Dymola interface to Java - A Case Study: Distributed Simulations

José Díaz López Hans Olsson
Dynasim AB
Research Park Ideon, S-223 70 Lund, Sweden
{jose.diaz, hans.olsson}@dynasim.com

Abstract

Running multiple simulations for the same model is useful for studying the influence of parameters, calibrating parameters from measurement series, and optimizing designs to be robust with respect to operating conditions. Dymola's [1] Design-package allows a user to specify such experiments for experimenting, calibrating and optimizing simulation studies [2].

Many of the simulations are independent allowing them to be run in parallel, i.e. coarse-grained parallelism. This paper describes the extension that distributes the simulations to multiple CPUs, while keeping the original setup for the simulation study. This makes the distributed nature transparent for the user, and the only difference is that the studies are done faster.

The implementation is implemented in Java using a Modelica external interface, but in contrast to [3] with direct interface inside Dymola. Implementing the distributed simulations in Java makes it possible to leverage the multi-threading, graphical, and communication capabilities of Java; while using Modelica's strengths for modeling, setup of simulations, and numeric algorithms.

Keywords: dynamic simulation, distributed simulations, Java, Modelica

1 Introduction

Dymola broadens the external interface possibilities to Java.[1] A case study is distributed simulations for model experimentation.

[1] Java and any other Sun trademarks that are referred to or displayed in the document are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. or other countries.

Dymola is a tradermark of Dynasim AB, and a registered trademark of Dynasim AB in Sweden.

There are several needs for interfaces between Dymola/Modelica and other tools/languages that can be met in different ways. In this case we need to call external functions inside Dymola and in those functions have access to Dymola's API in a "safe" way.

Java with Java's Native Interface allows this in a natural fashion, and Dymola allows a user to directly call external functions written in Java from the scripting environment. In those functions written in Java it is also possible to access all of Dymola's API functions and normal Modelica functions.

This tight coupling requires that Dymola internally runs a Java Virtual Machine, where Dymola adds implementations of native functions to allow access to Dymola's API.

2 Dymola Interface to Java

2.1 Comparison with previous work

Previous implementation of interfaces to Java (based on top of the external C-interface) did not allow functions written in Java to call Modelica functions as seamlessly, but sufficed for calling functions written in Java inside models. The new implementation also allows this and ensures consistency since one external declaration in Modelica can be used for calling the function both in the scripting environment and inside models.

This implies that models using functions written in Java will include a special header and automatically link with the Java Virtual Machine.

Previous implementation of access to Dymola's API functions e.g. via DDE [3] only allows the external tool to control Dymola, but not Dymola to call the external tool.

Modelica is a registered trademark of the Modelica Association.

The new implementation allows calls from Modelica functions to Java, *and* allows this function in Java to call Dymola/Modelica.

2.2 Mapping of functions

The interface in Modelica to functions written in Java is limited to static member functions of classes. The declarations in Modelica are of the kind:

```
function foo
  input Real in1;
  input String in2;
  output Boolean out;
external "Java" out='P.C.S'(in1,in2);
end foo;
```

In this example P is a package, C a class and S a static function in this class.

This is according to the Modelica specification (including the use of quoted identifiers for the name of the function) except that Java is not one of allowed external language in the specification. The package-name can be a hierarchical name, with dot-notation.

For the future it might be possible to extend Modelica's external objects to also handle objects in Java.

Calls of Modelica functions (and Dymola's API-functions callable as Modelica functions) from Java go through one generic function accessible in Java as com.dynasim.dymola.interpretMainStatic. For other Modelica functions a wrapper in Java can be constructed in a mechanical way that maps arguments, calls this bridge function, and maps the result.

Having one entry point to Modelica from Java makes it straightforward to transparently redirect all calls to a remote instance of Dymola, i.e. remote method invocation.

This interface is only intended for accessing one instance of Dymola. Distributed simulations require access to multiple instances of Dymola, and thus require additional efforts as will be explained later.

Asserts and other errors in Modelica are mapped to exceptions to in Java and vice versa. Specific exceptions are thrown for errors specific to calling Modelica functions from Java (illegal types for arguments, unknown Modelica function, etc).

2.3 Mapping of data-structures

Simple types in Modelica (e.g. Real) are normally mapped to corresponding simple types in Java. Strings are non-simple in Java, the mapping is still direct, and is made simpler by the fact that JVM and Dymola internally use the same UCS-8 implementation of Unicode strings.

Note: The UCS-8 mapping is the result of applying the UTF-8 mapping to UTF-16 strings, and the recommendation is that even though it can be used internally in programs it should not be used for interfaces. In this case we make an exception in order to be compatible with the pre-existing C-interface of JVM.

Arrays in Modelica correspond to (possibly nested) arrays in Java. Special care is needed to detect heterogeneous arrays in Java, and convert zero-sized matrices from Java.

Records in Modelica are mapped to a class implementing a map interface in Java. This ensures that the semantics of Modelica records (named based type equivalence) is preserved.

To summarize we first present how arguments are mapped when calling a function written in Java from Modelica.

Modelica	**External Java**
Real	double
Integer	int
Boolean	boolean
String	java.lang.String
Record	com.dynasim.record
Real[]	double[]
Integer[]	int[]
Boolean	boolean[]
String[]	java.lang.String[]
Record[]	com.dynasim.record[]

The mapping when a function in Java calls interpretMainStatic is similar, but has special handling of simple types as presented below. This is necessary since the simple types such as double are not objects and thus cannot be part of the generic argument list of interpretMainStatic.

Modelica	**interpretMainStatic**
Real	java.lang.Double
Integer	java.lang.Integer
Boolean	java.lang.Boolean
String	java.lang.String
Record	com.dynasim.record
Real[]	double[]
Integer[]	int[]
Boolean	boolean[]
String[]	java.lang.String[]

Record[]	com.dynasim.record[]

The record class, com.dynasim.record implements the map-interface, and the contents is mapped as for interpretMainStatic (also when calling a function written in Java from Modelica). The reason is the same as for interpretMainStatic. However, for easy access to simple variables there are also special functions, getDouble, getInt, and getBoolean.

The map for records is straightforward to use and by being name-based avoid issues with declaration order and future extensions of the records in Modelica.

2.4 Mapping of errors

Exceptions thrown from Java called from Modelica are automatically mapped to assertions, which is the normal error handling primitive in Modelica. Currently an assertion stop Dymola's interpreter are there is no way of catching the error inside Modelica.

When an assertion (or other error) is trigged in Modelica originating from a call to interpretMainStatic this is mapped to an exception in Java as follows:

- com.dynasim.DymolaException base-class of the other exception – introduced in order to make it easy to catch all exceptions.
- com.dynasim.DymolaNoSuchFunction(<name of function>) when the function is not found by interpretMainStatic.
- com.dynasim.DymolaIllegalArgumentException for problems with transforming results or argument between Java and Dymola, and incorrect type of arguments to function.
- com.dynasim.DymolaEvaluationException when evaluation fails – e.g. assertions and division by zero.

These exception classes all inherit from java.lang. RunTimeException, this ensures that no 'throws' clause is needed for routines calling Dymola-functions.

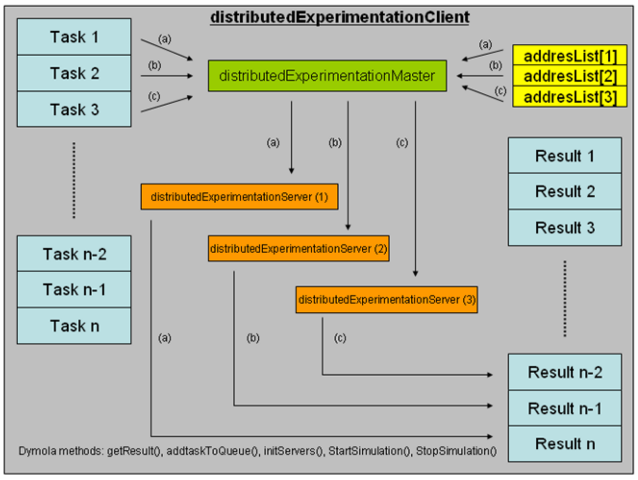
This corresponds to the Modelica environment where no "assert-clause" is needed.

3 Distributed simulations with Java

The new facilities accessible from Java for calling a Dymola instance, allows the use of transparent RMI [4] for distributed tasks. Since Dymola simulations for parameter experimentation are independent, long simulation times can be shortened by using several processors. A version of the Experimentation package [5] has been adapted to distributed simulations.

3.1 Setup

The command setup is of distributed Experimentation package is identical to the one described in [5].


distributedExperimentationClient
Task 1
Task 2
Task 3
Task n-2
Task n-1
Task n
distributedExperimentationMaster
addresList[1]
addresList[2]
addresList[3]
distributedExperimentationServer (1)
distributedExperimentationServer (2)
distributedExperimentationServer (3)
(a)
(b)
(c)
Result 1
Result 2
Result 3
Result n-2
Result n-1
Result n
Dymola methods: getResult(), addtaskToQueue(), initServers(), StartSimulation(), StopSimulation()

The main architecture implemented in Java is depicted in the previous figure.

The architecture of the Java code is simple:

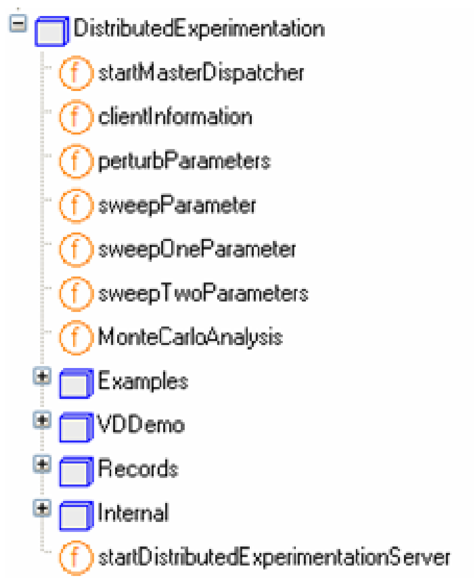
- A master dispatcher with three lists working as queues: Task Queue, ResultQueue and AddressList
- Independent threads for each server. The thread starts, monitors and reports results to the static lists in the dispatcher.
- External Java functions in Modelica to access the different Queues.

Single signals of whole trajectory files can be sent back to the master computer if requested, for animation purposes.

Having all this functionalities at hand, the Distributed Experimentation package was written, and we describe it in the following.

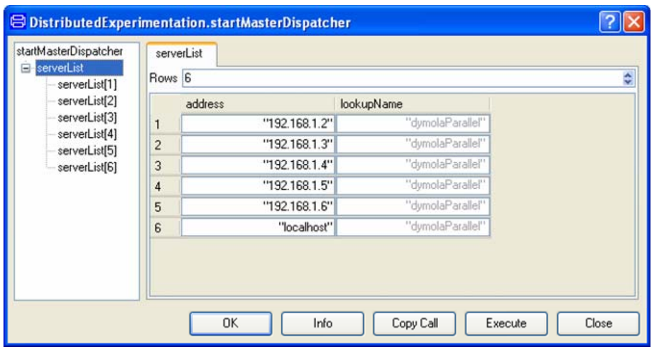
3.2 Distributed Experimentation package

The structure of the Distributed Experimentation package is very similar to the Experimentation package. The functionalities are the same, as shown below.



```
DistributedExperimentation
    startMasterDispatcher
    clientInformation
    perturbParameters
    sweepParameter
    sweepOneParameter
    sweepTwoParameters
    MonteCarloAnalysis
  Examples
  VDDemo
  Records
  Internal
    startDistributedExperimentationServer
```

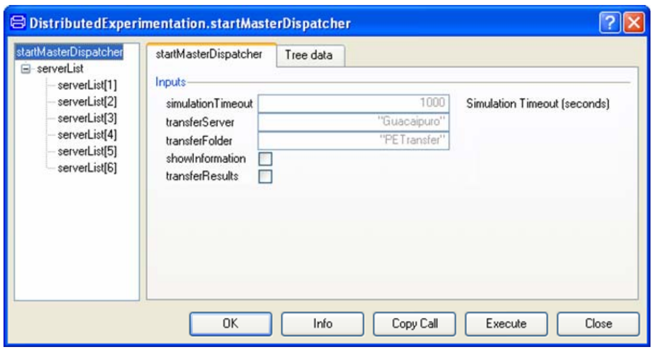
The main difference to the experimentation package is the distribution of tasks from client to server. The GUI and setup are the same, and the function startMasterDispatcher has to be run before any simulation is performed. The information of the servers and their lookup names is reported to the dispatcher with this function. See the figure below.



Rows 6

	address	lookupName
1	"192.168.1.2"	"dymolaParallel"
2	"192.168.1.3"	"dymolaParallel"
3	"192.168.1.4"	"dymolaParallel"
4	"192.168.1.5"	"dymolaParallel"
5	"192.168.1.6"	"dymolaParallel"
6	"localhost"	"dymolaParallel"

The names "192.168.1.2", "192.168.1.3" and so on are IP numbers of the server hosts. DNS name resolving is also supported, and therefore possible to use names instead. The name "localhost" represents the address 127.0.0.1 as usual. Use it to include the own computer which also runs a server.



simulationTimeout	1000	Simulation Timeout (seconds)
transferServer	"Guacaipuro"	
transferFolder	"PETransfer"	
showInformation	☐	
transferResults	☐	

In the view above it is included the set up for trajectory file transferring. Check "transferResults" if transfer files to the dispatcher computer is to be done. The variable "transferServer" is the network name of the client computer and the "transferFolder" is the **network name** of the shared folder.

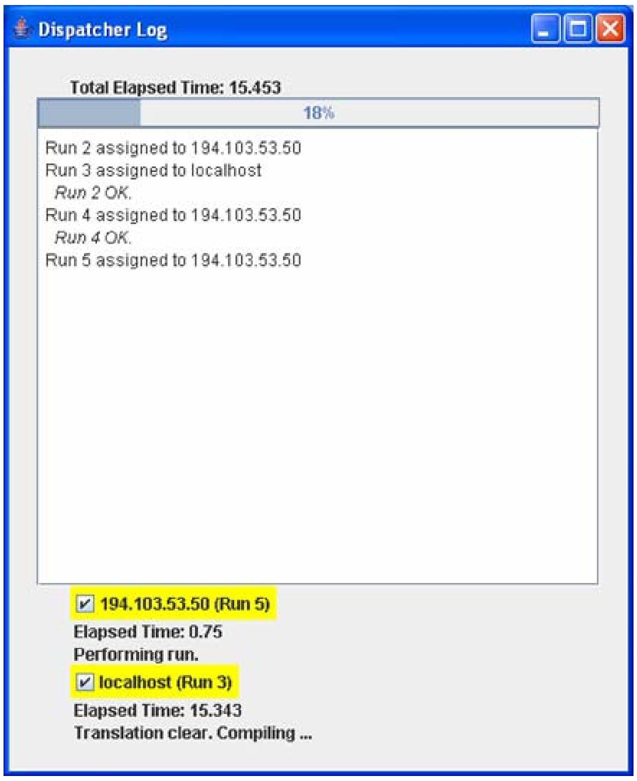
The flag showInformation makes Dymola show a window with the registered servers for debugging purposes.

The functionalities of perturbParameter, sweepParameter, sweepOneParameter, SweepTwoParameters and MonteCarloAnalysis are described in more detail in Dymola Additions document, and can be used directly here. We will focus here on the setup and running of the examples. We consider coupledClutches as a reference case study.

The subpackage VDDemo has two functions: vehicleSweepParameter and animateResultFile, used for demonstration purposes.

3.2.1 Progress Monitoring

The task dealer presents in this window which task was sent to which server, as well as the overall progress. The progress bar indicates the percentage of work done and received by the client.



Total Elapsed Time: 15.453

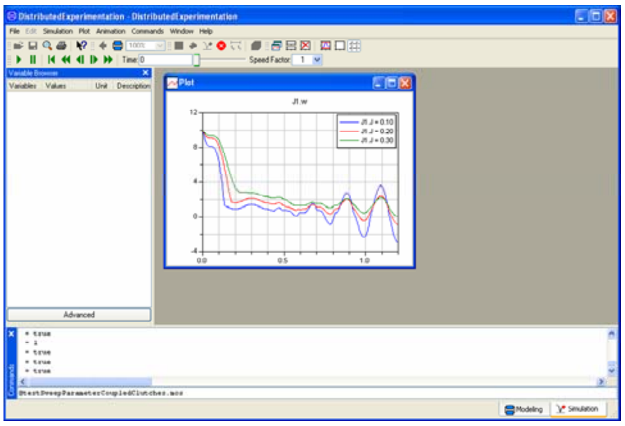
```
Run 2 assigned to 194.103.53.50
Run 3 assigned to localhost
 Run 2 OK.
Run 4 assigned to 194.103.53.50
 Run 4 OK.
Run 5 assigned to 194.103.53.50
```

☑ 194.103.53.50 (Run 5)
Elapsed Time: 0.75
Performing run.
☑ localhost (Run 3)
Elapsed Time: 15.343
Translation clear. Compiling ...

If any task could not be performed by a server, it will be back to the task queue and the respective server will be disabled (marked with red background). Yellow background denotes ongoing simulation on server. Green background denotes free server.

3.2.2 Collecting results

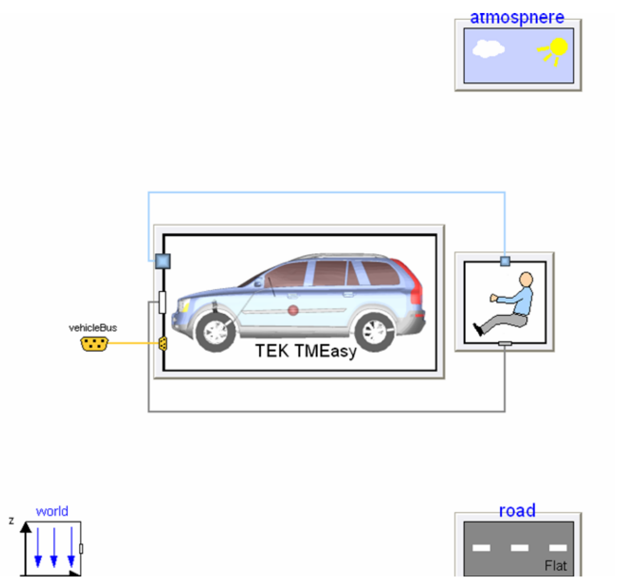
During simulation time, Dymola inspects the result queue in the master dispatcher waiting for results with a simple Modelica Function call. For instance, results are presented directly in sweepParameter as the results are arriving. It is up to the Modelica function whether to present the incoming results or not. In the figure below, the results of sweepParameter on CoupledClutches example are depicted, after three results are arrived.



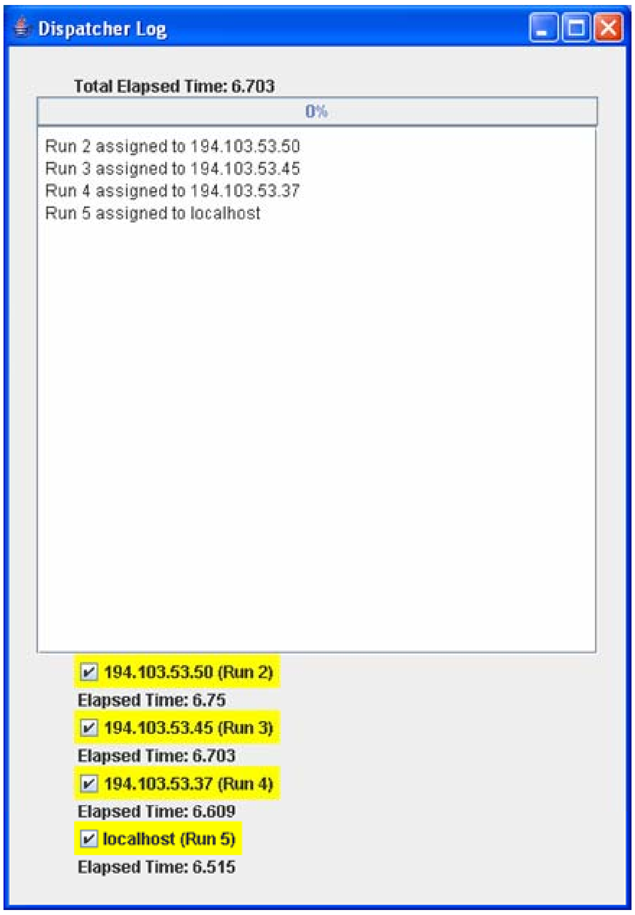
```
= true
= 1
= true
= true
= true
```

```
@startSweepParameterCoupledClutches.mos
```

Running a sweep with Payloads-example can be done by clicking on "Commands/Roof load (11 cases) with weights from 0 to 200 Kg", and then execute. The model is depicted below.



For this example we used four servers. The dispatcher log evolves as depicted in the figure below.


Dispatcher Log

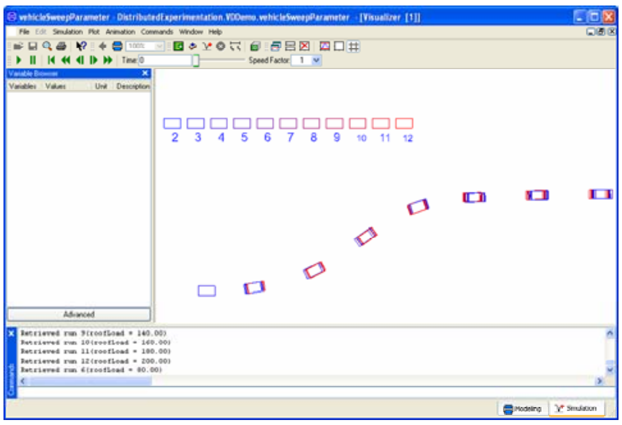
Total Elapsed Time: 6.703

0%

```
Run 2 assigned to 194.103.53.50
Run 3 assigned to 194.103.53.45
Run 4 assigned to 194.103.53.37
Run 5 assigned to localhost
```

✓ 194.103.53.50 (Run 2)
Elapsed Time: 6.75
✓ 194.103.53.45 (Run 3)
Elapsed Time: 6.703
✓ 194.103.53.37 (Run 4)
Elapsed Time: 6.609
✓ localhost (Run 5)
Elapsed Time: 6.515

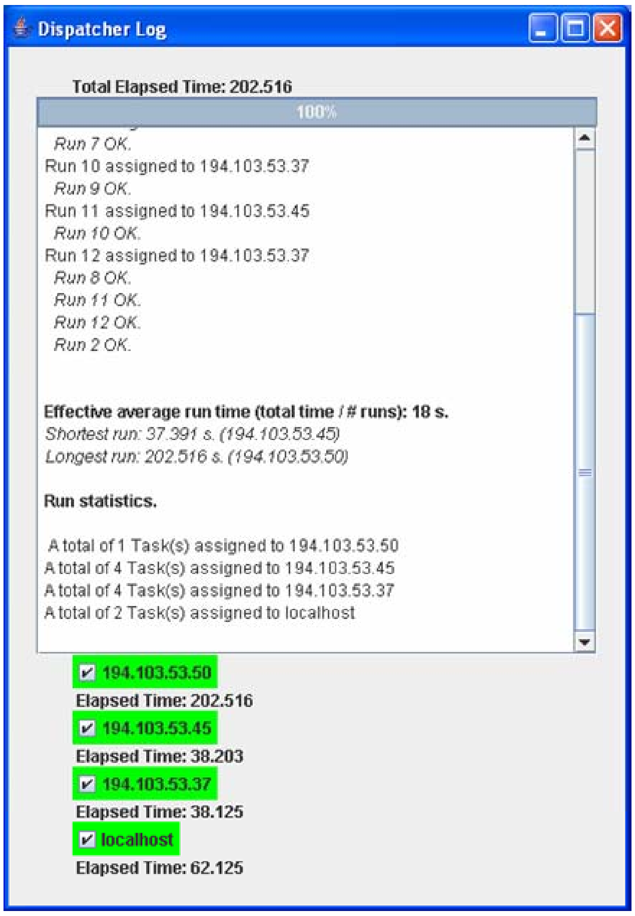
Dymola shows the following view when all 11 cases are completed. The visualisation is performed by a special visualizer written for this example. In the figure below, we observe a rectangle defined by the position of the four wheels and the identifier of the case.


vehicleSweepParameter - DistributedExperimentation.VDDemo.vehicleSweepParameter - [Visualizer (1)]
2
3
4
5
6
7
8
9
10
11
12
Advanced
Modeling
Simulation

```
Retrieved run 9(roofLoad = 140.00)
Retrieved run 10(roofLoad = 160.00)
Retrieved run 11(roofLoad = 180.00)
Retrieved run 12(roofLoad = 200.00)
Retrieved run 6(roofLoad = 80.00)
```

We observe with his experiment that the car is stable with regard to a payload from 0 to 200 Kg.

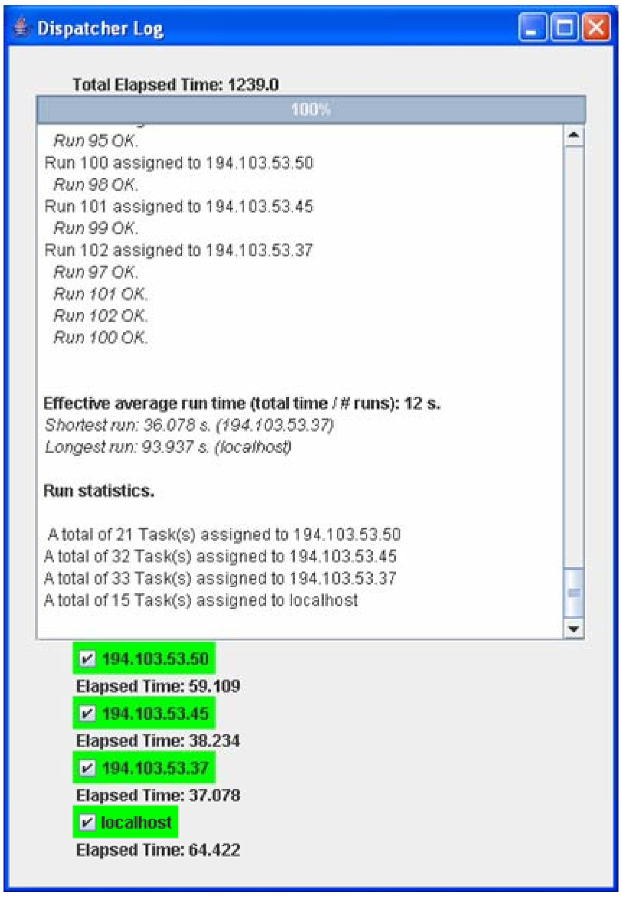
The final log window has the following information and statistics. The first simulation was ready after 86.5 seconds (translation, compilation with Visual Studio 2003 and running). We analyse now two runs of sweepParameter.



```
Dispatcher Log
Total Elapsed Time: 202.516
100%
  Run 7 OK.
Run 10 assigned to 194.103.53.37
  Run 9 OK.
Run 11 assigned to 194.103.53.45
  Run 10 OK.
Run 12 assigned to 194.103.53.37
  Run 8 OK.
  Run 11 OK.
  Run 12 OK.
  Run 2 OK.

Effective average run time (total time / # runs): 18 s.
Shortest run: 37.391 s. (194.103.53.45)
Longest run: 202.516 s. (194.103.53.50)

Run statistics.

 A total of 1 Task(s) assigned to 194.103.53.50
A total of 4 Task(s) assigned to 194.103.53.45
A total of 4 Task(s) assigned to 194.103.53.37
A total of 2 Task(s) assigned to localhost

✓ 194.103.53.50
Elapsed Time: 202.516
✓ 194.103.53.45
Elapsed Time: 38.203
✓ 194.103.53.37
Elapsed Time: 38.125
✓ localhost
Elapsed Time: 62.125
```

To interpret the results, we consider the total time against the best time. Using the best computer, the total time for eleven simulations would be 411 seconds. As a result of the distributed simulation, the whole task needed only 202.5 seconds. This means that the speedup factor was of 2,036. We observe also that the localhost and the first server are slowing down the simulation.

For a second run, we make the same sweep but for one hundred simulations. In this way, we get rid of initialisation overhead. Observe the final dispatcher log.



```
Dispatcher Log
Total Elapsed Time: 1239.0
100%
  Run 95 OK.
Run 100 assigned to 194.103.53.50
  Run 98 OK.
Run 101 assigned to 194.103.53.45
  Run 99 OK.
Run 102 assigned to 194.103.53.37
  Run 97 OK.
  Run 101 OK.
  Run 102 OK.
  Run 100 OK.

Effective average run time (total time / # runs): 12 s.
Shortest run: 36.078 s. (194.103.53.37)
Longest run: 93.937 s. (localhost)

Run statistics.

 A total of 21 Task(s) assigned to 194.103.53.50
A total of 32 Task(s) assigned to 194.103.53.45
A total of 33 Task(s) assigned to 194.103.53.37
A total of 15 Task(s) assigned to localhost

✓ 194.103.53.50
Elapsed Time: 59.109
✓ 194.103.53.45
Elapsed Time: 38.234
✓ 194.103.53.37
Elapsed Time: 37.078
✓ localhost
Elapsed Time: 64.422
```

We analyse this log in the next section.

3.3 Asymptotic resulting speed up

Confirmation of theoretical Speed up factor (approx. number of servers) has been observed using long simulation times using VehicleDynamics 1.0.2.

The effective time used per simulation for the second sweep is of 12 seconds. The fastest simulation was 36.07 seconds. If we just only used the fastest computer, the total elapsed time would have been 3607 seconds. The total elapsed time with these two servers was 1239 seconds. The resulting speedup factor is 2.911 compared to the most optimistic scenario.

It is our experience that localhost behaves slower than its equivalent server, since the dispatcher runs also in the same computer. We added a computer double so slow than the other two. Equivalently, we were running with three equally fast servers in total, reflected in the example.

4 Conclusions

A new interface between Modelica and Java has been implemented. This enables Modelica and Dymola to take advantage of all Java features, making possible the incorporation of java objects in Modelica.

As such an example of Java applications with Dymola, distributed simulations with Dymola where implemented using the transparent RMI Java package. High speed up factors were observed with low overhead. The main advantage is that RMI handles directly all concerning network communication, while Dymola handled all simulation aspects.

References

[1] Dymola User's Manual, www.dynasim.com

[2] Elmqvist H., Olsson H., Mattsson S.E., Brück D., Schweiger C., Joos D., Otter M., Optimization for Design and Parameter Estimation, Proceedings of the 4th International Modelica Conference, Hamburg-Haburg, Germany, 2005.

[3] Olsson H., External Interface to Modelica in Dymola, Proceedings of the 4th International Modelica Conference, Hamburg-Haburg, Germany, 2005.

[4] Gur-Ari, G. Empower RMI with TRMI. http://www.javaworld.com/javaworld/jw-08-2002/jw-0809-trmi.html

[5] Dymola Users Manual – Dymola 6.0 Additions, shipped together with Dymola distribution.