Analysis of Different Future Objects Update Strategies in ProActive

Nadia Ranaldo[1] and Eugenio Zimeo[2]

[1]Department of Engineering
University of Sannio
82100 Benevento – ITALY
ranaldo@unisannio.it

[2]Research Centre on Software Technology
University of Sannio
82100 Benevento – ITALY
zimeo@unisannio.it

Abstract

In large-scale distributed systems, asynchronous communication and future objects are becoming wide spread mechanisms to tolerate high latencies and to improve global performances. Automatic continuation, that is the propagation of a future object outside the activity that has generated it, can be used to further increase concurrency at system level through the anticipation of tasks. An important aspect of automatic continuation, which can cause different performance in different application and deployment scenarios, is the mechanism for updating result values of future objects, when they are ready. In this paper, we analyze the behaviour of the implementation of different updating strategies, by comparing them with the one currently implemented in ProActive. The experimental results show that the lazy home-based strategy behaves better than other strategies in some application scenarios that are very common in distributed applications.

1. Introduction

In distributed and parallel computing, applications, such as scientific data analysis and simulation, are typically composed of compute intensive tasks deployed on distributed machines that concurrently deal with huge amount of data. Performance of such applications can benefit from mechanisms that (1) improve concurrency among task activities and overlapping among computation and communication tasks and (2) minimize overhead tied to network latency involved especially in data transfer.

Asynchronous communication represents a valid candidate for this optimization since it permits the sender to immediately continue its computation after the communication has been started towards the receiver.

In many object-oriented systems, this mechanism is exploited to implement asynchronous method invocation of a remote object. With these kind of interactions, a high-level mechanism to coordinate activities of distributed objects through automatic data synchronization is based on *future objects* [1], which permits the calling object to obtain a reference to the future result of a method invocation, immediately returned by the asynchronous call, in order to permit the calling object to continue its computation concurrently with the called one, delaying any blocking until the result value is strictly necessary for next computations. The future object reference returned by an asynchronous method invocation is used by the requester to explicitly access to results. Many distributed systems aim to exploit performance optimization of future objects.

In [2] the authors propose a RMI extension that requires a pre-compilation phase of pseudo Java interfaces to introduce asynchronous communication and future objects through embedding threads into the RMI communication protocol. Another project based on RMI is RMIX [3], a framework that supports the configuration of different communication mechanisms based on synchronous and asynchronous protocols. The mechanisms delivered to update result values of asynchronous method calls are futures, completion callbacks and result queues. Also implementations of CORBA [4] over Java deliver asynchronous remote method invocations using IIOP as the underlying protocol.

In the Kan System [5] the asynchronous communication and future object mechanism is implemented through a special compiler, which produces byte-code compliant with any standard Java Virtual Machine (JVM).

In these distributed systems, future objects require explicit management of programmers, for example requiring to deal with the specific problem of data

1-4244-0910-1/07/$20.00 © 2007 IEEE.

synchronization and update, such as to wait for and access to result values, causing a decrease of programmer productivity who can not focus on functional aspects of applications.

A transparent and easy-to-use future object mechanism is delivered by ProActive [6] (version 3.1), a pure Java middleware that aims to simplify programming of distributed and parallel applications and their deployment on Local Area Network (LAN), on cluster of workstations, or on Internet-based grid systems, also featuring mobility, security and interoperability mechanisms.

Often in a distributed computation, data produced by some activities are used as input to other activities, and so on, until final results are produced. For this reason, data dependencies determine the chain of distributed activity activations and their implicit synchronization.

A classical distributed programming model based on such data-driven synchronization is the pipeline pattern, in which each pipe stage receives as input the output data of the previous stage and produces input data for the next one. Another example is the master-slave pattern, in which the master splits input data in sub-parts and sends each of them to slaves, while each slave computes the partial result and replies it to the master. Finally the master can complete its task of assembling the final result only when all partial results are received.

Another scenario of data-driven synchronization is in the context of workflow management systems, in which a workflow enactment engine invokes distributed activities and coordinates them passing input data and receiving output data, following a well-defined workflow process description.

Especially in the case of distributed activities, which require considerable execution times, concurrency and consequently system efficiency could benefit from *automatic continuation* [7], which is the propagation of a future object outside the activity that has generated it.

Automatic continuation can be adopted to immediately start method invocations that require as input parameter the output of previous asynchronous remote method calls. If at the moment of invocation, the input parameter has not yet been computed, a future object is used instead of the real input value. So, only when the called method actually uses the parameter, the activity is blocked until the value is available. By anticipating operations that are not tied to input parameters, automatic continuation is a mechanism that permits to improve performance through the overlapping of distributed activities that are invoked in sequential order [8].

One of the problems to address to efficiently implement future objects in middleware platforms is the definition of a strategy that minimizes data propagation in the network avoiding the distribution of future values to activities that do not use them.

One of the most widespread middleware platforms whose programming model is based on future objects and continuation is ProActive [9].

In this paper we study the behaviour of different updating strategies, namely eager home-based and lazy home-based strategies, and analyze them with respect to the eager forward-based strategy currently implemented in ProActive

The rest of the paper is organized as follows. Section 2 presents future object and automatic continuation mechanisms in ProActive. Section 3 presents a classification of update strategies and a comparison among them. Section 4 presents an experimental analysis of eager home-based and lazy home-based strategies with respect to the eager forward-based strategy implemented in ProActive. Finally Section 5 presents conclusions and future work.

2. Future Objects in ProActive

ProActive adopts extensively reflection mechanisms implemented through a run-time proxy based Meta-Object Protocol (MOP). This ensure to the middleware flexibility and extensibility, making the system open for adaptations and optimizations for improving performance without changes to the Java Virtual Machine (JVM), and without pre-processing or compiler modifications. In particular ProActive permits to configure the protocol used to export the remote objects. The default version of ProActive currently uses Java RMI as a portable communication protocol, but other protocols are supported, such as IBIS [10] and JINI [11].

ProActive proposes a heterogeneous programming model characterized by passive and active objects. *Passive objects* are common objects, whereas *active objects* are distributed objects with the following mechanisms managed by the framework: location transparency, activity transparency and synchronization. Thanks to polymorphism among the passive and active objects, these are used like passive ones but can be remotely created and accessed in a transparent way via asynchronous remote method invocations based on automatic and transparent future object mechanism called *wait-by-necessity* [12].

Transparent future objects of ProActive are implemented through MOP, which allows to customize the system behavior in order to react to specific run-time events, such as object creations and method invocations. To avoid virtual machines modifications and grant transparency, MOP uses the Proxy design pattern and the automatic generation of stubs (by using the tool BCEL [13] or ASM [14]): surrogate objects (stubs) intercept and reify object creation and method invocation events and pass them to the meta-level.

In ProActive the future object stub is a subclass of the

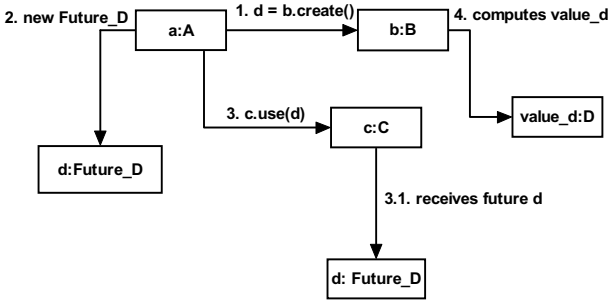

Figure 1. Automatic Continuation

return parameter class, in which each method is overwritten in order to verify whether the future object value was already replied by the invoked active object.

In ProActive, automatic continuations can occur when sending a request of remote invocation containing a future object as input parameter or when replying a result of a remote method invocation containing a future object not immediately used.

Figure 1 shows automatic continuation of a future object as input parameter for method invocation.

The updating of the result value, when it is computed, on all the active objects that have received the future object, is an important aspect of Automatic Continuation, which can determine different performance in different application and deployment scenarios.

3. A Classification of Updating Strategies

Several strategies can be adopted to update future objects [7]. They can be classified on the basis of the active object responsible of updating future objects and of the moment in which active objects receive updates.

Typically, the updating object can be the caller of an asynchronous invocation or the producer of the future result value. Consequently, two strategies are possible:

- *forward-based*: the active object that propagates the future object, called in the following *forwarder object*, after receiving the result value by the producer, is responsible to propagate the result to active objects that received the future;
- *home-based*: the producer active object is responsible to propagate the result to active objects that received the future;

On the basis of the moment in which the update is performed, two main strategies are possible:

- *eager-based*: when the result value of the future object is available, then it is immediately propagated on all the active objects that have previously received the future, including forwarder objects;
- *lazy-based*: when the result value of the future object is available, then it is propagated only on the active objects that received the future and that effectively required it.

Combining the strategies described above, four different solutions are possible: *eager forward-based*, *lazy forward-based*, *eager home-based* and *lazy home-based* strategies. The behaviors of such strategies are reported in figure 2, 3, 4 and 5.

In the forward-based strategy the forwarder manages a table (*called Waiting Active Object Table*) in which it registers the active objects that have to receive the result value of a future object. In the home-based strategy, instead, such table is managed by the producer.

As it is possible to note observing figure 3, the lazy forward-based strategy needs the transmission from objects requiring result value to the forwarder of a *registration request message.* Such message notifies to forwarder the actual need of result value of the future object. In the eager-based one, instead, such transmission is not necessary because the active object which receives the future object is automatically registered by the forwarder object at the moment of method invocation.

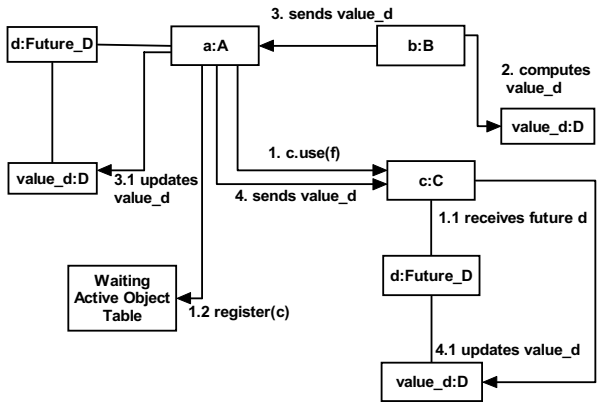

Figure 2. Eager forward-based Strategy

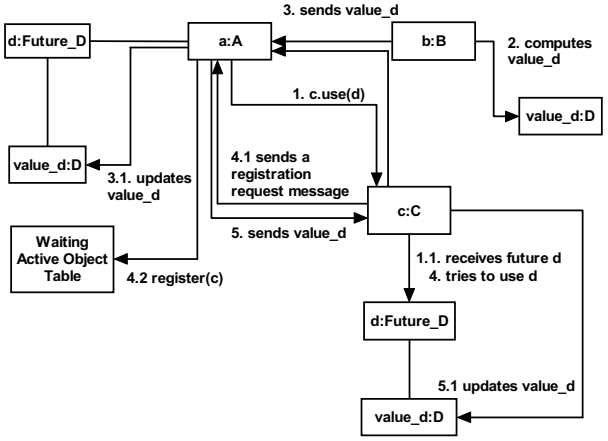

Figure 3. Lazy forward-based Strategy

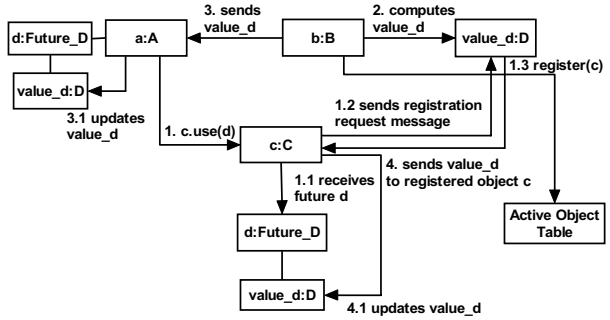

Figure 4. Eager home-based Strategy

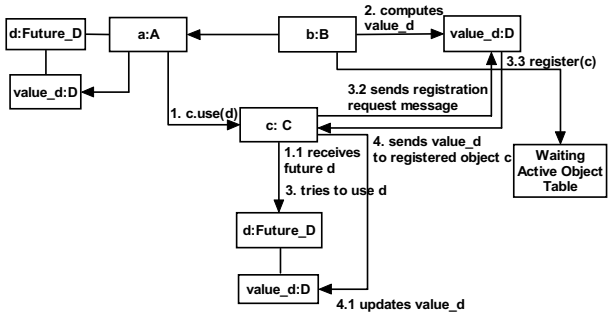

Figure 5. Lazy home-based Strategy

Analogously, the eager home-based strategy requires the transmission, from active objects receiving future object to the producer object, of the registration request message necessary to receive the result notification.

The lazy home-based strategy (see Figure 5), with respect to the eager-based one and to the forward-based strategies, in the case the forwarder object does not use the future object, avoids the transmission of the result value to it, permitting to decrease network traffic and communication overhead.

In scenarios in which data network overhead is low (such as local area networks or clusters), or size of data to update is small, or it is possible to predict that all the distributed objects that receive future objects will use the related value, the forward-based strategy permits to obtain good performance, since does not require registration, so limiting communication overhead.

On the other hand, we think that home-based strategies could be adopted in distributed programming to increase concurrency, reduce network overhead and so to increase performance. In particular, the home-based strategy could be helpful in distributed programming models in which one or more coordination entities are exploited to control the activities of multiple distributed entities synchronized on data flows and not to directly manage data. A typical example of such scenario is a workflow enactment engine, which has the task to invoke and coordinate activities and does not manipulate directly data, since it is not concerned in the result of such invocation, but only in the invocation occurring itself.

As it is possible to see in figure 6 and 7, in such scenario the home-based strategy, with respect to the forward-based one, permits to completely move the data-driven synchronization among producers and the forwarded ones, that permit a workflow enactment engine (that acts as forwarder object) to perform other tasks, simultaneously to enacted activities, such as starting other activities or code, input, and output data management tasks.

Another consideration is that the lazy-based strategy, especially when result values to update are huge and when not all the receiving objects will use the result value, could be adopted to decrease communication overhead, especially in wide-area distributed environments in which high latency and limited bandwidth of interconnection links can affect overall performance.

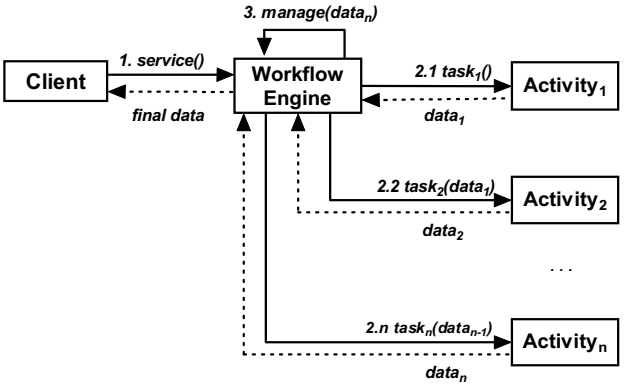

Figure 6. Forward-based Workflow Engine

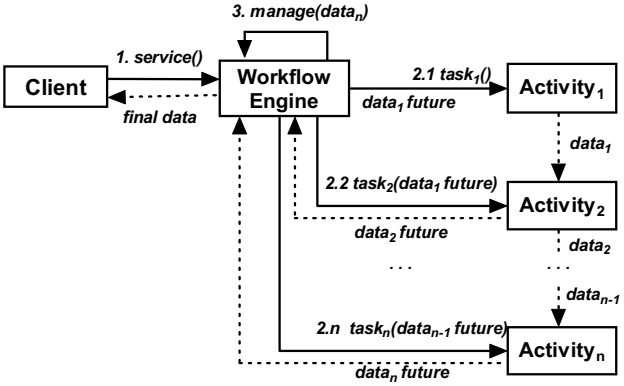

Figure 7. Home-based Workflow Engine

For example for a workflow management system, the lazy home-based strategy could permit to completely avoid the transmission of the result value to the workflow enactment engine if the result is not explicitly used.

In order to show the feasible performance improvements adopting lazy home-based strategy, in

figure 8 it is shown a simple scenario of a sort of workflow enactment engine in which a forwarder object sends as input parameter the future object returned by the remote method invocation on producer object to another active object that requires it to execute its operations.

The following code exemplifies this scenario.

```
public class Main {
....
 public static void main(String[] args){
        ......
   A a1 = newActive(A.class.getName(),…);
   A a2 = newActive(A.class.getName(),…);

   //r1 is a future
   Result r1 = a1.invoke1();
   Result r2 = a2.invoke2(r1);
   r2.display();
  }
}
public class A {
...
 public Result invoke1(){
    ...
 }

 public Result invoke2 (Result r) {
    ...
 }
}
```

In the case of forward-based strategy, the future object has to be updated on the main active object, which has the task to update the future object `r1` on active object `a2`, requiring so the transmission of two results. In the case of home-based strategy the home object (that is `a1`) has the task to directly update future object `r1`.

In the case of eager- home based strategies, the result is updated on both main object and active object `a2`, so permitting only a little performance improvement. In the case of the lazy-based strategy, the future result is updated only on active object `a2`, which actually requires such value, avoiding communication overhead among producer object and main active object, so permitting to obtain a more significant performance improvement.

4. Experimental Analysis

We implemented the eager home-based and lazy home-based strategies in ProActive and performed some experimental analyses in order to show the behaviour of such strategies with respect to the default forward eager-based strategy in some application scenario. The application scenario is represented by an active object pipe (`C1`, `C2`, `C3` and `C4`), deployed on different resources, each of which receives, by the previous active object in the chain, an asynchronous method invocation including as input parameter a future object whose result value is delivered by active object `B` (producer object).

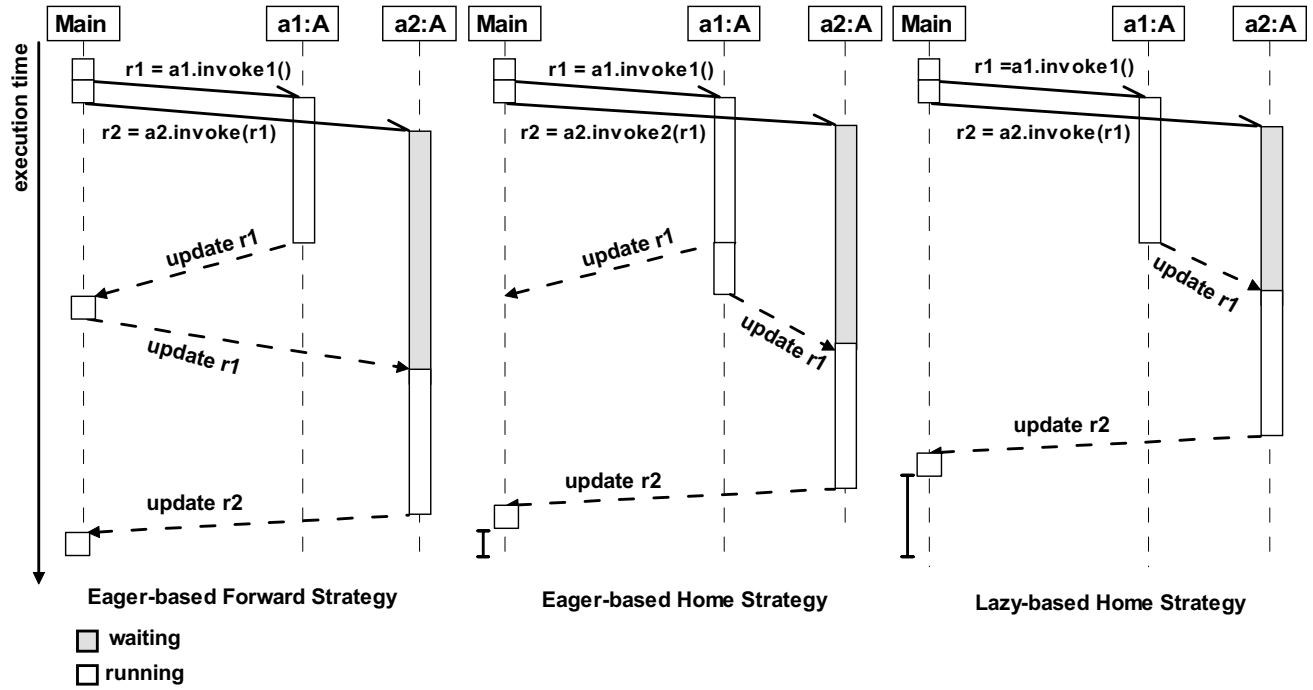

Figure 8. Comparison of different update strategies

The producer is started by the active object A, which acts as a coordinator object, (such as a workflow enactment engine) that does not use the result value of future object, but is only responsible to start the computation on the pipe by invoking the first active object, that is `C1`, passing it the future object returned by `B` (see figure 9).

We considered two cases: (1) all the active objects in the chain (`C1`, `C2`, `C3` and `C4`) actually access to the result value of the future object; (2) only the last active object, `C4`, in the chain accesses to the result value of the future object.

Figures 10 and 11 reports the update times on active object `C4` respectively in case (1) and in case (2), considering different future value sizes. Each active object is deployed on a different node of a cluster, which is equipped with an Intel Xeon Dual-Core, 2 GB of RAM, and runs Linux 2.6.9 operating system. The cluster nodes are interconnected by a Gigabit Ethernet. As figure 10 shows, in case (1), in which the result value is accessed by all the active objects, the eager home-based strategy does not permit to obtain significant improvement in update times on `C4`, while the lazy-based one permits to obtain lower update times because the active object `A` does not necessitate the result value, so avoiding the result transfer.

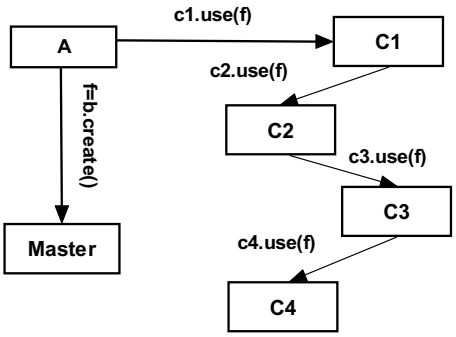

Figure 9. Benchmark scenario

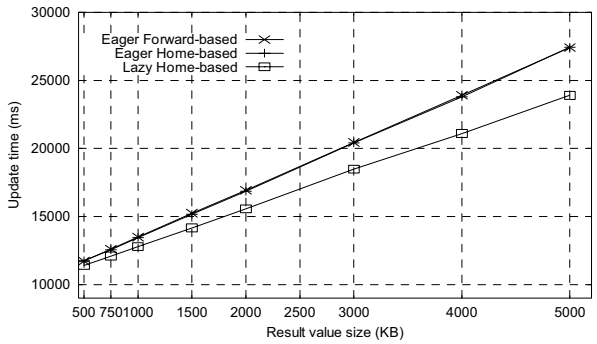

Figure 10. Case (1): all the active objects use the future object

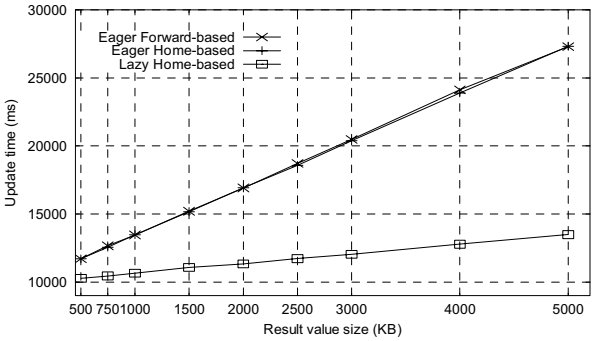

Figure 11. Case (2): only C4 uses the future object

Figure 11, finally, shows the deep decrease of update times using the lazy home-based strategy in case (2): producer has to send the result value only to active object `C4`, permitting to save a great amount of time.

An additional experimentation was conducted on the matrix multiplication application implemented through the classical master-slave pattern (see figure 12).

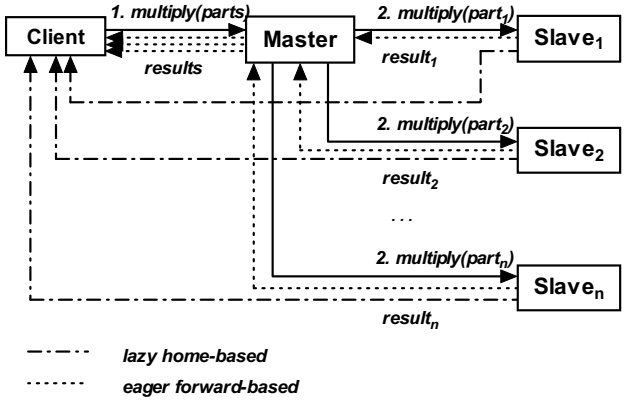

Figure 12. Matrix multiplication

The master invokes the slaves, passing to each of them a part of the left matrix. The slaves perform multiplication among the received parts and the right matrix, received during the initialization phase, and finally the client waits for the set of results.

Figure 13 shows the execution times obtained exploiting eight nodes of the cluster, varying the size matrix and adopting the default ProActive future update strategy (eager-forward based strategy) and the lazy home-based strategy. As it is possible to note, the lazy home-based strategy decreases the execution time, thanks to more light communication tasks. Using such strategy, in fact, the result computed by each slave can be directly replied to the client.

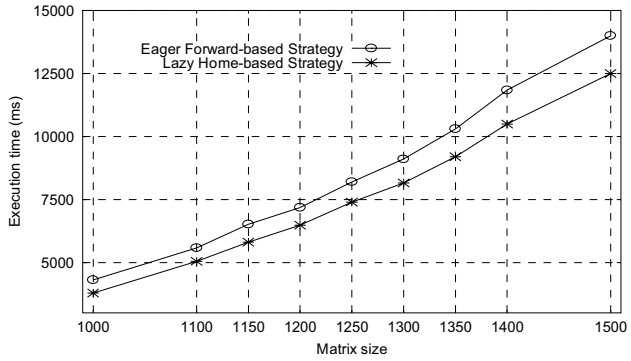

Figure 13. Experimental results

5. Conclusion

In this paper we studied different future object update strategies in the context of asynchronous communication, that are the eager and lazy forward-based strategies and the eager and lazy home-based strategies.

The implementation and experimentation conducted adopting the ProActive framework showed as the lazy home-based strategy can be exploited to improve performance thanks to the decrease update times of huge future objects when not every active object that received the future needs a strict access to the result. This is the case of distributed and parallel applications that include coordination entities, such as applications that follow the pipeline and master-slave patterns and workflow management systems in which a workflow enactment engine acts as a coordinator of various activities.

The lazy-based strategy, moreover, with respect to the eager-based one, introduces the problem of storage management and in particular of de-allocation of future objects when it is sure that no more objects will access to it. This issue can be solved introducing a mechanism of distributed memory management and a sort of distributed garbage collection for future objects. Such problem will be taken into account in a future work.

Finally, we intend to experiment with a different implementation of the eager home-based strategy through the adoption of multicast communication. In this case we expect to reduce update times by avoiding registration messages and by reducing significantly the transmission overhead, since only one message per group of a future object owners is necessary to deliver the result.

References

[1] E. F. Walker, R. Floyd, P. Neves. Asynchronous Remote Operation Execution. *In the 10th International Conference on Distributed Computing Systems*, pp. 253-259, May/June 1990.

[2] K. E. Kerry Falkner, P. Coddington, M. J. Oudshoorn. Implementing Asynchronous Remote Method Invocation in Java. *In the 6th Australian Conference on Parallel and Real-Time Systems, Springer-Verlag*, pp. 22-34, 29 November-1 December 1999.

[3] D. Kurzyniec, T. Wrzosek, V. Sunderam, A. Slominski. RMIX: A Multiprotocol RMI Framework for Java. *In 2003 International Parallel and Distributed Processing Symposium (IPDPS '03),* Nice, France, pp. 140–146, April 2003.

[4] A. Gokhale and D. C. Schmidt. Principles for Optimising CORBA Internet Inter-ORB Protocol. *In HICSS Conference*, January 1998.

[5] Kan, Web page: http://www.ittc.ku.edu/kan/.

[6] Proactive, Web page: http://www-sop.inria.fr/oasis/ProActive.

[7] D. Caromel and L. Henrio. A Theory of Distributed Objects. *Springer*, February 2005.

[8] G. Tretola and E. Zimeo. Workflow Fine-grained Concurrency with Automatic Continuation. In *2006 International Parallel and Distributed Processing Symposium (IPDPS '06)*, Rodhes, Greece, April 25-29 2006.

[9] D. Caromel, W. Klauser, J. Vayssiere. Towards Seamless Computing and Metacomputing in Java. *Concurrency & Computation: Practice & Experience*, 10(11-13), pp.1043-1061, 1998.

[10] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, H. E. Bal. Ibis: a Flexible and Efficient Java-based Grid Programming Environment. *Concurrency & Computation: Practice & Experience*, 17(7-8) pp. 1079-1107, 2005.

[11] K. Arnold, B. O'Sullivan, R. Scheifler, J. Waldo, A. Wollrath. The Jini Specification. *Addison Wesley*, December 2000.

[12] D. Caromel. Service, Asynchrony, and Wait-by-Necessity. *Journal of Object-Oriented Programming,* 2(4), pp. 12-18, 1989.

[13] The Byte Code Engineering Library, Web page: http://jakarta.apache.org/bcel.

[14] ASM Java Bytecode Manipulation Framework, Web page: http://asm.objectweb.org.