Combining Mobile Agents with Persistent Systems: Opportunities and Challenges

Miguel Mira da Silva and Malcolm Atkinson

mms@dmat.uevora.pt, Dept of Computing Science, University of Glasgow

In the last three years we have been working with persistence and distribution, in particular migration of higher-level objects (such as procedures) between autonomous persistent programs. In this paper we introduce persistence and the suitability of Napier88, the persistent system we have used for our experiments, as an agent language. We then present a few examples of opportunities and many more challenges that exist in the combination of persistence with agents.

1 Introduction

The main motivation for this paper is our perception that free variables and persistence have not been properly addressed by the agent research community. Persistence is important because many of the examples presented in the agent literature include local or remote database access. Free variables are also of interest, and unavoidable in a persistent environment, because data and code are highly and strongly inter-connected in the persistent store.

This paper is based on our experience with persistent systems and higher-level distributed computation, two research areas that combined address many of the same issues as that of mobile agents. One of our goals is to offer the application programmer a simple agent system with: general and flexible support for free variables; migration of code and data in an integrated manner between autonomous programs; and provision for persistent agents to better access local and remote databases.

Recently we have started to address mobile agents specifically in our research work, a natural consequence of trying to support higher-order objects as arguments to remote procedures. In the process we have encountered some *opportunities* for combining persistence and agents. However, we have also encountered many difficulties and problems.

2 Persistent Systems

The persistence of an object is the period of time for which the object exists and is usable [1][5]. A persistent programming language ensures that objects remain available as long as they are required for computation, thus eliminating the need for files or databases. A conventional—as opposed to persistent—programming language manipulates only objects that are resident in memory. If these objects are to be used later by the same or another program, then the data has to be explicitly transfered from (volatile) program data structures to some sort of permanent (non-volatile) storage, generally utilising a different representation.

A persistent language avoids the need for the programmer to worry about these two worlds and to write code to transfer data between them. All code and all data are available for computation as long as they are required. This simplicity permits to use more much complex data structures because they do not have to be flattened to a file or relational database.

Napier88 [66][47] is a research persistent programming language in which persistence of an object is defined by reachability from a single persistent root. Objects that cannot be reached are unusable and thus can be garbage collected. Napier88 also incorporates a number of other interesting features, such as: first-class procedures; dynamic binding; linguistic reflection; threads and concurrency control.

3 Opportunities

In this section we give examples of opportunities in the combination of persistent systems with mobile agents.

Mobility as persistence

The basic idea behind persistence is to exchange objects automatically between the store and the program as needed by the application. Many researchers have noted that mobility and persistence are intrinsically related because mobility is conceptually similar to persistence if the store is replaced by a program [25][9][11][22][20][57][67][75][78].

Higher-order RPC

An RPC system permits to call a remote procedure in another program. Conventional RPC systems (especially commercial implementations) are usually very restrictive in the types they support as arguments to the remote procedures. However, there is a new brand of *higher-order RPC systems* that support much richer data types as arguments to remote procedures, namely procedures themselves [77][14][8]. Recently, this research area has been extended towards persistence with some novel proposals [56][57][61][62].

Database agent language

Relational database systems acting as servers accept queries from client programs written in SQL, a declarative language. In this sense, SQL is a well-know, established *database agent language* (although there are not many references to SQL in the agent literature).

4 Challenges

Most challenges in this section are based on the fact that agents should migrate by copy—to avoid long-term dependencies between persistent programs— and the complexity and size of their transitive closures.

Reducing the transitive closure

Transitive closures in a persistent program may be very large and sometimes even reach the entire store [64]. A well-know approach to reduce the transitive closure of an object is to ask application programmers to make a distinction at compile-time between those objects that migrate by copy and those that migrate by reference only (i.e., that are not actually sent) [69]. Lazy migration can also be used to create remote references at run-time automatically by the system [11][42][45][50][88].

However, all these techniques use remote references that create dependencies between programs and as such are not appropriate for large, unreliable distributed environments like that of the Internet. Another approach to reduce the transitive closure which does not create remote references is simply not to migrate objects that already exist in the target program [8][42][57][56]. We have generalised this technique with our scheme of *migration by substitution* [62][63][64].

Sharing objects between stores

There are basically two approaches to solve the sharing problem in a distributed system: one copy (by remote access, see previous paragraph) or many copies (using some sort of replication protocol) [20]. The problem with replication is to maintain the replicas consistent when they are updated. Strict replication protocols that maintain the replicas always consistent can be used, but they require remote references and thus suffer from similar problems as the one-copy approach. In order to deal with distributed object sharing we have proposed and implemented a compromise between the one-copy and many-copies approach called *persistent spaces* [62][63].

Maintaining sharing semantics

Another problem with free variables occurs when objects that locally share another object(s) migrate separately to a target, since these shared objects will be duplicated remotely for each migration and destroy the original sharing semantics. Persistent spaces [62][63] can also be used as a first step towards solving this problem because within persistent spaces sharing is maintained between successive migrations.

Large and/or complex objects

Even after applying migration by substitution and using persistent spaces, the number of migrating objects may still be very large. The problem is that the time to pack and unpack large and/or complex objects is clearly dominant over transmission time [62][63]. More worryingly, migration time may increase *quadratically* to the number of objects being migrated depending on the packing algorithm. Finally, because packing and unpacking are CPU intensive, performance is limited by the implementation of the language itself (that is, if the language is slow, migration will be slow).

Mobility orthogonal to type

Most agent languages identify an agent at compilation time, thus restricting the set of objects which can later migrate. On the other hand, one of the requirements for Napier88 is *all objects in the language have the right to persist*. So at least for a Napier88 programmer it makes perfect sense that *all objects in the language have the right to migrate*. Migration orthogonal to type is also possible to implement as there are no such restrictions on mobile types in both Tycoon/RPC [57][56] and Napier88/RPC [62][63][64].

Failure model

Unlike static analysis of code or compile-time rules to decide if an object can migrate, the decision to migrate can instead be delayed until migration time. When migration cannot be achieved, the attempt should raise an exception that should ideally explain (in terms understandable to application programmers) why the migration did not succeed.

Standard agent language

Each agent system uses its own model and algorithms to migrate procedures and other objects between programs. Although probably not a technical challenge, the following questions still make an interesting topic for the agent community. Does it make sense to design and build an agent system independent of the actual programming languages in which the source and target programs are written? If yes, what kind of agent language, e.g., based on an existing language or a common sub-set of popular agent languages?

5 Conclusion

In the last three years we have been working with persistence and distribution. During this time we have built an higher-order RPC mechanism that permits us to exchange procedures and other complex data structures between autonomous persistent programs. In order to deal with some of the problems presented in section 4, we have proposed migration by substitution and persistent spaces. Migration by substitution offers a flexible mechanism to reduce the transitive closure of an agent by cutting links to objects that already exist in the destination store. Persistent spaces help with sharing objects between stores and maintaining sharing semantics, even between transmissions.

Remaining challenges for which we have not found a satisfactory solution yet include: marshaling algorithms that scale for large data structures; efficient migration orthogonal to type; a simple but comprehensive failure model; and the need and form of a standard agent language.

Acknowledgements

The authors would like to acknowledge Fritz Knabe and Bernd Mathiske who read previous versions of this paper and made numerous comments. Many other people have contributed to this paper, especially Andrew Black, Joachim Baumann, Peter Dickman, and Peter Larsson. The research environment in Glasgow has been partially supported by projects FIDE2, IMIS, DRASTIC, PJava and ZEST. Miguel Mira da Silva is partly supported by JNICT.