



# J A D E <sup>TM</sup>

## Multi-Threading JADE Applications – A Primer

Jade Software Corporation Limited cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages, or loss of profits. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Jade Software Corporation Limited.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Copyright © 2009 Jade Software Corporation Limited.

All rights reserved.

JADE is a trademark of Jade Software Corporation Limited. All trade names referenced are the service mark, trademark, or registered trademark of the respective manufacturer.

---

# Contents

Contents ii

<b>Multi-Threading JADE Applications</b>	<b>3</b>
Introduction .....	3
The Basics .....	3
Nodes and Processes .....	3
Client Nodes.....	3
The Server Node .....	4
Server Applications .....	4
Server Methods .....	4
System, Nodes and Processes in the JADE Database .....	4
Initiating New Application Processes .....	5
Application::startApplication .....	6
Application::startApplicationWithParameter .....	7
Application::startApplicationWithString .....	8
Application::startAppMethod.....	9
Node::createExternalProcess .....	9
Controlling Multi-Threaded Applications.....	10
Debugging Multi-Threaded Applications.....	12
Debugging Processes Started via startApplication or createExternalProcess .....	12
Debugging Processes Started via startApplicationWithParameter, startApplicationWithString or startAppMethod .....	13

---

# Multi-Threading JADE Applications

---

## Introduction

In JADE applications it is sometimes desirable to initiate asynchronously executing applications, threads or processes. This may be required for performance reasons, or to separate the execution of specific functions. Another reason may be a requirement to initiate a separate process on another machine.

This paper describes the basics of multithreading using JADE and provides some examples of the use and control of multithreaded applications.

## The Basics

### Nodes and Processes

In JADE, each application **process** or 'thread' executes within a JADE **node**. Multiple application processes can execute within a single node. A JADE node is physically one of the following executing programs:

- A **JADE.EXE** program, i.e. a fat client (not a thin client).
- A **JADAPP.EXE** program, i.e. an application server for JADE thin clients.
- A **JADRAP.EXE** program, i.e. the JADE database server (Remote Access Program).
- A user program that uses the C-API level of JOM, including .NET and Java Class Libraries.

For the purposes of this paper, there are two main types of nodes: server nodes and client nodes. Note that the distinction between the two is not black-and-white, since at certain times a server node can behave as a client node, and vice versa.

### Client Nodes

A client node is either an executing JADE.EXE program or a JADAPP.EXE program. Multiple application processes can be run within a client node, either by programmatically starting additional process (as we will show later), or by specifying **newcopy=false** in the command line of the operating system shortcut used to initiate the application.

For method execution, the application developer can specify that a given method is to be executed at the client node (using the **clientExecution** command in the method signature) or on the server node (by using the **serverExecution** command in the method signature). Methods that are specified for **serverExecution** are referred to as **server methods**.

By default, a method executes in the same node as its calling method.

## The Server Node

There are two main types of process that execute within the server node (i.e. in a JADRAP.EXE).

These are server applications and server methods.

## Server Applications

Server applications run in the server node and cannot have a user interface. They must be defined as application type Non-GUI. Server applications are normally started at the same time as the JADRAP starts, by including a **ServerApplicationX** statement in the **[JadeServer]** section of the JADE initialization file.

However, as we will see later, server applications can also be initiated programmatically at any time, by any other JADE process.

## Server Methods

Server methods are methods that specify **serverExecution** in the method signature, or methods that are called by another method which has **serverExecution** in its signature. These methods execute on a thread in the JADE database server (JADRAP).

## System, Nodes and Processes in the JADE Database

A JADE system consists of the database and the collections of nodes and processes that go to make up the system. There is one **System** object for each JADE system. The **System** object can be referenced by the developer, using the reserved word **system**.

A persistent **Node** object in the JADE database represents each node in a running JADE system. JADE automatically maintains these node objects. An application process can directly access its own node object through the use of the JADE reserved word **node**.

Each process running in a JADE system is represented by a persistent **Process** object in the database. The Process object is created automatically by JADE when the application process starts, and is destroyed when the application process terminates.

An application process can directly access its own process object through the use of the JADE reserved word **process**.

The RootSchema property **System::nodes** returns a reference to a collection of all nodes in a JADE system. The property **Node::processes** returns a reference to a collection of all processes running on the node.

As a simple example of the use of this, if you wanted to know whether any node was executing a particular application, you could code a method as follows:

```
findApp(appName : String) : Boolean serverExecution;
    //more efficient on server
vars
    allNodes : NodeDict;
    allProcesses : ProcessDict;
```

```
nod : Node;
proc : Process;
begin
  allNodes := system.nodes.cloneSelf(true);
  foreach nod in allNodes do
    allProcesses := nod.processes.cloneSelf(true);
    foreach proc in allProcesses do
      if proc.persistentApp.name = appName then
        return true;
      endif;
    endforeach;
    delete allProcesses;
  endforeach;
  return false;
epilog
  delete allNodes;
  delete allProcesses;
end;
```

It is important not to directly use **system.nodes** and **node.processes** with the **foreach** statements, because that causes them to be locked. These two collections are updated whenever nodes and processes start up or terminate, so locking them could disrupt system accessibility.

---

**Note** In actual use, the above example should arm an exception handler, to protect against the possibility of a node or process terminating while the code is executing.

---

## Initiating New Application Processes

There are five JADE RootSchema methods which allow the developer to start new application processes from an existing running JADE process. These are:

- Application::startApplication,
- Application::startApplicationWithParameter,
- Application::startApplicationWithString,
- Application::startAppMethod,
- Node::createExternalProcess.

The following diagram and sections describe the use of each of these.

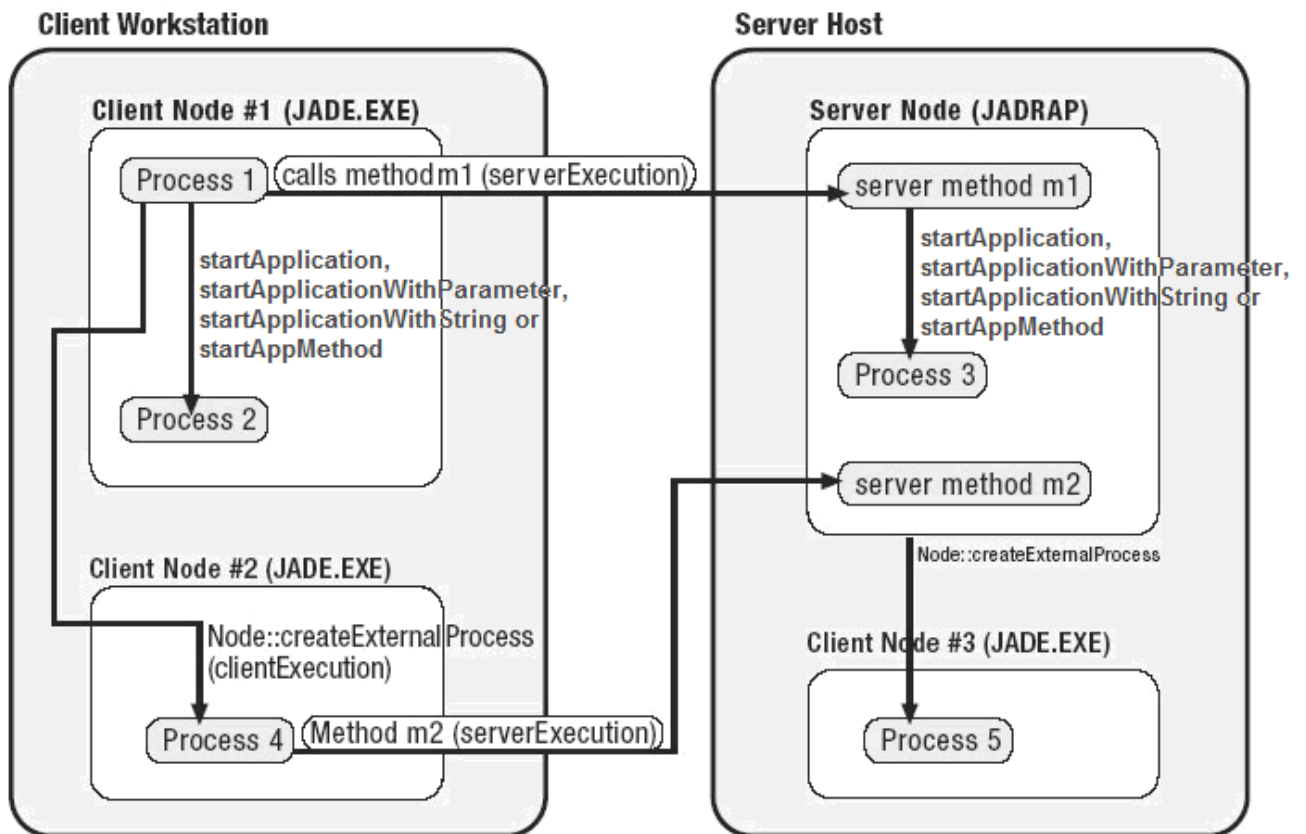


Figure 1: Nodes and Processes

## Application::startApplication

This method is the easiest way to start a separate application process. The method only requires that the schema name and application name be passed as parameters. For example:

```
app.startApplication("MySchema", "MyApp");
```

If the **startApplication** method is executed on the client node then the new process is initiated in the same node as the initiator process. In the diagram above this is depicted as **process1** starting **process2**.

If the **startApplication** method is executed on the server node i.e. **serverExecution** is specified for the method that calls **startApplication**, then the new process is initiated on the server node, effectively as a server application. In the diagram, this is depicted as **process1** via server method **m1** starting **process3**.

Processes that are started using the **startApplication** method behave exactly like a JADE process that was started by any other method e.g. started via an operating system short-cut. This means that the application will execute the application's initialize method and form load event (if specified) at process startup, and the application's finalize method at application shutdown. Of course, don't forget that applications that are executed on the server node cannot display forms and must be of application type **Non-GUI**.

## Application::startApplicationWithParameter

**startApplicationWithParameter** is similar to *startApplication*, but allows you to pass an object reference to the *initialize* method of the application being started. This can be useful if you want to pass parameters to the initiated process telling it what to do.

Shared transient objects are ideal for this, provided that the application to be started is to run on the same node. The example below illustrates this.

```
startReport (reportName);  
  
vars  
    param : Param;  
begin  
    beginTransientTransaction;  
    create param sharedTransient;  
    param.reportName := reportName;  
    commitTransientTransaction;  
  
    app.startApplicationWithParameter (currentSchema.name,  
                                       "ReportApp",  
                                       param);  
  
end;
```

In the diagram, this is the code that **process1** would execute.

In your application subclass, the initialize method executed by **process2** of the diagram, could be coded like this:

```
reportAppInit(param : Param io);  
  
vars  
    repManager : ReportManager;  
begin  
    create repManager transient;  
    repManager.runTheReport (param.reportName);  
epilog  
    delete repManager;  
    beginTransientTransaction;  
    delete param;  
    commitTransientTransaction;  
end;
```

If the application was to be started on a different node, i.e. the server node, the object being passed will need to be persistent. Shared transient objects can only be accessed by processes that are running in the node where the objects were created.

Of course, the above example is simplistic, and we will expand on this in **Controlling Multi-Threaded Applications**, later in this paper.

---

**Note** Do not attempt to pass a non-shared transient object to the application being started. Non-shared transient objects only exist in the context of the process that creates them, so cannot be accessed by the other application.

---

## Application::startApplicationWithString

**startApplicationWithString** allows you to start an application, passing a single String parameter to the initialize method of the application being started. If this is suitable, it avoids having to create and delete an object to contain particular specifications for the application.

A simple example of this is:

```
startReport (reportName) ;  
  
begin  
    app.startApplicationWithString (currentSchema.name,  
                                    "ReportApp",  
                                    reportName) ;  
end;
```

The initialize method for the ReportApp application could be like this:

```
reportAppInit(strParam : String);  
  
vars  
    repManager : ReportManager;  
begin  
    create repManager transient;  
    repManager.runTheReport(strParam);  
epilog  
    delete repManager;  
end;
```

## Application::startAppMethod

**startAppMethod** allows you to initiate a separate application process, starting with a specified method of your Application subclass. When the execution of the method is complete then the process terminates, unless the application calls **app.allowZeroForms** while in the initial method. If you call **allowZeroForms** then you need to code a **terminate** instruction in order to end execution of the process.

Similarly to **startApplicationWithParameter**, an object reference is passed to the method that is executed.

Note that application processes that are started using **startAppMethod** do not automatically execute the **finalize** method for the application when they terminate, even if the method is defined for the application.

## Node::createExternalProcess

This RootSchema method was designed to allow you to initiate any operating system task (e.g. you can start a Word or NotePad session using this method) but you can also use it to start a JADE process in a new node. In the diagram this is shown by **process1** starting **process4**, and by **process4** via serverExecution method **m2** starting **process5**. Note that in this last case, the new node runs on a different host machine from the initiating client (i.e. the server). **createExternalProcess** provides considerable control over the relationship between the initiator and the initiated process, because you can optionally specify that the new process is to be run modally i.e. the initiator process will not continue until the new process terminates, and for modally initiated processes a return code can be captured when the initiated process terminates.

The method signature for **Node::createExternalProcess** is:

```
createExternalProcess ( directory: String;
                       command: String;
                       args: StringArray;
                       alias: String;
                       thinClient: Boolean;
                       modal: Boolean;
                       result: Integer output): Integer;
```

Here is a code fragment where a client process (e.g. **process4** in the diagram) which starts a non-modal background client process on the server host machine in a new JADE.EXE:

```
startBackgroundClient() serverExecution;

vars
  binPath : String;
  argumentsArray : StringArray;
  iResult1 : Integer;
  iResult2 : Integer;
```

```

begin
  create argumentsArray transient;
  argumentsArray.add(" ");

  iResult1 := node.createExternalProcess
    (app.jadeInstallDir & "\jade.exe " &
     "path=" & app.dbPath,
     argumentsArray,
     null, //
     false, // no thin client
     false, // not modal
     iResult2);

epilog
  delete argumentsArray;
end;

```

## Controlling Multi-Threaded Applications

Once you have started a separate JADE process, you may want to be able to terminate it programmatically at will, or you may want it to submit progress reports to the initiating process. This can be done easily through the use of notifications. Shared transient objects or JADE's **Process** objects are extremely useful for this sort of inter-process communication.

Obviously there are many ways that you can use notifications to achieve the specific inter-process communication and control that you need. The following material should provide basic ways to do this, upon which you can build.

To show how termination of a process, or progress reports from another process can be achieved, let's expand the code that was shown earlier in the discussion of **startApplicationWithParameter**.

First the initiator process sets up additional information in the Param object that is passed to the process being initiated, and subscribes to events that may be caused on its own process object:

```

startReport(reportName : String); // e.g. as a method of a form

vars
  param : Param;
begin
  beginTransientTransaction;
  create param sharedTransient;
  param.reportName := reportName;
  param.initiatorProcess:= process; // new info
  commitTransientTransaction;

  beginNotification(param, Progress_Report, 0, 0);
  beginNotification(param, Process_Registration, Response_Cancel, 0);

```

```

beginClassNotification(Process, false, Object_Delete_Event, 0, 0);
app.startApplicationWithParameter(currentSchema.name,
                                "ReportApp",
                                param);
end;

```

The class that receives the notifications could be coded thus:

```

userNotification(eventType : Integer;
                 theObject : Object;
                 eventTag : Integer;
                 userInfo : Any) updating;
begin
  if eventTag = Progress_Report then
    // issue progress report to GUI
  elseif eventTag = Process_Registration then
    // save the reference provided
    runningReportProcess := userInfo.Process;
  endif;
end;

```

Note that the above code fragment uses a shared transient to pass information to the new process.

This assumes that the new process being started will run in the same node and will therefore have visibility to that shared transient. If you are using *startApplicationWithParameter* in a server method, then you need to use a persistent object to pass information to the new process. In the example above, you would also need to code the **sysNotify** method to handle deletion of process objects by JADE, so that when your initiated process ends, your initiator process handles the **Object\_Delete\_Event** notification.

Next, we look at what the initiated process needs to do.

In your application subclass, the **reportAppInit** method (to be executed by **process2** of the diagram) could be coded like this:

```

reportAppInit(param : Param io); // method of Application subclass

vars
  repManager : ReportManager;

begin
  beginNotification(param, Terminate_Request, 0, 0);
  // save param.initiatorProcess somewhere if required
  param.causeEvent(Process_Registration, true, process);
end;

```

```
param.causeEvent(Progress_Report, true, "Starting Report " &
    param.reportName);
create repManager transient;
repManager.runTheReport(param.reportName);

epilog
delete repManager;
beginTransaction;
delete param;
commitTransientTransaction;
end;
```

```
userNotification(eventType : Integer;
    theObject : Object;
    eventTag : Integer;
    userInfo : Any) updating;

begin
    if eventTag = Terminate_Request then
        terminate;
    endif;
end;
```

## Debugging Multi-Threaded Applications

There are some special considerations and techniques to be used when debugging multi-threaded applications using the JADE development environment. This section offers some suggestions and techniques that will help you with this.

Referring to the diagram again, if **process1** is running under the control of the JADE debugger and executes a **startApplication**, **startApplicationWithParameter**, **startApplicationWithString** or **startAppMethod** method call, then note that the spawned process (**process2** or **process3**) will **not** run under the control of the debugger. This is just the way that JADE works, but you can use one of the following techniques to debug a spawned process.

### Debugging Processes Started via startApplication or createExternalProcess

This is easily done from the development environment. You simply comment out the **startApplication** statement and substitute a **read** statement or message box. Run your main application from the development environment. When the message box appears, start the required application using the debugger in the normal way, then allow the initiator application to continue.

If your application is a server application, and you are using **startApplication** to initiate the new process, then this doesn't exactly replicate the true run-time environment, since the application you're running in debug mode will be running in a JADE.EXE on your workstation, rather than in the server node.

If you need to debug applications while they are running as server applications then the best that can be done is to sprinkle **write** statements through your code (remembering that the interpreter output window will appear on the server, not necessarily on your workstation). The **executeWhen** statement can be used around the **write** statements so that they do not have to be removed, but only get executed when testing.

---

**Note** For more information about **executeWhen**, refer to the JADE 6.3 Developer's Reference Manual.

---

## Debugging Processes Started via **startApplicationWithParameter**, **startApplicationWithString** or **startAppMethod**

This involves a little more effort, since you generally want to have your initiated debugger process pick up the parameter that you're passing in the call that starts the application. One way to do this is as follows:

Temporarily modify the signature of the application method that will be executed to have no parameters, and ensure that method is the initialize method for the application.

For *startApplicationWithParameter* and *startAppMethod* at the start of the initial method add code to fetch the object that would have been passed. For debugging, this can be done using the **firstInstance** or **firstSharedTransientInstance** method.

For *startApplicationWithString*, modify the method to use an explicit string value, or insert a **read** statement or message box to specify the string value.

In the primary application, substitute a **read** statement or message box for the call.

Run the primary application from the development environment without the debugger.

When the message box appears, run the secondary application with the debugger from the development environment in the normal way.

Note that the above technique isn't applicable to server applications. As before, the best that can be done there, without temporarily changing the applications to run outside the server, is to use **write** statements in the code, possibly in conjunction with the **executeWhen** statement.