HAL
open science

First Class Futures: Specification and implementation of Update Strategies

Ludovic Henrio, Muhammad Uzair Khan, Nadia Ranaldo, Eugenio Zimeo

To cite this version:

Ludovic Henrio, Muhammad Uzair Khan, Nadia Ranaldo, Eugenio Zimeo. First Class Futures: Specification and implementation of Update Strategies. CoreGRID/ERCIM Workshop on Grids, Clouds and P2P Computing, Aug 2010, Ischia - Naples, Italy. inria-00544594

HAL Id: inria-00544594

https://inria.hal.science/inria-00544594

Submitted on 8 Dec 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

First Class Futures: Specification and implementation of Update Strategies

Ludovic Henrio[1], Muhammad Uzair Khan[1], Nadia Ranaldo[2], and Eugenio Zimeo[2]

[1] INRIA – CNRS – I3S – Université de Nice Sophia-Antipolis
{mkhan,lhenrio} @sophia.inria.fr
[2] University of Sannio
{ranaldo,zimeo} @unisannio.it

Abstract. A natural way to benefit from distribution is via asynchronous invocations to methods or services. Upon invocation, a request is enqueued at the destination side and the caller can continue its execution. But a question remains: "what if one wants to manipulate the result of an asynchronous invocation?" First-class futures provide a transparent and easy-to-program answer: a future acts as the placeholder for the result of an asynchronous invocation and can be safely transmitted between processes while its result is not needed. Synchronization occurs automatically upon an access to the result. As references to futures disseminate, a strategy is necessary to propagate the result of each request to the processes that need it. This paper studies the efficient transmission of results: it presents three strategies in a semi-formal manner, providing experimental results highlighting their benefits and drawbacks.
Key words: programming languages, futures, update strategies

1 Introduction

Futures are language constructs that improve concurrency in a natural and transparent way. A *future* is a place holder for a result of a concurrent computation [6, 17]. Once the computation is complete and a result (called *future value*) is available, the placeholder is *replaced* by the result. Access to an unresolved future results in the caller being blocked, until the result becomes available. Results are only awaited when they are really needed which helps in improving the parallelization. Futures may be created transparently or explicitly. For explicit creation, specific language constructs are necessary to create the futures and to fetch the result. *Transparent* futures, on the other hand, are managed by the underlying middleware and the program syntax remains unchanged; futures have the same type as the actual result. Some frameworks allow futures to be passed to other processes. Such futures are called *First class futures* [2]. In this case additional mechanisms to update futures are required not only at the creator, but also on all processes that receive a future. First class futures offer greater flexibility in application design and can significantly improve concurrency both in object-oriented and procedural paradigms like workflows [15, 14]. They are particularly useful in some design patterns for concurrency, such as master-worker and pipeline.

Our work focuses on various future update strategies; it can be considered as an extension of [2] and [12] through a language-independent approach that makes it applicable to various existing frameworks that support first class futures. The experiments are performed with ProActive [1], which is a middleware providing first-class futures. The main contributions of this paper are: a semi-formal event-like notation to model the future update strategies, and a description of three different update strategies using this notation (Section 3); results from experiments carried out to study the efficiency of strategies. (Section 4).

2 Related works

Futures, first introduced in Multilisp [6] and ABCL/1 [17] are used as constructs for concurrency and data flow synchronization. Frameworks that make use of explicit constructs for creating futures include Multilisp [6, 5], λ-calculus [11], SafeFuture API [16] and ABCL/f [13]. In contrast, futures are created implicitly in frameworks like ASP [2], AmbientTalk [4] and ProActive [1]. This implicit creation corresponds to asynchronous invocation. A key benefit of the implicit creation is that no distinction is made between local and remote operations in the program. Additionally, the futures can be accessed explicitly or implicitly. In case of explicit access, operations like *claim* and *touch*, etc., are used to access the future [10, 9, 13]. For implicit access, the synchronization on the future is triggered automatically by the operations manipulating the actual result value. Accessing a future that has not been updated, results in the caller being blocked.

Creol [9] allows for explicit control over data-flow synchronizations. In [3], Creol has been extended to support first class futures. In contrast to our work, future creation and manipulation in Creol is explicit. ASP [2] and ProActive [1], have transparent first-class futures and the synchronization is transparent and data-flow oriented. In AmbientTalk, futures are also first-class and transparently manipulated; but the future access is a non-blocking operation thus avoiding the possibility of a dead lock as there is no synchronization. Processes interested in the future value are registered as observers, and results are sent to registered observers when they are computed. The future update strategy in AmbientTalk is closed to the eager-message based strategy presented here. [16] provides a *safe* extension to Java futures, but with explicit creation and access.

Our previous work, [8] and [7] presented a formal semantics for GCM-like components with first class futures. We focused on proving the correctness of the component model and the future update strategies. In contrast, this paper presents the future update strategies in a wider context. We present a generalized semi-formal notation that can be used to specify future update mechanisms in a language independent manner, thus making it applicable to other frameworks like [3, 4] as well. Instead of making proofs on correctness, here we are more interested in studying the efficiency of update strategies.

3 Modeling Different Future Update Strategies

This section gives a semi-formal definition for the three main future update strategies. Strategies are called *eager* when all the references to a future are updated as soon as the future value is calculated. They are called *lazy* if futures are only updated upon need, which minimizes communications but might increase the time spent waiting for the future value. Two eager and one lazy strategies are presented here: *eager forward-based* (following the future flow), *eager message-based* (using a registration mechanism, also called home-based in [12]), and *lazy message based*. One could also consider a lazy forward-based strategy, but as it is extremely inefficient, we do not discuss it here.

3.1 General Notation

This section presents a brief overview of the various notation and entities that we use to model the future update strategies. We denote by $\mathcal{A}$ the set of processes (also called *activities*); α, β, $\ldots \in \mathcal{A}$ range over processes. $\mathcal{F}$ denotes the set of future identifiers, each future identifier is of the form $f^{\alpha\rightarrow\beta}$, which represents the future f created by the activity α, and being calculated by β. As each object needs to keep track of the futures it has received, we make use of some local lists for this purpose. There is one *future list* for each activity α. It represents the location where the futures are stored in local memory.

$$\mathcal{FL}_\alpha : \mathcal{F} \mapsto \mathcal{P}(Loc)$$

Locations, called *loc* in the following and of type *Loc*, refer to the in-memory position of the future. To keep track of activities to which a future is to be sent, a *future recipient* list is stored in each process.

$$\mathcal{FR}_\delta : \mathcal{F} \mapsto \mathcal{P}(\mathcal{A})$$

$\gamma \in \mathcal{FR}_\delta\,(f^{\alpha\rightarrow\beta})$ if the future value for $f^{\alpha\rightarrow\beta}$ has to be sent from δ to γ. It should be noted that each $f^{\alpha\rightarrow\beta}$ can be mapped to several locations in $\mathcal{FL}$ or several activities in $\mathcal{FR}$. $\mathcal{FR}$ and $\mathcal{FL}$ are initialized to empty mapping on all processes. We use an event-like notation to define the different strategies. Operations triggered by the strategies, and events triggered by the rest of the middleware are described respectively in bellow. Events are indexed by the activity on which they occur, or $\alpha \rightarrow \beta$ for a communication from α to β.

Operations

Register Future - Reg: $\mathcal{F} \times \mathcal{B} \times \mathcal{F} \mapsto \mathcal{P}(\mathcal{B})$
We define an operation *Reg* that is given a future, a process and a mapping $\mathcal{F} \mapsto \mathcal{P}(\mathcal{B})$ (either $\mathcal{FL}$ when $\mathcal{B} = Loc$, or $\mathcal{FR}$ when $\mathcal{B} = \mathcal{A}$). $Reg_\gamma(f^{\alpha\rightarrow\beta}, b, L)$ replaces the list L by the list L' defined as follows:

$$L'(f_2^{\alpha'\rightarrow\beta'}) = \begin{cases} L(f^{\alpha\rightarrow\beta}) \cup \{b\} & \text{if } f_2^{\alpha'\rightarrow\beta'} = f^{\alpha\rightarrow\beta} \\ L(f_2^{\alpha'\rightarrow\beta'}) & \text{else} \end{cases}$$

The *Reg* operation replaces the old mapping L with a new one containing the additional mapping. An example of its usage could be $Reg_\gamma(f^{\alpha\rightarrow\beta}, loc, \mathcal{FL}_\gamma)$ which adds to the $\mathcal{FL}_\gamma$ list, a new location *loc* associated to future $f^{\alpha\rightarrow\beta}$.

Locally Update future with value - Update: $Loc \times Value$
Once the value for a given future is received, this operation is triggered to update all corresponding local futures with this value. The operation $Update_\gamma(f^{\alpha\to\beta}, v)$ replaces, in the activity γ, each reference to the future $f^{\alpha\to\beta}$ by the value v. Remember the set of locations of these references is $\mathcal{FL}_\gamma(f^{\alpha\to\beta})$.

Clear future from list - Clear: $\mathcal{F} \times \mathcal{F} \mapsto \mathcal{P}(\mathcal{B})$
The clear operation $Clear(f^{\alpha\to\beta}, L)$ removes the entry for future $f^{\alpha\to\beta}$ from the list L; it replaces the list L by the list L' defined by:

$$L'(f_2^{\alpha'\to\beta'}) = \begin{cases} L(f_2^{\alpha'\to\beta'}) & \text{if } f_2^{\alpha'\to\beta'} \neq f^{\alpha\to\beta} \\ \emptyset & \text{else} \end{cases}$$

It will be used after a future update to clear entries for the updated future.

Send future value: SendValue: $\mathcal{F} \times Loc \times Value$
Send operation is used when a process needs to send the value of a computed future to another process in order to update the future there. $SendValue_{\delta\to\gamma}(f^{\alpha\to\beta}, loc, v)$ sends the value v for the future $f^{\alpha\to\beta}$ from δ to γ. Sending a future value can trigger send future reference events, $SendRef$, for all the future references contained in the value v. The details of this operation appear in Sections 3.2, 3.3, and 3.4

Events Future update strategies react to events, triggered by the application or the middleware, presented below.

Create future: Create: $\mathcal{F} \times Loc$
$Create_\alpha(f^{\alpha\to\beta}, loc)$ is triggered when α creates a future that will be calculated by the process β. The semantics of this event is similar for all strategies: it registers the future in the future list $\mathcal{FL}$ of the creating process.

$$Create_\alpha(f^{\alpha\to\beta}, loc) \triangleq Reg_\alpha(f^{\alpha\to\beta}, loc, \mathcal{FL}_\alpha)$$

Send future reference: SendRef: $\mathcal{F} \times Loc$
$SendRef_{\delta\to\gamma}(f^{\alpha\to\beta}, loc)$ occurs when the process δ sends the future reference $f^{\alpha\to\beta}$ to γ and the future is stored at the location loc on the receiver side. The details of this operation will be described in Sections 3.2, 3.3, and 3.4.

Future computed: FutureComputed: $\mathcal{F} \times Value$
$FutureComputed_\beta(f^{\alpha\to\beta}, val)$ occurs when the value val of future $f^{\alpha\to\beta}$ has been computed by β.

Wait-by-necessity: Wait: $\mathcal{A}$
This event is triggered when a process accesses an unresolved future. This corresponds to *get* or *touch* operation in [10, 9, 13]. For the two eager strategies it simply causes the process to be blocked until the value is received. For the lazy strategy, this event retrieves the future value, see Section 3.4.

3.2 Eager Forward-Based Strategy

In this strategy, each process remembers the nodes to which it has forwarded the future. When the value is available, it is sent to all such nodes. The list of processes to which a process β should send the future value for $f^{\alpha\rightarrow\beta}$ is $\mathcal{FR}_\beta(f^{\alpha\rightarrow\beta})$. It is the list of processes to which β has sent the future reference.

Figure 1 shows an example illustrating this strategy. Process A makes an asynchronous call on process H and receives the future $f^{A\rightarrow H}$. A then passes this future to B, which in turn passes the future to C, D and E. Finally C passes the future to F. Each time a future is forwarded, i.e. upon a *SendRef* message, the forwarding process δ adds the destination to its $\mathcal{FR}_\delta(f^{A\rightarrow H})$. When the result for $f^{A\rightarrow H}$ is available, it is communicated to A using *SendValue* message. A then forwards the update on B ($\mathcal{FR}_A(f^{A\rightarrow H}) = \{B\}$). B can make concurrent updates on C, E and D ($\mathcal{FR}_B(f^{A\rightarrow H}) = \{C, E, D\}$). Finally, the occurrence in F is updated by C ($\mathcal{FR}_F(f^{A\rightarrow H}) = \{C\}$).

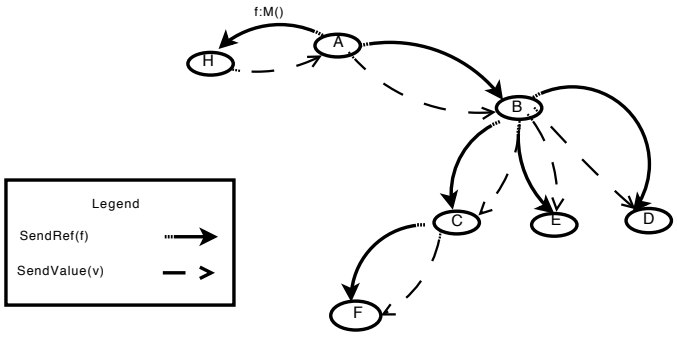

Fig. 1. Future-update in eager forward-based strategy

Send Future Reference When a process δ sends a future $f^{\alpha\rightarrow\beta}$ to a process γ, the sender registers the destination process in $\mathcal{FR}_\delta$, and the destination process registers the location of the future in $\mathcal{FL}_\gamma$.

$$SendRef_{\delta\rightarrow\gamma}(f^{\alpha\rightarrow\beta}, loc) \triangleq Reg_\delta(f^{\alpha\rightarrow\beta}, \gamma, FR_\delta); \quad Reg_\gamma(f^{\alpha\rightarrow\beta}, loc, \mathcal{FL}_\gamma)$$

Future Computed Once the value of a future $f^{\alpha\rightarrow\beta}$ has been computed at process β, it is immediately sent to all the processes that belong to $\mathcal{FR}_\beta(f^{\alpha\rightarrow\beta})$. This will trigger chains of *SendValue* operations. Once the future value have been sent, the future recipient list is no longer useful:

$$FutureComputed_\beta(f^{\alpha\rightarrow\beta}, value) \triangleq \forall\, \delta \in \mathcal{FR}_\beta(f^{\alpha\rightarrow\beta}),\ SendValue_{\beta\rightarrow\delta}(f^{\alpha\rightarrow\beta}, value)$$
$$Clear_\beta(f^{\alpha\rightarrow\beta}, \mathcal{FR}_\beta))$$

Send Future Value When a future value is received, the receiver first updates all the local references, and then sends the future value to all the processes to which it had forwarded the future (the processes in its $\mathcal{FR}$ list). The operation is recursive, because the destination process of *SendValue* may also need to update

further futures. This operation can potentially trigger the $SendRef$ operation in case of nested futures. The future locations and future recipient lists for this future are not needed anymore after those steps:

$$\begin{aligned} SendValue_{\delta\to\epsilon}(f^{\alpha\to\beta}, value) \triangleq\ & \forall\, loc \in \mathcal{FL}_\epsilon(f^{\alpha\to\beta}),\ \ Update_\epsilon(loc, value), \\ & Clear_\epsilon(f^{\alpha\to\beta}, \mathcal{FL}_\epsilon) \\ & \forall\, \gamma \in \mathcal{FR}_\epsilon(f^{\alpha\to\beta}),\ \ SendValue_{\epsilon\to\gamma}(f^{\alpha\to\beta}, value), \\ & Clear_\epsilon(f^{\alpha\to\beta}, \mathcal{FR}_\epsilon) \end{aligned}$$

3.3 Eager Message-based Strategy

In eager message-based strategy, the process β, computing the future value, is responsible for updating all nodes which receive a future. Opposed to forward-based strategy where futures updates are performed in a distributed manner, here all updates are performed by same process β (home) in a centralized manner. Whenever, a process δ forwards a future to another process γ, it sends a message *SendRegReq* to the home process β, and updates the list of future recipients $\mathcal{FR}_\beta$. $\mathcal{FR}_\beta(f^{\alpha\to\beta})$ contains the set of processes to which $f^{\alpha\to\beta}$ has been forwarded.

Figure 2 shows an example of this strategy. When A forwards the future to process B a registration message *SendRegReq* is sent from A to H, registering B in $\mathcal{FR}_H$. Similarly we have a registration message sent to H from B adding C, E, and D to $\mathcal{FR}_H$; finally we have $\mathcal{FR}_H(f^{A\to H}) = \{A, B, C, D, E, F\}$.

Once the future result is available, H uses the *SendValue* message to communicate the value to all processes in $\mathcal{FR}_H(f^{A\to H})$.

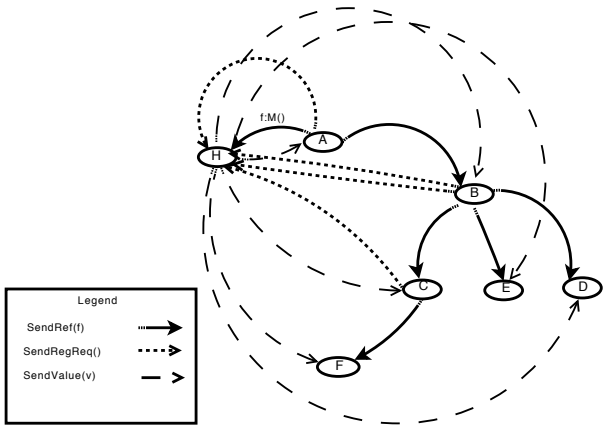

Fig. 2. Future-update in eager message-based strategy

Send Future Reference In the message-based strategy when a future $f^{\alpha\to\beta}$ is forwarded by a process δ to a process γ, a registration message is sent to the process that will compute the future, β.

$$SendRef_{\delta\to\gamma}(f^{\alpha\to\beta}, \gamma, loc) \triangleq Reg_\beta(f^{\alpha\to\beta}, \gamma, \mathcal{FR}_\beta);\quad Reg_\gamma(f^{\alpha\to\beta}, loc, \mathcal{FL}_\gamma)$$

The registration $Reg_\beta(f^{\alpha\to\beta}, \gamma, \mathcal{FR}_\beta)$ is performed using a communication addressed to the home process β, and is called *SendRegReq* in Figure 2.

Future Computed Once the execution is completed and the value is available in β, the process β sends the value to all the processes in $\mathcal{FR}_\beta(f^{\alpha\to\beta})$.

$$\begin{aligned} \mathit{FutureComputed}_\beta(f^{\alpha\to\beta}, \mathit{val}) \triangleq\ & \forall\, \delta \in \mathcal{FR}_\beta(f^{\alpha\to\beta})\ \mathit{SendValue}_{\beta\to\delta}(f^{\alpha\to\beta}, \mathit{val}); \\ & \mathit{Clear}_\beta(f^{\alpha\to\beta}, \mathcal{FR}_\beta) \end{aligned}$$

Send Future Value Contrarily to forward-based strategy, there is no need to forward the future value when received, only local references are updated, and then the $\mathcal{FL}$ list can be cleared.

$$\begin{aligned} \mathit{SendValue}_{\beta\to\gamma}(f^{\alpha\to\beta}, \mathit{val}) \triangleq\ & \forall\, \mathit{loc} \in \mathcal{FL}_\gamma(f^{\alpha\to\beta})\ \mathit{Update}_\gamma(\mathit{loc}, \mathit{val}); \\ & \mathit{Clear}_\gamma(f^{\alpha\to\beta}, \mathcal{FL}_\gamma) \end{aligned}$$

The received future value may contain other futures as well. In this case, it can potentially trigger the send future reference operation.

3.4 Lazy Message-based Strategy

The lazy strategy differs from the eager strategies in the sense that future values are only transmitted when absolutely required. When a process accesses a unresolved future, the access triggers the update. This strategy is somewhat similar to message-based strategy except the futures are updated only when and if necessary. In addition, each process now needs to store all the future values that it has computed. For this, we introduce another list, $\mathcal{FV}$ that stores these values: $\mathcal{FV}\colon \mathcal{F} \mapsto \mathcal{P}(Value)$. $\mathcal{FV}_\beta(f^{\alpha\to\beta})$, if defined, contains a singleton, which is the future value of $f^{\alpha\to\beta}$.

Compared to Figure 2, in the lazy strategy only the processes that require the future value register in $\mathcal{FR}_H$, $\mathcal{FR}_H(f^{A\to H}) = \{C, D\}$ if only C and D access the future. When the result is available, H communicates it to processes in $\mathcal{FR}_H(f^{A\to H})$. In addition, the value is stored in $\mathcal{FV}_H(f^{A\to H})$. If the future value is required later, it will be retrieved from $\mathcal{FV}_H(f^{A\to H})$.

Send future reference This strategy does not require registration with home process when forwarding a future. Incoming futures are registered in FL_γ on the receiver. Once the value is received, all local references can be updated.

$$\mathit{SendRef}_{\delta\to\gamma}(f^{\alpha\to\beta}, \gamma, \mathit{loc}) \triangleq \mathit{Reg}_\gamma(f^{\alpha\to\beta}, \mathit{loc}, \mathcal{FL}_\gamma)$$

Wait-by necessity Wait-by-necessity is triggered when the process tries to access the value of the future. We register the waiting process at β:

$$\mathit{Wait\text{-}by\text{-}necessity}_\gamma(f^{\alpha\to\beta}) \triangleq \mathit{SendRegReq}_{\gamma\to\beta}(f^{\alpha\to\beta}, \gamma)$$

If the future has already been computed by β, the value is transmitted immediately. Otherwise, the request is added to the Future receivers list of β.

$$\mathit{SendRegReq}_{\gamma\to\beta}(f^{\alpha\to\beta}, \gamma) \triangleq \begin{cases} \mathit{SendValue}_{\beta\to\gamma}(f^{\alpha\to\beta}, \mathit{val}) & \text{if } \mathcal{FV}_\beta(f^{\alpha\to\beta}) = \{\mathit{val}\} \\ \mathit{Reg}_\beta(f^{\alpha\to\beta}, \gamma, \mathcal{FR}_\beta) & \text{if } f^{\alpha\to\beta} \notin \mathit{dom}(\mathcal{FV}_\beta) \end{cases}$$

Future Computed When a result is computed, the value is stored in the future value list. Moreover, if there are pending requests for the value, then the value is sent to all the awaiting processes.

$$FutureComputed_{\beta}(f^{\alpha\to\beta}, val) \triangleq \forall\, \delta \in \mathcal{FR}_{\beta}(f^{\alpha\to\beta}) SendValue_{\beta\to\delta}(f^{\alpha\to\beta}, val) \\ Clear_{\beta}(f^{\alpha\to\beta}, \mathcal{FR}_{\beta});\quad Reg_{\beta}(f^{\alpha\to\beta}, val, \mathcal{FV}_{\beta})$$

Send Future Value The $SendValue$ operation is the same as for the eager message-based strategy:

$$SendValue_{\beta\to\gamma}(f^{\alpha\to\beta}, val) \triangleq \forall\, loc \in \mathcal{FL}_{\gamma}(f^{\alpha\to\beta})\ Update_{\gamma}(loc, val); \\ Clear_{\gamma}(f^{\alpha\to\beta}, \mathcal{FL}_{\gamma})$$

4 Experimental Evaluation

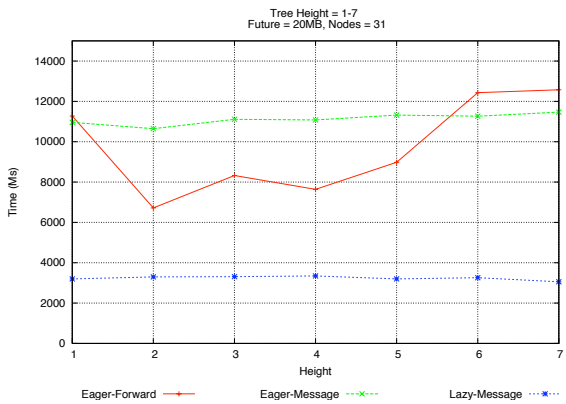

Fig. 3. Comparison of strategies for a tree configuration

We conducted an experimentation with a real system in order to test the efficiency of the various strategies. To this end, we adopted ProActive version 3.9. ProActive is based on the notion of active objects, abstracting processes with a unique thread and message queue. We used a cluster of 11 nodes equipped with Intel(R) Xeon(TM) CPUs at 2.80GHz with 1 GB RAM running Linux kernel 2.6.9. The cluster nodes are connected via a Gigabit Ethernet link. To measure the various parameters of interest, we deployed an application featuring a tree topology where each node is an active object. For the scope of the analysis, we kept the number of nodes accessing future value constant. In addition, only the leaf nodes of the tree make use of future values. The graph in Figure 3 compares the time needed to update futures for the evaluated strategies. Experiments are realized over trees of varying heights. Lazy strategy takes less time to update the futures since much less updates have to be made than for the two eager strategies. The experience shows that update time required for lazy and eager message-based strategies is roughly independent of the height of the tree. Eager-forward based strategy can take advantage of concurrent updates. On the other

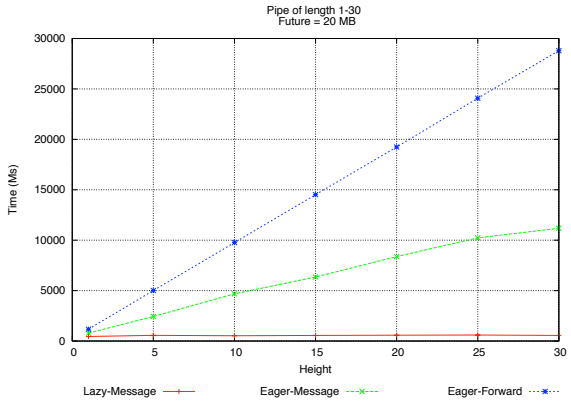

Fig. 4. Comparison of strategies for a pipe configuration

hand, it also gets more time to reach the bottom of high trees as shown by the shape of the graph. As the height of the tree increases, overheads increases due to time spent at intermediate nodes. As a result, at height 7, the time needed for updates is higher.

Figure 4 shows the time necessary to update a future along a simple chain of processes. Time taken by the lazy strategy is again constant and is very small because only one update is made (for the leaf node). It can be easily observed from the graph that forward-based and message-based strategies scale in a linear manner. There is no parallelization of the updates, neither for the forward-based strategy, nor for the message-based (as it is implemented in a single threaded manner). Future updates in eager forward-based strategy go through a number of intermediate steps before arriving at the last node in the chain. This introduces and additional delay for forward-based strategy. In message-based strategies, all updates are performed by same node in single step. Thus the update time is relatively constant.

5 Conclusion

This paper presented a semi-formal description of the three main strategies for updating first class futures. We build upon the work presented in [2, 12] to model and evaluate each of the strategies with experimental results. Our main contributions are:

Semi-formal event-like notation. We present and use a general (language independent) notation for modeling future update strategies. Consequently, other frameworks involving first class futures may benefit from our work.

Experimental results. We implemented the different strategies in the ProActive middleware to study the efficiency of various strategies.

We hope this article will help answering to the non-trivial question: "Which is the best future update strategy"? There is no single *best* strategy, rather the strategy should be adopted based on the application requirements, to summarize:

Eager forward-based strategy: it is more suitable for scenarios where the number of intermediate nodes is relatively small and the future value is not too big. Also, the distributed nature of future updates results in less overloading at any specific node.
Eager message-based strategy: it is more adapted for process chains since it ensures that all updates are made in relatively constant time. Due to its centralized nature, it may require more bandwidth and resources at the process that computes the future.
Lazy strategy: it is better suited for cases where the number of processes that require future value is significantly less than total number of processes. Considerable savings in network load can be achieved but this has to be balanced against the additional delay inherent in the design of lazy approach. Also, all computed results have to be stored which requires more memory resources.

With an understanding of the various strategies, a good next contribution could be to study hybrid strategies to improve performance.

References

[1] D. Caromel, C. Delbé, A. di Costanzo, and M. Leyton. ProActive: an integrated platform for programming and running applications on grids and P2P systems. *Computational Methods in Science and Technology*, 12(1):69–77, 2006.
[2] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Object*. Springer-Verlag, 2005.
[3] Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *ESOP*, pages 316–330, 2007.
[4] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-oriented programming in ambienttalk. In *ECOOP 2006*, pages 230–254, 2006.
[5] Cormac Flanagan and Matthias Felleisen. The semantics of future and an application. *Journal of Functional Programming*, 9(1):1–31, 1999.
[6] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4), 1985.
[7] Ludovic Henrio, Florian Kammüller, and Muhammad Uzair Khan. A framework for reasoning on component composition. In *FMCO 2009*. Springer, 2010.
[8] Ludovic Henrio and Muhammad Uzair Khan. Asynchronous components with futures: Semantics and proofs in isabelle/hol. In *Proceedings of the Seventh International Workshop, FESCA 2010*. ENTCS, 2010.
[9] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. In *SEFM '04: Proceedings of the Software Engineering and Formal Methods*, 2004.
[10] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: a type-safe object-oriented model for distributed concurrent systems. *Theor. Comput. Sci.*, 365, 2006.
[11] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364, November 2006.
[12] Nadia Ranaldo and Eugenio Zimeo. Analysis of different future objects update strategies in proactive. In *IPDPS 2007: Parallel and Distributed Processing Symposium, IEEE International*, pages 23–66, 2007.

[13] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. Abcl/f: A future-based polymorphic typed concurrent object-oriented language - its design and implementation. In *DIMACS '94*, volume 18, 1994.

[14] G. Tretola and E. Zimeo. Extending semantics of web services to support asynchronous invocation and continuation. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*, pages 208–215, 2007.

[15] G. Tretola and E. Zimeo. Activity pre-scheduling for run-time optimisation of grid workflows. *Journal of Systems Architecture*, 54(9), 2008.

[16] Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for java. *SIGPLAN Not.*, 40(10):439–453, 2005.

[17] Akinori Yonezawa, Etsuya Shibayama, Toshihiro Takada, and Yasuaki Honda. Modelling and programming in an object-oriented concurrent language ABCL/1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*. MIT Press, 1987.