Multithreading JADE Applications - A Primer White Paper

VERSION 2020

jade

Contents

Multithreading JADE Applications

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 white paper describes the basics of multithreading using JADE, and provides some examples of the use and control of multithreaded applications. For more details, see the following subsections.

The Basics

This section contains the following topics.

Nodes and Processes

In JADE, each application *process* (*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 for a fat client (that is, a standard client; not a thin one).
- A **jadapp.exe** program, which is an application server for JADE thin clients.
- A **jadrap.exe** program, which is the JADE database server (the JADE Remote Access Program).
- A user program that uses the C-API level of the JADE Object Manager, including .NET class libraries.

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

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 (discussed later in this white paper) 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 specific 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.

Server Node

The two main types of process that execute within the server node (that is, in a **jadrap.exe**) are:

- Server applications
- 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.exe** starts, by including a **ServerApplication<*application-number*>** parameter in the [JadeServer] section of the JADE initialization file.

However, server applications can also be initiated programmatically at any time, by any other JADE process. For details, see "Initiating New Application Processes", later in this document.

Server Methods

Server methods are methods that specify **serverExecution** in the method signature, or methods that are called by another method that has **serverExecution** in its signature. These methods execute on a thread in the JADE database server; that is, the JADE Remote Access Program (**jadrap.exe**).

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. You can reference the **System** object, by 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 **System**::**nodes** property returns a reference to a collection of all nodes in a JADE system. The **Node**::**processes** property returns a reference to a collection of all processes running on the node.

If you want to know whether any node is executing a specific application, you could write a method like that shown in the following example.

```
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 use **system.nodes** and **node.processes** directly 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.

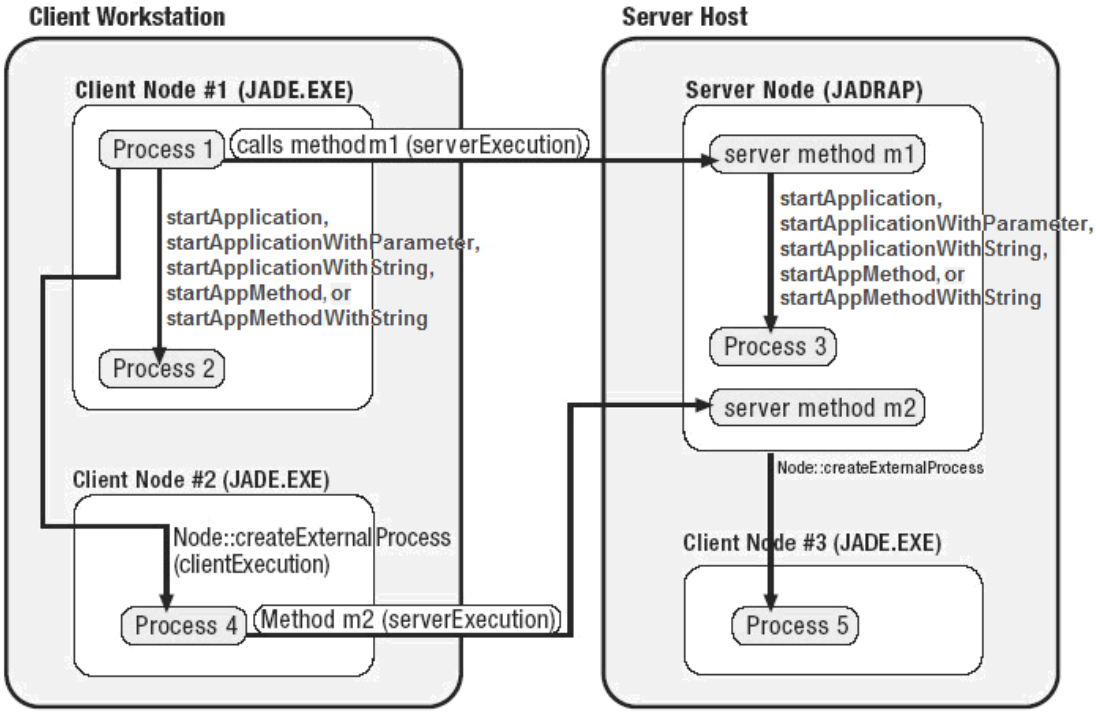
Note In actual use, the method in the previous example should arm an exception handler, to protect against the possibility of a node or process terminating while the code is executing.

Initiating New Application Processes

The five **RootSchema** JADE methods that allow you to start new application processes from an existing running JADE process are:

- Application::startApplication
- Application::startApplicationWithParameter
- Application::startApplicationWithString
- Application::startAppMethod
- Application::startAppMethodWithString
- Node::createExternalProcess

The following diagram and subsections describe the use of each of these methods.



Application::startApplication

The **Application** class **startApplication** method is the easiest way to start a separate application process. The method requires only 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, the new process is initiated in the same node as the initiator process. In the image in "Initiating New Application Processes", this is depicted as **process1** starting **process2**.

If the **startApplication** method is executed on the server node (that is, **serverExecution** is specified for the method that calls **startApplication**), the new process is initiated on the server node, effectively as a server application. In the image in "Initiating New Application Processes", 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; for example, started via an operating system shortcut. 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

The **Application** class **startApplicationWithParameter** method is similar to the **startApplication** method, but it enables you to pass an object reference to the **initialize** method of the application that is 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, as shown in the following example.

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

In the image in "Initiating New Application Processes", this is the code that **process1** would execute. In your **Application** subclass, the **initialize** method executed by **process2** of the image could be similar to the following example.

```
reportAppInit(param: Param);
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 (that is, the server node), the object being passed must be persistent. Shared transient objects can be accessed only by processes that are running in the node in which the objects were created.

Of course, the method in the previous example is simplistic, as is explained under "Controlling Multithreaded Applications", later in this white paper.

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

Application::startApplicationWithString

The **Application** class **startApplicationWithString** method enables you to start an application, passing a single **String** parameter to the **initialize** method of the application that is being started. If this is suitable, it avoids having to create and delete an object to contain particular specifications for the application, as shown in the following simple example.

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

The **initialize** method for the **ReportApp** application could look similar to the following example.

```
reportAppInit(strParam: String);
vars
```

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

Application::startAppMethod

The **Application** class **startAppMethod** method enables you to initiate a separate application process, starting with a specified method of your **Application** subclass. By default, the application's process terminates when the execution of the method is completed, unless a form is created.

If the **allowZeroForms** method was called before the **startAppMethod** method completes, the process continues afterwards, even if a form was not created, so you will need to code a **terminate** instruction.

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

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

Application::startAppMethodWithString

The **Application** class **startAppMethodWithString** method enables you to initiate a separate application process, starting with a specified method of your **Application** subclass. By default, the application's process terminates when the execution of the method is completed, unless a form is created.

If the **allowZeroForms** method was called before the **startAppMethodWithString** method completes, the process continues afterwards, even if a form was not created, so you will need to code a **terminate** instruction.

Like the **startApplicationWithString** method, a string is passed to the method if that is executed.

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

Node::createExternalProcess

The **Node** class **createExternalProcess** method enables you to initiate any operating system task (for example, you can use this method to start a Word or NotePad session). You can also use it to start a JADE process in a new node. In the image in "Initiating New Application Processes", this is shown by **process1** starting **process4**, and by **process4** via a **serverExecution** method **m2** starting **process5**.

Note In this last case, the new node runs on a different host machine from the initiating client (that is, the server).

The **createExternalProcess** method 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 (that is, 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;
```

In the following code fragment, a client process (for example, **process4** in the image in "Initiating New Application Processes") 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.getJadeInstallDir(),
                                           "\jade.exe " & "path=" & app.dbPath(),
                                           argumentsArray,
                                           null,
                                           false, // not thin client
                                           false, // not modal
                                           iResult2);
epilog
    delete argumentsArray;
end;
```

Controlling Multithreaded 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. You can build on the basic ways to achieve this, documented in this section.

To show how termination of a process or progress reports from another process can be achieved, let's expand the code examples shown in "Application::startApplicationWithParameter", earlier in this white paper.

The initiator process first 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; for example:

```
startReport(reportName: String); // for example, 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 like the method shown in the following example.

```
userNotification(eventType: Integer;
                 theObject: Object;
                 eventTag: Integer;
                 userInf : 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 method 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** method in a server method, you need to use a persistent object to pass information to the new process. In the above code example, 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 image in "Initiating New Application Processes") could be as follows.

```
reportAppInit(param: Param); // 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);
    param.causeEvent(Progress_Report, true, "Starting Report " & param.reportName);
    create repManager transient;
    repManager.runTheReport(param.reportName);
epilog
    delete repManager;
    beginTransientTransaction;
    delete param;
    commitTransientTransaction;
end;

userNotification(eventType: Integer;
                 theObject: Object;
                 eventTag: Integer;
                 userInfo: Any) updating;
begin
    if eventTag = Terminate_Request then
        terminate;
    endif;
end;
```

Debugging Multithreaded Applications

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

In the image in "Initiating New Application Processes", if **process1** is running under the control of the JADE debugger and executes a **startApplication**, **startApplicationWithParameter**, **startApplicationWithString**, **startAppMethodWithString**, or **startAppMethod** method call, 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 techniques in the following subsections to debug a spawned process.

For more details, see the following subsections.

Debugging Processes Started via startApplication or createExternalProcess

To debug an application initiated by calling the **startApplication** method, replace the **startApplication** call with a **debugApplication** call, which requires that the client is also running the JADE development environment. Executing this command within a server method will attempt to start and debug the application on the client.

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

If your application is a server application and you are using **startApplication** to initiate the new process, this doesn't exactly replicate the true runtime 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, the best that can be done is to sprinkle **write** instructions through your code (remembering that the interpreter output window will appear on the server, not necessarily on your workstation). You can use the **executeWhen** instruction around the **write** instructions so that they do not have to be removed but are executed only when testing.

Note For more information about the **executeWhen** instruction, see Chapter 1 of the *JADE Developer's Reference*.

Debugging Processes Started with a startApplication Method Call

Debugging processes started with a **startApplicationWithParameter**, **startApplicationWithString**, **startAppMethodWithString**, or **startAppMethod** call 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.

1. Temporarily modify the signature of the application method that will be executed so that it has no parameters, and ensure that the method is the **initialize** method for the application.
2. For **startApplicationWithParameter** and **startAppMethod** calls at the start of the initialize method, add code to fetch the object that would have been passed.

   For debugging, this can be done using the **firstInstance** or **firstSharedTransientInstance** method.
3. For **startApplicationWithString** and **startAppMethodWithString** calls, modify the method to use an explicit string value, or insert a **read** instruction or message box to specify the string value.
4. In the primary application, substitute a **read** instruction or message box for the call.

5. Run the primary application from the development environment without the debugger.
6. When the message box is displayed, run the secondary application with the debugger from the development environment in the normal way.

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