



open middleware  
infrastructure institute

# Developing with OMII

---

The Java Client Tutorial  
Stephen Crouch  
Steven Newhouse



# What we will cover...

- The client tutorial documentation
- A typical client process
- Setting up a development environment
- Example Java clients
  - Detailed example of one job
  - How to extend to 2 sequential jobs (overview)
  - How to extend to n parallel jobs (overview)

# The client tutorial documentation

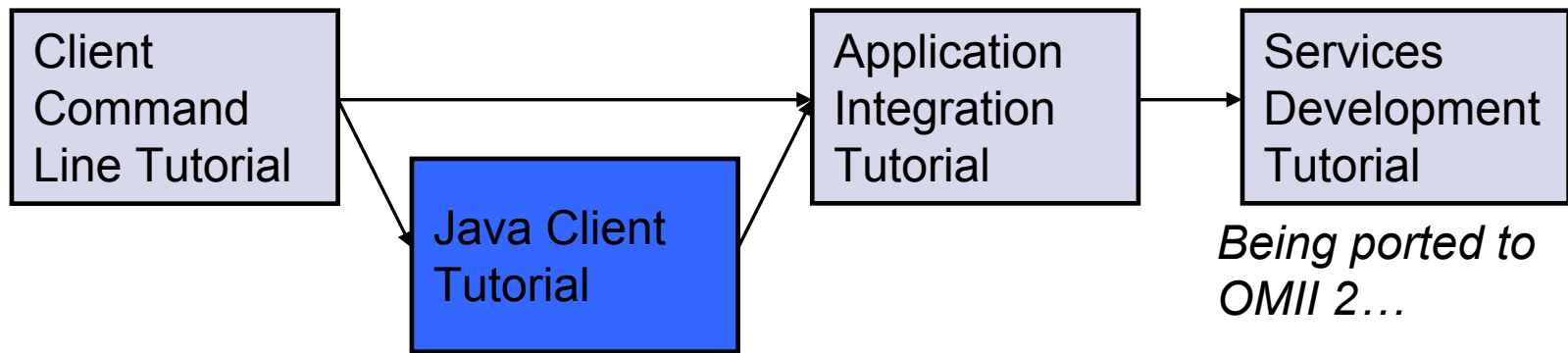


- Now available:
  - OMII command line client tutorial
  - OMII Java client tutorial
  - See OMII 2.0 documentation
- Java client tutorial teaches aspects of using the client Java API...
  - To implement simple, typical grid workflow models to drive execution of jobs on an OMII server
  - Discusses how to use the four OMII basic services

# The client tutorial documentation



- Developer pre-requisites...
  - Intermediate Java expertise
  - Knowledge of overall OMII architecture
    - In particular, the client and the four services
  - Familiarity with OMII Client command line
- In context...





# Tutorial pre-requisites

- Installation of an OMII Client (Windows/Linux)
  - Development platform
  - Used to submit jobs
- Access to an OMII Server
  - With the OMII Stack installed
  - An open, credit-checked account on the server
  - A working and tested OMII TestApp application
    - Installed by default as part of OMII Services installation



# Scenario Background

- Common class of problem...
  - User has data to be processed by an application
  - Wishes to use computational and data resources provided by another machine to expedite process
  - May involve running application many times over each element in data to produce output



# Scenario Background

- Using OMII, the process involves the following steps:
  1. Ensure application installed on service provider
  2. Obtain account on the service provider
  3. Obtain resource allocation
  4. Upload input data
  5. Execute application on input data
  6. Download output data produced by application
  7. Finish the allocation
- Assuming 1 & 2 achieved, can automate steps 3-7

# Jobs from a client perspective...



- Following this process, any submitted job may:
  - Run as a local process on the server, **or**
  - Be submitted to a resource manager (PBS, Condor)
- Depends on server configuration
- Cannot influence this configuration from client
- The client is not exposed to these differences
- Major advantage:
  - Java (and command line) clients completely independent of resource configuration - easily portable to other servers
  - Client development does not require knowledge of any underlying resource managers



# Our test application

- All examples use OMII TestApp on server
  - Written in Perl
  - Inputs 1 or more text files
  - Performs some string sorting on input files
  - Outputs same number of text files
  - Accepts input and produces output as zipped files



# Development environment

- For compilation and execution, essentially:
  - Need *classpath* to contain OMII API .jar files
  - Located in *OMIICLIENT/lib* subdirectory
  - Require *OMIICLIENT/conf* dir in classpath for execution
- Setting up:
  - Download tutorial.zip ('OMII Java Development...' from <http://www.ecs.soton.ac.uk/~stc/omii/training.html>)
  - Copy the [the tutorial material zip file] into the OMIICLIENT directory
  - Unzip it at that location
    - e.g. `cd ~/OMIICLIENT, unzip tutorial.zip`



# Development environment

- Contains a subdirectory, *tutorial*, which contains:
  - SmallApp.java – 1 job
  - SmallAppSeq2.java – 2 sequential jobs
  - SmallAppParN.java – n parallel jobs
  - OMIIUtilities.java – a few very small convenience functions
    - Logging, handling XML, obtaining an allocation
  - Work-SmallApp.xml – requirements in XML for one job
  - CompileEx.sh & RunEx.sh – for compiling and executing examples
  - Input?.zip – example zipped text files (for OMII TestApp)



# Compilation script

- Compilation script: CompileEx.sh

```
#!/bin/sh
export OMII_CLIENT_HOME=..
export MY_CLASSPATH=.../lib:

for j in `ls ../lib/*.jar`; do
    MY_CLASSPATH=$j:$MY_CLASSPATH
done

echo "Compiling the sample OMII TestApp client..."
javac -classpath $MY_CLASSPATH $1.java
```



# Execution script

- Execution script: RunEx.sh

```
#!/bin/sh
export OMII_CLIENT_HOME=..
export MY_PARAMS=-Dgria.config.dir=../conf
export MY_CLASSPATH=../lib:../conf:

for j in `ls ../lib/*.jar`; do
    MY_CLASSPATH=$j:$MY_CLASSPATH
done
```

Client support class –  
creates a default  
Requirements.xml file

```
echo Create Requirements.xml file...
rm Requirements-SmallApp.xml
java -cp .. CreateRequirementsFile
http://omii.ac.uk/OMIITestApp Requirements-SmallApp.xml
```

```
echo "Running the sample OMIITestApp client..."
java -cp $MY_CLASSPATH $MY_PARAMS $1
```



# The OMIIUtilities support class

- Used to enhance clarity of tutorial examples
- Some trivial convenience functions:
  - `setupLogging()` – configures LogManager, log4j
  - `xmlParse()` – Given XML file, returns Document object
  - `getRequirements()` – Given Requirements Document object, returns ResourceAllocationType of that object
  - `getJobSpec()` – Given Work Document object, returns JobSpec type
  - `getAllocation()` – given the client state, and a Requirements object, handles the process of obtaining an allocation
- Will be highlighted as they are used.



# Compilation

- Look at test-services/test/Account-test.xml
  - Make note of account URI for omiideo3
  - e.g. `http://omiideo3.omii.ac.uk:18080/axis/services/AccountService#xxx`
- Edit `SmallApp.java`
  - Change `serviceProviderAccount` to reflect account URI
- Type the following:

```
prompt> cd ~/OMICLIENT/tutorial
prompt> ./CompileEx.sh SmallApp
```



# Execution

In the tutorial  
subdirectory, type:  
`./RunEx.sh SmallApp`

```
Create Requirements.xml file...
```

```
Creating a requirement file Requirements-  
SmallApp.xml for job http://omii.ac.uk/OMIITestApp
```

```
parent directory /root/OMIICLIENT/tutorial  
Running the sample OMIITestApp client...  
Small App - OMIITestApp Java demonstration
```

```
-- Setting up logging --
```

```
Logger initialised
```

```
-- Beginning OMIITestApp client demo --
```

```
Creating memory (non persistent) repository  
Retrieving allocation requirements  
Tendering requirements to service providers
```

```
Creating input/output data staging areas  
Uploading input data (input.zip)  
Retrieving individual job requirements  
Initialising job  
Starting job  
Waiting for job to finish
```

```
.....  
Downloading results (output.zip)  
Cleaning up allocation  
Small App complete
```



# Process

- Preparation
- Initialise the repository and validate accounts
- Obtain an allocation
- Initialise data stage areas & upload input data
- Execute application on the input data
- Download output produced by application



# Code – preparation

- First, define overall structure of the class.

```
class SmallApp extends OMIIUtilities {  
    public static StateRepository repository = null;  
    public static final String serviceProviderAccount =  
"http://<host>:<port>/axis/services/AccountService#<account_conv_  
id>";  
  
    public static void main(String[] args) throws ... {  
        setupLogging();  
        ...  
    }  
}
```

The account to use

*repository* will hold client  
*conversation* state  
(account, allocation, data  
and job)

For clarity have  
omitted exception  
handling within the  
class – refer to API

# Code – initialise repository, validate accounts



- Create a repository to hold state
  - Using MemoryStateRepository – non-persistent
  - Could use FileStateRepository for persistence
- Check accounts are accessible

```
repository = new MemoryStateRepository();  
repository.importAccountConversation(new URL(serviceProviderAccount),  
    "Test");  
AccountConversation[] accounts =  
    repository.getAccountConversations();
```

```
for (int i = 0; i < accounts.length; i++)  
    accounts[i].getAccountStatus();
```

A local client reference (used for persistence)

Contacts service provider, checking each account (only 1 here)



# Code – obtain allocation

- We now have repository with valid account
  - Can tender requirements to service provider
  - Requirements held in OMII Client Requirements file, auto-generated by *CreateRequirementsFile* class in *RunEx.sh* script

```
ResourceAllocationType requirements =  
    getRequirements (xmlParse ("Requirements-SmallApp.xml")) ;
```

```
AllocationConversation allocation = getAllocation(repository,  
    requirements, "SmallAppTest") ;
```

A local client reference (used for persistence)



# Code – obtain allocation

- The function **getAllocation()** invokes an OMII Client helper that does the following:
  - Sends requirements to Resource Allocation Service of all service providers represented by accounts in repository.
  - Receives offers from service providers who make one.
  - Displays offers to user in GUI, allowing user to select offer that best meets their requirements.
  - Confirms selected offer with service provider and receives allocation in return.
  - Returns allocation to client as an **AllocationConversation**.
- Client can then use the **AllocationConversation** to create data stage areas and jobs.



# Requirements File

```
...
<ns1:max-work>
  <ns1:std-CPU-seconds>20000</ns1:std-CPU-seconds>
</ns1:max-work>
<ns1:max-upload-data-volume>
  <ns1:bytes>100000</ns1:bytes>
</ns1:max-upload-data-volume>
<ns1:max-download-data-volume>
  <ns1:bytes>100000</ns1:bytes>
</ns1:max-download-data-volume>
<resource-allocation-start>2005-02-01T17:17:35.451Z</resource-
  allocation-start>
<resource-allocation-end>2050-12-09T13:55:40.451Z</resource-
  allocation-end>
<service-name>http://omii.ac.uk/OMIITestApp</service-name>
<ns1:max-store-data-volume>
  <ns1:bytes>100000</ns1:bytes>
</ns1:max-store-data-volume>
...
```

# Code – initialise data stage areas, upload input data



- Define input/output stage areas
  - Locations of input and output required by job when it is started.
- Upload input data

Local client references (used for persistence)

```
DataConversation DSinput = allocation.newData("input.zip");  
DataConversation DSoutput = allocation.newData("output.zip");
```

```
DataHandler handler =  
    new DataHandler(new FileDataSource("input.zip"));  
DSinput.save(handler);
```

Upload the input data

# Code – execute the application on input data



- First, initialise job:
  - Pass URI of application to run
  - Local reference to job
- Then start the job, passing:
  - Requirements for individual job
    - When obtaining allocation, we sent requirements for entire allocation
    - `Number_of_jobs * Work_Requirements <= Obtained_Allocation_Limits`
  - Locations of input and output data stage areas
- Finally, wait for job to finish

# Code – execute the application on input data



```
JobSpecType work = getJobSpec(xmlParse("Work-SmallApp.xml"));
```

```
JobConversation JSjob =  
    allocation.newJob("http://omii.ac.uk/OMIITestApp/TestApp"  
        "ASmallAppTest");
```

```
JSjob.startJob(work, new DataConversation[] { DSinput },  
    new DataConversation[] { DSoutput });
```

```
while (JSjob.stillActive()) {  
    System.out.print(".");  
    Thread.sleep(2000);  
}
```

Individual job requirements

Check job status until it is finished

Input/output data stage areas created earlier

# Code – download output produced by application



- When execution is finished, the job's output is staged to output stage area defined earlier
  - When output staging is complete, job is complete
- Can then download output

```
DSoutput.read(new File("output.zip"));
```



# Code – finish allocation

- Assume we no longer need our allocation
- Can signal server we have finished
  - Release our resources back to provider

```
Conversation[] children = allocation.getChildConversations();  
for (int i = 0; i < children.length; i++)  
    children[i].finish();  
allocation.finishResourceAllocation();
```

Finish job and data  
conversations first

Finish allocation  
conversation last



# Extend to 2 sequential jobs?

- Output of first job is input to the second
- Very similar to how one job is done
- Changes required:
  - An additional, intermediate data staging area
  - Start first job, wait for completion
  - Then start a newly created second job, passing output of first job, wait for completion
  - (Need to ensure our allocation request reflects larger set of requirements)



# Compilation

- Edit test-services/test/Account-test.xml
  - Make note of account URI for omiideo3
  - e.g. `http://omiideo3.omii.ac.uk:18080/axis/services/AccountService#xxx`
- Edit SmallAppSeq2.java
  - Change serviceProviderAccount to reflect account URI
- Type the following:

```
prompt> cd ~/OMIIClient/tutorial
prompt> ./CompileEx.sh SmallAppSeq2
```



# Execution

In the tutorial subdirectory,  
type:

```
./RunEx.sh SmallAppSeq2
```

```
...
Creating input/output data staging areas
Uploading input data (inputn.zip)
Retrieving individual job requirements
Initialising job 1
Starting job 1
Waiting for job 1 to finish
.....
Initialising job 2
Starting job 2
Waiting for job 2 to finish
.....
Downloading results (output2.zip)
Cleaning up allocation
Small App complete
```

**This is the output you should see when you run your program....**

**Now you try it!**



# Code overview – 2 sequential jobs



```
DataConversation DSinput1 = allocation.newData("input1.zip");
DataConversation DSoutput1 = allocation.newData("output1.zip");
DataConversation DSoutput2 = allocation.newData("output2.zip");
...
JobConversation JSjob1 =
allocation.newJob("http://omii.ac.uk/OMIITestApp/TestApp",
"ASmallAppTest");

JSjob1.startJob(work, new DataConversation[] { DSinput1 },
new DataConversation[] { DSoutput1 });

while (JSjob1.stillActive())
System.out.print(".");
JobConversation JSjob2 =
allocation.newJob("http://omii.ac.uk/OMIITestApp/TestApp",
"ASmallAppTest2");

JSjob2.startJob(work, new DataConversation[] { DSoutput1 },
new DataConversation[] { DSoutput2 });

while (JSjob2.stillActive())
System.out.print(".");
```

Output from first job is  
input to second



# Extend to N parallel jobs?

- Jobs run completely independently
- Quite similar to how one job is done
- Changes required:
  - Create input/output stage areas (hold in arrays)
  - Upload multiple input files to input stage areas
  - Initialise all jobs (hold in array)
  - Start all jobs (hold in array)
  - Wait for all jobs to finish
  - Download all results



# Compilation

- Compilation:
  - Edit test-services/test/Account-test.xml
    - Make note of account URI for omiidemo3
    - e.g. `http://omiidemo3.omii.ac.uk:18080/axis/services/AccountService#xxx`
  - Edit SmallAppParN.java
    - Change serviceProviderAccount to reflect account URI
  - Type the following:

```
prompt> cd ~/OMIICLIENT/tutorial
prompt> ./CompileEx.sh SmallAppParN
```



# Execution

In the tutorial subdirectory,  
type:  
./RunEx.sh SmallAppParN

```
...
Creating input data staging area 1
Creating input data staging area 2
Creating input data staging area 3
Creating output data staging area 1
Creating output data staging area 2
Creating output data staging area 3
Uploading input data (input1.zip)
Uploading input data (input2.zip)
Uploading input data (input3.zip)
Retrieving individual job requirements
Initialising job 1
Initialising job 2
Initialising job 3
Starting job 1
Starting job 2
Starting job 3
Waiting for jobs to finish
.....

Downloading results (output1.zip)
Downloading results (output2.zip)
Downloading results (output3.zip)
Cleaning up allocation
Small App complete
```

# Code overview – N parallel jobs



- Create input/output data stage areas:

```
DataConversation[] DSinput = new DataConversation[NUM_JOBS];
for (int inCount = 0; inCount < NUM_JOBS; inCount++) {
    DSinput[inCount] = allocation.newData("input" +
        String.valueOf(inCount+1) + ".zip");
}
DataConversation[] DSoutput = new DataConversation[NUM_JOBS];
for (int outCount = 0; outCount < NUM_JOBS; outCount++) {
    DSoutput[outCount] = allocation.newData("output" +
        String.valueOf(outCount+1) + ".zip");
}
```

- Upload input data:

```
for (int inCount = 0; inCount < NUM_JOBS; inCount++) {
    DataHandler handler = new DataHandler(new FileDataSource("input" +
        String.valueOf(inCount+1) + ".zip"));
    DSinput[inCount].save(handler);
}
```

# Code overview – N parallel jobs



- Initialise and start jobs:

```
JobConversation[] JSjob = new JobConversation[NUM_JOBS];
for (int inCount = 0; inCount < NUM_JOBS; inCount++) {
    JSjob[inCount] =
allocation.newJob("http://omii.ac.uk/OMIITestApp/TestApp",
"SmallTestApp");
}
for (int inCount = 0; inCount < NUM_JOBS; inCount++) {
    JSjob[inCount].startJob(work,
        new DataConversation[] { DSinput[inCount] },
        new DataConversation[] { DSoutput[inCount] });
}
```

# Code overview – N parallel jobs



- Wait until all jobs are complete:

```
boolean stillRunningJob;  
do {  
    stillRunningJob = false;  
    for (int inCount = 0; inCount < NUM_JOBS; inCount++) {  
        if (JSjob[inCount].stillActive())  
            stillRunningJob = true;  
    }  
} while (stillRunningJob);
```

# Code overview – N parallel jobs



- Download all results:

```
for (int outCount = 0; outCount < NUM_JOBS; outCount++) {  
    System.out.println("Downloading results (output" +  
        String.valueOf(outCount+1) + ".zip)");  
    DSoutput[outCount].read(new File("output" +  
        String.valueOf(outCount+1) + ".zip"));  
}
```