PROCEEDINGS OF THE WORKSHOP ON APPLICATIONS OF SOFTWARE AGENTS
ISBN 978-86-7031-188-6, pp. 32 - 36, 2011

Investigating F# as a development tool for distributed multi-agent systems[⋆]

Extended abstract

Alex Muscar

Software Engineering Department, University of Craiova, Bvd. Decebal 107, Craiova, 200440, Romania, `amuscar@software.ucv.ro`

1 Introduction

Recently we have set up a research project to investigate the development of software systems based on the metaphor of distributed multi-agent organizations. In what follows we focus on the design and implementation of distributed algorithms using two programming frameworks that we consider appropriate: JADE[1] and F#[2]. For experimental evaluation we have considered the distributed version of the Depth-First Traversal (DDFT) algorithm presented in [3].

We adopt a very weak notion of “agency” by understanding an agent as a computational entity that (i) has its own thread of control and can decide autonomously if and when to perform a given action; (ii) communicates with other agents by asynchronous message passing. We consider asynchronous programming as being characterized by many simultaneously pending reactions to internal or external events [5].

The rest of the paper is structured as follows. In section 2 we outline the DDFT algorithm that we considered in our experimental implementations using both F# in section 3 and JADE in section 4. We proceed with an experimental evaluation and comparison of those two implementations in section 5. Finally we draw some conclusions in section 6.

2 Background and Problem formulation

The DDFT algorithm operates over a graph in which vertices are entities and edges are communication links between entities. Each entity has a set of neighbors which does not necessarily coincide with the set of all vertices. In the context of multi-agent systems this can be seen as a simplified form of organization. The key point of DDFT is to visit the graph by forwarding a *token* for as long as possible and *backtracking* when forwarding is not possible. In this scenario there is exactly one *initiator*. The other entities will be *idle* waiting for the token. When visited, an entity forwards the message

⋆ This work was supported by the strategic grant POSDRU/CPP107/DMI1.5/S/78421, Project ID 78421 (2010), co-financed by the European Social Fund Investing in People, within the Sectoral Operational Programme Human Resources Development 2007 - 2013.

1 `http://jade.tilab.com/`

2 urlhttp://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/spec.html

to each of its neighbors in turn and when it runs out of neighbors it sends the token back to the sender. If the entity has already been visited it will just reply signaling that the link is a *back-edge* and it will not proceed to forwarding the token. For a more detailed description ad the pseudocode of the algorithm we refer the interested reader to [3].

We can identify four states of the DDFT algorithm: *Initiator*, *Idle*, *Visited* and *Done*. As stated above only one entity will start in the *Initiator* state, whereas the rest will have the state *Idle*. When visited, an entity enters the state *Visited*. After the algorithm has ran, all the entities will be in the *Done* state.

3 The F# approach

F#'s rich type system provides for both expressive and safe programming [4]. More information can be encoded at the type level and it can be statically checked by the compiler. For instance, by using *discriminated unions*[3] we can defined message types that agents can exchange in a succinct and type-safe manner. This is achieved in conjunction with *pattern matching*, which can be used to decompose complex structures.

Programs that implement message passing software agents in F# follow a certain pattern: sets of recursive functions, each defining an asynchronous computation, are used to model the states of a state machine out of which only one is identified as the initial state [4]. This idiom can be used to write any agent who can be modeled as a state machine. Note that the state transitions are implicitly coded by allowing the functions that implement each state to call each other.

3.1 Asynchrony

F# has built in support for asynchronous programming via blocks of the form *async* {...} which in F# parlance are called *asynchronous computation* [5]. Inside *async* blocks the some control constructs (e.g. *let*!, *return*!) of the language take a special meaning which helps in composing asynchronous computations in a manner that's close to the core language. When executed, an asynchronous computation will eventually produce a value.

4 The JADE approach

JADE is a very flexible agent platform that provides the basic functionalities for the development of distributed multi-agent systems. A JADE-based system is composed of a set of software agents. Each software agent has its own thread of control. The actual program performed by a JADE agent is defined as a set of plugin components known as *behaviors*. A behavior consists of a sequence of primitive actions. Behaviors are executed in parallel using interleaving of actions on the agent's thread with the help of a non-preemptive scheduler, internal to the agent [1].

Our JADE implementation of the DDFT algorithm is fairly straight forward. We have extended the generic *Behaviour* class to represent our agent's behavior. State transitions

[3] `http://msdn.microsoft.com/en-us/library/dd233226.aspx`

are implemented by updating the value of the an instance variable in a *switch* statement. As in the case of F# we have omitted the implementation of the JADE agent for brevity.

Whenever an agent must receive a message, we use the following JADE idiom: (i) invoke the *receive*() method, (ii) test if there is a message in the mailbox, (iii) do the work if an appropriate message was received, otherwise invoke *block*(). The invocation of *block*() is necessary because the *receive*() method is non-blocking, i.e. it immediately returns either a message or *null* if the mailbox is empty.

5 Results and discussion

So far we have seen two approaches for implementing the same algorithm: (i) one based on a functional programming language and (ii) the other using the more familiar imperative paradigm combined with a middleware framework. We are now going to take a closer look at the implications of our choice in each case.

5.1 Safety

In F# we use statically typed messages and we constrain our mailboxes to operate only on those message types that are relevant to our agent. By doing so we ensure that we won't send messages that we can't handle or that our agent can't understand. This method is also applicable when using serialization since the .NET runtime embeds type information in the serialized entities so they can safely be re-constructed at the destination.

In JADE there are three possibilities: (i) plain strings, the least safe method since strings cannot be checked to be of an appropriate type; (ii) serialization which is a convenient way of packing messages that are sent between agents written in Java, and (iii) ontologies which set up a vocabulary and a nomenclature of elements that can be used as message contents.

5.2 Usability

The structure of the programs derived using both approaches follows the algorithm specification closely, but the F# solution (including the code to generate a tree shaped hierarchy of agents) is nearly half the size of the JADE solution. The agent logic itself spans over 30 lines of F# code whereas in JADE it spans over 70 lines of code. Note that this is the case in which we used strings for messages in JADE. If we were to take the solution using serialization or the one using ontologies the amount of code would grow further.

Another advantage we get from using F# is that we don't have to explicitly manage the state of the agent ourselves. Instead of keeping track of it in a variable, like we do in Java, we let the runtime take care of that. We just call functions corresponding to states whenever we want to transition to a new state. This makes the code more declarative, allowing the programmer to state *what* we want done instead of *how* we want to do it.

5.3 The threading model

While the code structure of the solutions is quite similar, following closely the algorithm, the underlying systems behave quite differently. Whereas JADE uses one thread per agent [1], F#'s runtime uses the thread pool approach [5].

F#'s approach of using lightweight tasks which are scheduled for execution in a thread pool leads to a more scalable solution–as long as asynchronous operations don't block the executing thread [5]. In this scenario a (rather small) number of OS threads run tasks that are scheduled in the thread pool. Thus a very large number of tasks can be ran without paying the price of spawning a thread per task.

JADE does offer a *block*() method that marks a behavior as being blocked, but it has a different purpose. While blocked a behavior does not poll for messages but the agent continues to take up a thread. When a message is delivered all blocked behaviors will be marked as active again.

5.4 Performance

While DDFT is a good example for getting a feel of developing distributed systems using both solutions and the means they offer for this task, in order to evaluate the performance of both platforms we implemented an example inspired from the real world: Google's *mapreduce* [2]. In mapreduce a machine–the *master*–splits the work in chunks and assigns each chunk to a *mapper*–more than one chunk can be assigned to a mapper if there are more chunks than mappers. When done, the intermediary results generated by mappers are forwarded to one or more *reducers* for final processing.

Using the mapreduce model we implemented an application that counts the number of occurrences of each word in a collection of documents–4 documents, 11.8 Mb each. In our setup we had a master, a reducer and between 1 and 4 mappers, each running on its own machine. In order to simplify the implementation each mapper had the 4 documents available locally.

We ran the test on network of computers with Core 2 Duo processors running at 2.2 GHz each with 4 GBs of RAM. The computers are connected by a Myrinet-2000 network[4]. The F# benchmark was run on .NET 4.0 and the Java one on Java 1.6.0_22. Figure 1 shows a comparison of the total time for each solution with a varying number of mappers–from 1 to 4.

6 Conclusions and Future Work

In light of our experiment we think F# is a viable option for writing distributed algorithms. F#'s asynchronous computation model proved to be a scalable option. This is confirmed by the results shown in section 5.4 where the F# solution performed consistently better than the JADE implementation. Also the fact that F# doesn't use one thread per agent means that a larger number can be spawned thus allowing for finer grained solutions to be implemented. Also our F# implementation of the DDFT algorithm is half the size of the JADE version.

[4] `http://www.myri.com/myrinet/overview/`

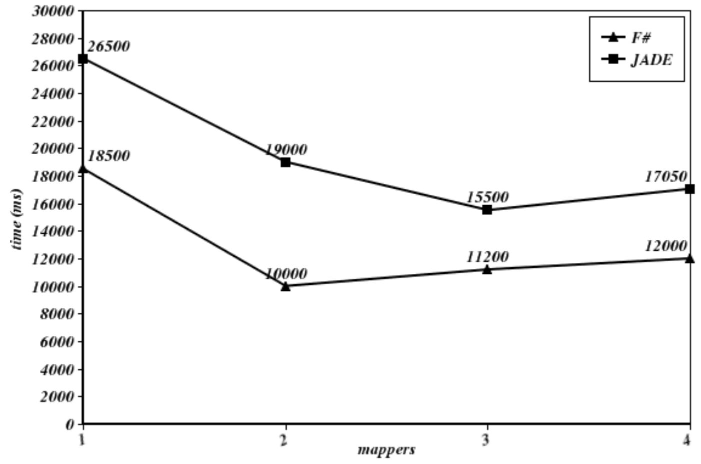

Fig. 1. mapreduce performance for F# and JADE.

Based on our initial results we would like to use F# to investigate computational properties of distributed multi-agent organizations, with a focus on advanced properties like self-configuration and adaptivity. One line of research is to see if F#'s advantages outlined in this paper will scale-up to significantly more complex systems.

References

1. Bellifemine, F., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. Wiley (2007)
2. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM 51, 107–113 (January 2008)
3. Santoro, N.: Design and analysis of distributed algorithms. Wiley (2007)
4. Syme, D., Granicz, A., Cisternino, A.: Expert F# 2.0. Apress Berkley (2010)
5. Syme, D., Petricek, T., Lomov, D.: The F# asynchronous programming model (2011)