Performances and Mobile Code in Coordination Environments: the Jada Approach

Davide Rossi
Davide.Rossi@CS.UniBO.IT

Dipartimento di Scienze dell'Informazione
University of Bologna - Italy

Abstract. Coordination was born in the context of parallel/distributed computation community; from there it expanded to distributed systems design where it is usually seen as an enabling technology to manage complex relationships between components. Recently coordination technologies have been applied to the Internet and the World Wide Web (entering the Open Distributed Systems arena) in order to provide a middleware to support distributed collaborative applications. This evolution of the coordination technology often left behind one important issue: performances. In this paper we show that, if implemented with performances in mind, a coordination system developed for the WWW can be used also for computational-intensive applications. We also introduce a mechanism based on code mobility that shows its usefulness for enhancing (from both a performances and fault tolerance point of view) the coordination of distributed components.

1 Introduction

The evolution of coordination systems towards an Internet middleware frequently left performance issues behind. It is true that the target applications are rarely computationally intensive but when we have to deal with systems that have to coordinate hundreds of users and agents, good performances in the coordination technology trace the difference between really usable systems and *wait please* systems; the most obvious outcome is that performances really affects the scalability of a multi-user application. Having performances issues in mind when we design a coordination system is then a good thing, moreover the use of new technologies (at least in the context of Open Distributed Systems) such as *mobile code* [FPV98] can help our seek for speed and scalability. In this paper we test the performances of Jada, a coordination package for Java. We then introduce a new coordination operation based on code mobility and we test its usefulness for both performance enhancements and network failure overcoming. The structure of the exposition is as follows: in Sect. 2 we introduce Jada; in Sect. 3 we show the result of a set of performance test run on Jada; in Sect. 4 we show how to exploit code mobility in coordination systems; in Sect. 5 we show some related approach; Sect. 6 concludes the paper.

2 Jada

Jada is coordination package for Java. Using Jada we can coordinate parallel/distributed components using a set of operations introduced by the coordination language Linda [CarGel92]. The main idea is to have a basic set of primitives that allow the user to implement both data sharing and synchronization using a shared, structured, data spaces. Jada extends this basic concepts by implementing object spaces (i.e. specialized object containers) as data spaces and by allowing the creation of multiple spaces. The basic entity in Jada is in fact a *space*. Concurrent threads can access a space by using a small set of simple primitives:

– `out`: put an object in the space;
– `read`: associatively retrieve an object from the space; if more object can be retrieved the one returned is chosen non deterministically; if no object is available at the time of the call the calling thread is blocked until the operation can be successfully performed;
– `in`: works just like `read` but the returned object is removed from the space;

Jada also provides non blocking and time-out aware versions of the `read` and `in` operations.

Jada supports a client/server based mechanism that allows even distributed components to access an object space (two version of the server/client pairs are provided: one based on a simple stub/skeleton mechanism that uses socket connections and another one based on Java RMI).

Since an object space is a simple object each application can create several (either local or remote) object spaces. We can then use the same paradigm to archive data-driven coordination [PapArb98] in both parallel and distributed applications (with the obvious difference that accessing a remote object space can fail because of network troubles).

2.1 Remote Evaluation

`eval` is a Jada operation that can be performed on an object space. The result of this operation is that the specified object (that must implement a specific interface) is associated to a new execution thread and the handle to the object space is exported to it. While the operation performed by `eval` on a local space are easily implementable using the standard Java threads creation calls, it becomes interesting to perform this operation when we apply it on a remote space. In this case the specified object is serialized (using the standard Java mechanisms) and sent to the host where the space server is running. What we are doing this way is to exploit mobile code technology with Jada. In our case we are moving code and data but not execution state (this is not possible with Java). This allows us to exploit the many well known advantages of the mobile code technology [MO97] such as the ability to move a part of the client code that needs a big exchange of data with the server directly to the server host, thus overcoming the network's introduced delays and failures.

3 Performance Tests

Parallel/distributed calculation is not the main application target for Jada but we acknowledge that the performances of a remote space server can have great impact on the scale of a distributed application. Performances of shared-memory based coordination systems has been an hot topic for sever years and many works have been proposed to address this issue. Performances can, in fact, be improved with various solutions but the inherent speed of the coordination kernel (i.e. the implementation of the primitives to access the shared space) is, of course, the basis to archive good overall speed.

We run a set of tests on Jada using an interactive Mandelbrot set generator (see Fig. 1): in our opinion it is a good choice since, while the algorithm is inherently parallel, the computational effort requested for each step is not excessively high (and thus the parallelization of the program is non-trivial).

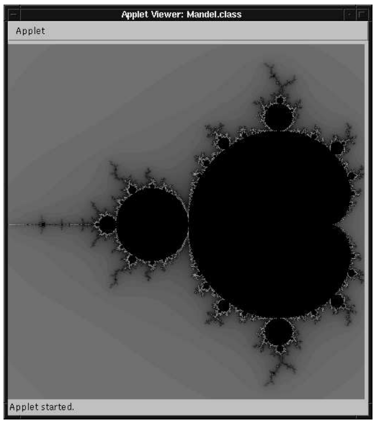

Fig. 1. The interactive Mandelbrot set generator

All the tests have been performed on Intel based hardware running Linux. The tools used are: the port of Sun's JDK for Linux; TYA: a JIT for Linux; Toba: a Java to C translator with JIT support. It should be noted that, as of version 1.1.7a, the port of Sun's JDK for Linux also supports native threads and thus is able to take advantage of SMP machines. All the tests have been run on a 500 x 500 pixels image split among 64 (8 x 8) regions.

The first test is a comparison between the sequential (1 worker) and parallel (2 workers, 4 workers, 100 workers) versions of the application.

It is easy to see from Fig. 2 that, where SMP hardware is available, the application is able to take full advantage of that. It is also worth to remark that a large number of computational threads (used to run the workers) has little effect on the overall speed of the program. We can conclude that the access to local object spaces using Jada is fast enough not to affect the speed of the

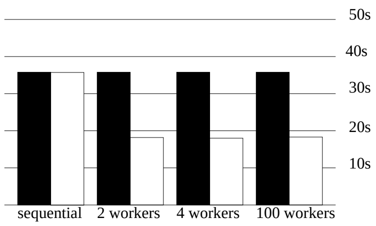

Fig. 2. Legenda:
black boxes: Intel Pentium II 300 MHz based workstation
white boxes: dual Intel Pentium II 300 MHz based workstation

computation; of course this apply to our test case where just two processors are involved in the calculation. Unluckily we don't currently have access to machines with more than 2 CPUs and thus we don't have any test on that kind of hardware. It should really be interesting to see how many processors are needed before the performances start to drop; it is our opinion, however, that a similar speedup should be archived with 8-16 CPUs.

Now we present the results of a set of tests run to evaluate the performances of our application in a distributed environment. The components of our system are: one master (that also runs the user interface); one remotely accessible object space (that, if the master is run as an applet, must run on the same host that runs the HTTP server from which the HTML document is downloaded because of the well known security limitation imposed to applets networking); one or more workers. A well known parameter that affect the evaluations of this kind of applications is the ratio between computational power of a single host and the network speed. For this reason we decided to run our distributed components with both "plain" JDK and with a JIT. This way we're able to indirectly affect the computational power of an host and thus we can obtain interesting figures (like: what happens if the remote space server runs on a slower/faster host? Is this a big bottleneck for the application?). All our tests are run on a LAN using a 100 Mbit/s ethernet connection. The first test uses just one worker. In this test all the three components are run on the same host. The results are shown in Fig. 3.

From this test we can see that the speed of the worker is important for the overall speed of the application but the same doesn't apply to the space server when we have to deal with just two remote components.

Now we compare the first and the second case of the previous test with the same configurations but with the worker running on a remote host. The results are shown in Fig. 4.

Now we compare the speed of the application when adding workers. In the next graph we compare the time needed to calculate the Mandelbrot set when using from 1 to 8 workers. In this test (see Fig. 5) we used the TYA JIT for the master and the workers and plain JDK for the remote space server, all the tests are run with Intel Pentium 200 MHz based workstations.

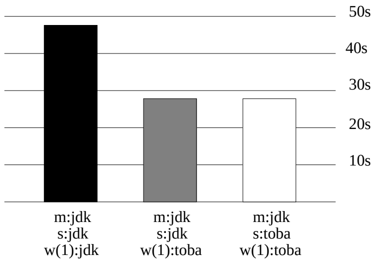

Fig. 3. Legenda:
black box: master on plain JDK (JDK), space server on JDK, worker on JDK
gray box: master on JDK, space server on JDK, worker on TOBA with JIT compiler (TOBA)
white box: master on JDK, space server on TOBA, worker on TOBA

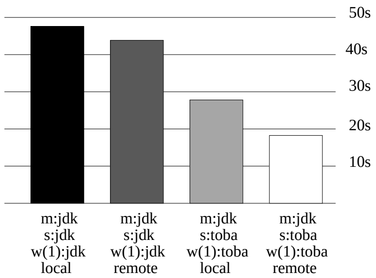

Fig. 4. Legenda:
black box: master on JDK, space server on JDK, local worker on JDK
dark gray box: master on JDK, space server on JDK, remote worker on JDK
light gray box: master on JDK, space server on TOBA, local worker on TOBA
white box: master on JDK, space server on TOBA, remote worker on TOBA

The speedups (with respect to the case of one worker) are: 1w: 1, 2w: 1.93, 4w: 3.36, 6w: 4.87, 8w: 5.67. The next graph shows the results of a test very similar to the previous one but in this case we put the remote space server on a dual Intel Pentium II 233 MHz based workstation. We implemented Jada's object space in order to take the maximum advantage of SMP workstations by taking great care of avoiding unneeded synchronizations and locks, leaving as much concurrency as possible in the access methods; our idea is in fact that the best way to archive performances enhancements with Jada, when the remote space server is the bottleneck of the system, is to put the server on a SMP machine with as many processors as possible. The results of the test are shown in Fig. 6.

In this case the speedups are: 1w: 1, 2w: 1.91, 4w: 3.57, 6w: 5.07, 8w: 5.72. It is easy to see that we have just a small improvement and thus we probably already got the best possible performance for this application.

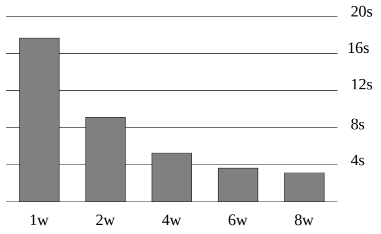

Fig. 5. Distributed application test

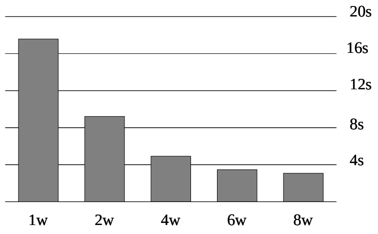

Fig. 6. Distributed application test: remote space on dual processor workstation

4 Exploiting Remote Evaluation

As explained before calling the eval operation on a remote space allows the remote client to move an object (code and data) to the host where the remote space is running. As soon as the object is received a new execution thread is created which task is to run the object's code; this mechanism resemble the usual techniques used by mobile agents systems and thus we will often refer to this object as an agent. The agent is then run on the same Java Virtual Machine (JVM) where the server is running and can then now access the space locally (using the standard space operators) with improved performances and without network troubles. The agent can perform just a limited set of operations and exit or could life in symbiosis with the space server for all of its lifetime (we call these *permanent agents*). Suppose, for example, that want to have a space that, as soon as an object of type `Item` is put on the space copies it to an object of type `IndexedItem` where the index field is set to an increasing number (so that we can enforce an ordering on the elements). To implement this behavior we can use `eval` (usually the non blocking version) to send to the space on agent that loops waiting for elements of type `Item`, gets them, makes the copy to an `ItemIndex` type and puts the results back into the space. With small changes we can use this technique to implement a wide set of data structures; what we obtain is then an extended space semantics: the behavior of the space is extended (but its interface is unchanged). Permanent agents can also be used to perform operations on the server when requested by clients. To implement this the agent should be waiting for a request that have to be put on the space by clients; when the request is got the requested operation is performed, when a result from the operation is requested the agent puts a result item on the space that contains the result of the operation. This kind of behavior is quite similar to that of a server and thus we refer to this kind of agents as *server agents*. Suppose we want to know how many objects of a given kind are present on a remote space at a given time. Without `eval` we have to `in` all of the objects, count them, and `out` them back to the space. Using `eval` we can run a server agent that counts the items and put a result element into the space. To take advantage of the server agent a remote client have then to perform an `out` operation (to put the request into the space) followed by an `in` operation (to get back the result). Is it also, of course, possible to extend the basic remote server interface with a new class that "hides" this two step operation inside a single method (and eventually performs the eval operation to run the server agents in its constructor) in order to obtain an extended server interface. Obviously we can get the same results without the `eval` operation (whatever can be performed by a server agent can be performed by a remote client too since both access the space using the same set of primitives) by implementing the algorithm that has to be executed from the server agent directly inside the new method; but there are some major differences (mostly when the algorithm is complex and needs lots of access to the space) that we'd like to remark:

- the only remote operation needed when we have a server agent are `out` request / `in` result (or even just `out` request when no result is needed): this can mean a difference in the execution speed of the algorithm of some order of magnitude.
- the `out` request / `in` result sequence needed when we have a server agent is very simple and much less likely to fail because of a network trouble; this becomes very important because, if the failure happens in the middle of the algorithm, the data in the remote server could be in an "inconsistent" state. Consider this very simple example: we want to put items on the remote space so that each item has an unique, progressive index. The algorithm is then: `in` index; set item index field to index; increment index; `out` index; `out` item. Now suppose that the client machine crashes right after the first operation: we got the index but we cannot put its updated version back into the space. Other remote clients can still access the space but, if they try to execute the same algorithm they are going to block (and eventually fail because of a time out) because the index item is no more available.
- the client can ask the server agent to execute the needed task without having to know anything about the algorithm used: the server agent could in fact be generated by another client. This way if we change the algorithm we don't have to update all of the clients but just the one that performs the `eval` needed to run the server agent.

It should be noted, however, that the activation of a server agent is an asynchronous task: synchronization must be directly addressed (using `out in` pairs) where needed.

4.1 Performances

Using the eval operation we can move part of the client code to the server host allowing a faster execution path to space primitives. An obvious side effect is that also computation effort is moved from clients to the server: abuse of this technique can lead to overloading of the remote server. When is correct to use `eval` to improve performances then? First of all it should be noted that if we just move sets of space primitives no real overload is put on the server: the same operations would have been performed as accomplishments of remote requests.

The real overhead is introduced by the calculation operations moved to the server host. Consider this simple example: the increment of an index item. The sequences of operations is: `in` index; increment index; `out` index. We have two space primitives and a calculation statement. The computational effort needed for the calculation is, however, very small and thus we can expect not to introduce any relevant overhead on the server. But what about performances? The code moved in this case is very small and, being the `eval` operation more complex than a simple `out` have we to expect an improvement of the performances using `eval` instead on `in` and `out`? Our tests show that already for such a small amount of code as the one needed to increment the index for five times eval gives a 10 percent speedup over the `in/ out` sequences. Of course, the larger the number of space primitives used in the code the bigger the archived speedup. We performed a set of tests using a simple algorithm: finding the maximum among all the integer items stored inside a space. Unless some kind of ordering is enforced on the elements when they are generated there is not a fast path to solve this problem: we have to `in` all of the items, find the smaller, and put all of the items back into the space (it should be noted here that if you really find yourself using this technique in a real program developed using Jada you're probably doing something wrong and you should have solved this problem in some smarter way). We performed our tests to find the maximum among a set of 5, 50 and 500 elements; to compare the performances of `in/ out` sequences and `eval` we run the program with a variable number of clients (from 1 to 6) connected to the same remote space and performing the same algorithm (on disjoint items sets); this way we can compare not just the difference of performances for a single client but also the overhear put on the server to handle the requests. The first test (see Fig. 7) compares the speed of execution (average of client's execution time) using `in/ out` sequences.

For each number of items we have 6 test (and thus 6 bars): with increasing number of concurrent clients. The white bars behind the first six bars are 10x magnification of the same values.

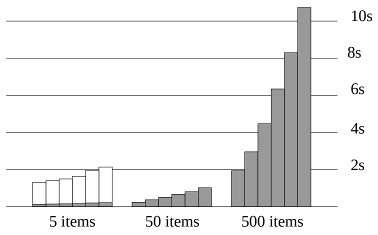

Fig. 7. Execution speed of `in/ out` sequences

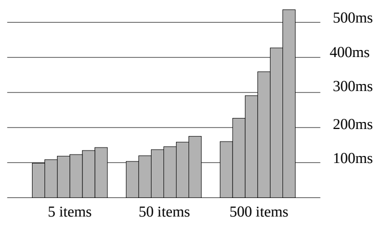

Fig. 8. Execution speed of `in/ out` sequences using `eval`

The second test (see Fig. 8) is just like the previous one but using `eval` instead of `in/ out` sequences (please note that the time scale is changed).

It is easy to see that we don't just have an improvement of the performances but we also have a better behavior when the number of elements is increasing. Another important issue that we should look at is the modification of the performances when adding more clients. Since it is not easy to evaluate this issue from the two previous graph we have two more graphs that show the ratio among the cases with 1 and 2 to 6 clients. The first graph (Fig. 9) is about the tests with `in/ out` sequences, the second (Fig. 10) is about the tests with `eval`.

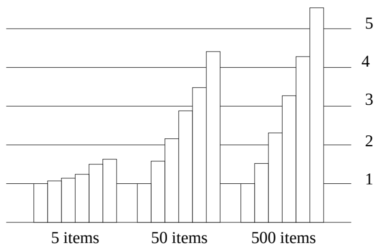

Fig. 9. `in/ out` sequences: speed ratios

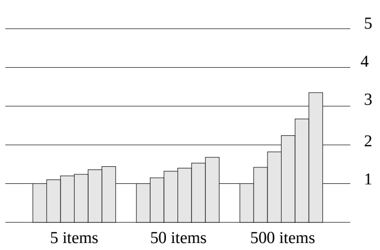

Fig. 10. `in/ out` sequences with `eval`: speed ratios

What these graph are saying is that the overhead put on the server by `eval` is less than the overhead due to `in/ out` sequences. Of course in our simple tests `eval` has a very small set of calculation operation, nevertheless the results are impressive: we could have thought that since `eval` explicitly puts an extra overhead on the server it should have been outperformed by `in` and `out`, instead it turns out that the hidden overhead due to `in` and `out` is worse than that of `eval`. The conclusion is then that, as long as we don't move too many calculation operations with it, eval does not just allow us to improve performances of single clients but even helps our multi-client applications to scale better.

5 Related Approaches

Two major projects by leading companies have been developed to bring the data driven coordination model to Java: TSpaces (from IBM) and JavaSpaces (from Sun Microsystems). The main differences between Jada and these two projects are the open matching mechanism provided by Jada and the ability of Jada to deal with local spaces (spaces in both TSpaces and JavaSpaces are always remote); many other differences, of course, exist but the ability to use the coordination model in both shared and distributed memory systems seems really the major issue. It should also be noted that Jada was born before these two systems.

An approach very similar to that of Jada's `eval` operation was developed independently by Antony Rowstron and is presented in [Row99]; the advantages of the system outlined in this work are, however, about fault tolerance rather than performances.

6 Conclusions and Future Work

Jada provides a set of classes to coordinate concurrent threads and/or distributer applications using shared object spaces that are accessible using a set of Linda-like operators. We run a set of benchmarks on Jada to test its speed and scalability. We also did a set of tests with a real word computational-intensive application in order to test the usefulness of Jada for parallel/distributed computations. We introduced the new `eval` operation, that makes use of mobile code technology, and we showed its usefulness for both performances and remote server loading. We had not the space to show that here but `eval` can also be used to overcome network troubles in a way that allows us to avoid transactions (that are often used in data-driven coordination systems to address fault tolerance).

Jada turns out to be a very flexible coordination package; it can be used in many different application area: from parallel computation to multi user distributed collaborative applications. The new `eval` operation extends this flexibility even further and allows to archive better performances, better scalability and failure protection.

All the work presented in this document is based on Jada and, then, on Java. It is our opinion, however, that many of the conclusion drawn in this context could easily extend to other similar systems based on different platforms. The `eval` operator could be implemented by other coordination languages and systems (probably introducing some small technological problem we avoided thanks to the platform independence of Java). The future work we are planning includes the extension of the `eval` mechanism in order to move agents not just to remote server's host but even to other hosts that have a faster access to the server (like an host in its same LAN, or an host dynamically choose with respect to the actual network parameters) allowing a better scalability for Internet-based applications while keeping most of the advantages of our implementation.

References

[Wood99] A. Wood. Coordination with Attributes. In *Proc. 3rd International Conference on Coordination Models and Languages (Coordination '99)*, pages 21–36. Springer-Verlag, Berlin, 1999.

[CarGel92] N. Carriero and D. Gelernter. Coordination Languages and Their Significance. *Communications of the ACM*, February 1992, pages 97–107.

[PapArb98] G. A. Papadopoulos and F. Arbab. Coordination Models and Languages. *Advances in Computers*, Academic Press, 1998, Vol. 46.

[Row99] A. Rowstron. Mobile Co-ordination: Providing Fault Tolerance in Tuple Space Based Co-ordination Languages. In *Proc. 3rd International Conference on Coordination Models and Languages (Coordination '99).* pages 196–210. Springer-Verlag, Berlin, 1999.

[MO97] J. Vitek and C. Tschudin editors. *Mobile Object Systems: Towards the Programmable Internet*, LNCS 1222, Springer-Verlag, Berlin, 1997.

[FPV98] A. Fuggetta and G. Picco and G. Vigna. Understanding Code Mobility. *IEEE Transaction on Software Engeneering*, 24(5) 1998, pages 342–361.