MODELLING OVERHEAD OF DISTRIBUTED SYSTEMS USING STATISTICAL TECHNIQUES

Frederic Hancke, Frans Arickx, Jan Broeckhove and Tom Dhaene
Department of Mathematics and Computer Science
University of Antwerp, Belgium
frederic.hancke@ua.ac.be

ABSTRACT

This contribution is an attempt to model the overhead of distributed systems using appropriate statistical techniques. The communication and platform dependent overhead of different distributed systems are estimated by comparing experimental and model-based data.

INTRODUCTION

In our research group CoMP (*Computational Modelling and Programming*) research is done in the field of computational science, aimed at understanding physics and engineering problems through modern modelling techniques, using new software development paradigms and advanced mathematical techniques.

Many problems, as in the area of quantum physics, often involve very intense and large computations. Therefore, distributed platforms, such as JavaSpaces [7], MPICH [3] (an MPI [8] implementation), etc. provide a better and faster solution. The goal of our project is to characterise different existing platforms for certain kinds of calculations.

The process of testing distributed systems could be illustrated with a cycle. The first step is to define the test to be performed, the second to run the test and the third to analyse the results obtained from the test. The analysis may detect interesting areas for further investigation which leads to redefining the test and thus concluding the cycle.

This contribution is a first attempt at modelling overhead of distributing tasks in JavaSpaces, a Linda tuple spaces [5] implementation. Firstly, a framework for easily building distributed applications using the tuplespaces paradigm, in particular farmer-worker applications, will be discussed. Secondly, the setup of the testcase will be explained, and finally, results of the tests performed are analysed.

FRAMEWORK

In our research activities we are often confronted with the question which distributed platform to choose for a certain calculation, often based on its brute performance. Therefore, different distributed systems have to be set up and tested. In the past, separate implementations for different platforms had to be made. The framework briefly discussed here has been developed to minimize the extra work when writing a test for a different, previously not tested, distributed platform.

The framework has been developed in Java using the farmer-worker pattern with a communication concept of tasks and results. The first being distributed by the farmer and processed by the worker. The latter being returned by the worker upon finishing task processing, and collected by the farmer. The underlying communication system is based on the tuple spaces paradigm. Thus, tasks and results are being *sent* between farmer and workers using the concept of a space with basic operations read, write and take. This means farmer and worker do not communicate directly with each other, but rather via a space, although this could be forced using appropriate task descriptions. As distributed systems other than tuple space implementations could also be plugged in, at least when providing the three basic operations, the framework will not be perfect for these. The reason for this is of course that distributed platforms such as MPICH use concepts (like direct message passing) that are very different from tuple spaces.

The architecture of this framework consists of layers. As previously mentioned, the framework is implemented in Java. The first layer around Java provides the structure for building farmer-worker applications using a task-result communication via a space. The outer layer provides a frame for testing platforms using XML based sleep tasklists [9]. When using this framework, a couple of methods/classes need to be implemented or extended.

TESTCASE FOR JAVASPACES

The aim of this testcase is to find out whether JavaSpaces is a good potential candidate to solve high performance calculations, such as the quantum physics problems mentioned in the introduction. Other platforms such as MPICH, TSpaces [4, 12] and GigaSpaces [1] will be tested the same way in the near future. Besides the platforms functionality, we are also interested in its performance measured against a simple formula.

JavaSpaces

JavaSpaces is one of many implementations of the so called Linda Spaces distributions concept. The underlying idea is that objects can be thrown into a virtual space and taken out, or simply read, by any object connected with the space. Many distributed platforms have been built using this idea. Other implementations, besides JavaSpaces, are TSpaces and GigaSpaces.

JavaSpaces was built on top of Jini [2, 6] as a service of the Jini technology. Its functionality was kept very simple, but nevertheless is very powerful. In fact there are only three basic actions on the JavaSpace itself: **write** to write an object *into* the space, **read** to read an object from the space, but leaving the object *in* the space, and **take** to take an object *out* of the space. There is a fourth operation possible, but this one does not really perform on the space: **notify**, which notifies an object of objects being added to the space.

Setup

In total, five machines have been used to perform the tests. One Intel PII 400 for the HTTP server, RMI Activation Daemon, Lookup service, JavaSpace service and the farmer, and four Intel PIV 1.7, each for a separate worker.

Measures

The tests have covered different values (in the near future, tests will be performed with more different values for w and t) for four different parameters. These are:

- w: the number of workers ($w \in \{1, 2, 3, 4\}$),
- t: the number of tasks ($t \in \{10, 50, 100\}$),
- m: the mean (in ms) of the Gauss distribution for tasks ($m \in \{1, 10, 100, 500, 1000, 2000, 5000, 10000\}$), and
- v: the standard deviation (as a percentage of m; $0 \leq v \leq 100$) of the Gauss distribution for tasks ($v \in \{1, 2, 5, 10, 20, 30\}$).

Thus, for every combination of t, m and v, a tasklist has been generated, except for the combinations of $m = 1$ for all v, and $m = 10$ with $v \in \{1, 2, 5\}$. For these combinations mv becomes real. Instead, tasklists for $m = 1$ and $m = 10$ were generated with task duration m for all tasks. So a constant distribution was used instead of the normal distribution.

This means 123 different tasklists were generated. Execution of the farmer-worker process with a given tasklist resulted in one value representing the duration wct (wall-clock time) of the whole process. Every test ran 10 times for each tasklist and for each w, resulting in 4920 data points.

Statistical Analysis

As the data set for evaluating JavaSpaces is, as mentioned in the previous subsection, not yet complete, we will give a first brief analysis in this subsection.

First, the mean duration of all tasks in one tasklist was calculated. This was done to prevent using the theoretical mean that was used to generate the tasklist, because this would corrupt further calculations. A simple formula was used to get a first indication of the performance of JavaSpaces:

$$T = \lceil \frac{N_{task}}{N_{work}} \rceil T_{task} \qquad (1)$$

where N_{task} denotes the number of tasks, N_{work} the number of workers, and T_{task} the mean duration of one task (note that this is the most simplistis form of approximating the real execution time in distributed systems). In the rest of this section we work with the overhead of distributing one task in JavaSpaces, which is given by

$$\frac{wct - T}{t}$$

As robustness of data sets [10] is not self-evident, we first took out a number of potential outliers. *Boxplots* and *Stem-and-Leaf Plots* are two possible methods to identify potential outliers. Each of these does not necessarily produce the same results. The decision which outliers will finally be discarded, is up to the user. Each potential outlier must be investigated carefully and may only be discarded if there is a good reason. In our case, occasionally high network or processor load might be good reasons.

The method we used is that of boxplots. Using these, one can distinguish potential *mild* outliers from potential *extreme* outliers. A data point is marked as a mild outlier if $d < Q1 - 1.5IQR$ or $d > Q3 + 1.5IQR$ [11] with d the value of the data point, $Q1$ and $Q3$ respectively the first and the third quartile and $IQR = Q3 - Q1$ the interquartile range. A data point is marked as an extreme outlier if $d < Q1 - 3IQR$ or $d > Q3 + 3IQR$ (remark that extreme outliers are also mild outliers).

This technique was applied on our data set. Table 1 shows a comparison of the resulting statistics. The table shows a significant difference between the mean, standard deviation, minimum and maximum using the complete data set (ALL) and using the complete data set discarding extreme outliers (ALL - EO). The median, $Q1$ and $Q3$ shrink only slightly, which means that the complete data set was corrupted by only a few (171) data points. The difference between discarding extreme outliers and discarding mild outliers confirms this. As it is better to discard as few data points as possible, we will use the complete data set discarding only the extreme outliers in the rest of this subsection.

	ALL	ALL - EO	ALL - MO
N	4920	4749	4589
N (%)	100.00	96.52	93.27
Mean	50.5536	41.1575	39.8202
Median	40.0600	39.2000	38.7000
Std. Deviation	184.1168	30.8262	26.5052
Minimum	-559.60	-75.00	-34.30
Maximum	11859.77	168.03	113.20
$Q1$	20.3550	20.1300	20.2450
$Q3$	57.4950	56.1300	55.0000

Table 1: Statistics on the overhead (in ms) using JavaSpaces for distributing one task, using respectively the complete data set (ALL), the complete data set discarding extreme outliers (ALL - EO) and finally the complete data set discarding mild outliers (ALL - MO)

The graph for the overhead using JavaSpaces for distributing one task is shown in figure 1. The mean is marked with a solid line at $41.1575ms$. The graph is divided in four columns grouped by the number of workers. So, from left to right we have first the data points based on 1 worker, then 2 workers, etc.

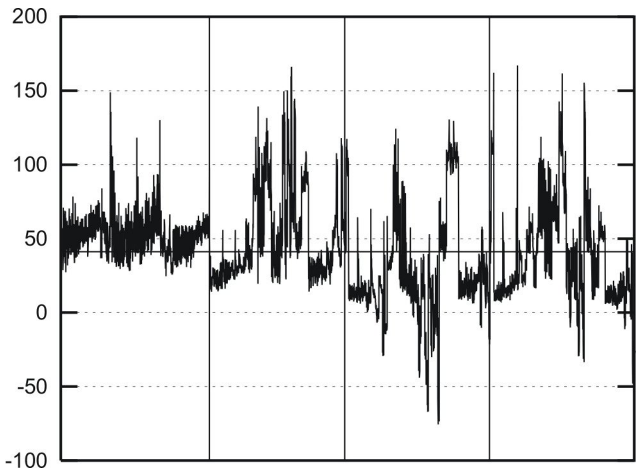

Figure 1: Graph representing the overhead using JavaSpaces for distributing one task. The X axis represents the data points, the Y axis the overhead (in ms)

There are still a few problems with this visualization. As mentioned earlier, the graph does not represent data measured with more than 4 workers. Another, more important, problem is the fact that some data points in the graph have negative values, which influence the mean badly. The reason for these negative values lies in equation (1). Indeed, all tasks are supposed to have the same duration (the mean of the tasklist, not the theoretical mean). In the calculations with 3 or more workers, the following situation may occur: suppose 4 tasks must be distributed with each a length of respectively 3, 4, 3 and 2. The mean of this tasklist is clearly 3. So, 4 tasks of length 3 will be theoretically distributed, resulting in a total computation time of 6. However, in practice, worker 1 could compute the tasks with length 3 and 2, worker 2 the task with length 4 and worker 3 the task with length 3, resulting in a total computation time of 5. Thus, situations in which tasks are distributed in such a way that they outperform the theory might occur, leading to a negative overhead.

CONCLUSIONS

This contribution discussed briefly the design and implementation of a framework created to minimise efforts for building distributed applications using the farmer-worker pattern with tasks and results. Furthermore, JavaSpaces served as a testplatform for a first attempt at modelling the overhead using statistical techniques. When more detailed data sets will become available, more thoroughgoing analysis tools will be used, such as analysis of variance (ANOVA), to obtain better predictions of the overhead.

REFERENCES

[1] Gigaspaces. URL: http://www.j-spaces.com.

[2] Jini. URL: http://www.jini.org.

[3] Mpich. URL: http://www-unix.mcs.anl.gov/ mpi/ mpich/.

[4] Tspaces. URL: http://www.almaden.ibm.com/ cs/ TSpaces/.

[5] Yale linda group. URL: http://www.cs.yale.edu/ Linda/ linda.html.

[6] W. Keith Edwards. *Core Jini*. Prentice Hall, second edition, 2001.

[7] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison Wesley, 1999.

[8] William Gropp, Lusk Ewing, and Anthony Skjellum. *Using MPI*. The MIT Press, second edition, 1999.

[9] Frederic Hancke, Gunther Stuer, David Dewolfs, Jan Broeckhove, Frans Arickx, and Tom Dhaene. Modelling overhead in javaspaces. In *Proceedings Euromedia*, pages 77–81. Eurosis, 2003.

[10] David C. Hoaglin, Frederick Mosteller, and John W. Tukey. *Understanding Robust and Exploratory Data Analysis*. Wiley, 2000.

[11] Neil A. Weiss. *Introductory Statistics*. Addison Wesley, sixth edition, 2002.

[12] P. Wyckoff. Tspaces. Technical report, IBM Almaden Research Center, 1998.