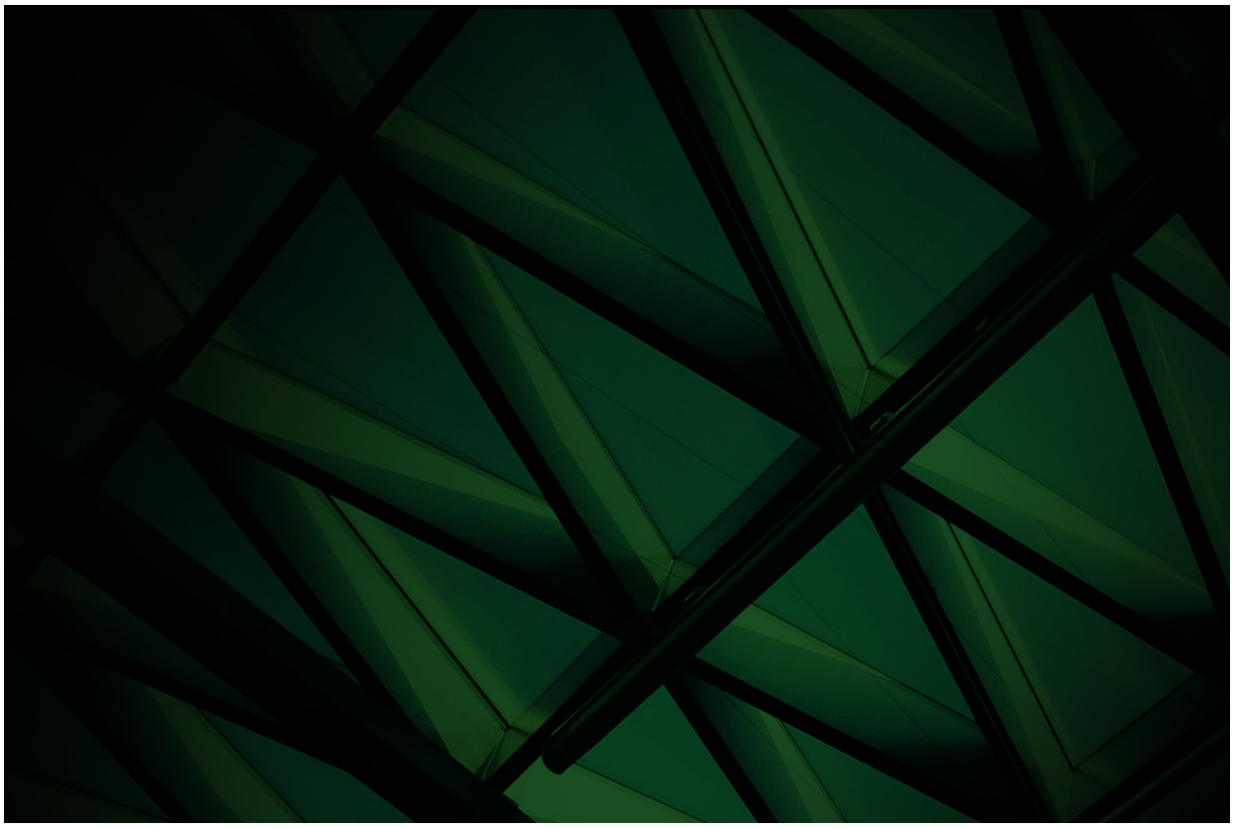


Asynchronous Method Calls White Paper

VERSION 2022

jade platform

Contents

Asynchronous Method Calls

The asynchronous method calls feature provides the ability to execute in parallel different tasks of an application. The parallel execution of tasks provides more potential to achieve improved performance than that achieved by handling tasks sequentially.

In essence, the feature is a framework around Generic Messaging, which enables you to perform parallel tasks in your applications with a few simple calls.

For more details about asynchronous method calls, see the following subsections.

Purpose of this Document

This document outlines:

- How the feature works
- How to make best use of the feature
- When it is or it is not appropriate to use the feature

An example application was written to simulate different execution scenarios and to measure the tasks that have to be performed. The example **AsynchMethodExample** application, which can be downloaded from the **JADE-WP-Asynchronous-Method-Calls** link at https://github.com/jadesoftwarenz, is described in the appendix of this document and is available for you to use and modify, running it against your own specific hardware.

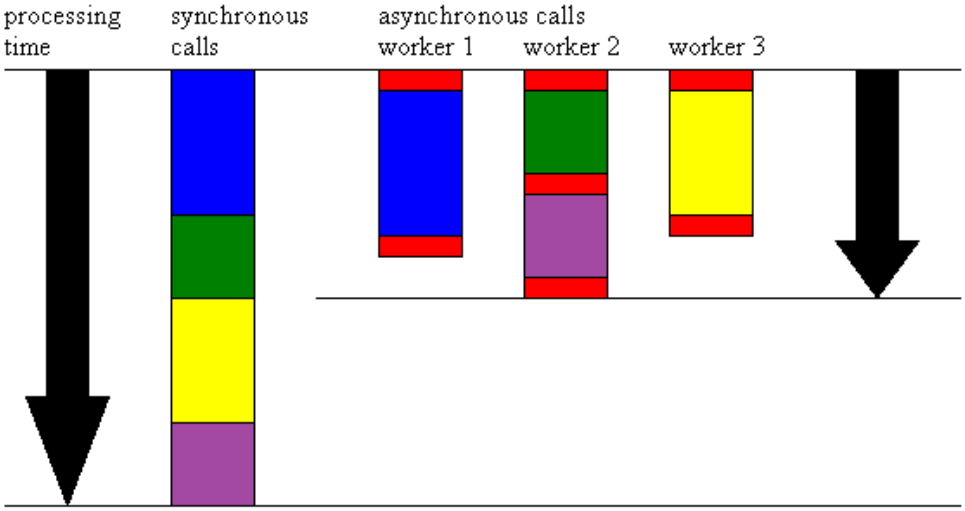
Example performance tests utilizing the described application are also contained in the appendix. The results of these tests give an indication of possible performance gains that are achievable using asynchronous method calls. However, the actual gains that can be obtained in practice are highly dependent on application characteristics and hardware configuration.

Feature Overview

To make the best use of asynchronous method calls, your application must have tasks that can be executed in parallel.

An asynchronous application generally sends multiple tasks to worker processes. Web service processes (for example, web consumers or providers) are a good example of this feature.

The following image shows how you can achieve performance gains with four parallel tasks (blue, green, yellow, and purple).



The red section of the bar represents the time utilized by the framework to manage the asynchronous method calls.

Using the Feature

The use of this feature requires the following elements.

- A caller process
- A receiver object
- An intermediate object (**JadeMethodContext**)
- One or more worker application processes

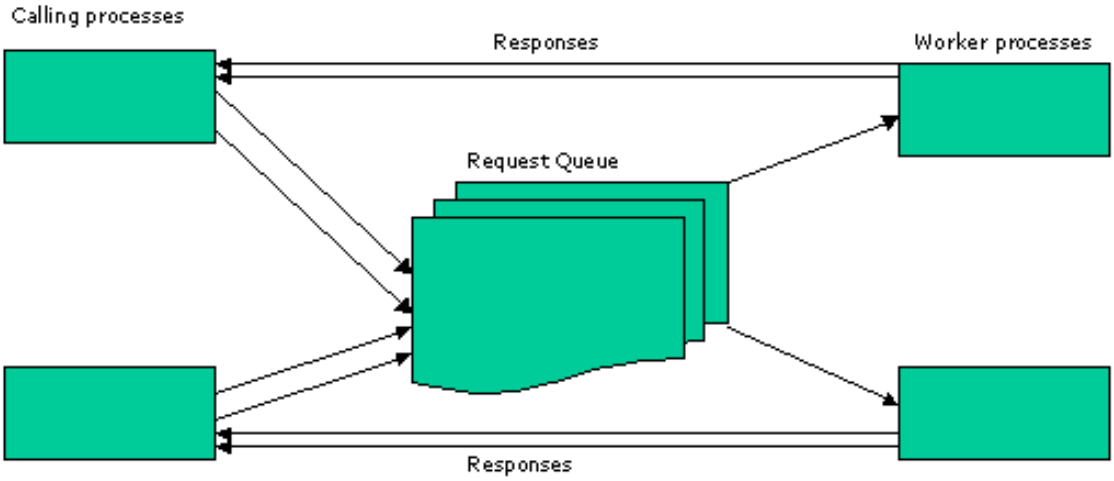
The calling process creates a **JadeMethodContext** object (as a non-shared transient instance) and uses the **JadeMethodContext**::**invoke** method to queue the request, passing the receiver object for the method execution to the worker process.

The object that is passed in must be a shared transient or a persistent object, or the **app** application environmental variable.

- The method called must be **public**.
- All parameters for the target method must be **constant** or **input**.
- The worker process must be launched in the same node as the caller process.

The application must pre-launch the worker processes that are intended to process the request queue. As a general guideline, the more tasks that are required to be processed simultaneously, the more worker processes need to be launched.

The caller process does not specify which of the worker processes handles a specific request. When all worker processes are busy and a new request is queued, the request waits in the queue until a worker process is available to service the request.



Code Example

The code example in this section is not the same as the application example in the schema that accompanies this white paper.

In the following code example, we have a class called **Person**. The **Person** class implements a method called **getAddress**, which returns an address string for the specified date. The signature of the method is:

```
getAddress(time: TimeStamp input): String;
```

If we were to call the method asynchronously, we could implement it as follows.

```
asynchronousCall(aPerson: Person; ts: TimeStamp input);
vars
    mtdCtx1, mtdCtxX : JadeMethodContext;
    result : String;
begin
    ...
    create mtdCtx1 transient;
    mtdCtx1.workerAppName := "WorkerApplication";
    mtdCtx1.invoke(aPerson, getAddress, ts);
    mtdCtxX := process.waitForMethods(mtdCtx1);
    if mtdCtxX.getErrorNumber() = 0 then
        result := mtdCtxX.getReturnValue().String;
    else
        // handle call error
    endif;
    ...
epilog
    delete mtdCtx1;
end;
```

The two conditions that must be true before this method will run to completion are as follows.

- The **aPerson** object of the class **Person** must be **persistent** or **sharedTransient**.
- There must be a minimum of one worker application process with the name **WorkerApplication** running on the current node.

To invoke an asynchronous method call, you must utilize the **JadeMethodContext** class. You will need to create an object of this class and define the name of the worker application, by assigning the property **workerAppName**. Initiation of the asynchronous call occurs when the **invoke** method is called, at which time the request is inserted in the queue. The first parameter for the **invoke** method is the object on which to make the call and the second parameter is the name of the method to be called. If the method to be called receives parameters (as is the case in the previous example), you must also supply these parameters.

To receive the response, you must wait for the methods to complete. You should always call the **waitForMethods** method of the current **Process** object before retrieving the result to ensure the method has finished processing. This method receives the **JadeMethodContext** object as a parameter. The **waitForMethods** method blocks the current sender process until a worker process processes the request and returns the specified object. In the earlier example, we receive back the same **JadeMethodContext** object as the one we passed to the **waitForMethods** method as a parameter. If the call was successful, we can query the **JadeMethodContext** object for the return value of our asynchronous method call.

In this example, there is no benefit from calling this one method asynchronously. Our current process has to wait until the method has executed and does nothing else during this period. However, if two or more method calls are performed asynchronously, as outlined under "Multiple Asynchronous Method Calls" later in this document, total elapsed times may be reduced.

Preparing a Worker Process

A standard application process is not created with the necessary internal structure required to handle the processing of asynchronous method calls. However, these internal structures can be constructed for any such process via a single call to the **Application** method **asyncInitialize**. To disable this ability and to ensure that the process is cleaned up appropriately when destroyed, the **Application** method **asyncFinalize** should be invoked.

Typically, these two method calls are made in the worker application's specific initialize and finalize methods, respectively. For example, a worker application could be defined as follows.

- Application name **WorkerApplication**
  - Initialize method **init**

```
init() updating;
begin
    app.asyncInitialize();
end;
```

  - Finalize method **finalize**

```
finalize() updating;
begin
    app.asyncFinalize();
end;
```

Multiple Asynchronous Method Calls

Continuing with the earlier example, we now show you how to call the same method as a parallel task and call the same method for another instance of the **Person** class.

Although you can call different methods for the same object, for this example we are simply outlining one scenario of how parallel method calls can be handled.

```
asynchronousCall(aPerson1, aPerson2: Person; ts: TimeStamp input);
vars
    mtdCtx1, mtdCtx2, mtdCtxX : JadeMethodContext;
    result1, result2 : String;
begin
    ...
    create mtdCtx1 transient;
    create mtdCtx2 transient;
    mtdCtx1.workerAppName := "WorkerApplication";
    mtdCtx2.workerAppName := "WorkerApplication";
    mtdCtx1.invoke(aPerson1, getAddress, ts);
    mtdCtx2.invoke(aPerson2, getAddress, ts);
    while true do
        mtdCtxX := process.waitForMethods(mtdCtx1, mtdCtx2);
        if mtdCtxX = null then
            break; // all requests complete
        endif;
        if mtdCtxX.getErrorNumber() = 0 then
            if mtdCtxX = mtdCtx1 then
                result1 := mtdCtxX.getReturnValue().String;
            endif;
            if mtdCtxX = mtdCtx2 then
                result2 := mtdCtxX.getReturnValue().String;
            endif;
        else
           write "Error: " & mtdCtxX.getErrorNumber().String;
        endif;
    endwhile;
    ...
epilog
    delete mtdCtx1;
    delete mtdCtx2;
end;
```

As in the first code example earlier in this document, a number of conditions must be met before the above example completes successfully. The **aPerson1** and **aPerson2** instances of the **Person** class must be **persistent** or **sharedTransient** and there must be a minimum of one application with the name **WorkerApplication** running on the current node.

If only one application with the name **WorkerApplication** is running, there will be no parallel processing of requests observed, because the single worker process would handle the first asynchronous method call and after finishing it, the process would then handle the second request. In other words, at least two worker processes must be run concurrently to achieve parallel execution of the intended asynchronous method calls.

The above example is similar to the first code example, as we have to prepare all of the objects in a similar manner to the first example. However, this example shows how results can be retrieved from each asynchronous method call. An important point to note is that the order in which the results are received may not be the same as the order in which the calls were invoked if you are running multiple server processes (that is, the result order is undefined). One consequence of this is that you have to check which **JadeMethodContext** object is received after calling the **invoke** method.

When you know which object has been received, you can assign the results for the different asynchronous method calls. If all requests are handled and the **invoke** method is invoked again, the current process no longer gets blocked and the method immediately returns **null**, which confirms that all calls have been processed.

Additional Information

The requests are sent to the queue for asynchronous method calls when the **invoke** method is invoked and not when the **waitForMethods** method of the current process is invoked. This is an important distinction, as it means that requests are processed even if the sender process never asks for the result.

The completed request stays in a process queue until the sender process gets terminated or if a timeout has been specified, the request can be deleted when the timeout expires.

If the **invoke** method is invoked while there is no worker application running with the specified name, the **invoke** call fails with an exception.

The **waitForMethods** method accepts a comma-separated list of **JadeMethodContext** objects, an array of such objects, or a mixture of array references and object references.

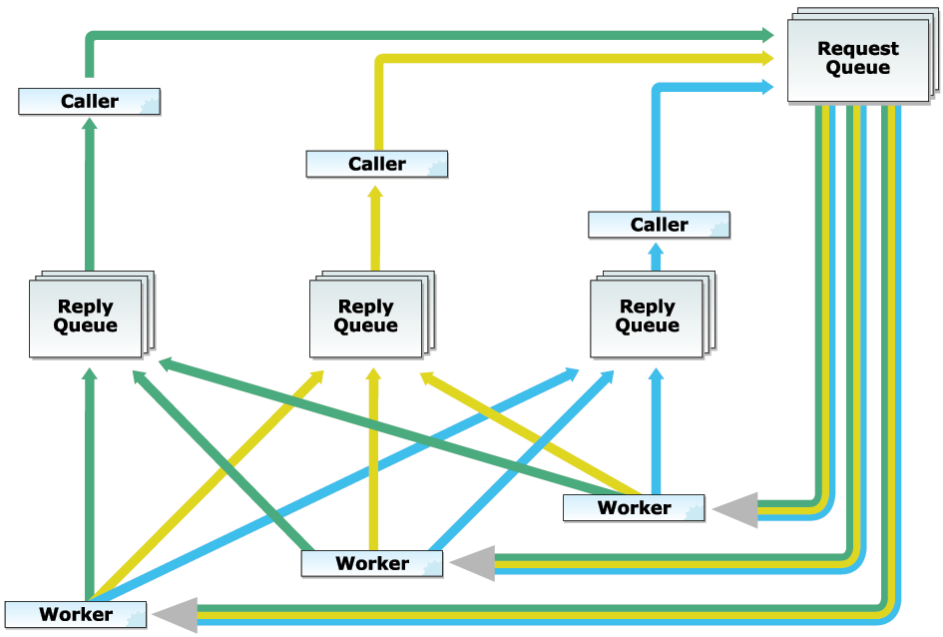
After invoking the asynchronous method call, you can use **JadeMethodContext** class methods to query the internal state of the framework; for example, you can ask for requests already waiting in the queue by using the **getTimeStamps** method.

Appendix

This appendix contains the following topics.

Example Application Overview

The example application demonstrates how to use the asynchronous method calls feature. The scenario the application simulates is based on the use of web services. In addition, the application allows you to view the effects of changing runtime parameters such as the number of workers and callers.



The above image depicts a number of worker and caller processes. The worker processes offer different services to work on for the caller processes. The caller processes are waiting for a number of external requests. Every caller process periodically receives an updated list of requests.

If a caller receives a request, the caller process sends this request to a queue. One of the worker processes takes the request from the queue and processes it.

The example application allows you to change the number of caller and worker processes, the number of different tasks per external request, the approximate time difference between every external request, the kind of tasks, and finally, an *operation* parameter, which enables the application to specify the type of work the worker processes perform.

The application offers four different kinds of task, as follows.

1. Use no resources

   This option enables you to enter the operation parameter in milliseconds as the duration for a process to sleep. None of the activity for the other task types occurs for this task.

2. Use CPU

   The worker processes utilize an inefficient algorithm to count prime numbers lower than a specified number. For this task, the operation parameter is the specified number. This is intended to be a CPU-intensive task.

3. Use I/O

   The workers create the specified number of objects in the database and then delete them all again. The operation parameter for this task is the number of objects. Each object create or delete is in its own transaction. This is intended to be an IO-intensive task.

4. Use CPU and I/O

   This combines the work done in task types 2 and 3, above. The operation parameter in this scenario represents the number of objects to create and delete, along with the start number to begin the descending prime number search.

The application writes a Comma-Separated Values (CSV) file on completion, with the run times for each worker process, caller process, and for each single request.

Using the Example Application

This section describes the use of the example application, as follows.

The Framework

The framework offers you two distinct options to make use of the application.

1. The first option is a basic interface with a few method calls used to control the whole application. In this scenario, you can view the application as a black box that is doing everything for you. You cannot start a new run of the application if the previous one is not finished.

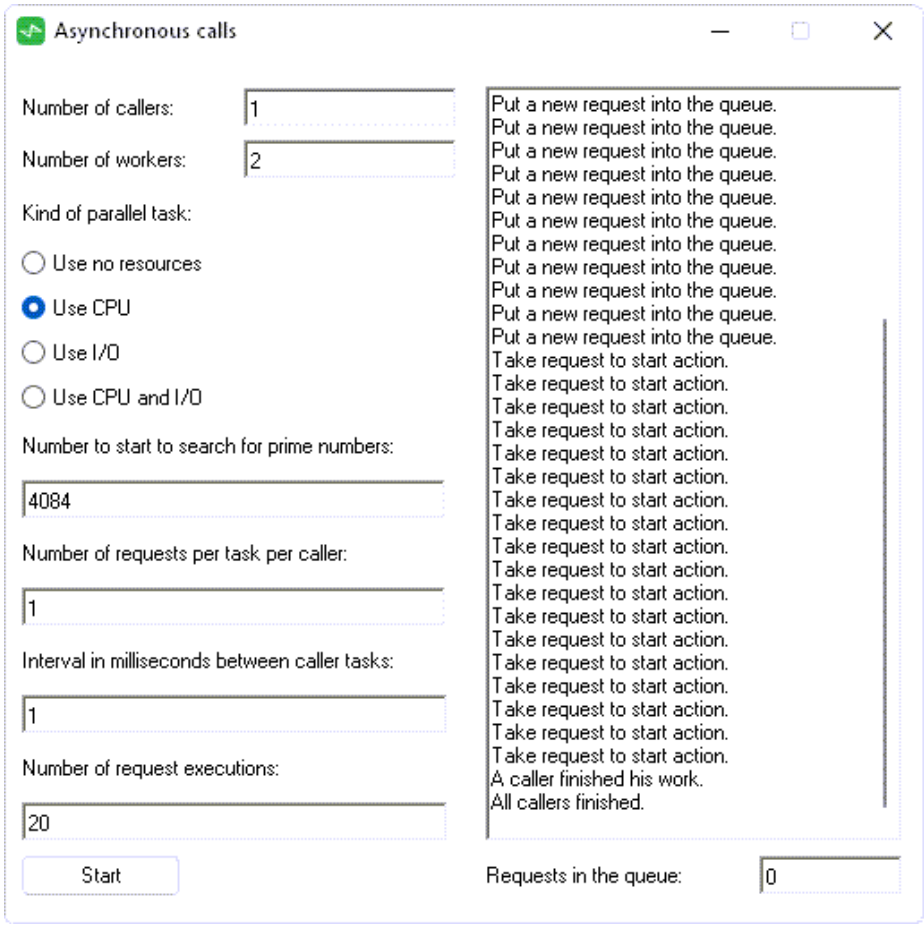
   Caution Take care if you want to run multiple scenarios, as initiation of concurrent runs results in errors.

2. The second option uses the example GUI, which also makes use of the first option as the interface. You can enter each of the described parameters and start the simulation.

   The form receives regular updates, informing you of what is happening inside the application. At the completion point, a Save File dialog prompts you to specify where the performance results should be saved.

Application GUI

The following form enables you to control the example application. The left side of the form represents the input area for each of the parameters you can change. The right side of the form displays the execution output.



The first two input fields enable you to enter the number of workers and caller processes.

The option buttons enable you to select the kind of tasks that are received by the callers and that the worker processes work on.

The first text box under the option buttons enables you to enter the operation parameter.

The second text box under the option buttons enables you to specify the number of requests for each external task. This number starts a separate request for the worker processes every time an external task arrives at a caller process.

The third text box enables you to specify in milliseconds a time interval between caller tasks. In the above image, the caller queues another request every millisecond.

The fourth text box enables you to specify the number of intervals before the application stops assigning new tasks to the caller processes.

Using the Application without the Predefined GUI

To run the application, it is not necessary to start the GUI. The GUI is merely an example that shows how the application framework can be used. If you want to develop your own user interface, consider the following.

A class that implements the **IUIProcessingUnit** interface needs to be implemented. The implementation of the interface receives all of the callbacks from the framework. In the GUI implementation, the **MainForm** class implements this interface.

An object of the **ProcessingUnit** class needs to be initialized by calling the **init** method, which needs the callback object as a parameter. After initializing, you can perform the call for the asynchronous method calls. The two options are as follows. To run your application:

- Without defining a result output file, call the **startAsynchronousCalls** method of your **ProcessingUnit** object. After the application finishes and all requests are completed, a Save File dialog prompts you for a folder and a file name in which to save the performance results. This file must be a CSV file.
- With an output CSV file defined at startup time, call the **startAsynchrnousCallsAndSave** method of your **ProcessingUnit** object. The results are automatically written to the specified file, and no Save File dialog is displayed after the application completes.

Take care not to delete the **ProcessingUnit** object (or your callback object that implements the **IUIProcessingUnit** interface) before the callback object receives the **finishedAction** callback. After receiving the callback, you can call the **startAsynchronousCalls** or **startAsynchronousCallsAndSave** method again, perhaps with different parameters, or terminate it. In the latter case, you must call the **finalize** method of your **ProcessingUnit** object.

Modifying the Application

This section describes modification of the example application, as follows.

- General Process Handling
- Report System
- Changing the Kind of Task Implementations
- Constants

General Process Handling

If you start the application using the predefined GUI, there are four different kinds of processes or Jade applications that can be launched.

The first is the process for the GUI. Each worker process and each caller process is also launched as a separate process. These three kinds of process normally run continuously. When a new request arrives for a caller process the application starts an additional process that tracks that request, puts the tasks from that request onto the queue, and calls back to the caller when all the tasks from that request are complete.

The simulation first starts the GUI process and after the application has initialized, the worker and the caller processes are launched. When the last of these processes is running, the application starts the first task for the callers.

When the application completes, the processes end in reverse order. A caller process terminates only when the last of its requests has been finished. All of the worker processes run until the last caller process terminates. This leaves only one process remaining after the simulation is finished: the GUI's process.

Information about the start and the end of each process is sent via notifications.

Instances of the **Worker** and **Caller** classes hold the information about the application's parameters.

Report System

A report class structure saves the individual times for the processes and the actions between them. The main class is **Report**, which holds collections of other report units.

Each type of process has its own report unit class, which is derived from the abstract base **ReportUnit** class. The associations are as follows.

- Request process **RequestReportUnit**
- Caller process **CallerReportUnit**
- Worker process **WorkerReportUnit**

The worker process holds another report unit, **ContinueableReportUnit**, which collects all of the active working time and the passive idling time.

The report system collects all of the units and after completion of an application run, you can print the report or write it to a file.

Changing the Kind of Task Implementations

To change the implementations of the tasks, modify the methods of the **Worker** class. The methods are **doNothing**, **useCPU**, **useIO**, and **useCPUandIO**. All of these methods receive an **Integer** parameter that represents the operation parameter.

Constants

Several constants are defined that are used all over the framework. Most method parameters are **Integer** values.

The most-important constants are those used to represent the four different kinds of task and the different user event numbers for the notifications.

The Performance Output File

At the end of each run of the application, a CSV file is written with the run times for the caller and worker processes and for each request. Use spreadsheet software to view and edit CSV files.

The CSV file contains the following sections.

- Worker Report
- Caller Report
- Request Report

Worker Report

The first section of the CSV file contains information about the worker processes. Each row of this section of the report represents a single worker process, identified by an ID in the first column. The second column contains the elapsed time of the process, which should be similar for each worker process.

Each worker process is started at the start of the application and terminates at the end, after all requests are processed. As such, you can use these times to determine how long it took to process all of the requests.

The third column contains the number of request tasks the worker process processed. Adding all of the numbers in the third column should give the total number of requests for your current application run.

The fourth and fifth columns contain the active work and the idle time in microseconds. The idle time is the time the worker spends waiting for a request. This can happen only if the asynchronous method call queue was empty. The active work time is the time the worker spends processing a request.

The last column mirrors the relationship between the two times and gives the percentage of busy time for the worker process for the duration of the application execution.

Caller Report

In terms of performance, a caller process plays a secondary role in this example application. The caller processes wait for incoming tasks (requests) and pass them to the request queue; they do not get blocked at all. Because of this, the caller report section of the CSV file contains only the identifier of the caller processes and the elapsed run time.

Each of the caller processes' run times are shorter than the run times of the worker processes in our scenario, because the worker processes can terminate only if the last caller process has already terminated.

If a caller process has no additional tasks to perform, it terminates even if other caller processes are still waiting for other tasks. Because of this, a caller process can have a significantly shorter run time than other caller processes.

Request Report

The requests report section of the CSV file includes the request information encountered for the selected application parameters during the application's execution. Elapsed time data is provided for each request.

Requests do not have an ID, as they are executed via a caller process and they are not aware of other caller processes and their requests.

The report output for the callers is ordered by the ID of the caller process that initiated the request. The ID of this caller process is displayed in the sixth column. The seventh column contains the ID of the worker process that processed the request.

The first column of the request report section contains the number of requests that were already in the queue waiting for a free worker process to get processed. The second column contains the number of microseconds that elapsed from the application start to the start of the current request.

The third column of the requests report contains the elapsed response time in microseconds for the requests. This is the time from when the request was queued until the response was returned. This time can be split into two parts. The first part is the time that the request was queued waiting for a free worker process, which is the information in the fourth column. The fifth column contains the second part: the active working time in milliseconds of a worker process on this request.

The three columns can be summarized by the following formula.

```
Response time = Queue time + Active work time
```

Example Test Results

The section describes example test results, as follows.

- Hardware Configuration
- Test Configuration

Hardware Configuration

The example application was run on the following configuration.

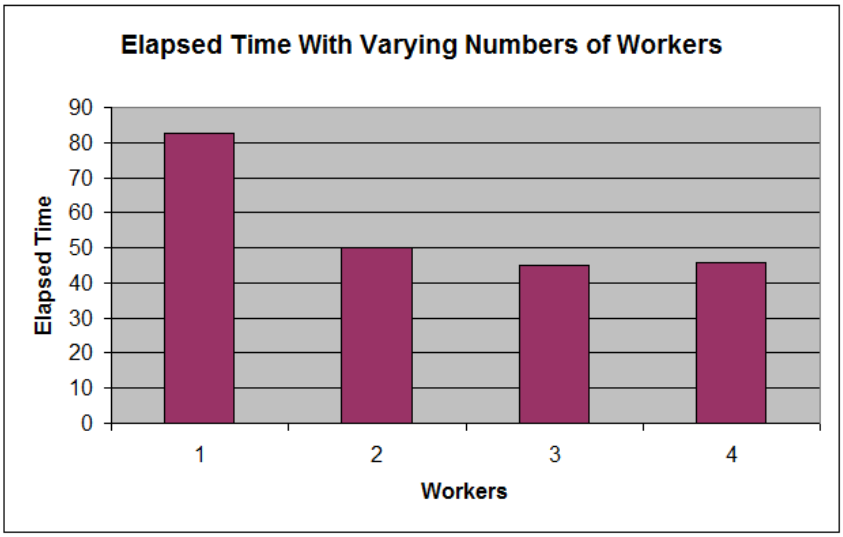
- 2 x Intel with hyperthreading, 2.4GHz, 4G bytes RAM, Windows 2000 Server

Test Configuration

A simple test was conducted to determine the potential savings in elapsed time that can be achieved by running multiple workers.

One caller was used. It quickly queued a number of requests for the calculate primes task. A varying number of workers processed the queue.

Results



There is a big saving in elapsed time going from one worker to two. Adding a third worker saved a little more, but adding a fourth slowed it down slightly.

Interpretation

The calculate primes task uses a lot of CPU but it does not do any IO, nor does it use any shared Jade node resources such as object caches. The overall performance is therefore highly dependent on the amount of CPU power available. The machine used has two real CPUs and two hyper-threaded CPUs. Using two workers instead of one nearly cut the elapsed time in half, as one might expect.

The hyper-threaded CPUs are not as quick as real CPUs for this workload, though, so four CPU-bound workers did not run much quicker than the two workers.