Update Strategies for First-class Futures

Ludovic Henrio Muhammad Khan
INRIA, Univ. Nice Sophia Antipolis, CNRS
{lhenrio,mkhan}@sophia.inria.fr

I. Introduction

In the context of distributed programming, several different notions have been defined to make parallel or concurrent programming more efficient and more intuitive. Futures represent an example of such notions to improve concurrency in a natural and transparent way. A *future* is a temporary object that is used as a place holder for a result of a concurrent computation [5], [7]. Once the computation is complete and a value for the result (called *future value*) is available, the future is *updated* with the computed value. Access to an unresolved future is a blocking operation. As results are only awaited when they are really needed, computation is parallelised in a somehow optimal way. The future creation can be *transparent* or *explicit*. With explicit futures, specific language constructs are necessary to create the futures and to convert them into normal objects (fetching the future value). *Transparent* futures, on the other hand, are managed by the underlying middleware and the program syntax remains unchanged, since futures have the same type as the actual result. Some frameworks allow futures to be passed to other (remote) processes. Such futures are called *First class futures* [2]. In this case additional mechanisms to update futures are required not only on the creating node, but also on all nodes 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. They are particularly useful in some design patterns for concurrency, such as master-worker and pipeline.

II. Contribution

Our work analyses several strategies that may be used to update first class futures; it can be considered as an extension of the works presented in [2] and [6] through a language-independent approach that makes it applicable to various existing frameworks that support first class futures.

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

Cost analysis of the strategies. For better understanding of the strategies and the relative costs (in terms of number of messages and time) involved, we developed a simplified cost analysis of the protocols. This helps in understanding which strategy is more suitable for a given application.

Experimental results. We implemented the different strategies in the ProActive middleware and experimentally verified the results of our analysis.

Impact and Related Works This work, is a study of future update strategies. We present the strategies in greater detail compared to previous work [2], and analyse the costs associated with each strategy. We present our work in a language independent manner and as such it can be applied to various existing frameworks that support first class futures, like [3], [4]. This cost analysis is evaluated by experiments using the ProActive library [1].

III. Future update Strategies

Future updates strategies can be classified as either *Eager* or *Lazy*. 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 minimises 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 [6]), 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.

Eager Forward-based Strategy. In this strategy, each process remembers only the nodes to which it has sent the future, and forward them the value when available. Therefore, the flow of future updates follow the same path as the futures themselves. The updates, are therefore performed in a distributed manner.

Eager Message-based Strategy.Future values are sent to all future recipients as soon as future is computed. Opposed to forward-based strategy where futures updates are performed in a distributed manner, future updates in message-based strategy are centralised. All updates are performed by the process responsible for computing the future value.

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 future the synchronisation on the future access triggers the future update. This strategy is somewhat similar to message-based strategy except the fact that futures are updated only when and if necessary.

IV. Comparison of Different Strategies

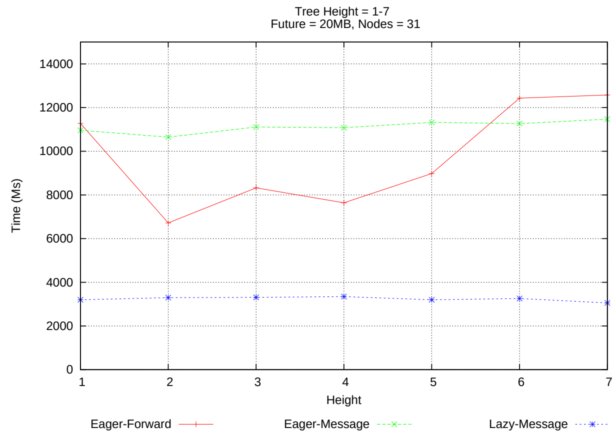

Fig. 1. Comparison of strategies for a tree configuration

The graph in Figure 1 compares the time needed to update futures for the evaluated strategies. Experiments are realized over trees of varying heights. As can be seen from Fig 1 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 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 much higher. Note that for height 1, both eager strategies perform in a similar way because in that case both algorithms are roughly identical.

We try here to answer 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 summarise:

- *Eager forward-based strategy* 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* is more adapted for process chains since it ensures that all updates are made in relatively constant time. Due to its centralised nature, it may require more bandwidth and resources at the process that computes the future.
- *Lazy strategy* is better suited for applications 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.

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 Dave Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 230–254. Springer, 2006.

[5] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.

[6] 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.

[7] 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.