />

            <!-- Agent Version -->

            <xs:attribute name="v" type="xs:string" use="required" />

            <!-- Agent Class Path -->

            <xs:attribute name="cp" type="xs:string" use="optional" />

            <!-- Agent Library Path -->

            <xs:attribute name="lp" type="xs:string" use="optional" />

            <!-- Agent Working Dir -->

            <xs:attribute name="wd" type="xs:string" use="optional" />

            <!-- Agent Main Class -->

            <xs:attribute name="mc" type="xs:string" use="optional" />

            <!-- Agent Flags -->

            <xs:attribute name="f" type="xs:unsignedByte" use="required" />

            <!-- Agent Runtime Flags -->

            <xs:attribute name="rf" type="xs:unsignedByte" use="required" />

        </xs:complexType>

    </xs:element>

</xs:schema>

```

STATS\_FRAME

```

<?xml version="1.0" encoding="utf-8"?>

<xs:schema attributeFormDefault="unqualified"

    elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <!-- StatsFrame -->

    <xs:element name="stf">

        <xs:complexType>

            <!-- Frame Id -->

            <xs:attribute name="aid" type="xs:unsignedInt" use="required" />

            <!-- Time -->

            <xs:attribute name="t" type="xs:unsignedLong" use="required" />

            <!-- Cpu Usage -->

            <xs:attribute name="cpu" type="xs:unsignedByte" use="optional" />

            <!-- Heap Memory Usage -->

            <xs:attribute name="h" type="xs:unsignedInt" use="optional" />

            <!-- Non Heap Memory Usage -->

            <xs:attribute name="nh" type="xs:unsignedByte" use="optional" />

            <!-- IO Input -->

            <xs:attribute name="ioi" type="xs:unsignedByte" use="optional" />

            <!-- IO Output -->

            <xs:attribute name="ioo" type="xs:unsignedByte" use="optional" />

            <!-- GC count -->

            <xs:attribute name="gcc" type="xs:unsignedByte" use="optional" />

            <!-- GC Time -->

            <xs:attribute name="gct" type="xs:unsignedByte" use="optional" />

        </xs:complexType>

    </xs:element>

</xs:schema>

```

## TRANSACTION\_FRAME

```

<?xml version="1.0" encoding="utf-8"?>

<xs:schema attributeFormDefault="unqualified"

    elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <!-- TransactionFrame -->

    <xs:element name="tf">

```

```

<xs:complexType>

  <xs:sequence>

    <!-- Thread Name -->

    <xs:element name="tn" type="xs:string" use="optional" />

    <!-- Frame Object -->

    <xs:element name="src" type="xs:string" use="optional" />

    <!-- Method Arguments -->

    <xs:element name="args" type="xs:string" use="optional" />

    <!-- Method Return -->

    <xs:element name="ret" type="xs:string" use="optional" />

    <!-- Frame State -->

    <xs:element name="state" type="xs:string" use="optional" />

    <!-- Frame Request -->

    <xs:element name="req" type="xs:string" use="optional" />

    <!-- Frame Response -->

    <xs:element name="resp" type="xs:string" use="optional" />

    <!-- LEK Data -->

    <xs:element name="lek" type="xs:string" use="optional" />

    <!-- Children TransactionFrame -->

    <xs:element name="tf" type="tf" minOccurs="0" maxOccurs="unbounded"/>

  </xs:sequence>

  <!-- Frame Id -->

  <xs:attribute name="id" type="xs:string" use="required" />

  <!-- Parent Frame Id -->

  <xs:attribute name="pid" type="xs:string" use="required" />

  <!-- Transaction Id -->

  <xs:attribute name="tid" type="xs:string" use="required" />

  <!-- Agent Id -->

  <xs:attribute name="aid" type="xs:unsignedInt" use="required" />

  <!-- Session Id -->

  <xs:attribute name="sid" type="xs:string" use="optional" />

  <!-- Frame Category -->

  <xs:attribute name="cat" type="xs:unsignedByte" use="required" />

  <!-- Frame Flags -->

  <xs:attribute name="f" type="xs:unsignedByte" use="required" />

```

```
<!-- Frame Class -->

<xs:attribute name="c" type="xs:string" use="required" />

<!-- Frame Method -->

<xs:attribute name="m" type="xs:string" use="required" />

<!-- Frame Method Signature -->

<xs:attribute name="sig" type="xs:string" use="required" />

<!-- Frame Method Modifiers -->

<xs:attribute name="mods" type="xs:unsignedByte" use="required" />

<!-- Time -->

<xs:attribute name="t" type="xs:unsignedLong" use="required" />

<!-- Ordinal -->

<xs:attribute name="ord" type="xs:unsignedByte" use="required" />

<!-- Clock Duration -->

<xs:attribute name="cd" type="xs:unsignedInt" use="required" />

<!-- CPU Duration -->

<xs:attribute name="cpu" type="xs:unsignedInt" use="required" />

<!-- Local IP Address -->

<xs:attribute name="lip" type="xs:string" use="optional" />

<!-- Local Port -->

<xs:attribute name="lp" type="xs:unsignedShort" use="optional" />

<!-- Remote IP Address -->

<xs:attribute name="rip" type="xs:string" use="optional" />

<!-- Remote Port -->

<xs:attribute name="rp" type="xs:unsignedShort" use="optional" />

<!-- Complexity -->

<xs:attribute name="cx" type="xs:unsignedLong" use="optional" />

</xs:complexType>
```

```
</xs:element>

</xs:schema>
```

## TICKET

```
<?xml version="1.0" encoding="utf-8"?>

<xs:schema attributeFormDefault="unqualified"

  elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <!-- Ticket -->

  <xs:element name="ti">

    <xs:complexType>

      <xs:sequence>

        <!-- Component -->

        <xs:element name="cmp" type="xs:string" use="optional" />

        <!-- Title -->

        <xs:element name="ttl" type="xs:string" use="required" />

        <!-- Description -->

        <xs:element name="desc" type="xs:string" use="optional" />

        <!-- Screenshot data (BASE64 encoded) -->

        <xs:element name="ss" type="xs:string" use="optional" />

      </xs:sequence>

      <!-- Ticket Id -->

      <xs:attribute name="id" type="xs:string" use="required" />

      <!-- Session Id -->

      <xs:attribute name="sid" type="xs:string" use="required" />

      <!-- Frame Id -->

      <xs:attribute name="fid" type="xs:string" use="required" />

      <!-- Time -->

      <xs:attribute name="t" type="xs:unsignedLong" use="required" />

      <!-- Defect Id -->

      <xs:attribute name="did" type="xs:unsignedShort" use="optional" />

      <!-- Customer Id -->

      <xs:attribute name="cid" type="xs:string" use="optional" />

      <!-- Reporter Id -->

      <xs:attribute name="rid" type="xs:string" use="optional" />

    </xs:complexType>

  </xs:element>

</xs:schema>
```

```
<!-- Reporter Email -->

<xs:attribute name="e" type="xs:string" use="required" />

<!-- Status -->

<xs:attribute name="st" type="xs:unsignedByte" use="required" />

<!-- Severity -->

<xs:attribute name="sev" type="xs:unsignedByte" use="required" />

</xs:complexType>
```

```

    </xs:element>

</xs:schema>

```

## VSE\_FRAME

```

<?xml version="1.0" encoding="utf-8"?>

<xs:schema attributeFormDefault="unqualified"
  elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <!-- VSE Frame -->

  <xs:element name="vf">

    ...

  </xs:element>

</xs:schema>

```

## Required Files for Agent Deployment

The following files are required for the JVM Agent:

- JavaBinder.xxx
- LisaAgent.jar
- rules.properties

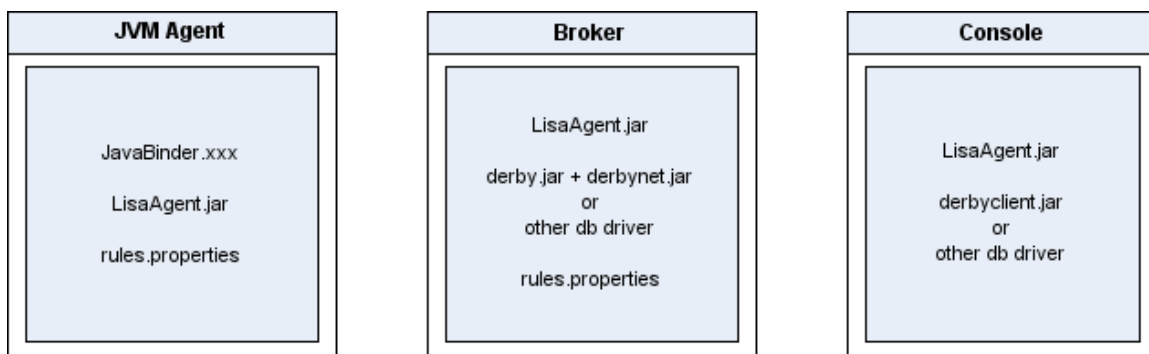
The following files are required for the Broker:

- LisaAgent.jar
- derby.jar and derbynet.jar, or other database driver
- rules.properties

The following files are required for the Console:

- LisaAgent.jar
- derbyclient.jar or other database driver

This diagram shows a graphical representation of the required files.



## Agent Algorithms

The startup order of agents, broker, and consoles has no relevance because all communications are asynchronous and can reestablish themselves automatically. This is especially important for agents to not have their performance suffer because the broker goes down for example.

When an agent comes online, it starts pulsing its information over the **lisa.agent.info** topic at regular (short) intervals. For more information about the **lisa.agent.info** topic, see [Agent Components and Data Flow](#).

If no broker is available, then it will not try to notify any listeners of anything else until a connection is (re)established.

If the broker is available, then all interested parties will become quickly notified of which agents are online. The broker and consoles keep a running list of those agents so that when they stop pulsing their information, they are expired and removed from the list after a while.

## LISA Agent Installation

This chapter describes how to install the LISA Agent.

If you use Java 1.4, you must install the native agent.

If you use Java 1.5 or newer, install the pure Java agent unless you need the superstacks feature or heap walking.

This chapter contains the following sections:

[Installing the Native Agent](#)  
[Installing the Pure Java Agent](#)  
[Agent Downloads](#)

## Installing the Native Agent



**Important:** It is critical you find out about the target environment ahead of time and make sure it has been tested, or test it yourself. Most issues will be operating system or JVM-dependent, not application-dependent. Then carefully follow the instructions supplied in this documentation. If that does not help, look at the [troubleshooting guide](#) before contacting support.

To turn on the LISA Agent on any Java process, follow these steps.

### Step 1: Download Agent Files

Download **LisaAgent.jar** and **(lib)JavaBinder(.dll, .so, .jnilib)** from the links in [Agent Downloads](#).

Place the files anywhere on your disk that has read permissions from your Java process and is not automatically loaded in the application classpath (such as **/WEB-INF/lib** directories).

### Step 2: Define Environment Variable

Start your Java process as you typically do after defining the following environment variable:

On Sun's JRE 1.5+ or JRockit 1.5+:

```
JAVA_TOOL_OPTIONS=-agentpath:<Path to JavaBinder.xxx>[url=<broker url>][,name=<agent name>]
```

On IBM's JRE 1.5+:

```
IBM_JAVA_OPTIONS=-agentpath:<Path to JavaBinder.xxx>[url=<broker broker>][,name=<agent name>]
```

On Sun's JRE 1.4:

```
_JAVA_OPTIONS=-Xbootclasspath/a:<Path to LisaAgent14.jar> -Xdebug -Xnoagent -Djava.compiler=NONE  
-XrunJavaBinder:jar=file:<Path to LisaAgent14.jar>[url=<broker url>][,name=<agent name>]
```

On IBM's JRE 1.4:

```
IBM_JAVA_OPTIONS=-Xbootclasspath/a:<Path to LisaAgent14.jar> -Xdebug -Xnoagent -Djava.compiler=NONE  
-XrunJavaBinderIBM14:jar=file:<Path to LisaAgent14.jar>[url=<broker url>][,name=<agent name>]
```

### Step 3: Start the Broker

Run the following command to start the broker:

```
java -jar LisaAgent.jar -broker [broker url]
```

Setting **JAVA\_TOOL\_OPTIONS** in a shell will apply the agent to all Java processes started in that shell. To avoid this, you can instead supply the



arguments as a JVM option in the command line. For example, many J2EE containers support **JAVA\_OPTS**, as in:  
**JAVA\_OPTS=-agentpath:<Path to JavaBinder.xxx>**



If you specify `-agentpath:...` as a JVM argument (as opposed to an environment variable) then you **MUST** include the `jar` option **in addition** to the `dll`. Something like this:

```
java -agentpath:c:\agent\JavaBinder.dll=jar=c:\agent\LisaAgent.jar,url=tcp://192.168.1.66:2009,name=myAgent
```



With 1.4 JVMs, the native library must be in the **PATH** (or **LD\_LIBRARY\_PATH**).

For more options and troubleshooting tips, see the [Agent User Guide](#).

## Installing the Pure Java Agent

LISA supports a pure Java agent for any JRE 1.5 and above. This agent does not have the full functionality of the native-based agent, but all of Pathfinder and Java Virtualize are available.

To install it, follow the steps in [Installing the Native Agent](#) but replace **(lib)JavaBinder(.dll, .so, .jnilib)** with **LisaAgent2.jar** and in **Step 2**, start your Java process as you typically do after defining the following environment variable:

```
JAVA_TOOL_OPTIONS=-javaagent:<Path to LisaAgent2.jar>[url=<broker url>][,name=<agent name>]
```

## Agent Downloads



The Agent will not work with a native library not compiled for your OS.

Using this download site, you are responsible for backing up the files you overwrite if you need to revert to a previous version later.

| Item   | Link                                     |
|--|--|
| Cross-Platform Java Agent                    | <a href="#">LisaAgent.jar</a>            |
| Cross-Platform Pure Java Agent               | <a href="#">LisaAgent2.jar</a>           |
| Cross-Platform Java Agent for Java 1.4       | <a href="#">LisaAgent14.jar</a>          |
|  |  |
| Windows (32 bit) Native Agent                | <a href="#">JavaBinder.dll</a>           |
| Windows (32 bit) Native Agent (IBM Java 1.4) | <a href="#">JavaBinderIBM14.dll</a>      |
| Mac OSX (x86) 32-bit (Java 1.5) Native Agent | <a href="#">libJavaBinder.x86.jnilib</a> |
| Mac OSX (x64) 64-bit (Java 1.6) Native Agent | <a href="#">libJavaBinder.x64.jnilib</a> |
| Linux (x86) 32-bit Native Agent              | <a href="#">libJavaBinder.x86.so</a>     |

|                                     |  |
|-------------------------------------|--|
| Linux (x64) 64-bit Native Agent     | <a href="#">libJavaBinder.x64.so</a>   |
| Solaris (x86) 32-bit Native Agent   | <a href="#">libJavaBinder.solaris.x86.so</a>   |
| Solaris (x64) 64-bit Native Agent   | <a href="#">libJavaBinder.solaris.x64.so</a>   |
| Solaris (Sparc) 32-bit Native Agent | <a href="#">libJavaBinder.solaris.sparc32.so</a>   |
| Solaris (Sparc) 64-bit Native Agent | <a href="#">libJavaBinder.solaris.sparc64.so</a>   |
|                                     |  |
| Sample configuration file           | <a href="#">rules.properties</a>   |
|                                     |  |
| Agent client for LISA               | <a href="#">lisa-agent.jar</a> , <a href="#">_misc-agent.jar</a>                           |
| Derby db and drivers                | <a href="#">derby.jar</a> , <a href="#">derbynet.jar</a> , <a href="#">derbyclient.jar</a> |
| MySQL drivers                       | <a href="#">mysql.jar</a>  |
| postgres drivers                    | <a href="#">postgresql.jar</a>   |
|                                     |  |
| Native MQ Pathfinder                | NativeMQ_Pathfinder.zip  |

## LISA Agent Platforms

This chapter contains the following sections:

- [Configuring the Agent on BEA Oracle WebLogic](#)
- [Configuring the Agent on IBM WebSphere](#)
- [Configuring the Agent on Sun GlassFish](#)
- [Configuring the Agent on TIBCO BusinessWorks](#)
- [Configuring the Agent on Tomcat, JBoss, Geronimo, and Resin](#)
- [Configuring the Agent on WebMethods IS](#)

## Configuring the Agent on BEA Oracle WebLogic

The default install instructions apply for WebLogic but to follow the officially supported way to add arguments to the JVM, you can go to the WebLogic administration console and specify the agentpath in there:

**ORACLE WebLogic Server® Administration Console**

Home Log Out Preferences Record Help

Welcome, weblogic Connected to: base\_dom

Home > Summary of Servers > AdminServer > Roles > AdminServer

**Settings for AdminServer**

**Configuration** Protocols Logging Debug Monitoring Control Deployments Services Security Notes

General Cluster Services Keystores SSL Federation Services Deployment Migration Tuning Overload Health Monitoring **Server Start**

Save

Node Manager is a WebLogic Server utility that you can use to start, suspend, shut down, and restart servers in normal or unexpected conditions. Use this page to configure the startup settings that Node Manager will use to start this server on a remote machine.

**Java Home:** The Java home directory (path on the machine running Node Manager) to use when starting this server. [More Info...](#)

**Java Vendor:** The Java Vendor value to use when starting this server. For example, BEA, Sun, HP etc. [More Info...](#)

**BEA Home:** The BEA home directory (path on the machine running Node Manager) to use when starting this server. [More Info...](#)

**Root Directory:** The directory that this server uses as its root directory. This directory must be on the computer that hosts the Node Manager. If you do not specify a Root Directory value, the domain directory is used by default. [More Info...](#)

**Class Path:** The classpath (path on the machine running Node Manager) to use when starting this server. [More Info...](#)

**Arguments:** The arguments to use when starting this server. [More Info...](#)

```
-agentpath:c:\itko\main\lisa-remote\dist\agent
/javaBinder.dll=jar:file:c:\itko\main\lisa-remote\dist\agent
\lisaAgent.jar,name=wls -Dlisa.log.level=debug
```

Alternatively, you can edit the `WEBLOGIC_HOME/user_projects/domains/base_domain/config/config.xml` file and add the `-agentpath` property to the `<server>/<server-start>/<arguments>` node of the target server.

## Configuring the Agent on IBM WebSphere

WebSphere is typically started as a service and thus makes it more difficult to set the variable used to turn on the agent. There are multiple ways to accomplish this.

### Using the Admin Console (WebSphere 6.1, 7.0 and 8.0)

You can use the Admin Console.

1. Go to the WebSphere Admin Console (<https://localhost:9043/ibm/console/secure/securelogin.do> by default).
2. Navigate to your WAS Application Server and click Process Definition.

Integrated Solutions Console Welcome admin Help Logout

Views: All tasks

- Welcome
- Guided Activities
- Servers
  - Application servers**
  - Web servers
  - WebSphere MQ servers
- Applications
- Resources
- Security
- Environment
- System administration
- Users and Groups
- Monitoring and Tuning
- Troubleshooting
- Service integration
- UDDI

Runtime Configuration

**General Properties**

Name: server1

Node Name: sauronNode01

☐ Run in development mode

☒ Parallel start

Access to internal server classes: Allow

**Server-specific Application Settings**

ClassLoader policy: Multiple

Class loading mode: Parent first

Apply OK Reset Cancel

**Container Settings**

- Session management
- SIP Container Settings
- Web Container Settings
- Portlet Container Settings
- EJB Container Settings
- Container Services
- Business Process Services

**Applications**

- Installed applications

**Server messaging**

- Messaging engines
- Messaging engine inbound transports
- WebSphere MQ link inbound transports
- SIB service

**Server Infrastructure**

- Java and Process Management
  - Class loader
  - Process Definition**
  - Process Execution

3. Click Java Virtual Machine.

Integrated Solutions Console Welcome admin Help Logout

Views: All tasks

- Welcome
- Guided Activities
- Servers
  - Application servers
  - Web servers
  - WebSphere MQ servers
- Applications
- Resources
- Security
- Environment
- System administration
- Users and Groups
- Monitoring and Tuning
- Troubleshooting
- Service integration
- UDDI

Application servers Close page

**Application servers**

**Application servers > server1 > Process Definition**

Use this page to configure a process definition. A process definition defines the command line information necessary to start or initialize a process.

Configuration

**General Properties**

Executable name:

Executable arguments:

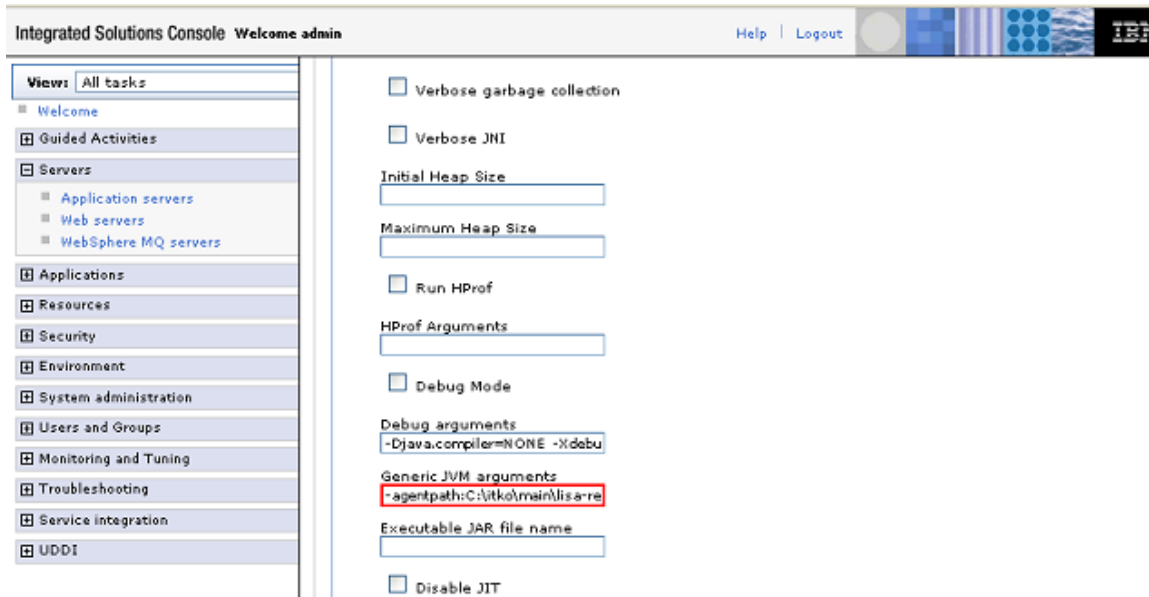
startCommand:

startCommandArgs:

**Additional Properties**

- Java Virtual Machine**
- Environment Entries
- Process Execution
- Process Logs
- Logging and Tracing

4. In the Generic JVM arguments field, enter the agent JVM arguments as described in Installing the Native Agent.



For WAS 7.0 on linux, you can use something like this in the generic JVM arguments:

```
-agentpath:/home/itko/agent/libJavaBinder.x86.so=jar=file:/home/itko/agent/LisaAgent.jar,url=tcp://192.168.1.100:2009,name=RDR
```

## Editing the server.xml File

You can manually edit the `<WebSphere Home>/AppServer/profiles/AppSrv01/config/cells/<machine name>Node01Cell/nodes/<machine name>Node01/servers/server1/server.xml` file.

At the bottom of the file, modify the following entry:

```
<jvmEntries ... genericJvmArguments="-agentpath:<Path to JavaBinder.dll>=jar=file:<Path to LisaAgent.jar>,name=was" .../>
```

## WebSphere 6.0

This version of WAS uses the JRE 1.4.2. Here's one way of getting it to work.

Download JavaBinderIBM14.dll, LisaAgent14.jar and rules.properties. Put all 3 into a new directory c:\agent.

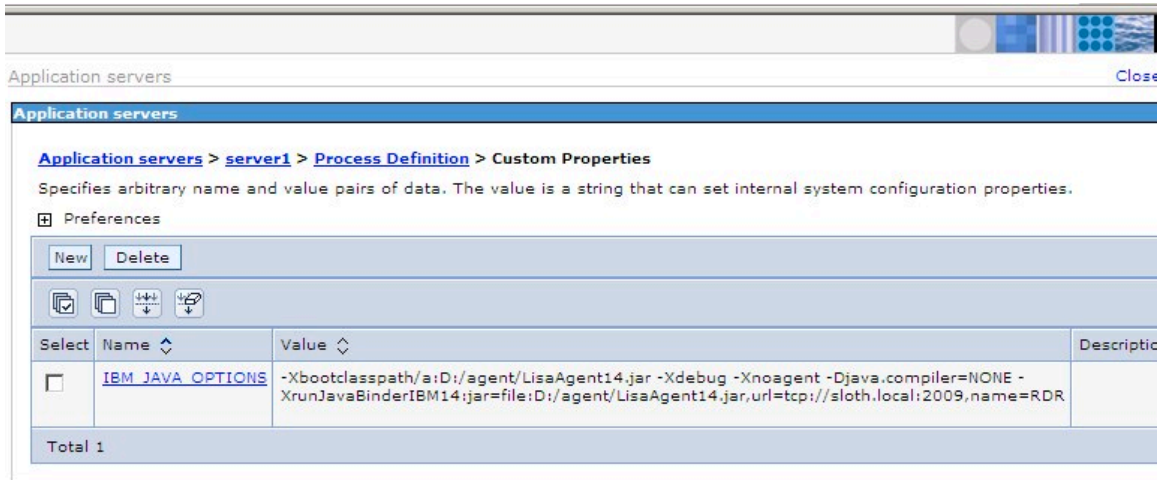
Make sure c:\agent is on the system PATH environment variable - not just the users path.

Restart the machine to verify that the PATH is being picked up by the services manager.

Using the admin console, add a new Environment entry for the server:

```
name="IBM_JAVA_OPTIONS" value="-Xbootclasspath/a:D:/agent/LisaAgent14.jar -Xdebug -Xnoagent -Djava.compiler=NONE -XrunJavaBinderIBM14:jar=file:D:/agent/LisaAgent14.jar,url=tcp://sloth.local:2009,name=RDR"
```

It will take forever to start up but it works.



## Using wsadmin

You can use the **wsadmin** tool. For example:

```
C:\IBM\WebSphere61\AppServer\bin>hostname
cam-aa74651f617

C:\IBM\WebSphere61\AppServer\bin>wsadmin
WASX7209I: Connected to process "server1" on node cam-aa74651f617Node01 using SOAP connector; The
type of process is: UnManagedProcess
WASX7029I: For help, enter: "$Help help"

wsadmin>set server1 [$AdminConfig getid
/Cell:cam-aa74651f617Node01Cell/Node:cam-aa74651f617Node01/Server:server1/ ]
server1(cells/cam-aa74651f617Node01Cell/nodes/cam-aa74651f617Node01/servers/server1|server.xml#Server_1
jvm [$AdminConfig list JavaVirtualMachine $server1]
(cells/cam-aa74651f617Node01Cell/nodes/cam-aa74651f617Node01/servers/server1|server.xml#JavaVirtualMach
modify $jvm genericJvmArguments "-agentpath:<Path to JavaBinder.dll>=jar=file:<Path to
LisaAgent.jar>,name=was"

wsadmin>$AdminConfig save

wsadmin>quit
```

## wsadmin jacl scripts

### Zip of wsadmin jacl scripts

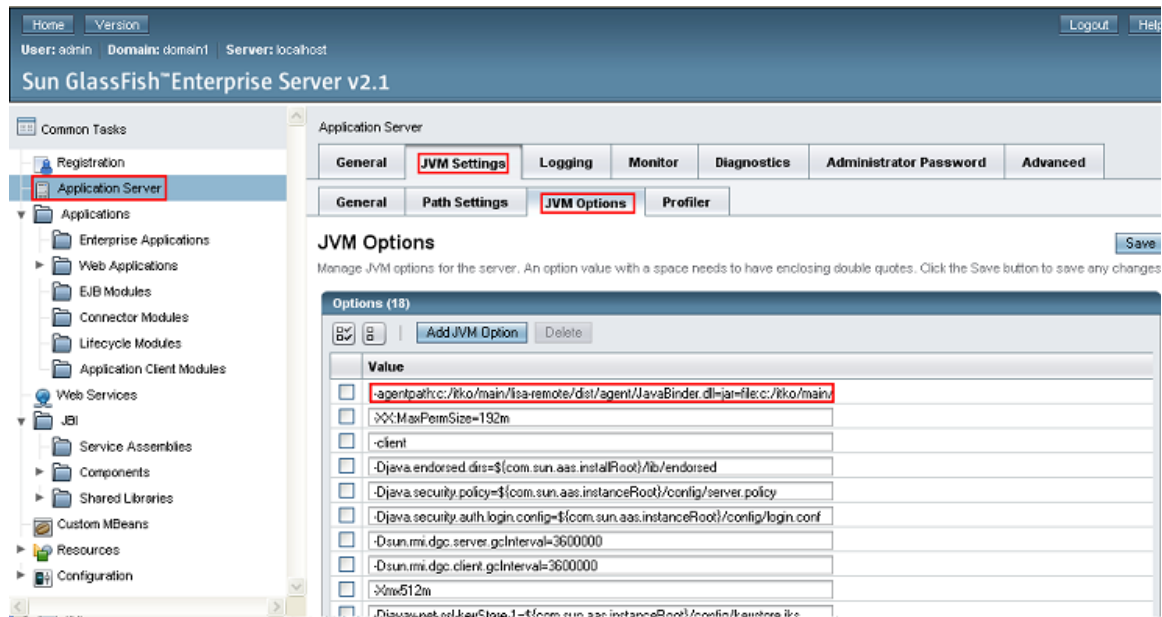
More complex IBM products that are built on top of WebSphere Application Server have complex configurations. Not all "servers" that are identified in **wsadmin** are truly "application servers" that can have LISA Agent installed on them. The linked attachment in the previous example contains a zip file of **wsadmin** scripts that can be used to deploy LISA Agent into such environments. To use it:

1. Download, unzip, and copy the contents into your LISA Agent directory.
2. Put a Java Binder library, LisaAgent.jar, and rules.properties into the LISA Agent directory as usual.
3. Edit **agent.env** per the comments in the file, changing the values appropriately.
4. With WebSphere up and running, run **deployAgent.sh** to deploy the LISA Agent to all application servers in a WAS install. Use **undeployAgent.sh** to undeploy the Agent.

The JACL scripts have the "smarts" to not remove existing JVM arguments unrelated to Agent, and to not add or remove duplicate LISA Agent arguments.

## Configuring the Agent on Sun GlassFish

You can apply the usual instructions with GlassFish or you can use the administrative console, which is the officially supported way to do it. Go to <http://localhost:4848/>, navigate to JVM Options and enter the **-agentpath:...** option before saving.



Alternatively, you can manually edit the configuration file that stores these entries at **<Glassfish home>/domains/domainXXX/domain.xml**. You need only add an entry with the **-agentpath:...** value under **/domain/configs/java-config/jvm-options**.

For Glassfish version 3, you also need to modify **<glassfishHome>/glassfish/osgi/felix/conf/config.properties** to include the **com.itko.\*** packages in the boot delegation list as follows:

```
# There is no need to use bootdelegation except for the following issues:
# 1. EclipseLink
# 4. NetBeans profiler packages exist in parent class loader (see issue #8612)
# 5. BTrace exists in bootclasspath.
org.osgi.framework.bootdelegation=${eclipseLink.bootdelegation}, \
com.sun.btrace, com.sun.btrace.*, \
org.netbeans.lib.profiler, org.netbeans.lib.profiler.*, com.itko.*
```

## Configuring the Agent on TIBCO BusinessWorks

This topic is applicable to TIBCO BusinessWorks 5.x.

To configure the Agent for all applications, you edit the following file:

```
...\tibco\bw\5.x\bin\bwengine.tra
```

Editing this file does not affect applications that have already been created.

To configure the Agent for an existing application, you edit the following file:

```
...\tibco\tra\domain\<domain name>\application\<application name>\<application name>-<service name>.tra
```

where:

- <domain name> is the name of your TIBCO domain
- <application name> is the name of the deployed TIBCO application
- <service name> is the name of the deployed service

For example:

```
D:\tibco\tra\domain\itko-tibcobw\application\MyBWProjectXml\MyBWProjectXml-itkoUserCRUD.tra
```

In either file, add the following line:

```
java.extended.properties=-agentpath:<lisa-home>/agent/JavaBinder.dll=jar=file:<lisa-home>/agent/LisaAgent.jar,name=<name>
```

where:

- <lisa-home> is the absolute path of the LISA installation directory.
- <name> is a descriptive name to give the LISA Agent. This can be the name of your service if editing the individual service file, or just something like bwengine if editing the main bwengine.tra file.

## Configuring the Agent on Tomcat, JBoss, Geronimo, and Resin

The standard instructions apply well for those containers:

### Tomcat

In Tomcat's **startup.bat(.sh)**, define **set JAVA\_OPTS=-agentpath:<path to JavaBinder>** before calling **catalina.bat(.sh)**.

### JBoss

In JBoss's **run.bat(.sh)**, define **set JAVA\_OPTS=-agentpath:<path to JavaBinder>** before calling java executable.

### Geronimo

In Geronimo's **startup.bat(.sh)**, define **set JAVA\_TOOL\_OPTIONS=-agentpath:<path to JavaBinder>** before calling **geronimo.bat(.sh)**.

### Resin

In Resin, simply define **set JAVA\_TOOL\_OPTIONS=-agentpath:<path to JavaBinder>** in the launching shell of **resin.exe(.sh)** or pass **-J-agentpath:...** as an argument.

## Configuring the Agent on WebMethods IS

In WebMethods's **IntegrationServer/bin/server.bat(.sh)** define **set JAVA\_TOOL\_OPTIONS=-agentpath:<path to JavaBinder>** before the line calling java. **server.exe** just calls into **server.bat** so there is no need for special instructions to use it.

If IS is defined as a Windows service, you can edit the **server.bat** by inserting the **-agentpath:...** argument in the **/jvmargs** value defined in the if **"1%1"=="1-service"** switch, for example:

```
%S_DIR%\bin\SaveSvcParams.exe" /svcname %2 /jvm "%JAVA_DIR%\.\" /binpath "%PATH%" /classpath %CLASSPATH% /jvmargs "
-agentpath:c:/Lisa/agent/JavaBinder.dll=jar=file:C:/Lisa/agent/LisaAgent.jar,name=MyIS %JAVA2_MEMSET%" /progargs
"%S_DIR%\bin\ini.cnf"#"-service
%2"##PREPENDCLASSES_SWITCH%##PREPENDCLASSES%##APPENDCLASSES_SWITCH%##APPENDCLASSES%##ENV_CLASSP.
```

## LISA Agent User Guide

This chapter contains the following sections:



Agent Getting Started  
Agent Configuration Properties  
Agent Data Categories  
Using the Dev Console  
Dev Console Videos  
Developing Against the Agent  
Agent Custom Extensions

Agent Transaction Weight

Load Balancers and Native Web Servers

## Agent Getting Started



**Important:** It is critical you find out about the target environment ahead of time and make sure it has been tested, or test it yourself. Most issues will be operating system or JVM-dependent, not application-dependent. Then carefully follow the instructions supplied in this documentation. If that does not help, look at the [troubleshooting guide](#) before contacting support.

After starting an agent-enabled Java process (see [LISA Agent Installation](#)), we need to communicate with it. This is accomplished by using a JMS broker. If running standalone, the broker can be started with the command line:

```
java -jar LisaAgent.jar -broker [connection url]
```

The only piece of configuration that is needed on the agent and console sides is to tell them where the broker is located; that is, they need the broker connection string. If the string is not specified, it will default to **tcp://localhost:2009**.

In addition, persistent information captured by the agents (transaction trees, statistics, and so on) is automatically flushed by the broker to a database that agent clients can later query if the data is too old to still be in memory.

This database is configured through the **sink** property, as described in [Agent Configuration Properties](#). If no sink is configured, then the broker automatically starts an embedded Derby database. For this to work, the Derby JARs must be placed in the broker directory: [derby.jar](#) and [derbynet.jar](#). Similarly, the Derby client JARs must be placed in the client JAR directory: [derbynet.jar](#) and [derbyclient.jar](#).

Any other database that supports JDBC can be used to flush this information. The JDBC driver JARs must be placed in the same directory as the Broker JAR. The way that agents are told of where these databases reside is described in the next section.

See [Required Files for Agent Deployment](#) for a description of these file requirements.

## Startup Options

### url

If you want to specify a non default broker connection string, you set the **url** parameter. For example:

```
JAVA_TOOL_OPTIONS=-agentpath:<Path to JavaBinder.xxx>=url=tcp://192.168.1.100:2009
```

This combines with starting the broker on that connection string with: `java -jar LisaAgent.jar -broker tcp://192.168.1.100:2009`

### name

You can give the Agent a name by using the **name** parameter. This helps with readability (agents are displayed with that name in the UI) and serves to identify processes across shutdowns and startups by keeping some persistent identifier.

```
JAVA_TOOL_OPTIONS=-agentpath:<Path to JavaBinder.xxx>=name=myjboss
```

### token

If you want to make access to the Agent secure, you can use a password in the environment variable using the **token** parameter. For example:

```
JAVA_TOOL_OPTIONS=-agentpath:<Path to JavaBinder.xxx>=token=password
```

### jar

If you want to put **LisaAgent.jar** and **JavaBinder(.dll, .lib, .so)** in different directories, you can specify the location of the jar with:

```
JAVA_TOOL_OPTIONS=-agentpath:<Path to JavaBinder.xxx>=jar=file:<Path to LisaAgent.jar>
```

### heap

This option, available only with the native agent, enables all the heap walking APIs (getting object instances, reference graphs, and so on) at a small performance cost (less than 5 percent for modern VMs). Can be set to true or false. The default value is false.

### ex

This option, available only with the native agent, enables the superstacks feature at a small performance cost (less than 5 percent for modern VMs). Can be set to true or false. The default value is false.

## Specifying Multiple Options

You can combine all options in any way by delimiting them with commas. For example:

```
JAVA_TOOL_OPTIONS=-agentpath:<Path to JavaBinder.xxx>=url=tcp://orion:61616,name=myjboss,token=password
```

## Agent Configuration Properties

The Agent makes the utmost effort to self-configure, but you can tune or override parameters by placing a text file named **rules.properties** in the Agent JAR directory. This file supports the following directives:

### Setting a General Property

For the full list, see the sample **rules.properties** file in [Agent Downloads](#).

```
wsproperty key=<key> value=<value>
```

Example:

```
property key=lisa.broker.transaction.buffer.size value=10
```

### Setting a Database Sink for Agent Persistent Data

This setting will stay in use for the lifetime of the Agent.

```
sink driver=<Driver class> url=<jdbc url> user=<username> password=<password>
```

Example:

```
sink driver=org.postgresql.Driver url=jdbc:postgresql://localhost:5432/agtdb user=postgres  
password=postgres
```

### Adding a Class for Tracking

The purpose of this is so instances can easily be retrieved from the heap later.

```
track class=<class name>
```

Example:

```
track class=java.io.File
```

### Adding a method for interception

The purpose of this would be so the method's invocations will be tracked, either for Pathfinder or VSE purposes.

```
intercept class=<class name> method=<method name> signature=<signature>
```

Example:

```
intercept class=com.itko.lisa.test.Client method=method1 signature=
```

You can also add methods to intercept by using the Dev Console. For more information, see [Viewing Classes](#).

### Excluding a method, class, or package from interception and virtualization

Exclusion takes precedence over interception and virtualization, but it works only for the classes specified and not their descendants.

```
exclude class=<class name> method=<method name> signature=<signature>
```

The following example is for a class:

```
exclude class=com.itko.lisa.test.Client method= signature=
```

The following example is for a package:

```
exclude class=com.itko.lisa.test.* method= signature=
```

The following example is for a package and all its subpackages:

```
exclude class=com.itko.lisa.test.** method= signature=
```

## Adding a class for virtualization (both recording and playback modes)

```
virtualize class=<class name>
```

Example:

```
virtualize class=javax.ejb.SessionBean
```

## Setting up a custom transform for virtualized method parameters and results before being XStream'ed

In these examples, "this" and "that" are keywords, with "this" referring to the object being transformed and "that" having a custom meaning depending on the object, usually its state.

```
virtualize transform=<class name> code=
```

Example:

```
virtualize transform=javax.servlet.http.HttpServletRequest code=[return this.getRequestURL().toString();]
```

Example:

```
virtualize transform=javax.servlet.http.HttpServletResponse code=[return that;]
```

## Telling the Agent how to determine what constitutes a session for a given protocol

You do this by giving a code snippet that returns a session identifier.

```
virtualize track=<class name> method=<method name> signature=<signature> code=[<code>] push=<true|false>
```

Example:

```
virtualize track=javax.servlet.http.HttpServletRequest method=service
signature=(Ljavax/servlet/http/HttpServletRequest;Ljavax/servlet/http/HttpServletResponse;)V code=[return
$1.getSession().getId();] push=false
```

```
virtualize track=javax.ejb.SessionBean method=setSessionContext signature=(Ljavax/ejb/SessionContext)V
code=[return $1.getEJBObject().getHandle().toString();] push=true
```

```
virtualize track=javax.ejb.EntityBean method=setEntityContext signature=(Ljavax/ejb/EntityContext)V
code=[return $1.getPrimaryKey().toString();] push=true
```

The preceding three lines are hard-coded in the Agent and thus unnecessary in your **rules.properties** file, but show how it works so it can be done for any number of protocols other than HTTP and EJB without recompiling the Agent.

The value of the **track** parameter is the class from which we gain access to a session.

The value of the **method** and **signature** parameters determine the method that, when invoked, will compute the session identifier for us, using the value of the **code** parameter (in which **\$0** represents the source object and **\$1**, **\$2**, and so on are the method arguments).

The **push** parameter decides how we store that session for later use by VSE frames: **push=false** means the session is basically thread-scoped and we store in a thread-local variable, whereas **push=true** means the session is set for us by the container and we keep a mapping of object to session. The innermost session (identifier) will be served back through VSE in the [com.itko.lisa.remote.vse.VSEFrame getSessionId\(\)](#) method.

## Extending the Agent with custom interceptors

```
interceptor class=<class name>
```

If you package Java classes in jar files located in the agent directory, those can be made available to the Agent. In particular, if a class

implements `com.itko.lisa.remote.transactions.interceptors.IInterceptor` and is specified with the interceptor directive, then its `block()`, `preProcess()`, and `postProcess()` methods will be invoked as hooks.

## Extending the Agent with custom formatters

```
formatter class=<class name> name=<formatter name>
```



A simple JDBC invoke program cannot be recorded until the program attached to Agent starts the Pathfinder path. This can be achieved by updating the `rules.properties` file to look for a specific method being invoked to start the path.



The Agent does not intercept methods in inner classes.

## Agent Data Categories

The Agent can capture the following categories of data:

- Client
- EJB
- GUI (AWT, Swing, SWT)
- Framework (Struts, Spring)
- JCA
- JDBC
- JMS
- Logging
- REST
- RMI
- Thread (a synthetic frame used to show stitching across threads)
- Throwable
- TIBCO ActiveMatrix BusinessWorks
- TIBCO ActiveMatrix Service Grid
- webMethods Integration Server
- WebSphere MQ
- WebSphere Process Server
- Web (HTTP/S)
- Web Service

## Using the Dev Console



This feature requires a separate license.

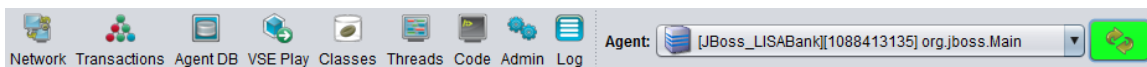
The Dev Console lets you view detailed information about the workings of the LISA Agent. The Dev Console also lets you add extensions.

To start the Dev Console, do either of the following:

- Go to the `LISA_HOME/bin` directory and run the **DevConsole** executable file. The Dev Console opens in a native application window.
- Go to LISA Workstation and select View > LISA Dev Console from the main menu.

The upper portion of the Dev Console includes a toolbar. The toolbar has a series of icons for displaying the various windows, a drop-down list for selecting the Agent, and a refresh button.

The following image shows the toolbar.



The refresh button also serves as a status indicator. For example, the button indicates when a new Agent comes online and when the selected Agent goes offline.

The following topics describe the functionality of the Dev Console:

- [Viewing the Network Summary](#)
- [Viewing Transactions](#)

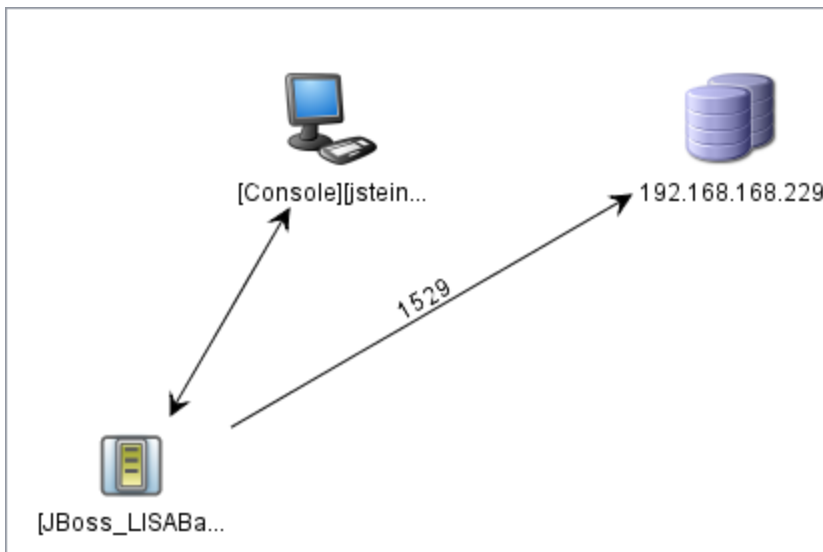
- Viewing the Call Tree
- Viewing Agent Database Information
- Viewing Agent Logging
- Viewing Agent Properties
- Viewing Classes
- Viewing Threads
- Managing Files
- Accessing a Remote Terminal
- Creating Agent Extensions
- Using the Virtual Service Playground

## Viewing the Network Summary

The Network window in the Dev Console is divided into a left panel and a right panel.

The left panel contains a graphical view of the network topology.

The following image shows an example of the left panel. The panel has three components: the Console Agent, the JBoss Agent, and the Agent database. Each component has a different icon. The JBoss Agent connects to the database on port 1529.



You can use the agents, consoles, and external systems check boxes to show and hide component types in the left panel.

If multiple agents are started with the same name, then the process ID is used as a secondary identifier. They will appear as separate agents in the Network window. This feature is not available with Java 1.4.

The right panel contains the following tabs:

- **VM Info:** Displays properties for the Agent and the VM in which it is running. The Agent version number is typically different from the LISA version number. The main class is the Java class containing the main method that was invoked for the Agent. The PID is the process ID in which the Agent is running. If the process ID is unknown, then the value is set to -1.
- **Performance:** Displays statistics on CPU usage, memory usage, and I/O throughput.
- **J2EE Web Info:** Displays information from the **web.xml** deployment descriptor file. For example, you can view the **servlet** and **servlet-mapping** elements.
- **JNDI Info:** Covers all the web applications in the Agent.

## Agent Alarm Mechanism

If an Agent is in an alarm state, then the component icon in the left panel becomes a lock icon. Any of the following situations can lead to an alarm state:

- The JVM is experiencing a deadlock.
- The percentage of time spent running on the CPU is too high for a specified number of intervals.
- The percentage of time spent in garbage collection is too high for a specified number of intervals.
- The percentage of heap memory used is too high for a specified number of intervals.
- The percentage of permgen memory used is too high.
- The number of exceptions printed to a log per second is too high.
- The Agent is sending too many messages per interval to the broker.

The heap and permgen are memory regions of the JVM.

The following properties in the **rules.properties** file let you control the alarm behavior:

- **lisa.agent.stats.alarm.threshold.cpu**: The percentage for the CPU alarm. The default value is 80.
- **lisa.agent.stats.alarm.threshold.cpu.num.intervals**: The number of intervals for the CPU alarm. The default value is 5.
- **lisa.agent.stats.alarm.threshold.gc**: The percentage for the garbage collection alarm. The default value is 30.
- **lisa.agent.stats.alarm.threshold.gc.num.intervals**: The number of intervals for the garbage collection alarm. The default value is 3.
- **lisa.agent.stats.alarm.threshold.heap**: The percentage for the heap alarm. The default value is 95.
- **lisa.agent.stats.alarm.threshold.heap.num.intervals**: The number of intervals for the heap alarm. The default value is 10.
- **lisa.agent.stats.alarm.threshold.permgen**: The percentage for the permgen alarm. The default value is 90.
- **lisa.agent.stats.alarm.threshold.exceptions.per.second**: The number of exceptions per second. The default value is 2.
- **lisa.agent.stats.alarm.threshold.messages.per.interval**: The number of messages. The default value is 20.
- **lisa.agent.stats.sampling.interval**: The number of intervals for the messages alarm. The value is in milliseconds. The default value is 1000.

If the messages alarm is triggered, try [configuring the Agent properties](#) to reduce the number of messages sent.

## Viewing Transactions

Each transaction contains a hierarchical set of frames. A frame encapsulates data about a method call that the Agent intercepted. This data includes the class, method, arguments, return value, and so on. The top-level frame in a transaction is referred to as the root frame.

The Transactions window in the Dev Console has two modes:

- Regular view
- Expert view

The default setting is regular view. You can switch to expert view by selecting the Expert View check box.

Both views are divided into an upper panel and a lower panel.

The Expand All and Collapse All buttons in the toolbar let you quickly expand and collapse all the nodes in the upper panel.

The upper panel can contain incomplete transactions, which are fragments that could not be assembled within the time specified by the **lisa.broker.transaction.assembly.expiry** property. These transactions are grayed out. If you do not want incomplete transactions to appear, then select the Exclude incomplete paths check box.

The lower panel contains the following tabs: Details, Request, Response, and Debug.

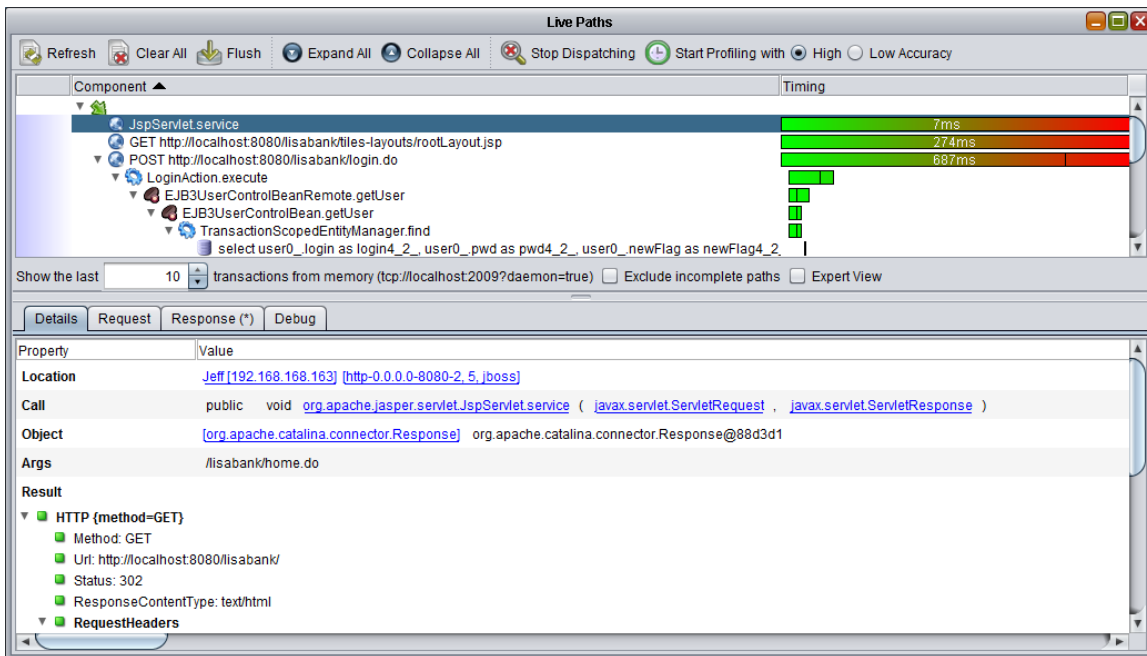
### Regular View

The Component column in the upper panel contains a high-level view of the transactions. Each node represents a frame. In some cases, a parent frame and its child frames are collapsed into a single frame.

You can view the Agent and thread for a frame by placing the mouse pointer in the first column.

The horizontal bars in the Timing column provide a visual indicator of the duration of each frame and when the frame started, in relation to the root frame. A horizontal bar might include a vertical divider, which indicates the proportion of time spent in CPU versus the wall clock time. You can view the time, duration, and CPU information by placing the mouse pointer over a horizontal bar.

The following image shows the regular view. The upper panel contains a set of transactions that resulted when a user opened the LISA Bank demo application and then logged in. The first transaction is selected. The third horizontal bar in the Timing column includes a vertical divider.



The Details tab contains the following rows:

- **Location:** The name and IP address of the computer that this object was generated on, and the globally unique identifier of the Agent.
- **Call:** The signature of the method that the Agent intercepted. Includes the access modifier, return type, class name, method name, and arguments.
- **Object:** The actual class name.
- **Args:** A stringified representation of the arguments to the method.
- **Result:** A stringified representation of the return value of the method.

Some of the rows may include an inspect icon, which you can click to display the value in a separate window.

If the Request tab or the Response tab has any content, then an asterisk appears next to the tab name.

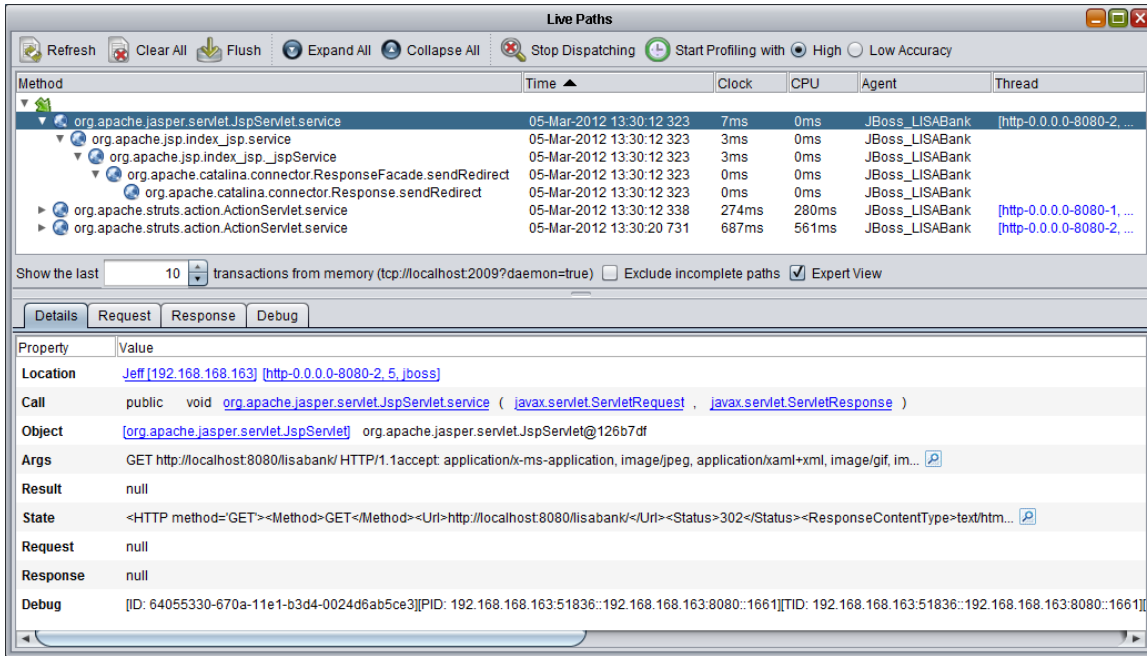
The Debug tab contains the following information for the selected frame:

- **FID:** Unique identifier of the frame
- **PID:** Unique identifier of the frame's parent
- **TID:** Unique identifier of the transaction
- **SID:** Unique identifier of the session
- **CAT:** Category
- **FLAGS:** Flags (if any)
- **SRC:** Source address (if applicable)
- **TARGET:** Target address (if applicable)
- **BAG:** A set of arbitrary, non-persistent properties set by agents on frames so they can be used by the broker to perform additional logic. Bags are currently used for secondary stitching (when there are [load balancers](#)).

## Expert View

The Method column in the upper panel contains a low-level view of the transactions. Each node represents a frame. When you select a frame in the upper panel, the lower panel displays detailed information about the frame.

The following image shows the expert view. The upper panel contains a set of transactions that resulted when a user opened the LISA Bank demo application and then logged in. The root frame of the first transaction is selected. This frame is for a call to the **service()** method of the **org.apache.jasper.servlet.JspServlet** class.



The Details tab contains the same rows as in the regular view, plus the following rows:

- **State:** A custom XML blob that contains information about the protocol that this frame captured.
- **Request:** A protocol-specific stringified representation of the arguments to the method. The request field is more detailed than the arguments field. For stream-based protocols, it captures the entire stream.
- **Response:** A protocol-specific stringified representation of the return value of the method. The response field is more detailed than the result field. For stream-based protocols, it captures the entire stream.
- **Debug:** The frame ID, parent ID, transaction ID, session ID, category, flags, source address, and target address. Each piece of information is enclosed within square brackets.

Some of the rows may include an inspect icon, which you can click to display the value in a separate window.

If the Request tab or the Response tab has any content, then an asterisk appears next to the tab name.

For a description of the Debug tab, see the regular view section.

## Viewing the Call Tree

The toolbar in the [Transactions window](#) includes a Start Profiling icon. If profiling is enabled when the frames are recorded, then you can click any link in the Thread column to display the full call tree in a separate window.

The following image shows the Call Tree window.



| Method   | Approximate Duration | %    |
|--|----------------------|------|
| [root]   |                      |      |
| java.lang.Thread.run   | 402 ms               | 100% |
| org.apache.tomcat.util.net.JioEndpoint\$Worker.run                       | 402 ms               | 100% |
| org.apache.coyote.http11.Http11Protocol\$Http11ConnectionHandler.process | 402 ms               | 100% |
| org.apache.coyote.http11.Http11Processor.process                         | 402 ms               | 100% |
| org.apache.catalina.connector.CoyoteAdapter.service                      | 402 ms               | 100% |
| org.apache.catalina.core.StandardEngineValve.invoke                      | 402 ms               | 100% |
| org.jboss.web.tomcat.service.jca.CachedConnectionValve.invoke            | 402 ms               | 100% |
| org.apache.catalina.valves.ErrorReportValve.invoke                       | 402 ms               | 100% |
| org.apache.catalina.core.StandardHostValve.invoke                        | 402 ms               | 100% |
| org.jboss.web.tomcat.security.JaccContextValve.invoke                    | 402 ms               | 100% |
| org.jboss.web.tomcat.security.SecurityAssociationValve.invoke            | 402 ms               | 100% |
| org.apache.catalina.core.StandardContextValve.invoke                     | 402 ms               | 100% |

| Property | Value   |
|----------|---|
| Location | Jeff [192.168.1.67] [http-0.0.0.0-8080-5, 5, jboss] |
| Call     | java.lang.Thread.run (...)                          |
| Object   |   |
| Args     |   |
| Result   |   |
| State    |   |
| Request  |   |
| Response |   |
| Debug    |   |

You can filter the call tree by absolute duration and relative duration.

You can click Export to display the call tree in a snapshot window that enables you to copy and paste.

## Viewing Agent Database Information

The Agent DB window in the Dev Console lets you view the following types of information in the Agent database:

- Transactions
- Java VSE
- Agents
- Tickets
- Statistics


The Transactions tab includes a variety of filters. You can choose to display only a certain type of root frame, such as Web, RMI, or EJB. You can specify a date and time range. You can remove one or more Agents.


The Tickets tab contains information about any Pathfinder cases that have been generated.


The Agent DB window also lets you enter SQL statements to query or update the database. Click the Raw SQL tab, enter the statement, and click Refresh. For a listing of the available tables, see [Agent Database Schema](#).

The following image shows a query of the LISA\_STATISTICS table.

Agent DB

 Refresh

 Export

 Import

Connected to  User  Password

Transactions

Java VSE

Agents

Tickets

Statistics

Raw SQL

select \* from lisa\_statistics

| AGENT_ID   | TIME          | CPU | HEAP      | NON_HEAP | IO_IN | IO_OUT | GC_COUNT | GC_TIME |
|------------|---------------|-----|-----------|----------|-------|--------|----------|---------|
| 1088413135 | 1311020829021 | 93  | 36522288  | 11690040 | 0     | 0      | 1        | 47      |
| 1088413135 | 1311020830035 | 68  | 55775176  | 12967648 | 0     | 0      | 0        | 0       |
| 1088413135 | 1311020831049 | 98  | 59013104  | 13014240 | 0     | 0      | 0        | 0       |
| 1088413135 | 1311020832063 | 95  | 63039976  | 13090592 | 0     | 0      | 0        | 0       |
| 1088413135 | 1311020833077 | 100 | 69031456  | 13104480 | 0     | 0      | 0        | 0       |
| 1088413135 | 1311020834091 | 95  | 81128352  | 13569464 | 0     | 0      | 0        | 0       |
| 1088413135 | 1311020835152 | 96  | 45005840  | 14726648 | 0     | 0      | 1        | 57      |
| 1088413135 | 1311020836182 | 74  | 28041904  | 15897800 | 0     | 0      | 2        | 255     |
| 1088413135 | 1311020837196 | 84  | 59025528  | 17298296 | 0     | 0      | 0        | 0       |
| 1088413135 | 1311020838210 | 67  | 73329976  | 17777768 | 0     | 0      | 0        | 0       |
| 1088413135 | 1311020839224 | 95  | 104008632 | 18991632 | 0     | 0      | 0        | 0       |
| 1088413135 | 1311020840238 | 92  | 122555352 | 19142528 | 0     | 0      | 0        | 0       |
| 1088413135 | 1311020841423 | 100 | 39353000  | 21459200 | 0     | 0      | 1        | 97      |
| 1088413135 | 1311020842484 | 90  | 68940632  | 22113608 | 0     | 0      | 0        | 0       |
| 1088413135 | 1311020843514 | 88  | 134854976 | 22295104 | 0     | 0      | 0        | 0       |
| 1088413135 | 1311020844528 | 100 | 168804552 | 23047432 | 0     | 0      | 0        | 0       |
| 1088413135 | 1311020845542 | 93  | 70974432  | 23739936 | 0     | 0      | 1        | 49      |

## Viewing Agent Logging

The Logging window in the Dev Console displays the log messages for the selected Agent.

You can specify how many lines to display.

You can also specify which log levels to display. The Default and Debug levels are sufficient for most purposes. The Dev level produces a lot of information.

The following example shows a partial set of log messages for the JBoss Agent.

```
[LISA AGENT:][INFO][5448][main][Thu Jan 12 09:57:34 PST 2012] Agent Log Path:
C:\Lisa\agent\lisa_agent_5448.log
[LISA AGENT:][INFO][5448][main][Thu Jan 12 09:57:34 PST 2012] Starting Agent (5.1.4.1)...
[LISA AGENT:][INFO][5448][main][Thu Jan 12 09:57:36 PST 2012] Broker Connection: [empty] -
defaulting to tcp://localhost:2009?daemon=true
[LISA AGENT:A][INFO][5448][main][Thu Jan 12 09:57:38 PST 2012] Connection to
tcp://localhost:2009?daemon=true established...
[LISA AGENT:A][INFO][5448][main][Thu Jan 12 09:57:38 PST 2012] *** Started LISA Agent
[LISA AGENT:A][INFO][5448][main][Thu Jan 12 09:57:38 PST 2012] *** Version      : 5.1.4.1 (4.7.8.1)
[LISA AGENT:A][INFO][5448][main][Thu Jan 12 09:57:38 PST 2012] *** Name        : JBoss_LISABank
(1088413135)
```

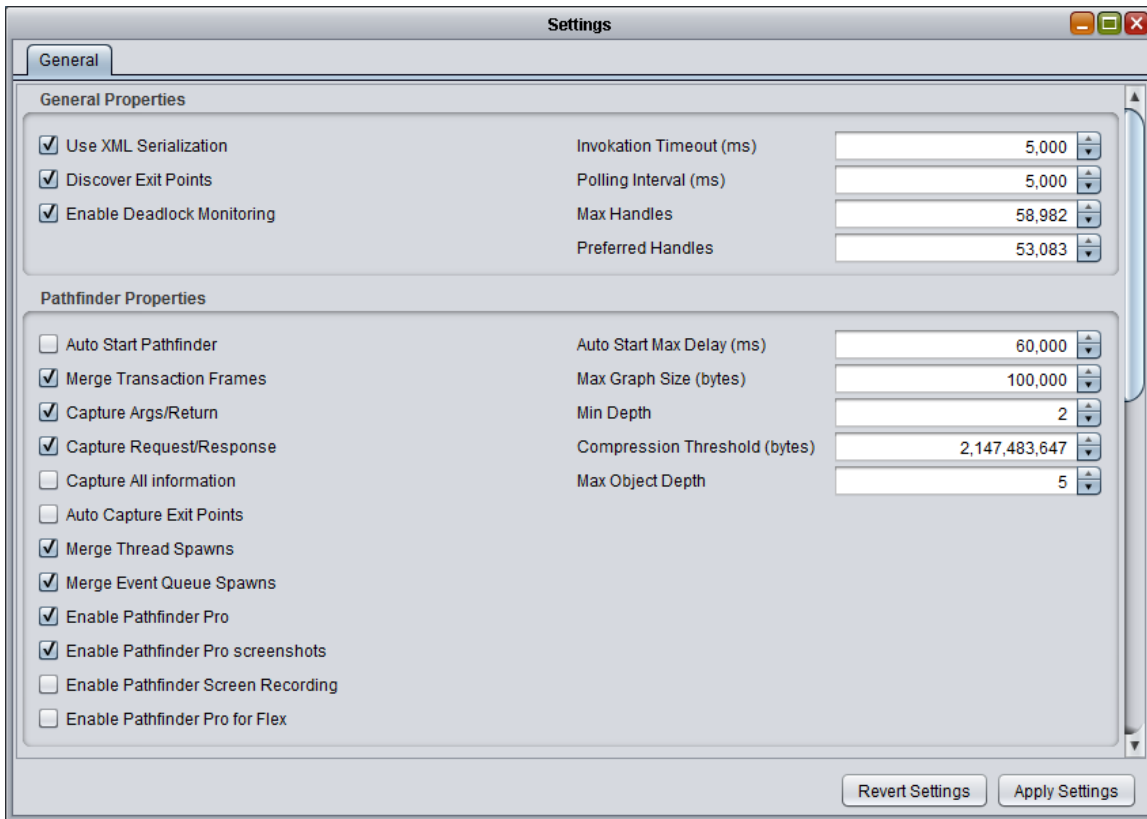
## Viewing Agent Properties

The Settings window in the Dev Console displays a list of properties for the selected Agent. To access the window, click the Admin icon in the main toolbar and then click Agent Settings.

The properties are divided into categories.

To update one or more properties, make the changes and click Apply Settings.

The following image shows the Settings window.



## Viewing Classes

The Classes window in the Dev Console lets you browse the full class hierarchy loaded in the JVM.

To display the Classes window, do either of the following:

- In the Dev Console toolbar, click the Classes icon.
- In the lower panel of the [Transactions window](#), click a class or object link.

You can view the class hierarchy by package or by archive.

In the left panel, icons indicate whether the node is a class or an interface. An orange icon indicates a class. A blue icon indicates an interface.

When you select a class or interface in the left panel, the right panel displays the following information:

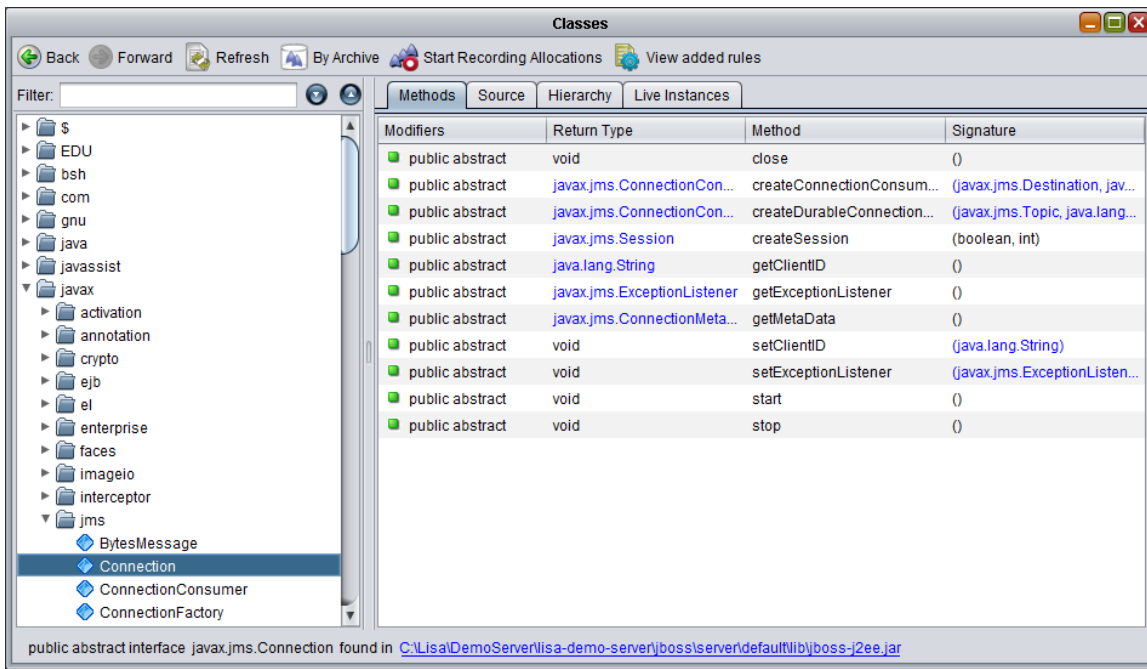
- Method signatures
- A decompiled version of the Java source code (if available)
- Superclasses and subclasses
- Live object instances

When you click the Source tab, you might be required to confirm that you have sufficient rights to view the decompiled code.

The **rules.properties** file contains the following entries, which represent packages for which you do not need to provide confirmation:

```
whitelist package=java.*
whitelist package=javax.*
whitelist package=org.*
```

The following image shows the Classes window. In the left panel, the class hierarchy is displayed by package. The **javax.jms.Connection** interface is selected. The right panel displays the method signatures.



## Adding Methods to Intercept

If you want the Agent to track an additional method in the current session, right-click the method and select Intercept Method.

You can see a list of the methods that you added by clicking **View added rules** in the toolbar. The Rules for this session window appears. The following example is a rule that intercepts the **setUsername()** method of the **com.itko.lisabank.action.login.LoginForm** class:

```
# Rules added by me@mycomputer at Thu Feb 09 09:37:44 PST 2012

intercept class=com.itko.lisabank.action.login.LoginForm method=setUsername
signature=(Ljava/lang/String;)V delay=false
```

The Rules for this session window lets you commit the rules that you added.

## Viewing Threads

The Threads window in the Dev Console shows all the threads that are currently active in the JVM.

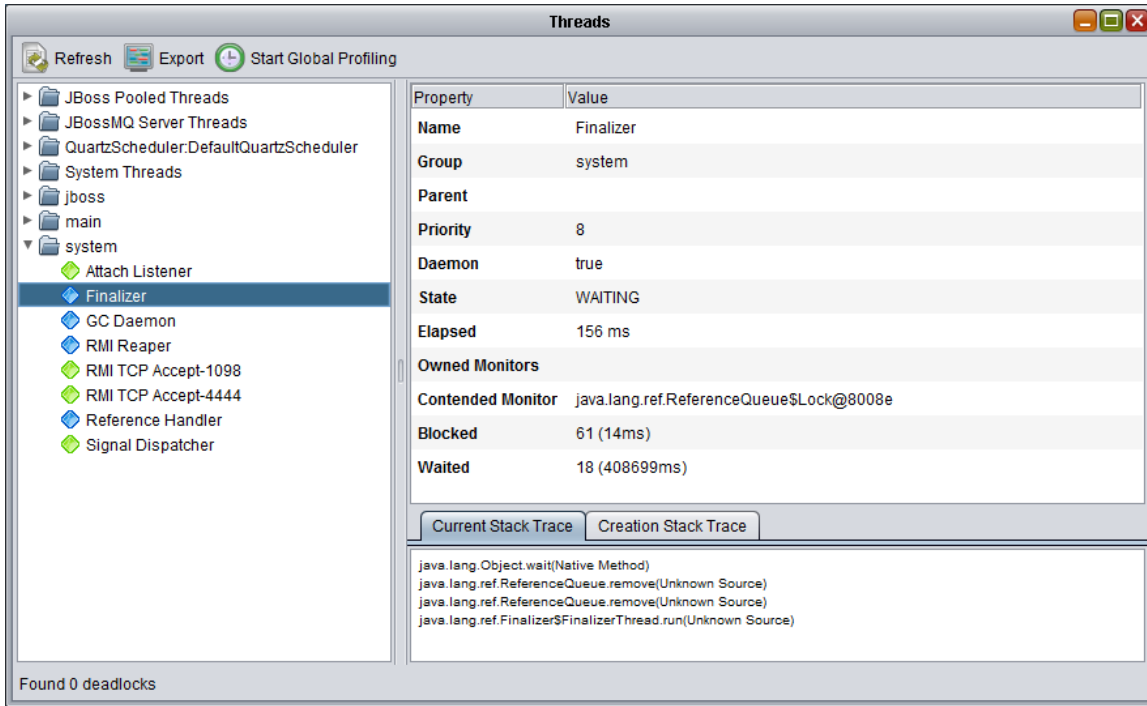
The left panel displays the threads in a tree structure. The icons indicate the thread state.

| Icon   | Thread State   |
|--------|--|
| Green  | New or Runnable  |
| Blue   | Waiting, Timed Waiting, Sleeping, Terminated, or Blocked |
| Purple | Deadlocked   |

The right panel displays thread properties and the current stack trace. If the thread was started by another thread, the stack trace of the creating thread is also shown.

The lower left corner displays the number of deadlocks.

The following image shows the Threads window. A waiting thread is selected in the left panel.



Click Export to display the stack traces in a snapshot window that enables you to copy and paste. However, the snapshot window does not include the thread names.

## Managing Files

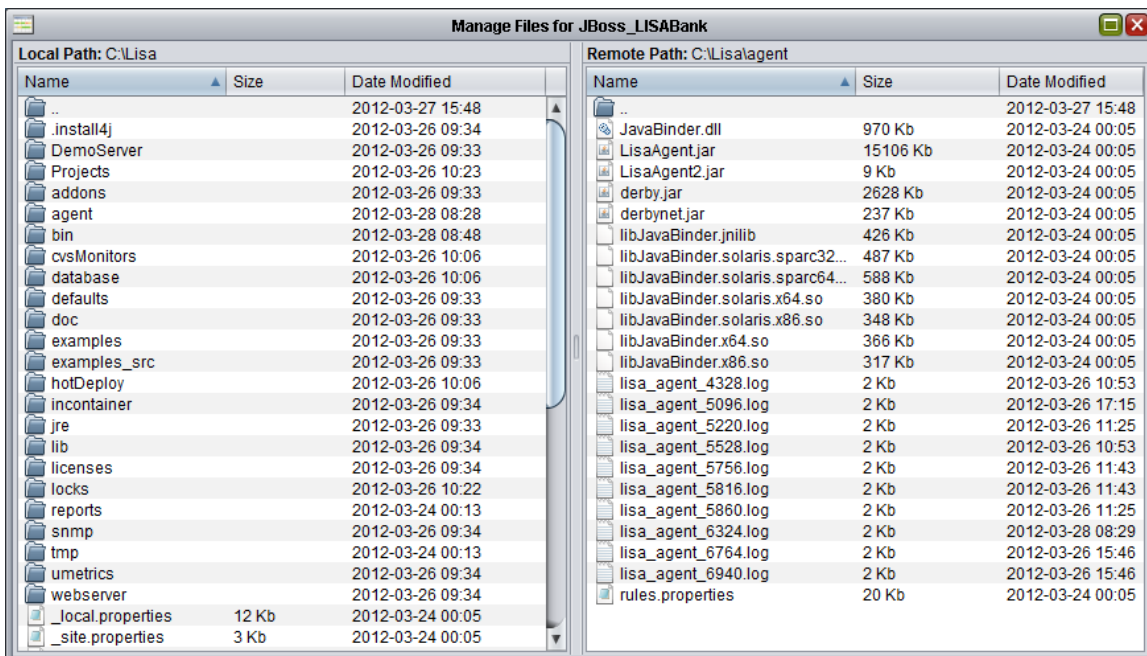
The Manage Files window in the Dev Console displays two sets of files:

- Files on the local computer
- Files on the computer where the selected Agent is installed

To access the window, click the Admin icon in the main toolbar and click File Commander.

You can move files between the computers by dragging and dropping. You can open files by double-clicking them. You can create new directories, rename files, and delete files by using the right-click menu.

The following image shows the Manage Files window.



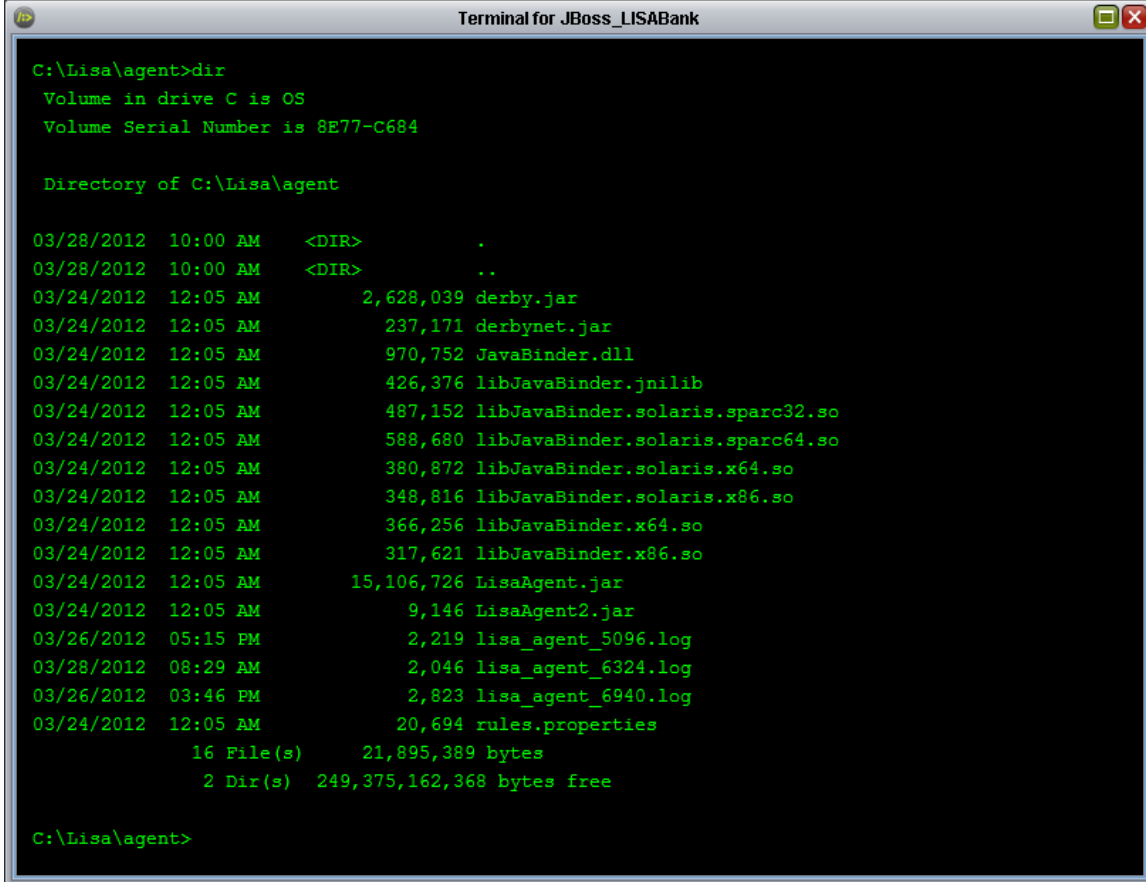
## Accessing a Remote Terminal

The Terminal window in the Dev Console displays a command prompt or shell for the computer where the selected Agent is installed.

To access the window, click the Admin icon in the main toolbar and then click OS Terminal.

You can perform any command that the operating system supports. For example, you can restart a process or check memory usage.

The following image shows the Terminal window.



```
C:\Lisa\agent>dir
Volume in drive C is OS
Volume Serial Number is 8E77-C684

Directory of C:\Lisa\agent

03/28/2012  10:00 AM    <DIR>          .
03/28/2012  10:00 AM    <DIR>          ..
03/24/2012  12:05 AM      2,628,039  derby.jar
03/24/2012  12:05 AM      237,171  derbynet.jar
03/24/2012  12:05 AM      970,752  JavaBinder.dll
03/24/2012  12:05 AM      426,376  libJavaBinder.jnilib
03/24/2012  12:05 AM      487,152  libJavaBinder.solaris.sparc32.so
03/24/2012  12:05 AM      588,680  libJavaBinder.solaris.sparc64.so
03/24/2012  12:05 AM      380,872  libJavaBinder.solaris.x64.so
03/24/2012  12:05 AM      348,816  libJavaBinder.solaris.x86.so
03/24/2012  12:05 AM      366,256  libJavaBinder.x64.so
03/24/2012  12:05 AM      317,621  libJavaBinder.x86.so
03/24/2012  12:05 AM     15,106,726  LisaAgent.jar
03/24/2012  12:05 AM        9,146  LisaAgent2.jar
03/26/2012  05:15 PM        2,219  lisa_agent_5096.log
03/28/2012  08:29 AM        2,046  lisa_agent_6324.log
03/26/2012  03:46 PM        2,823  lisa_agent_6940.log
03/24/2012  12:05 AM        20,694  rules.properties
               16 File(s)        21,895,389 bytes
               2 Dir(s)    249,375,162,368 bytes free

C:\Lisa\agent>
```

## Creating Agent Extensions

The Script window in the Dev Console lets you create, build, and upload Agent extensions. To access the window, click the Code icon in the main toolbar.



For additional guidance on writing Agent extensions, see [Agent Custom Extensions](#).

The following types of extensions are supported:

- Pathfinder
- Java VSE
- Broker

### To create an agent extension

1. In the Script window, click Create and select the extension type.  
The upper portion of the window provides a code template.
2. When you are finished, click Build.  
A JAR file is created that has the same name as the extension class.
3. Click Upload.
4. Specify the file name and click Choose file.  
The JAR file is uploaded to the selected Agent.

## To modify an agent extension

1. In the Script window, click Create and select Edit Extension.
2. Select the JAR file.
3. Click Choose file.

## Pathfinder

In a Pathfinder extension, you can overwrite any of the following methods: **block()**, **preProcess()**, and **postProcess()**.

If you want to stop data from being captured, then overwrite the **block()** method. This technique is similar to adding an exclude statement to the **rules.properties** file, but provides more flexibility.

The following example uses the **block()** method to prevent the Agent from capturing any method whose thread name starts with Event Sink Thread Pool.

```
public class MyInterceptor extends AbstractInterceptor {

    /** Returns true if this call should not be intercepted */
    public boolean block(boolean wayIn, Object src, String spec, String clazz, String method, String
signature, Object[] args, Object ret) {
        if (Thread.currentThread().getName().startsWith("Event Sink Thread Pool")) {
            return true;
        }
        return super.block(wayIn, src, spec, clazz, method, signature, args, ret);
    }

    ...

}
```

If you want the Agent to perform logic immediately before it captures a method, then overwrite the **preProcess()** method.

If you want the Agent to perform logic immediately after it captures a method, then overwrite the **postProcess()** method.

The following example uses the **postProcess()** method to capture data for the Response row in the [Transaction window](#).

```
public class MyInterceptor extends AbstractInterceptor {

    ...

    public boolean postProcess(TransactionFrame frame, Object src, String spec, String clazz, String
method, String signature, Object[] args, Object ret) {
        if (clazz.equals("com.itko.lisa.training.NotCaptured") && method.equals("doRequest")) {
            frame.setResponse((String) ret);
        }
        return super.postProcess(frame, src, spec, clazz, method, signature, args, ret);
    }

}
```

## Java VSE

In a Java VSE extension, you can overwrite any of the following methods: **onPostRecord()**, **onPreHijack()**, and **onPostHijack()**.

## Broker

In a Broker extension, you can overwrite the **onTransactionReceived()** method. The Broker calls this method after it receives a transaction fragment from an Agent.

You can review a code example in [Agent Custom Extensions](#).

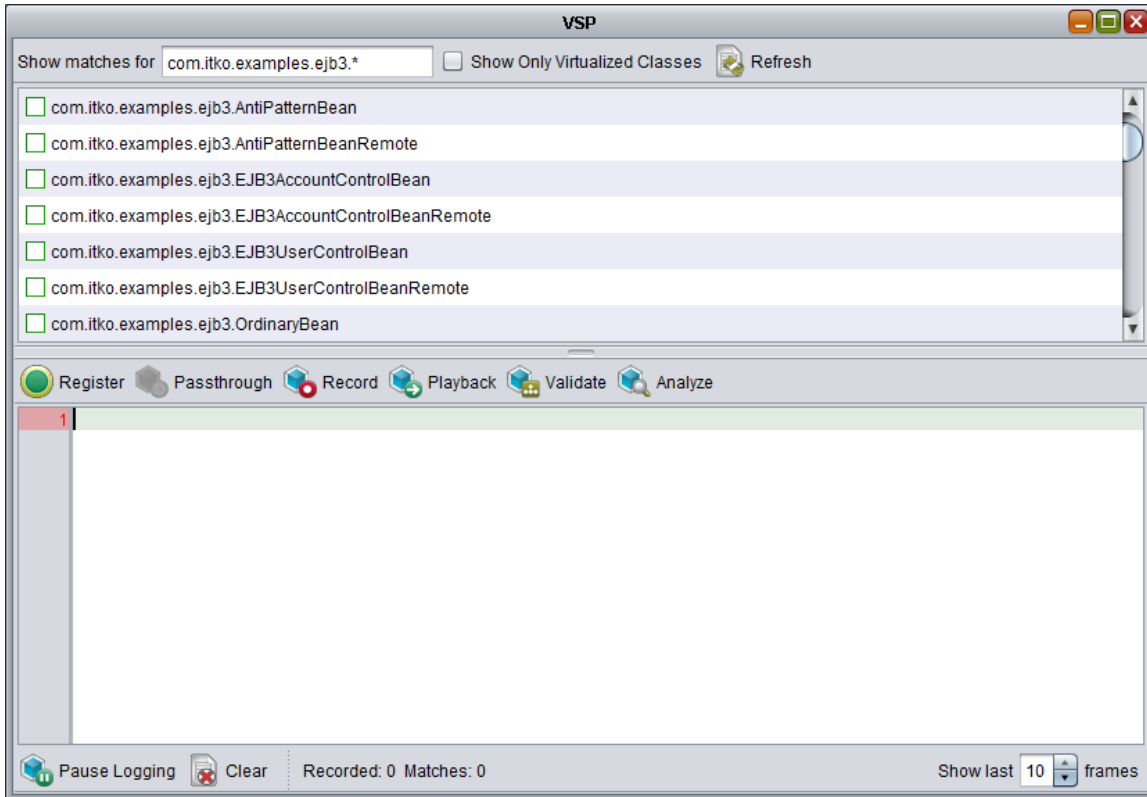
## Using the Virtual Service Playground

The VSP window in the Dev Console lets you perform the following tasks:

- Dynamically change the behavior of Java methods

- Obtain a list of suggested classes to virtualize
- Make the Dev Console act as a Virtual Service Environment (VSE)

The following image shows the VSP window. The window contains an upper panel and a lower panel. The upper panel is showing matches for any package that starts with **com.itko.example.ejb3**. The lower panel includes the following buttons: Register, Passthrough, Record, Playback, Validate, and Analyze.



The Passthrough button corresponds to the Live System execution mode. The Validate button corresponds to the Image Validation execution mode. For more information about execution modes, see [Running Live Requests Against LISA Virtualize](#).

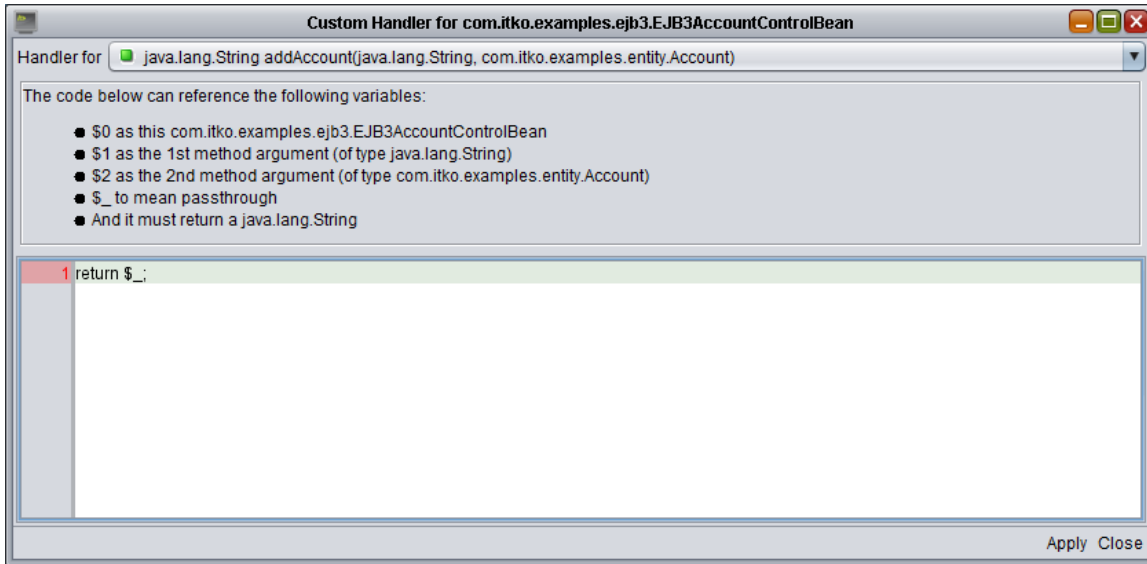
The appearance of the Agent node in the [Network window](#) changes when the Passthrough, Record, Playback, and Validate buttons are selected.

## Dynamically Changing the Behavior of Java Methods

This feature can be useful when you are debugging applications.

The following image shows the Custom Handler window. A drop-down list contains the available methods. A text area indicates what variables you can use in the handler, and what type of value the method must return. The remainder of the window consists of the source code editor.





### To dynamically change the behavior of Java methods

1. In the [Classes window](#), right-click the class that includes the method and select View Class in VSE Console. The VSP window appears.
2. In the upper panel, right-click the class and select Edit VSE Handler. The Custom Handler window appears.
3. Select the method from the drop-down list.
4. In the source code editor, write the custom handler. When you are done, click Apply and click Close.
5. In the VSP window, click Playback.

### Obtaining a List of Suggested Classes to Virtualize

#### To obtain a list of suggested classes to virtualize

1. In the VSP window, click Analyze. The User Action Required message appears.
2. Generate transactions.
3. Click OK. The VSE Candidates window appears. The window contains the list of suggested classes. For each class, a score is provided.

### Making the Dev Console Act as a Virtual Service Environment (VSE)

You can make the Dev Console act as a Virtual Service Environment (VSE) by clicking the Register button.

If an agent is in VSE recording mode, the console receives recording events. If an agent is in VSE playback mode, the console provides VSE responses to the agent.

When you are in VSE playback mode, the console has nothing to respond with because it does not have a virtual service model and service image. However, if you record some transactions while the console is listening, it will create a trivial model and image in-memory (it is basically a map of responses keyed by requests without any substitution or custom logic). You can use that as a trivial VSE to quickly test out some recordings before you go through the whole VSE deployment cycle.

The button label toggles between Register and Unregister. When you are ready to use a real VSE, click Unregister.

## Dev Console Videos

The following videos explore different aspects of using the Dev Console.

### Using Pathfinder to Test LISA Workstation and the Demo Server

[Using Pathfinder to Test LISA Workstation and the Demo Server](#)

### Rules and Extensions

[Rules and Extensions](#)

### Superstacks, Java VSE Intro, Heap Access Intro

[Superstacks](#), [Java VSE Intro](#), [Heap Access Intro](#)

## Exercises

[Exercises](#) (download the [sample application](#))

## Tickets in LISA Tests

[Auto Tickets](#)

# Developing Against the Agent

The main interface to the agent from the client side is [com.itko.lisa.remote.client.AgentClient](#). It has methods to invoke APIs on the Agents or the Broker, to discover Agents and to be notified of their status changes (online/offline).

It also gives access to the classes responsible for Agent interaction in the main areas of functionality:

[com.itko.lisa.remote.client.AgentClient](#)  
[com.itko.lisa.remote.client.DiscoveryClient](#)  
[com.itko.lisa.remote.client.TransactionsClient](#)  
[com.itko.lisa.remote.client.VSEClient](#)

[Agent General APIs](#)  
[Agent Discovery APIs](#)  
[Agent Transaction APIs](#)  
[Agent VSE APIs](#)  
[Agent LEK APIs](#)  
[Agent API Examples](#)

## Agent General APIs

Most of the client APIs need to specify an agent as their target. This is accomplished by passing a parameter of type [com.itko.lisa.remote.IAgentInfo](#). This represents an object that uniquely identifies an agent and some of its basic information.

```

/** A unique identifier for this agent or console. If it is named the guid is persistent across VM
lifespans */
public long getGuid();
public void setGuid(long guid);

/** A human-readable name to identify this agent, manually given or generated from system
properties */
public String getName();
public void setName(String name);

/** The name of the machine this object was generated on (if available) */
public String getMachine();
public void setMachine(String machine);

/** The IP of the machine this object was generated on (if available) */
public String getIp();
public void setIp(String ip);

/** The working directory of the JVM this object runs in */
public String getWorkingDir();
public void setWorkingDir(String workingDir);

/** The classpath as it is returned by the java.class.path property */
public String getClasspath();
public void setClasspath(String classpath);

/** The library path as it is returned by the java.library.path property */
public String getLibpath();
public void setLibpath(String libpath);

/** For agents, returns the java class containing the main method that was invoked */
public String getMainClass();
public void setMainClass(String mainClass);

/** A short-hand version of the command-line (for representation purposes as it may not be
accurate) */
public String getCommandLine();

/** Transient field to keep track of when this object is sent or received */
public Date getGenerationTime();
public void setGenerationTime(Date time);

/** Transient field to keep track of a required password to invoke APIs on this agent over JMS */
public String getToken();
public void setToken(String token);

```

We can now look at some of the APIs directly off [com.itko.lisa.remote.client.AgentClient](#). This list is not exhaustive but should cover most of the needs of most clients.

```

/** Gets the class responsible for all discovery and information for a given agent */
public DiscoveryClient getDiscoveryClient();

/** Gets the class responsible for all transaction related operations */
public TransactionsClient getTransactionClient();

/** Gets the class responsible for all VSE related operations */
public VSEClient getVSEClient();

/**
 * Get all the discovered agents - this is the main API to get IAgentInfos used in all other API
calls
 * @param tokens a map of agent guids or names to tokens, null if no agent has token-enabled
security
 * @return a set of IAgentInfo objects representing agents that are currently online
 */
public Set getRemoteAgentInfos(Map tokens);

/**
 * Forward agent online information to registered listeners
 * @param info the agent that was just detected to come online
 */
public void onAgentOnline(IAgentInfo info);

/**
 * Forward agent offline information to registered listeners
 * @param info the agent that was just detected to go offline
 */
public void onAgentOffline(IAgentInfo info);

/**
 * Evaluate arbitrary code on the specified agent
 * @param info the agent this code will be evaluated against
 * @param input the code to execute. The variable '_agent' represents the Agent instance.
 * @return the value returned by the code
 * @throws JMSInvocationException if the code throws on the agent
 */
public Object eval(IAgentInfo info, String input) throws JMSInvocationException

```

## Agent Discovery APIs

The [com.itko.lisa.remote.client.DiscoveryClient](#) class provides methods pertaining to discovering data on an Agent. You can get the **DiscoveryClient** class with the following call: **AgentClient.getInstance().getDiscoveryClient()**

```

/**
 * The system properties of the specified agent
 * @param info
 * @return
 * @throws JMSInvocationException
 */
public Map getVMProperties(IAgentInfo info) throws JMSInvocationException;

/**
 * Returns a list of StatsFrames recorded for the specified agent between the from and the to
dates.
 * @param agentInfo the agent for which to retrieve statistics
 * @param from      how far back to filter
 * @param to        how recently to filter
 * @return          the desired StatsFrames list ordered by decreasing time (starting at the to
date)
 */
public List getStatistics(IAgentInfo agentInfo, Date from, Date to);

/**
 * Returns a list of StatsFrames recorded for the specified agent between the from and the to

```

```

dates.
    * @param agentInfo the agent for which to retrieve statistics
    * @param from       how far back to filter
    * @param to         how recently to filter
    * @param interval   how to aggregate the results in seconds. 10 means average the results of every
10 secs, etc...
    * @param limit      the maximum number of results
    * @return           the desired StatsFrames list ordered by decreasing time (starting at the to
date)
    */
    public List getStatistics(IAgentInfo agentInfo, Date from, Date to, int interval, int limit);

    /**
    * TODO: (re)implement - currently will throw
    * @param info
    * @return
    * @throws JMSInvocationException
    */
    public Topology getTopology(IAgentInfo info) throws JMSInvocationException;

    /**
    * Exit points are MethodInfo that capture classes/methods that make network calls down the stack
    * @param info
    * @return
    * @throws JMSInvocationException
    */
    public Set getExitPoints(IAgentInfo info) throws JMSInvocationException;

    /**
    * Returns the name of the J2EE container (or java if it's not a J2EE container)
    * @param info
    * @return
    * @throws JMSInvocationException
    */
    public String getServerInfo(IAgentInfo info) throws JMSInvocationException;

    /**
    * Returns the web applications deployed in the specified J2EE container
    * @param info
    * @return
    */
    public WebApplication[] getWebApps(IAgentInfo info) throws JMSInvocationException;

    /**
    * Returns the JNDI hierarchy on the specified agent represented by a ClassNode tree
    * @param info
    * @return
    * @throws JMSInvocationException
    */
    public ClassNode getJNDIRoot(IAgentInfo info) throws JMSInvocationException;

    /**
    * The current threads on the agent VM
    * @param info
    * @return
    * @throws JMSInvocationException
    */
    public ThreadInfo[] getThreadInfos(IAgentInfo info) throws JMSInvocationException;

    /**
    * The current threads stacks on the agent VM
    * @param info
    * @return
    */
    public String[] dumpThreads(IAgentInfo info) throws JMSInvocationException;

    /**
    * The set of all files in the classpath of the specified agent
    * @param info
    * @return

```

```

    * @throws JMSInvocationException
    */
    public Set getClasspath(IAgentInfo info) throws JMSInvocationException;

    /**
     * Returns the class hierarchy under the specified path
     * @param info
     * @param fromPath
     * @return
     */
    public ClassNode getClassNodes(IAgentInfo info, String fromPath) throws JMSInvocationException;

    /**
     * The class hierarchy found in the archive at the given url
     * @param info
     * @param url
     * @return
     * @throws JMSInvocationException
     */
    public ClassNode getArchiveNodes(IAgentInfo info, URL url) throws JMSInvocationException;

    /**
     * A set containing data about the class (fields/methods/src)
     * @param info
     * @param className
     * @return
     * @throws JMSInvocationException
     */
    public Set getClassInfo(IAgentInfo info, String className) throws JMSInvocationException;

    /**
     * Decompile and return the source to a class
     * @param info
     * @param clazz
     * @param loc decompile on the client or in the agent
     * @return
     * @throws JMSInvocationException
     */
    public String getClassSrc(IAgentInfo info, String clazz, boolean loc) throws
    JMSInvocationException, IOException;

    /**
     * Returns the hierarchy this class belong to, i.e., all ancestors but also all
    extenders/implementers
     * @param info
     * @param className
     * @return
     * @throws JMSInvocationException
     */
    public ClassNode[] getClassHierarchy(IAgentInfo info, String className) throws
    JMSInvocationException;

    /**
     * Returns (references to) all objects in the heap of the specified class - use with caution
     * @param info
     * @param className
     * @return
     * @throws JMSInvocationException
     */
    public ClassNode getInstancesView(IAgentInfo info, String className) throws
    JMSInvocationException;

    /**
     * Returns (references to) all objects on the heap tracked by the agent
     * @param info
     * @return
     * @throws JMSInvocationException
     */
    public ClassNode getTrackedObjects(IAgentInfo info) throws JMSInvocationException;

```

```
/**
 * A crude graph representation of an object (recursively computed fields)
 * @param info
 * @param clazz
 * @param hashCode
 * @return
 * @throws JMSInvocationException
 */
public ClassNode getObjectGraph(IAgentInfo info, String clazz, int hashCode) throws
JMSInvocationException;

/**
 * Gets the path from an object to a GC root
 * @param info
 * @param clazz
 * @param hashCode
 * @return
 * @throws JMSInvocationException
 */
public ClassNode getRootPath(IAgentInfo info, String clazz, int hashCode) throws
JMSInvocationException;

/**
 * Gets a file on the agent filesystem, downloads it to the client in a temp location and return a
handle to it
 * @param info
 * @param file
 * @return
 */
```

```

    * @throws JMSInvocationException
    */
    public File getFile(IAgentInfo info, String file) throws JMSInvocationException, IOException;

```

In the statistics related APIs shown, the results are lists of [com.itko.lisa.remote.stats.StatsFrame](#) objects, which are POJOs used to hold various counter values (more platform-specific - ala `vmstat/iostat/netstat/sar/...` - may be added in the future):

```

/** The Agent Id this statistics frame was recorded for */
public long getAgentId();
public void setAgentId(long agentId);

/** The time at which this frame was recorded */
public long getTime();
public void setTime(long time);

/** The total CPU usage for this process during the last sampling interval */
public long getCpuUsage();
public void setCpuUsage(long cpuUsage);

/** The total memory consumed by the heap at the time this was recorded */
public long getHeapUsage();
public void setHeapUsage(long heapUsage);

/** The total memory consumed by non-heap resources at the time this was recorded */
public long getNonHeapUsage();
public void setNonHeapUsage(long nonHeapUsage);

/** The number of bytes received (over the network) during the last sampling interval */
public long getIoIn();
public void setIoIn(long ioIn);

/** The number of bytes emitted (over the network) during the last sampling interval */
public long getIoOut();
public void setIoOut(long ioOut);

/** The number of garbage collection events during the last sampling interval */
public long getGcCount();
public void setGcCount(long gcCount);

/** The time spent garbage collecting during the last sampling interval */
public long getGcTime();
public void setGcTime(long gcTime);

```

In all APIs shown, the sampling interval's default value is 1 second. You can modify the interval by using the **STATS\_SAMPLING\_INTERVAL** key in the **rules.properties** file. Other statistics configuration properties can be tuned as documented in the **rules.properties** file.

## Agent Transaction APIs

A transaction is a code path executed by one or more servers as the result of a client request. It is represented by a tree structure rooted at the client initiating the request, and nodes of that tree are [com.itko.lisa.remote.transactions.TransactionFrame](#) objects. They encapsulate information about a class, method and arguments that were invoked as part of the server processing. They also contains ancillary information, such as duration, time of execution, thread in which it occurred, and so on.

You can think of a transaction as a method call stack. The main differences are that transactions cross thread, process, or even machine boundaries and stacks contains all methods involved in a thread's code execution whereas transactions skip some levels and have frames only for chosen method of interest (those are named *intercepted methods*).

The main way to obtain and work with those **TransactionFrame** objects is through a few overloads of the following APIs supplied by [com.itko.lisa.remote.client.TransactionsClient](#), which in turn can be obtained with the following call:

**AgentClient.getInstance().getTransactionsClient()**

```

/**
 * Start recording transactions
 * @param info the agent to start recording on

```



```

    */
    public void startXRecording(IAgentInfo info) throws JMSInvocationException;

    /**
     * Stop recording transactions
     * @param info the agent to stop recording on
     */
    public void stopXRecording(IAgentInfo info) throws JMSInvocationException;

    /**
     * Start tracking socket usage on the client to use in reconciling client and server transactions.
     * This must be called prior to the call we're adding in addClientTransaction.
     * @param global install on all sockets or only on this thread
     */
    public void installSocketTracker(boolean global);

    /**
     * Stop tracking socket usage on the client to use in reconciling client and server transactions.
     * This should be called after the call we're adding in addClientTransaction.
     * @param global uninstall on all sockets or only on this thread
     */
    public void uninstallSocketTracker(boolean global);

    /**
     * As a client, you can invoke this method to root a transaction tree at a new client transaction
     created using
     * the specified parameters.
     * Note: you must call installSocketTracker before you initiate the client side transaction you're
     adding here
     * and it is recommended you call uninstallSocketTracker after you're done with the network call.
     * @param stamp a unique string you can later use to identify the root transaction
     * (in calls to getTransactions for ex.)
     * @param stepInfo a nice human-readable string that tells what the transaction is doing (can be
     null)
     * @param args the parameters the client passes to the transaction (can be empty)
     * @param result the result of the transaction (can be null)
     * @param duration the client's view of the transaction duration
     */
    public void addClientTransaction(String stamp, String stepInfo, Object[] args, Object result,
    long duration);

    /** Delete all transactions and dependent data that originated from this client. */
    public void clearTransactions();

    /**
     * Gets a flat list of TransactionFrames received from all agents that satisfy the filters passed
     as arguments
     * @param offset offset
     * @param limit max number of results
     * @param category filter by transaction category (see TransactionFrame.CATEGORY_XXX - 0 for no
     filter)
     * @param clazz filter by class name (null or "" for no filter)
     * @param method filter by method name (null or "" for no filter)
     * @param minTime filter by frame duration greater than minTime
     * @return a list of TransactionFrames satisfying the supplied criteria ordered by
     decreasing time
     */
    public List getTransactions(int offset, int limit, int category, String clazz, String method, int
    minTime);

    /**
     * Get a list of transaction trees rooted at the specified transaction(s) and satisfying the
     specified criteria
     * @param offset offset
     * @param limit max number of results
     * @param stamps an array of root client transaction stamps - returns all if this is empty
     * @param minTime duration below which transactions are pruned out of the results

```

```

    * @return ret      a list of transaction trees matching the criteria ordered by decreasing time
    */
    public List getTransactionsTree(int offset, int limit, String[] stamps, int minTime)

```

The parameters to these APIs do not specify an Agent or AgentInfo because transactions can span multiple Agents.

Those APIs returns list of [com.itko.lisa.remote.transactions.TransactionFrame](#) (or trees thereof) so let us look at what data they encapsulate:

```

/** A unique identifier for this frame */
public String getFrameId();
public void setFrameId(String frameId);

/** The frame id of this frame's parent frame
public String getParentId();
public void setParentId(String parentId);

/** An identifier shared by all frames belonging to the same transaction (same as global root
frame id) */
public String getTransactionId();
public void setTransactionId(String transactionId);

/** The parent TransactionFrame object */
public TransactionFrame getParent();
public void setParent(TransactionFrame parent);

/** The list of child TransactionFrame objects */
public List getChildren();
public void setChildren(List children);

/** The unique identifier of the Agent this frame was recorded in */
public long getAgentGuid();
public void setAgentGuid(long agentGuid);

/** Increasing counter that helps order the frames (time may not be precise enough) */
public long getOrdinal();
public void setOrdinal(long ordinal);

/** Network incoming or outgoing frames set this to tell us what IP they are talking from */
public String getLocalIP();
public void setLocalIP(String ip);

/** Network incoming or outgoing frames set this to tell us what port they are talking from */
public int getLocalPort();
public void setLocalPort(int port);

/** Network incoming or outgoing frames set this to tell us what IP they are talking to */
public String getRemoteIP();
public void setRemoteIP(String ip);

/** Network incoming or outgoing frames set this to tell us what port they are talking to */
public int getRemotePort();
public void setRemotePort(int port);

/** The name of the thread this frame was recorded in */
public String getThreadName();
public void setThreadName(String threadName);

/** Name of the class or interface this frame was recorded in (as per the interception spec) */
public String getClassName();
public void setClassName(String className);

/** Name of the class of the actual object this frame was recorded in */
public String getActualClassName();

/** Name of the method this frame was recorded in */
public String getMethod();

```

```
public void setMethod(String method);

/** Signature of the method this frame was recorded in */
public String getSignature();
public void setSignature(String signature);

/** Formatted representation of the object this frame was recorded in */
public String getSource();
public void setSource(Object source);

/** Formatted representation of the arguments to the method this frame was recorded in */
public String[] getArguments();
public void setArguments(Object[] arguments);

/** Formatted representation of the result of the method this frame was recorded in */
public String getResult();
public void setResult(Object result);

/** Number of times this frame was duplicated within its parent */
public long getHits();
public void setHits(long hits);

/** Server time at which the frame was recorded */
public long getTime();
public void setTime(long time);

/** Wall clock duration this frame took to execute */
public long getClockDuration();
public void setClockDuration(long clockDuration);

/** CPU duration this frame took to execute */
public long getCpuDuration();
public void setCpuDuration(long cpuDuration);

/** Custom representation of state associated with this frame */
public String getState();
public void setState(Object state);

/** Formatted LEK info as encoded/decoded by the LEKEncoder class */
public String getLekInfo();
public void setLekInfo(String lekInfo);

/** Pre-computed category this frame belongs to - see TransactionFrame.CATEGORY_XXX */
public int getCategory();
public void setCategory(int category);

/** Bitwise or'ed combination of various internal pieces of information - see
```

```
TransactionFrame.FLAG_XXX */  
public long getFlags();  
public void setFlags(long flags);
```

## Agent VSE APIs

The Virtual Service Environment (VSE) enables users to stub out processes, services, or parts of them along well-defined boundaries and replace their internals with a layer run by LISA according to custom user-defined rules - also known as a model. Usually a default starting point for those rules is obtained from a recording of live system interactions.

LISA already supports VSE for the HTTP protocol (allowing virtualization of web applications and web services), JMS, and JDBC to a certain extent. The Agent provides APIs to enable virtualization directly from within server processes, thus making it protocol agnostic. It can be enabled of course for HTTP, JMS, and JDBC but also for RMI, EJB, or any custom Java objects.

LISA (or any other client of the Agent) can achieve this virtualization by using the following APIs defined in [com.itko.lisa.remote.client.VSEClient](#) as obtained by **AgentClient.getInstance().getVSEClient()**.

```

/**
 * Returns a list of all class/interface names whose name matches the supplied regular expression
 * or that extend/implement a class/interface whose name matches the supplied regular expression
 * if implementing is true. Searching for annotations is supported through the syntax:
 * class regex@annotation regex (e.g. ".*.Remote@.*.Stateless").
 * @param agentInfo
 * @param regex
 * @param impl
 * @return
 */
public String[] getMatchingClasses(IAgentInfo info, String regex, boolean impl) throws
JMSInvocationException

/**
 * Register a VSE callback with the specified agent whose onFrameRecord will be invoked
 * in recording mode for all virtualized methods and whose onFramePlayback method will be invoked
 * in playback mode for all virtualized methods.
 * @param info
 * @param callback
 */
public void registerVSECallback(IAgentInfo info, IVSECallback callback);

/**
 * Unegister a VSE callback with the specified agent.
 * @param info
 * @param callback
 */
public void unregisterVSECallback(IAgentInfo info, IVSECallback callback);

/**
 * Start calling our virtualization recording callback on the specified agent.
 * @param agentInfo
 * @throws RemoteException
 */
public void startVSERecording(IAgentInfo agentInfo) throws JMSInvocationException

/**
 * Start calling our virtualization playback callback on the specified agent.
 * @param agentInfo
 * @throws RemoteException
 */
public void startVSEPlayback(IAgentInfo agentInfo) throws JMSInvocationException

/**
 * Stop virtualizing on the specified agent.
 * @param agentInfo
 * @throws RemoteException
 */
public void stopVSE(IAgentInfo agentInfo) throws JMSInvocationException

/**
 * Virtualizes the specified class/interface and all its descendants on the specified agent.
 * @param agentInfo
 * @param className
 * @return
 */
public void virtualize(IAgentInfo agentInfo, String className) throws JMSInvocationException

```

The interface for the callback APIs is defined by [com.itko.lisa.remote.vse.IVSECallback](#) and defines the following methods:

```

/**
 * This is the method that gets invoked by agents that have VSE recording turned on
 * when a virtualize method gets called. The VSE frame has all the information needed
 * to later replay the method in playback mode.
 * @param frame
 * @throws RemoteException
 */
void onFrameRecord(VSEFrame frame) throws RemoteException;

/**
 * This is the method that gets invoked by agents that have VSE playback turned on
 * when a virtualize method gets called. The VSE frame has all the information needed
 * to match an existing recorded frame so its result (and by reference arguments)
 * can be set appropriately.
 * @param frame
 * @return
 * @throws RemoteException
 */
VSEFrame onFramePlayback(VSEFrame frame) throws RemoteException;

```

Finally, the `com.itko.lisa.remote.vse.VSEFrame` object is a POJO with getters and setters for the following properties:

```

/** Get/sets a unique identifier for this frame */
public String getFrameId();
public void setFrameId(String frameId);

/** The agent id this frame originates from */
public long getAgentGuid();
public void setAgentGuid(long agentId);

/** The thread name this frame method was invoked on */
public String getThreadName();
public void setThreadName(String threadName);

/** The name of the class this frame method was invoked on */
public String getClassName();
public void setClassName(String className);

/** A unique identifier that tracks objects for the span of the VM's life */
public String getSourceId();
public void setSourceId(String srcId);

/** The session id of the innermost session-scoped protocol enclosing this frame */
public String getSessionId();
public void setSessionId(String sessionId);

/** The name of the method that was invoked */
public String getMethod();
public void setMethod(String method);

/** The XStream'ed arguments array to the method that was invoked */
public String[] getArgumentsXML();
public void setArgumentsXML(String[] argumentsXML);

/** The XStream'ed result of the method that was invoked */
public String getResultXML();
public void setResultXML(String resultXML);

/** The (server) time the method was invoked */
public long getTime();
public void setTime(long time);

/** The time the method took to execute */
public long getClockDuration();
public void setClockDuration(long duration);

/** Whether to use getCode or the ResultXML to compute the desired result in playback mode */
public boolean isUseCode();
public void setUseCode(boolean useCode);

/**
 * The code to execute on the server if isUseCode is true. This can be arbitrary code
 * that has access to the object ($0) and method arguments ($1, $2,...)
 */
public String getCode();
public void setCode(String code);

```

## Agent LEK APIs

The LISA Extension Kit (LEK) enables client to write code in their application that will be automatically be picked up by the agent and be sent back to LISA for further processing.

This generation of the LEK is designed in such a way that it requires no dependencies whatsoever on LISA code or files, thus the application can still run without the agent with no impact.

The generic way to make an LEK call is by creating a **java.util.Properties** object and setting some well-known keys on it. This is better understood using an example:

```

Properties p = new Properties();

/** Tell LISA to issue a log message event with message "The quick brown fox" upon step completion */
p.put("lisa.log.msg", "The quick brown fox");

/** Tell LISA to generate a warning event, including the stack trace of the supplied exception */
p.put("lisa.log.warning", new Object[] { "jumped over the lazy dog", new RuntimeException("oops")
});

/** Tell LISA to raise a property set event using the specified key-value pair */
p.put("lisa.set.prop", new Object[] { "lek", "cool" });

/** Sets the next node in the flow of the current LISA test (thus tests can be remote controlled
server-side) */
p.put("lisa.set.next", "abort");

/** Raises a LISA event using a TestEvent id, message, description and optionally a next node */
p.put("lisa.raise.event", new Object[] { 20, "short msg", "long msg" });

```

The last call is the general form of the other ones, which are just provided for convenience:

```

new Properties().put("lisa.raise.event", new Object[] { eventId, message, description, nextNode });

```

The message and description fields need not be strings. If they are throwables, their stack trace will be returned, if they are non-string objects, their XML representations (XStream'ed) will be returned.

The list of well-known keys is as follows (subject to change before release):

- `lisa.log.msg` (Event ID = 21)
- `lisa.log.warning` (Event ID = 41)
- `lisa.set.prop` (Event ID = 14)
- `lisa.set.next` (Event ID = 0)
- `lisa.raise.event` (Event ID = \*)

The **rules.properties** file contains a configurable Boolean property called **LEK\_LOG\_CALLS** that controls whether LEK API calls will generate log messages on the server side.

As a stylistic note, the following code is equivalent to the example shown previously (and some may prefer it):

```

System.getProperties().put("lisa.raise.event", new Object[] { eventId, message, description,
nextNode });

```

Finally, if client code makes extensive use of these LEK calls, it may be beneficial to create a tiny wrapper class that wraps these calls to help eliminate typos (the price of no JAR dependencies):



```

public class LEK {
    public void log(Object msg, Object desc) {
        new Properties().put("lisa.log.msg", new Object[] { msg, desc });
    }

    public void warn(Object msg, Object desc) {
        new Properties().put("lisa.log.warning", new Object[] { msg, desc });
    }

    public void setProperty(String key, Object value) {
        new Properties().put("lisa.set.prop", new Object[] { key, value });
    }

    public void setNext(String next) {
        new Properties().put("lisa.set.next", next);
    }

    public void raiseEvent(int id, String msg, Object desc, String next) {
        new Properties().put("lisa.raise.event", new Object[] { id, msg, desc, next });
    }
}

```

On the LISA side, when a [com.itko.lisa.remote.transactions.TransactionFrame](#) tree is received, all the LEK information that was generated on the server can be retrieved from it using the [com.itko.lisa.remote.lek.LEKEncoder](#) class, which has the following relevant APIs:

```

/**
 * Extracts LEK data from a frame (if any) and deserializes it as a list of LEKEvent objects.
 * Note this extracts LEK data only for the supplied frame and not its hierarchy.
 * @param frame
 * @return
 */
public static List decode(TransactionFrame frame);

/**
 * Finds the next node as set in a transaction tree using total node ordering on the tree
 * defined by left to right and top to bottom
 * (essentially - but not necessarily exactly - following the code timeline that generated this
 * tree).
 * @param frame
 * @return
 */
public static String findNext(TransactionFrame frame);

/**
 * Returns a list of all LEKEvent objects generated and stored on the supplied transaction tree
 * (in timeline order - see findNext).
 * This is essentially the same as decode but extracts LEK data for the whole hierarchy.
 * @param frame
 * @return
 */
public static List decodeAll(TransactionFrame frame);

```

The results returned by those APIs are encapsulated in POJOs called [com.itko.lisa.remote.lek.LEKEvent](#). Those are a lightweight version of LISA's **TestEvent** objects:

```

/** The TestEvent id that was used when invoking the "lisa.raise.event" LEK call */
public int getEvtId();
public void setEvtId(int evtId);

/** The xml-ized (if need be) message
public String getMsg();
public void setMsg(String msg);

/** The xml-ized (if need be) description
public String getDesc();
public void setDesc(String desc);

/** Optionally a next next node */
public String getNext();
public void setNext(String next);

/** Key-value pairs used in "lisa.set.prop" calls */
public String getKey();
public String getValue();
public void setKeyValue(String key, String value);

```

## Agent API Examples

**Example 1:** Generating a transaction from client code and retrieving the transaction tree it generated

```

private static void testAddTransaction() throws Exception
{
    String request = "http://localhost:8080/examples/servlets/servlet/HelloWorldExample";
    TransactionsClient tc = AgentClient.getInstance().getTransactionClient();

    tc.installSocketTracker(false);
    long start = System.currentTimeMillis();

    String response = testMakeRequest(request);

    long end = System.currentTimeMillis();
    tc.uninstallSocketTracker(false);

    String frameId = tc.addClientTransaction("test", new Object[] { request }, response, end - start);
    TransactionFrame frame = tc.getTransactionTree(frameId);

    System.out.println(frame.isComplete() ? "Yes!" : "No!");
}

```

This sample makes use of the following utility function that has nothing to do with the Agent but is listed for completeness sake.

```

/** Assuming Tomcat is running on localhost:8080 */
private static String testMakeRequest(String url) throws IOException
{
    StringBuffer response = new StringBuffer();
    HttpURLConnection con = (HttpURLConnection) new URL(url).openConnection();

    con.setRequestMethod("GET");
    con.setDoOutput(true);
    con.setUseCaches(false);

    BufferedReader br = new BufferedReader(new InputStreamReader(con.getInputStream()));
    for (String line = br.readLine(); line != null; line = br.readLine()) response.append(line);
    br.close();

    return response.toString();
}

```

### Example 2: Registering a logging VSE callback

```

AgentClient.getInstance().addListener(new IAgentEventListener()
{
    public void onAgentOffline(final IAgentInfo info) {}
    public void onAgentOnline(final IAgentInfo info)
    {
        AgentClient.getInstance().getVSEClient().registerVSECallback(info, new IVSECallback()
        {
            public void onFrameRecord(VSEFrame frame) { System.out.println("Recorded: " + frame); }
            public VSEFrame onFramePlayback(VSEFrame frame) { System.out.println("Played back: " + frame);
return frame; }
            public int hashCode() { return 0; }
            public boolean equals(Object o) { return o instanceof IVSECallback; }
        });

        try { AgentClient.getInstance().getVSEClient().startVSERecording(info); } catch
(JMSInvocationException e) {}
    });
}

```

## Agent Custom Extensions

### Agent Extensions

You can customize the Agent behavior by writing Java classes that implement the **com.itko.lisa.remote.transactions.interceptors.IInterceptor** interface or extend the **com.itko.lisa.remote.transactions.interceptors.AbstractInterceptor** class.

```

public interface IInterceptor {
    /**
     * Whether this custom interceptor is currently disabled
     */
    public boolean isDisabled();

    /**
     * Returns whether the interception should return (true) or proceed (false)
     */
    public boolean block(boolean wayIn, Object src, String spec, String clazz, String method,
String signature, Object[] args, Object ret);

    /**
     * Called after method entry to possibly modify the current frame based on interceptor logic.
     */
    public boolean preProcess(TransactionFrame frame, Object src, String spec, String clazz, String
method, String signature, Object[] args, Object ret);

    /**
     * Called before method exit to possibly modify the current frame based on interceptor logic
     */
    public boolean postProcess(TransactionFrame frame, Object src, String spec, String clazz,
String method, String signature, Object[] args, Object ret);
}

```

These methods are automatically invoked for all classes and methods that have been intercepted or tracked. To intercept a method or track a class, you can specify it using the previously documented syntax in the **rules.properties** file, or you can programmatically specify the interception in the extension class's constructor. The advantage of the latter is that it makes the extension completely self-contained.

To deploy an extension, compile it and package it in a JAR file with a manifest containing the following entry:

```

Agent-Extension: extension class name

```

When you drop this JAR in the LISA Agent JAR directory, the Agent will automatically pick it up. A hot load mechanism will help to ensure that if you add the file after the Agent has been started, it will be applied. If you update the extension JAR as the Agent is running, the JAR classes will be reloaded dynamically, making it easy and fast to test your extension code without restarting the server.

The extension JARs get loaded by a classloader that can see all the classes used in the extension class. You can compile your extension source against the `LisaAgent.jar` and all container JARs that define classes you may want to use, so you do not need to use reflection in your extension.

The following sample shows how to print the XML contents of a `WebMethods.com.wm.data.IData` objects as it gets invoked in a flow of Integration Server. The agent already supports WM and an extension is not necessary for it; this is just an example.

```

package com.itko.lisa.ext;

import java.io.ByteArrayOutputStream;
import com.itko.lisa.remote.transactions.interceptors.AbstractInterceptor;
import com.itko.lisa.remote.transactions.TransactionFrame;
import com.wm.app.b2b.server.ServiceManager;
import com.wm.app.b2b.server.BaseService;
import com.wm.data.IData;
import com.wm.util.coder.IDataXMLCoder;

public class IDataInterceptor extends AbstractInterceptor {

    public IDataInterceptor() {
        super.intercept("com.wm.app.b2b.server.ServiceManager", "invoke",
"(Lcom/wm/app/b2b/server/BaseService;Lcom/wm/data/IData;Z)Lcom/wm/data/IData;");
    }

    public boolean preprocess(TransactionFrame frame, Object src, String spec, String clazz, String
method, String signature, Object[] args, Object ret) {
        if (ServiceManager.class.getName().equals(clazz) && "invoke".equals(method)) {
            doCustomLogic((BaseService) args[0], (IData) args[1], false);
        }

        return super.preProcess(frame, src, spec, clazz, method, signature, args, ret);
    }

    public boolean postProcess(TransactionFrame frame, Object src, String spec, String clazz, String
method, String signature, Object[] args, Object ret) {
        if (ServiceManager.class.getName().equals(clazz) && "invoke".equals(method)) {
            doCustomLogic((BaseService) args[0], (IData) ret, true);
        }

        return super.preProcess(frame, src, spec, clazz, method, signature, args, ret);
    }

    private void doCustomLogic(BaseService flow, IData p, boolean output) {
        ByteArrayOutputStream baos = new ByteArrayOutputStream(63);

        try {
            new IDataXMLCoder().encode(baos, p);
        } catch (IOException e) {
            e.printStackTrace();
        }

        System.out.println("Flow: " + flow.getNSName());
        System.out.println((output ? "Output" : "Input") + "Pipeline: " + baos);
    }
}

```

To build this extension yourself, you need to compile this code against **LisaAgent.jar**, **wm-isclient.jar** and **wm-isserver.jar** then JAR the class file with a manifest containing the line:

```
Agent-Extension: com.itko.lisa.ext.IDataInterceptor
```

Alternatively, you can download this sample extension here: [IDataExtension.jar](#). A demo video is also available [here](#).

## Broker Extensions

Another type of extension is the broker extension. Instead of being loaded and invoked by the Agent while transactions are being captured, it is loaded and invoked by the broker after a transaction fragment is received from an Agent. You can change data contained in frames, add or eliminate certain frames, or (most commonly) customize the stitching algorithm (that is, how transaction fragments are assembled).

You must implement the **com.itko.lisa.remote.plumbing.IAssemblyExtension** interface:

```

public interface IAssemblyExtension {
    /**
     * The method invoked prior to assembly
     * @param frame the partial transaction root received from an agent
     * @return true to bypass normal assembly (i.e. if the extension wants to take care of it
     itself)
     */
    public boolean onTransactionReceived(TransactionFrame frame);
}

```

To deploy a broker extension, compile it and package it in a JAR file with a manifest containing the following entry:

```

Broker-Extension: extension class name

```

If you use the [Dev Console](#), all of this is done automatically.

When you drop this JAR in the LISA Broker (Registry) directory, the broker will automatically pick it up. A hot load mechanism helps to ensure that if you add the file after the Broker (registry) has been started, it will be applied. If you update the extension JAR as the Broker (Registry) is running, the JAR classes will be reloaded dynamically, making it easy and fast to test your extension code without restarting the Broker (registry).

A typical usage example would be when the normal stitching algorithm that uses TCP/IPs and ports is confused by a load balancer sitting between agents and is assigned a virtual IP, or when agents use a native library to do IO and we do not have direct access to the IPs and ports in use. In that case, the address and port fields of frames are either left blank or incorrect and we need to assemble the frames in an extension:

```

import com.itko.lisa.remote.plumbing.IAssemblyExtension;
import com.itko.lisa.remote.transactions.TransactionFrame;
import com.itko.lisa.remote.utils.Log;
import com.itko.lisa.remote.utils.UUID;

public class LoadBalancerExtension implements IAssemblyExtension {

    private TransactionFrame m_lastAgent1Frame;
    private TransactionFrame m_lastAgent2Frame;

    public boolean onTransactionReceived(TransactionFrame frame) {

        if (frame.getClassName().equals("Class1") && frame.getMethod().equals("method1")) {
            m_lastAgent2Frame = frame;
            if (m_lastAgent1Frame.getFrameId() == m_lastAgent2Frame.getParentId()) {
                stitch(m_lastAgent1Frame, m_lastAgent2Frame);
            }
        }

        if (frame.getClassName().equals("Class2") && frame.getMethod().equals("method2")) {
            m_lastAgent1Frame = frame;
            if (m_lastAgent1Frame.getFrameId() == m_lastAgent2Frame.getParentId()) {
                stitch(m_lastAgent1Frame, m_lastAgent2Frame);
            }
        }

        return false;
    }

    private void stitch(TransactionFrame parent, TransactionFrame child) {
        String newFrameId = UUID.newUUID();
        parent.setFrameId(newFrameId);
        child.setParent(parent);
        parent.getChildren().add(child);
    }
}

```

## Agent Transaction Weight

The **lisa.agent.transaction.weight** property lets you control how much data is recorded for each frame.

The following table describes the base values for this property.

| Value | Description  |
|-------|--|
| 1     | Do not merge frames for multiple invocations. This value is not currently used by the Agent. |
| 2     | Capture arguments and return value.  |
| 4     | Capture request and response.  |
| 8     | Capture request and response using the XStream library.                                      |

As of release 6.0.5, the value 4 and the value 8 are functionally equivalent.

The default value of this property is 7, which is produced by adding the first three base values. The maximum value for this property is 15, which is produced by adding all the base values.

## Load Balancers and Native Web Servers

The process of assembling partial transactions into complete transactions is referred to as *stitching*.

When agents are deployed on a network that has load balancers or native web servers, the stitching algorithm used by Pathfinder may not function properly. In this scenario, you should configure the **loadbalancer** rule in the broker's **rules.properties** file. For each appliance installed between agents, specify the IP address of the appliance. For example:

```
loadbalancer ip=172.16.0.0
loadbalancer ip=172.31.255.255
```

Alternately, you can set the **lisa.broker.transaction.always.wait** property to true.

```
property key=lisa.broker.transaction.always.wait value=true
```

## LISA Agent Troubleshooting



**Important:** It is critical you find out about the target environment ahead of time and make sure it has been tested, or test it yourself (most issues will be OS or JVM-dependent, not application-dependent). Then be sure to carefully follow the instructions supplied in this documentation. If that does not help, here is a review of the most common problems.

The majority of serious issues involving the Agent will happen at start-up (Agent not found, process crashing or hanging, and so on). Generally, after you get past this you are in good shape and it is a matter of tweaking the configuration or writing extensions.



Sometimes it is difficult to bounce servers to try various things in the agent environment, so it is simpler (and a worthy exercise) to test running **java <agent options> -version** on the target machine and JVM. This will let you (and us) know if the issue is OS/JVM-specific or container/application-specific.

### Issue: You get the following error at startup: Error occurred during initialization of VM. Could not find agent library in absolute path...

Make sure the library is indeed in that path and is using the same architecture as Java (both 32 bit or 64 bit).

You can also verify that the library is not missing any dependencies (using **depends.exe** on Win32, **otool -L** on Mac OS X, and **ldd -d** on Linux and UNIX). If libraries are missing, make sure to set **LD\_LIBRARY\_PATH** to include the directories where they reside. Sometimes containers override **LD\_LIBRARY\_PATH** in their start-up scripts, so do not assume that it is correctly set if you simply did it in a shell and not from inside the script or a container-specific admin tool.

If **ldd** does not return cleanly, then the Agent will not run properly, so this is the first thing to get right. If you cannot find an Agent version for the operating system, consider using the pure Java agent.

## Issue: The Agent exits immediately (or shortly after starting the process)

If you see the message **LISA AGENT: VM terminated**, then it is likely that the process terminated normally (several containers have launcher processes and it is normal for them to exit quickly).

If there is no such message, or a crash dump happens (after **ldd** returns cleanly), you may have run into a legitimate agent bug. Notify Support and try the pure Java agent. If it still crashes or produces a dump, you may have run into a JVM bug. One such well-known occurrence is for the IBM JVM 1.5 on some operating systems (caused by trying to instrument threads). In that case, try supplying the command-line JVM argument **-Disa.debug=true** and it should start properly.

## Issue: The Agent exits with the message "GetEnv on jvmdi returned -3 (JNI\_EVERSION)"

This issue may happen with some older 1.4 versions of the IBM JRE.

Try specifying the command line option: **-Xsow** to instruct the JVM to use its debug-enabled libraries. If that does not work (for example, you get an invalid option error message), then this operating system is not supported.

## Issue: The Agent exits with the message "UTF ERROR [\"./././src/solaris/instrument/EncodingSupport\_md.c\":66]: ..."

The message may vary depending on the exact version of the operating system and/or the JVM, but will generally have one of the following elements: UTF ERROR [\"./././src/solaris/instrument/EncodingSupport\_md.c\":66]: Failed to complete iconv\_open() setup.

This issue is due to a bug in some Solaris JVMs when some language packs are not installed. To fix this issue, install the en-US language pack by running **pkg install SUNWlang-enUS** and then **export LANG=en\_US.UTF-8**. Alternatively, you can try using the native LISA agent.

## Issue: The Agent hangs or throws a lot of exceptions at startup (LinkageErrors, CircularityErrors, and so on)

A side effect of instrumenting Java bytecode using Java is that it can subtly alter some of the class-loading order for early classes (**java.\*** and such), resulting in a deadlock or bytecode verification errors.

ITKO has made an effort to find and eliminate those for all combinations of JVMs and OSES, but it is possible that you have run into an untested combination. Notify Support, and if it is a hang, send a thread dump (CTRL+Break on Windows, CTRL+\ or kill -3 <pid> on UNIX/Linux). You may also try the Java agent, as the class-loading order is slightly different. If a class or package seems involved every time in the hung thread, you can try and add an exclude rule for it in the **rules.properties** file.

## Issue: The Agent throws java.lang.VerifyErrors or HotSwapping has no effect

Some older 1.4 JVMs have bugs in their support for hotswapping (instrumenting classes after they have been loaded). In that case, turn off hotswapping by specifying **property key=lisa.agent.enable.hot.instrumentation value=false** in the agent's **rules.properties** file. This means you will have to find out in advance what classes and/or methods you want to intercept or virtualize, add rules for them in **rules.properties**, and bounce the server. This process is more tedious than doing it on a live server, but it is the only known way to work around this issue.

Sometimes the **java.lang.VerifyError** is not even seen in the logs, but the agent behaves as though the designated class has not been instrumented or even yields random results, up to and including crashes.

## Issue: The Agent starts but the consoles or broker cannot see the Agent

This typically means there is a firewall or port problem between the Agent and the broker. At the top of the Agent log, you should see a warning that says **Can't connect to broker at tcp://<ip>:<port>**. Look at this to make sure the IP address and port that the Agent is using are correct. Then make sure the broker is listening on the specified port on the specified ip (**netstat -ano | grep <port>** should show a port listening on the supplied ip or 0.0.0.0). Finally, check for firewall issues by running **telnet <ip> <port>** from the Agent machine to make sure it can see the broker. If it can, the broker is likely in a bad state, so check the registry and broker logs (and restart it if need be).

## Issue: Abnormal resource consumption (CPU, memory, file handles, ...)

If the CPU usage is abnormally high or spikes periodically, note the period of the spikes because it will help determine the faulty thread(s). Also try to turn off Pathfinder (if on) and VSE (if on) from the [Dev Console](#), to see if it helps. Support may provide you with scripts to run from the Dev Console that will enable and disable various components of the Agent so the culprit can be identified.



If you get OutOfMemory errors, monitor the Java heap usage (using the Performance tab in the Dev Console, for example). If it is above the **-Xmx** limit, then increase that limit, unless it is already high and well in excess of normal app usage without the Agent. In that case, generate a heap dump (you can do it from the Dev Console's script sub-console for JRE 1.6 and above. You can use WAS's HeapDump utility in WebSphere's case or the free [Eclipse MAT](#) tool for older versions). If the memory usage is below the **-Xmx** limit at the time of the error, it is likely a leak in native code and you will need to notify Support.

If you get unexplained IOExceptions (like too many file handles), especially on UNIX or Linux, check the ulimit on the box (**ulimit -n -H** and **ulimit -n -S**). If it is low, consider asking the administrator of the box to raise it (under 4,096 is considered low for modern J2EE apps). Do not forget to reboot the machine afterward.

The Agent tries to keep the number of file handles under that limit by triggering garbage collection when the limit is approached. If the limit cannot be read by the Agent at startup, the Agent uses a default value of 1,024 which may result in excessive garbage collection (and semi-random frequent CPU spikes). You will know this is the case by observing these messages in the logs: **Max (or preferred) handles limit approaching - triggering GC...**

In that case, raise the value of the **lisa.agent.handles.max** property in the **rules.properties** file.

## Issue: You can see the Agent started but Pathfinder data is missing or incomplete

There are several stages in the data lifecycle process: transaction capture (in the Agent), transfer to the broker, partial transaction assembly (in the broker), transfer to the consoles, persistence to the database, and retrieval from the database. Missing or incomplete data can be the result of a problem in any of these stages.

First, look in the Agent log for capture exceptions, broker and console logs for transfer or persistence exceptions. Then try to look at the data in the [Dev Console](#) (because it has one fewer layer than the Web console) and in particular at the Live Paths window, because it takes all persistence issues out of the equation.

If there are no exceptions and the data is still nowhere to be found, turn on debug or dev logging in the agents (the easiest way to do this is from the Log window of the Dev Console). You should see statements such as "Sent partial transaction...". If you do not, then Pathfinder is probably not turned on. If none of this is conclusive, contact Support.

## Issue: I turned on Java VSE recording or playback, but VSE does not receive any requests from the Agent(s)

First, check that the Agent is in fact in VSE record or playback mode. You can do that by looking at its log (and look for "Starting VSE record/playback...") or by looking at the Java VSE window of the [Dev Console](#) and seeing the Record or Playback button being disabled.

Then look for exceptions in the Agent logs and the VSE logs. If they are clean, it is possible that the class you think is virtualized is not actually virtualized. Look in the agent log for a statement such as "Virtualized com.xxx....". If it is not there, it is possible that the class simply had not yet been loaded by the app. Another possibility is if you are using an early version of Java 1.4, some flavors do not support HotSwapping (which is what is used by Java VSE by default). In that case, you need to specify the virtualized classes in the **rules.properties** of the agent and restart it.

## Issue: Alt+Click functionality is not working

First, make sure the Agent is connected to a broker (or registry).

If it is, then look at the HTML source of the page that is exhibiting the problem. The bottom of the page should have a block of JavaScript that is easily identifiable by the variable names it uses (for example, **com\_itko\_pathfinder\_defectcapture\_xxx**). If that block is missing, check the Agent log for possible exceptions. Other reasons for its absence include the page HTML being unusual (for example, missing HTML tags) or the Agent having issues capturing it entirely (that could happen with untested containers or static pages).

If the block is present, check if a native web server (such as Apache) or a load balancer is not present in front of the Java container. If it is the case, it may need to be configured to forward request of the Pathfinder JavaScript and resource files to the Java container. Those will have the word **defectcapture** as part of their URL. IT admins generally know how to do this.