Tupleware Manual (version 0.1)

Alistair Atkinson (aatkinson@eit.ac.nz)

December 20, 2008

1 Overview

This short document is intended to provide sufficient information to enable someone to get started with using Tupleware.

2 Software Requirements

You will need to following pieces of software installed on your system and their relevant executables available in your `$PATH`:

- Java SDK version 6 or later (JRE 6 or above is fine for worker nodes),
- Apache Ant for compilation, and
- GNU/Linux is the recommended operating system. No other OSs have been tested.
- ClusterSSH (`cssh`) makes controlling all of the computers in the system much easier, but isn't strictly required.

3 System Components

The Tupleware middleware is implemented in the `TupleSpaceRuntime` class, which controls the operation of lower-level classes `TupleSpaceService`, `TupleSpaceStub`, and `TupleSpaceImpl`. From an application development perspective, the only interaction is with `TupleSpaceRuntime`. For an example is it's usage, see Section 5.1.

4 Compilation

An Ant compile script is included in the compressed file, which contains fairly standard settings. To compile Tupleware, simply use:

```
ant compile
```

Or, if you aren't working in the Tupleware directory:

```
ant -f /path/to/build.xml compile
```

5 Execution

A typical Tupleware system involves one designated *master* process, and one or more *worker* processes. At this stage, the number of worker processes must be specified at compile-time, and remain constant during the course of an application's execution.

Also, each process in the system must be given a different port number. This is necessary due to Tupleware's slight clumsy process registration mechanism, which occurs when the system initialises. The master process, by default, runs on port 6001, and the workers begin on 6002 and increment by one.

5.1 Example

One of the applications contained in the Tupleware package is a parallel sorting algorithm (a modified quicksort). To get it up and running the following steps should be used.

Assuming the application has been compiled and the source code is located on each machine which is taking part, we firstly execute the master process (`QSortMaster`) like so:

```
java -Xms512M -Xmx2048M apps.sorting.QSortMaster 6001 <NO_ELEMENTS>
    <THRESHOLD>
```

The `-Xms` and `-Xmx` options aren't necessarily needed, but they don't hurt (they set the initial and maximum heaps sizes respectively). `NO_ELEMENTS` specifies the number of elements you wish to sort (5 million would be a good one to start with), and `THRESHOLD` specifies the point at which the array stops being partitioned by the system and the remaining unsorted segments sorted sequential using insertion sort.

Once the master process is up and running, we start however many workers we need, using the following command:

```
java -Xms512M -Xmx2048M apps.sorting.QSortWorker <PORT>
```

As it was mentioned before, start the first worker on port 6002, the second on 6003, and so on.

At this point the registration process will begin (node discovery) followed by processing. After processing, the master process will have a copy of the fully sorted array to do with as it likes.

6 Application Development

For the purposes of our example, we'll describe how to go about developing a simple mapping application, which will apply a function, f, to each element of an array, A. Implemented sequentially, a solution would look something like the following:

```
for(int i = 0; i < A.length; i++)
    A[i] = f(A[i]);
```

In our parallel implementation, the master process will simply add the array elements to tuple space as individual tuples, and the worker processes will

retrieve an element, compute its new value, and add the result back to tuple space. Once all elements have been computed, the application is complete.

Firstly, we begin with the definition of our master process. In order for it to execute on the Tupleware platform, it must contain an instance of a `TupleSpaceRuntime` object, and this object, upon instantiation, must be given an integer argument specifying on which TCP port the tuple space service will listen for incoming connections. Also, the runtime system object's `start()` method must also be invoked to initialise the underlying Tupleware system. Other than these requirements, the master process will contain only application specific code; in this case, creating array A and adding its values to tuple space, and waiting for computed values to become available. A partial code listing for the master process is listed below.

```
public class ArrayMaster {
    public final boolean IS_MASTER = true;
    public final int N = 100;
    public TupleSpaceRuntime space;

    public ArrayMaster(int port) {
        space = new TupleSpaceRuntime(port, IS_MASTER);
        space.start();
    }

    public void init() {
        int[] a = new int[N];
        for(int i = 0; i < N; i++)
            a[i] = (int) Math.round(Math.random()*Integer.MAX_VALUE
                );

        Vector<Tuple> tpls = new Vector<Tuple>();
        for(int i = 0; i < N; i++)
            tpls.add(new Tuple("A",
                               new Integer(i),
                               new Integer(a[i]),
                               new Boolean(false)));

        space.outAll(tpls);
    }

    public Vector<Tuple> getResults() {
        TupleTemplate template = new TupleTemplate("A", null, null,
             new Boolean(true));
        Vector<Tuple> results = new Vector<Tuple>();

        for(int i = 0; i < N; i++)
            results.add(space.in(template));

        return results;
    }
}
```

One important decision that does need to be made when developing an application for Tupleware relates to the format of the tuples. More specifically, what fields do the tuples need, and in what order should they be arranged? A consistent approach is needed in order to allow Tupleware processes to communicate successfully, and it should be based on the characteristics of the data that is being shared, and also the application logic relating to the purpose of the communication itself.

In the example application presented here, the data being shared are elements of an array. A typical approach to sharing array elements is to include a name for the array, its index (or indices for multidimensional arrays), and finally the value of the element. This approach is adopted here, however we use an additional field to denote whether or not the element has yet been processed. We use a boolean value for this, with unprocessed elements having a false value in its final field, and processed elements having a true value. Thus, the format of our tuples are as follows: $< String : name, Integer : index, Integer : value, Boolean : processed >$.

The role of the worker processes is simply to retrieve an unprocessed tuple element, apply a specified function to it, and return the result to tuple space. Unprocessed tuples are retrieved from tuple space using a tuple template based on the tuple format described above. The *index* and *value* fields are left as null, and the *processed* field is set to false so that only unprocessed array elements are retrieved. Once the element has been processing, the result is written back to tuple space as a tuple with the same index and with the *processed* field set to true.

Like the master process, before accessing the tuple space, the worker process must first instantiate and start a tuple space runtime system, as shown in the `ArrayWorker` constructor below. The worker process will carry out these steps until all unprocessed elements have been exhausted, at which point it will terminate. A partial code listing for the worker process is shown below, and an illustration of the complete system can be found in Figure ??.

```
public class ArrayWorker {
    public final boolean IS_MASTER = false;
    public TupleSpaceRuntime space;

    public ArrayWorker(int port) {
        space = new TupleSpaceRuntime(port, IS_MASTER);
        space.start();
    }

    public void doWork() {
        TupleTemplate template = new TupleTemplate("A",
                                                   null,
                                                   null,
                                                   new Boolean(
                                                       false));

        for(;;) {
            Tuple t = space.inp(template);

            if(t == null)
                return;

            int result = f(t.field(2));
            space.out(new Tuple("A", t.field(1), new Integer(result
                ), new Boolean(true)));
        }
    }

    public int f(int n) {
        // this is the function to be applied to array element
    }
}
```

Several aspects of the above application are worth discussing from a development perspective. Firstly, the requirements for a process to participate in a Tupleware system are simple; all it needs is to instantiate and start a `TupleSpaceRuntime` object. Once this has been done, the process will have its own local tuple space, and is able to interact with other processes participating in the system. Secondly, the usage of the tuple space operations is also simple and straightforward; the developer does not need to concerns themselves with the problem of where a tuple is physically stored. It may exist in the process' local tuple space, or it may be in any one of the other participating processes' tuple space. The practicalities of searching for an retrieving a tuple are handled by the runtime system, and so the application code remains as simple as possible. Also, to this end, the runtime system handles any network communication errors, meaning that the tuple space operations used in an application do not need to be enclosed in endless try/catch blocks, which further assists in keeping the application code as simple as possible.

7 Summary

This manual has hopefully provided enough information to get started using Tupleware. It is still an experimental system at this stage, so it is guaranteed that there are bugs an anomalies galore. For more information, email `aatkinson@eit.ac.nz`.