A Reflective Infrastructure for Coarse-Grained Strong Mobility and its Tool-Based Implementation*

Noury M. N. Bouraqadi-Saâdani, Thomas Ledoux and Mario Südholt

Objects, Components, Models group
Département Informatique
École des Mines de Nantes, Nantes, France

Abstract

We argue that separation of concerns can be advantageously applied to the development of applications relying on coarse-grained strong mobility, i.e. distributed applications moving entities while these entities are executing. We present the design of an infrastructure for such mobile applications where the mobility concern is cleanly separated from other concerns.

We present an overview of a prototype implementation — called RAM (Reflection for Adaptable Mobility) — of such an infrastructure for strong mobility in Java by means of computational reflection. This infrastructure permits the development of mobile applications by plugging a mobility concern implemented at the meta-level into a sequential base program, and enables the dynamic introduction of different migration policies. We show how such an infrastructure can be implemented based on program transformation techniques using appropriate existing tools.

Invited presentation at the *International Workshop on "Experiences with reflective systems"*, Kyoto, Japan, 25 September 2001.

(Held in conjunction with Reflection 2001, the *"3rd International Conference on Meta-level Architectures and Separation of Crosscutting Concerns"*.)

Technical report no.: 01/7/INFO

*This work is part of the project "Reflection for Adaptable Mobility" (RAM) which has been partially funded by France Télécom, CTI no. 98-5.1069.

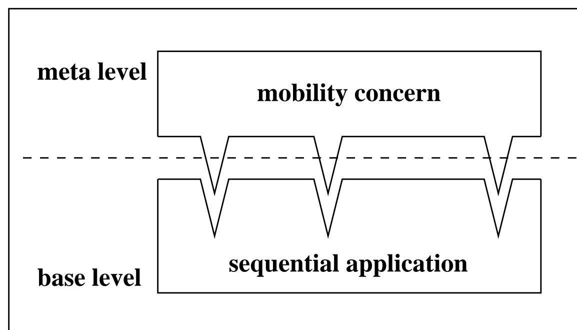


Figure 1: Building mobile applications by separating concerns

1 Introduction

Separation of concerns has been applied successfully as a guiding principle to the development process of many different kinds of distributed applications [HL95] [GGM96] [BCRP98] [Led99]. In this paper, we argue that separation of concerns can be advantageously applied to the development of applications relying on coarse-grained strong mobility. Such applications move entities during their execution, for instance, to achieve load balancing. We present the design of an infrastructure for such mobile applications which cleanly separates concerns and allows isolating mobility issues. Thus, a programmer can develop the functional part of his application without caring for mobility. The sequential program can then be turned into a mobile application by introducing mobility in a transparent way.

Computational reflection [Mae87] can be used as an implementation mechanism supporting separation of concerns in a distributed setting (see, for instance, the session on "reflective middleware" in [Coi99]). We present an overview of design and implementation of a reflective infrastructure — called RAM (Reflection for Adaptable Mobility) — for strong mobility in JAVA. It allows the development of mobile applications by plugging a mobility concern implemented at the meta-level into a sequential base program (cf. Figure 1). Besides, thanks to reflection, RAM enables the adaptation of mobile object systems by defining customized migration policies.

Finally, the prototype deserves some interest because of its implementation technique: instead of modifying the underlying JAVA virtual machine or standard libraries, reflective capabilities and mobility (essentially thread migration) are introduced using program transformation techniques performed by appropriate existing tools. We thus preserve full portability on standard JAVA platforms. We discuss constraints imposed by such an implementation method and compare them to the benefits it provides.

The paper is structured as follows. Section 2 gives an overview of the underlying mobility model. Section 3 presents the reflective infrastructure on which RAM is based in some detail. Then, Section 4 discusses related work and Section 5 sketchs out some future work. Finally, Section 6 presents a conclusion.

2 Mobility Model

In this section, we present the model of distribution and mobility of systems which our reflective infrastructure is intended to support.

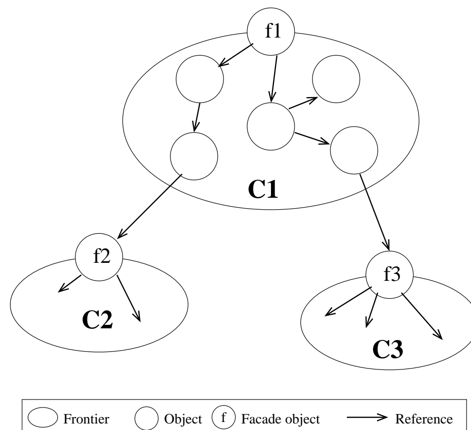


Figure 2: Example of clusters

2.1 Model Description

The model of distribution and mobility underlying our reflective infrastructure can be presented quite naturally in terms of its structural and behavioral features. Note that we did not look for a new distribution model but rather synthesized our model from existing ones — especially JAVANAISE [HL98] and JAVA RMI — in such a way that the model effectively supports coarse-grained distributed applications.

Structural part. Structurally, our model is built around two main abstractions, clusters and places:

- A *cluster* groups a number of objects and constitutes a unit of migration. Clusters are responsible of their own migration policy. Each cluster features a unique *facade object*[1] providing a unified interface for the whole cluster. The facade object of a cluster is the only object of that cluster which is visible to other clusters. Finally, a *frontier* of a cluster delimits the group of objects which can be accessed from the facade object of a cluster without accessing other clusters (see Figure 2).

- A *place* constitutes the run-time environment for clusters and provides, in particular, services for mobility (cluster migration, cluster localisation, etc.). It may host several clusters.

Note that we reduced the presentation of the structural part of the model to only those parts essential for the reflective infrastructure described below. We did not mention, for instance, the *host* abstraction, which represents the physical machine connected via a network to others in the distributed system. We suppose the existence of a *naming service* which is associated to hosts and maps (symbolic) names to clusters. We also suppose the existence of a *code repository* from where byte-code of migrating object classes can be retrieved.

This model reflects our interest in coarse-grained distributed applications, mainly because units of migration are not objects but clusters, that is, groups of objects having a specific structure (facade object, frontier).

[1] Named according to the analoguous design pattern [GHJV94].

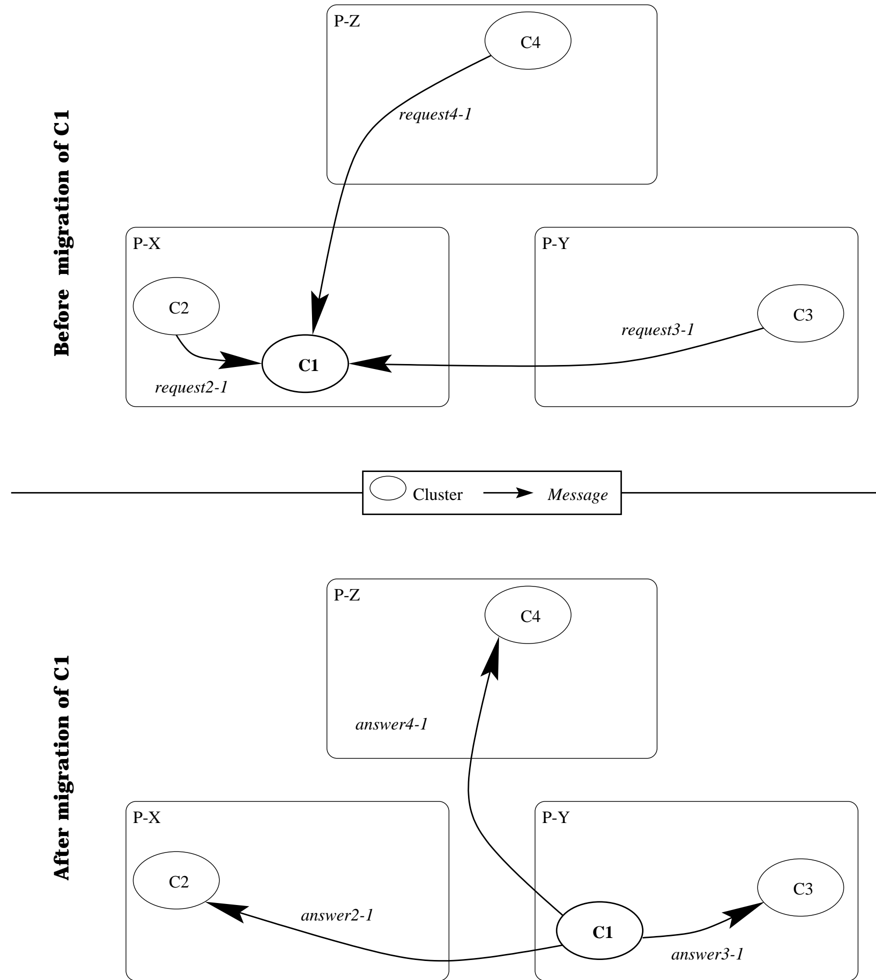


Figure 3: Example of cluster communication

Behavioral part. The behavior of mobile applications is modeled as follows:

- *Communication* is performed using ordinary Java method calls within a cluster and an *RPC-like* mechanism between clusters. In order to simplify the treatment of mobility, we model such RPC-like communication by two synchronous messages: a *request* message and an *answer* message. Both contain information about the two participants in the communication. Furthermore, the former contains the arguments and the latter a return value (potentially being an exception to be raised). This enables a communication partner to be found if one of the two partners moved to another place during the communication period. Note that in order to preserve the encapsulation property of clusters, cluster facades are passed by reference in message arguments but objects encapsulated in clusters are passed by (deep) copy.

  Figure 3 shows such an example, where three clusters C2, C3 and C4, located in three different places (respectively P-X, P-Y and P-Z) communicate with the cluster C1. Consider the case where C1 moves from place P-X to place P-Y while communication is taking place. Moreover, imagine that C1 receives all requests while it is still at the departure place P-X, and processes them at the arrival place P-Y. Thanks to the information identifying the communicating clusters

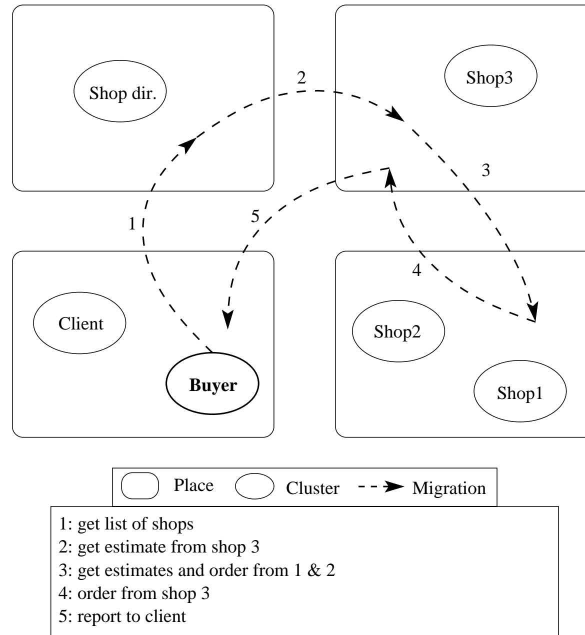


Figure 4: A mobile purchase application

that is stored in the messages, C1 can retrieve addresses of the requesting clusters and send them answers.

- *Reference management* is a particular important issue of mobile systems. Consider the critical case of reference update after cluster migration. When a cluster C1 migrates to a new place, it provides a list of clusters it references to that place and gets a list of references in return. Since each place knows all clusters it hosts and a place is able to retrieve references for remote clusters via the naming service. The departure place of the migrating cluster updates local references to the migrating cluster C1. References to C1 held by clusters that are neither hosted at the departure place of C1 nor by its arrival place are updated lazily, in the course of subsequent communications to or from C1.

- *Migration* of a cluster locks the execution of that cluster, constructs a structure representing its execution state, transfers it to the target place, rebuilds an executable cluster from the transferred state, destroys the originating cluster and resumes its execution at the target. Note that this requires synchronization between the cluster to be migrated and the departure place which has to suspend message sends to the cluster temporarily. It also requires synchronization between the two involved places in order to avoid cluster duplication on network failure.

Note that, similar to the structural part, we restricted the discussion to those features essential to our reflective approach to mobility.

2.2 Example in an E-Commerce Setting

In order to illustrate our model, we briefly discuss an example taken from the e-commerce domain. Consider the following scenario (see Figure 4). A client wants to buy a number of items but does not

know all the shops selling the items and their respective price tags. S/he therefore creates and orders another entity, called "buyer," to gather all necessary information and buy the required items. The buyer gets from a directory a list of suitable shops, which are located on different places, visits them in order to get the different estimates, determines the best price, and finally orders the items for the client.

The client, the buyer, the shop directory, and the shops may be implemented using our model by clusters; the estimates and orders may be implemented by objects encapsulated in the clusters. Each cluster has its own migration policy. In the context of the interactions illustrated in Figure 4, the buyer would migrate, for instance, each time it sends a message to a remote cluster.

3 Reflective Infrastructure

In this section, we present the model of JAVA based prototype RAM (Reflection for Adaptable Mobility). In order to introduce the mobility concern via our reflective infrastructure in a transparent way, we first analyzed the requirements for a such an infrastructure. Based on this analysis, we develop the reflective infrastructure followed by a brief description of the user-level development process enabled by this infrastructure. Finally, we discuss the implementation of the prototype in some detail.

3.1 Requirement Analysis

Based on usage scenarios, such as the example introduced in Section 2.2, we derived requirements for a reflective infrastructure. The most important of which are listed below (a more complete list is given in [BSDLS00]):

Cluster.

R1) *Creation of facade objects has to be distinguished from the creation of ordinary objects.* Instantiation of a facade object allows the creation of a cluster and leads to the construction of the cluster infrastructure (cluster manager, message queues, etc.).

R2) *The cluster infrastructure has to be installed before the constructor of a facade class is called.* Execution of a JAVA constructor can lead to an inter-cluster communication. The cluster infrastructure is thus to be installed and linked to the facade before the constructor of the facade is executed.

R3) *Cluster serialization during migration must stop at frontiers.* Serialization is used to build a representation of a cluster to be transferred during migration. If frontiers had not be respected, clusters would not be units of migration.

Inter-cluster communication.

R4) *Inter-cluster method calls must be detected and transformed into communication requests.* This requires detection of calls beyond a cluster's frontier.

R5) *Cluster encapsulation should be preserved.* This requires that during communications facades are passed by reference and other objects by deep-copy.

Reference management.

R6) *It is necessary to distinguish between local and remote references.* Besides being necessary to handle cluster migration, distinguishing these kinds of references is useful because of efficiency concerns.

R7) *It should be possible to switch references from local to remote and vice versa.* This is necessary to update references in the context of migration.

R8) *Invalid references to migrated clusters must be identifiable.* Invalid references can thus be updated (in an eager or lazy fashion).

3.2 A Reflective Model

Reflection is the process of reasoning about and acting upon itself [Smi84]. Computational reflection [Mae87] supports run-time adaptability because it supports the derivation of new behaviors from initial ones based on variations of the underlying computational model. Moreover, reflective programming allows separation of concerns because a clear separation between the application code (the base-level) and its description and/or control (the *meta-level*) is provided. In a reflective OO language, base-level objects are controlled by objects at the meta-level called *meta-objects*. A base-level object has a *meta-link* to its meta-object. A Meta-Object Protocol (*MOP*) provides a protocol followed by the meta-objects in order to control the base-level objects [KdB91].

Our reflective model is built around a kernel providing a general-purpose meta-object protocol useful for distributed applications and a set of meta-objects specialized for the implementation of the mobility concern. In the following, we justify the different issues raised by relating them the requirements listed in the previous section.

3.2.1 The Kernel

The reflective kernel allows the following four execution mechanisms to be controlled:

- *Control of object creation.* In order to install the cluster infrastructure in a transparent way, we have to treat cluster creation (that is, creation of the facade object) during the ordinary Java creation protocol, that is, execution of `new` statements (cf. requirement R1).
- *Initialization of the meta-link.* Technically, the meta-object protocol relies on meta-objects to be attached to some of base-level objects (e.g. facade objects), such that meta-objects control the execution of base-level objects. By initializing the meta-link right after object creation, our meta-objects defining the mobility infrastructure can handle inter-cluster communication that can take place during constructor calls (R2).
- *Control of method call receptions.* In our MOP, method call reception on a base-level object can be controlled by a meta-object. This way, we can have meta-objects that transform a JAVA method invocation into a communication protocol for inter-cluster communication (R4).
- *Control of object serialization.* During lifecycle of a cluster, objects can be serialized either on mirgration or on inter-cluster communication (parameter passing by deep copy). Control of serialization allows to respect clusters encapsulation (R5) and frontiers (R3) .

As noted above, the kernel of our reflective model directly addresses the five first requirements (R1-5). The other requirements (R6-8) and the mobility model are handled by a dedicated specialization of the kernel for mobility as described in the following section.

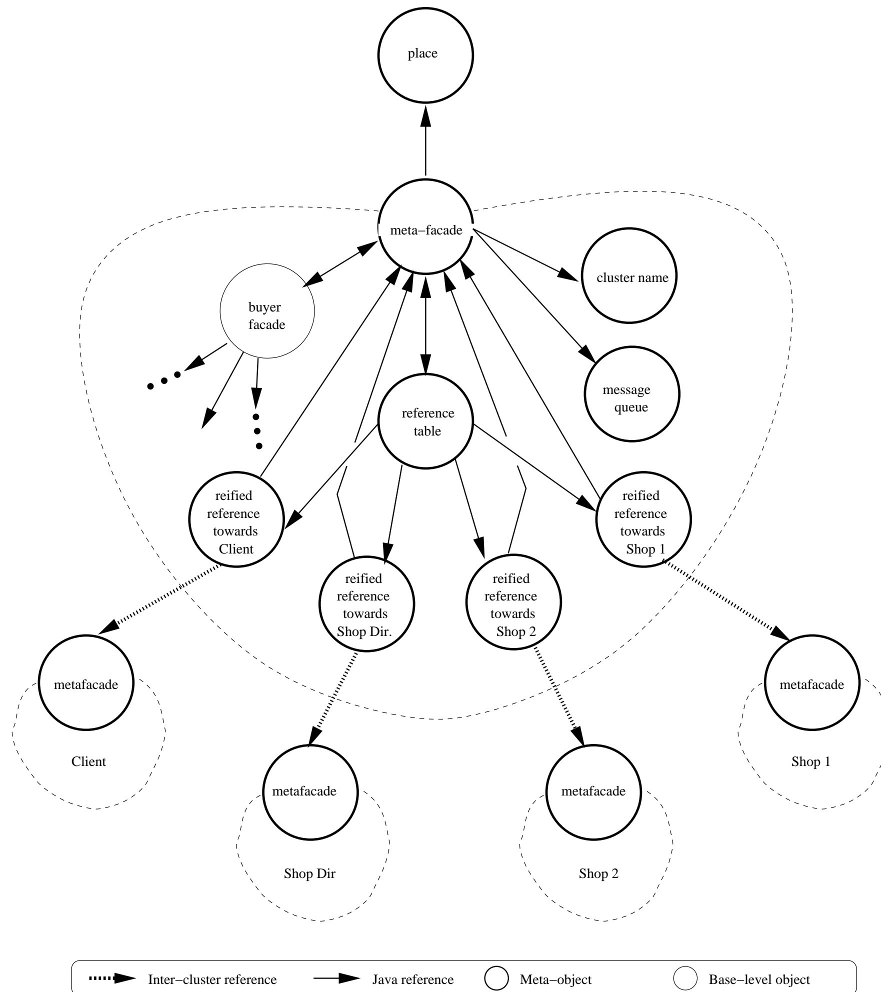


Figure 5: Overview of a cluster in our reflective infrastructure.

3.2.2 Meta-objects for Mobility

The reflective infrastructure of RAM is based on the general reflective kernel presented before. The mobility concern is built on top of the kernel in form of several dedicated meta-objects. In order to illustrate the different meta-objects, we show the structural representation of the cluster buyer in Figure 5.

Meta-facade. Figure 5 illustrates that the facade object of the cluster is attached to a meta-object called *meta-facade*. This meta-object plays several roles: the meta-facade initializes the cluster infrastructure (message queue, thread, etc.), manages message exchanges and the lifecycle of the cluster including the migration policy. Customization of the default migration policy is done by specialization of the default meta-facade class.

Reified reference. Figure 5 includes several meta-objects labelled *reified references*, which represent both local and remote references from a cluster towards another cluster (R6). These reified

references are created by meta-facades, essentially during deserialization of arguments contained in message requests. A reified reference holds the invariant name of the referenced cluster. Thus, it is able to update itself (R7) and retrieve the meta-facade of the referenced cluster for purpose of communication. Transforming JAVA method invocations into a request message is also done by reified references. Finally, reified references allow the computation of cluster frontiers: they stop the JAVA serialization process.

Reference table. All reified references of a cluster are stored in the cluster's *reference table*, which enables the meta-facade to retrieve and update all references towards other clusters during and after migration (R8). This table is important for several reasons: it ensures the unicity per cluster of a reified reference thus avoiding useless reference proliferation and ensuring that one update per reference and cluster is sufficient in the case of migration.

Reified references and the reference table enable references to be managed in the context of migration. When a cluster C migrates, it is necessary to update references from C to other clusters and references held by the other clusters towards C. The first case is immediate because the cluster C owns a table of reified references: after the migration has terminated, the migrated cluster immediately updates invalid references by help of its new place. In second case (updating the references held by the other clusters towards the migrated cluster) the update depends on whether the clusters linked by a reference were originally located on the same place or not. If they were co-located, the corresponding local references must be transformed into remote ones, otherwise the remote one has to be updated essentially using the same protocol as in the first case.

3.3 Adaptation of the Default Execution Policies

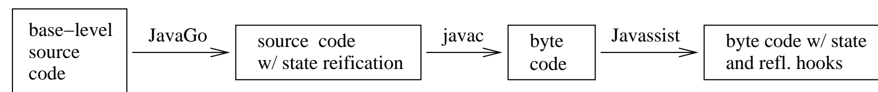
Reification of principal entities of a mobile objects system allows the definition of different migration policies. The meta-facade which is in charge of inter-cluster communication and migration is the main entity entitled for dynamic adaptation of new behavior.

As far as migration conditions are concerned, the default policy is to use systematic migration when inter-cluster communication takes place between two remote clusters. Other migration policies can, however, be easily implemented. For example, by specializing the default meta-facade, we can design stationary clusters, that is, clusters which execute only on the place where they start execution and communicate with other clusters using RPC. Also, a meta-facade could introspect dynamic network characteristics such as network traffic, network reliability, CPU performance, etc, in order to choose the best moment to migrate.

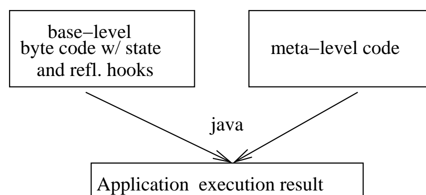
Similarly, resource control can be customized by specializing the meta-objects playing the role of reified references. The default update policy is to find the actual location of the referenced cluster (policy *by network reference*, cf. reconfiguration of resources in [FPV98]). By specializing the reified references, we can introduce new update policies such as making the referenced cluster move to the local place (policy *by move*) or to copy locally the referenced cluster (policy *by copy*).

Concerning thread management, the default policy is to provide strong mobility, that is, migration of the currently executing thread. This policy could turn out not to be optimal for a particular application (e.g. when strong mobility is not always needed). An extension of the default meta-facade could easily provide support for weak mobility.

Concerning communication, the default policy is to use synchronous message exchanges. Since the communication "mode" is defined by meta-facades, we can modify this behavior to achieve asynchronous communication (with future objects), one-way communication, communication with messages having a priority, etc.



a) compilation of base–level programs



b) execution of mobile code

Figure 6: Tool integration

3.4 Overview of the User-Level Development Process

The model and infrastructure we propose enable an incremental programming methodology: the programmer may design and implement its application in a non-distributed non-mobile environment, focusing only on its functional properties (i.e. services provided by application). Then, without modifying the previously written code, the programmer may turn its application into a mobile object system by introducing a mobility concern. To reach this goal, the programmer has to perform the following steps.

1. *Clusters assignment (design choice).* Developers should choose classes that will play the role of facades. This is done simply by setting up a *base-to-meta table* linking base-level classes and meta-object classes. When a base-level object is created, the right meta-object class is instantiated. Then, the object is attached to the newly created meta-object.

2. *Reflective capabilities and thread migration support through program transformation.* A base-level object has to become a *reflective object*, i.e. an object that is able to be controlled by meta-objects. This is done by transforming the initial code using a fully automatic program transformation which introduces the meta-link and hooks permitting to transfer control from the base-level code to the meta-level. Another automatic transformation introduces support for thread migration (see Section 3.5).

3. *Deployment.* Finally, we need to deploy different clusters to achieve the distributed start configuration. RAM uses an initialization cluster to this end: it migrates between several places to create the initial application clusters.

3.5 Implementation Based on Tool Integration

We built a prototype which implements the model and reflective infrastructure described above. This prototype relies on different existing tools for program manipulation. First, we give an overview of the tool-based structure of the prototype, followed by brief descriptions of how the essential parts of this architecture are implemented.

Figure 6 illustrates the architecture of the prototype. At compile-time (see Figure 6a), a base-level program is transformed using two program transformation tools:

- JAVAGO [SMY99] is used to transform sequential code such that the execution state of the currently executing cluster is reified and the cluster migrates whenever a specific migratory exception is raised.
- JAVASSIST [Chi00] is used to insert hooks into the base-level program which enable control to be transferred to the meta-level in case of object creations, method invocation, change of meta-links, and object serialization.

At runtime (see Figure 6b), the meta-level takes control of the program execution. It can throw the migratory exception and thus handle the mobility concern according to some policy. Note that this architecture substantiates our initial goal as illustrated in Figure 1.

3.5.1 Thread Migration Through Program Transformation

One of the main defining characteristics of the project on which we report here was to achieve maximal portability. This excluded changes to the JAVA virtual machine and to JAVA's basic run-time libraries. Hence, we chose a program transformation based approach to strong mobility [Fue98] [SMY99] and the tool JAVAGO developed by Sekiguchi et al. [SMY99]. JAVAGO implements strong mobility by transforming JAVA programs, such that the execution state of the current thread is reified whenever a specific exception is raised at run-time.

More precisely, this program transformation technique enables the contextual information of an executing thread to be stored by transforming all methods appearing on the execution stack when the migratory exception is raised in a two-fold manner:

- The exception is caught and the contextual information (such as local variables) is stored in an appropriate data structure in the exception handler.
- Statements are "labelled" such that execution can resume at any statement once the cluster has been migrated.

Being the most portable solution, this implementation technique for strong migration has an important drawback, though. Without (non-standard) support from the underlying virtual machine it is only possible to store the contextual information of the thread that is active when the migratory exception is raised. It is not possible to reify sleeping threads. Since portability was the prime requirement of the project definition, we chose to stick to this implementation method and restrict clusters to contain only one user-defined thread. Note that this restriction only applies to user-defined threads but not threads which are encapsulated in the underlying infrastructure. Furthermore, multiple threads can be supported using checkpointing-based methods as discussed by Fuenfrocken [Fue98]. Finally, this technique obviously introduces a run-time overhead. The results presented in [SMY99] indicate, however, that this should not be a problem in the context of coarse-grained mobile applications.

3.5.2 Reflective Hooks Through Program Transformation

We also used program transformation techniques in order to introduce the hooks that are used to transfer control from the base-level code to the meta-level code. To this end, we used JAVASSIST [Chi00], a tool for structural reflection in Java. Basically, each of the different features to be intercepted, such as method invocation (see Section 3.2), corresponds to a transformation step in the prototype which

has been implemented using JAVASSIST's transformation library. Such transformations consists in introducing indirections in the base-level code. Method wrappers [BFJR98] are an example of tranformation. We exploited these transformations in order to shift to the meta-level.

4 Related Work

The system we have presented has three main characteristics: it enables the implementation of (coarse-grained) strong mobile systems in JAVA, uses computational reflection in order to cleanly separate a mobility concern from the non-mobile functional concerns of applications, and preserves complete portability by not relying on modifications of the JVM or standard libraries. In the following, we restrict our discussion to (the few) reflective systems which approach the same goals.

The systems AL-1/D [OI94] and TJ [McA95] are systems supporting a mobility concern based on computational reflection. AL-1/D provides a model of adaptable migration policies, which is, however, simpler than ours in that it relies only on reflective support for the interception of messages. Furthermore, AL-1/D does not address the problem of how to resolve references during and after migration explicitly. TJ provides very fine-grained reflective support for distributed infrastructures and mashalling. This support is, however, of much lower level and finer-grained, and does therefore need an additional layer of abstractions in order to be fully comparable to our work. Finally, since both of these systems have not been implemented in JAVA, our compatibility requirements are not meaningful in this context.

PROACTIVE [CKV98] is the only related work on reflection in the context of mobility in JAVA. PROACTIVE enables method invocations to be intercepted without modification to the JVM by using an explicit specific object-creation protocol. PROACTIVE defines an infrastructure for distributed applications and a mobility concern including reference management. However, full transparency is lost due to the use of the specific object creation protocol. Besides, this platform is a weak mobile system: its migration primitive can only be called as the last instruction in a method. Hence, the problem of reconciling compatibility constraints with task reification is not addressed.

5 Future Work

Our prototype architecture has two layers. One is dedicated to mobility, and the other (the kernel) is a general-purpose meta-object protocol. This kernel can be specialized for other purposes than mobility. We currently explore alternative specializations by developing a meta-object library which deals with other middleware features such as authentication, disconnected mode, publish/subscribe interaction, etc.

Dynamic adaptability is another topic that we plan to explore more deeply. Such adaptability is necessary in contexts of scarce resources. We are investigating reflective middleware solutions for context-aware applications. Such applications have to be aware of, and adapt to, variations in the context of execution, such as fluctuating network bandwidth, decreasing battery power, etc. Adaptability can be achieved by replacing some meta-level entities.

Finally, an challenging future work is dealing with complex systems that have different non-orthogonal concerns. In such systems, it is necessary to handle overlapping between concerns while composing different concerns. In the context of reflective systems, the problem of concern composition can be expressed in terms of the composition of meta-objects.

6 Conclusion

In this paper, we have presented the model and infrastructure provided by RAM (Reflection for Adaptable Mobility). RAM enables the development of coarse-grained strong mobile application in JAVA without compromising portability. Strong mobility has been implemented using program transformation techniques which do not require the underlying virtual machine or standard libraries to be modified. Thanks to RAM's infrastructure, we achieved complete separation of the mobility concern from the non-mobile functional concerns of an application by means of reflection. This enables the dynamic adaptability of mobile object systems by addressing new execution policies for their infrastructure.

References

[BCRP98] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, London, 1998. Springer-Verlag.

[BFJR98] J. Brant, B. Foote, R. E. Johnson, and D. Roberts. Wrappers to the rescue. In *ECOOP'98*, pages 396–417, 1998.

[BSDLS00] N. M. N. Bouraqadi-Saâdani, R. Douence, T. Ledoux, and M. Südholt. Une infrastructure réflexive pour la mobilité forte. Technical report, École des Mines de Nantes, 2000. in French.

[Chi00] S. Chiba. Load-time structural reflection in Java. In *Proceedings of ECOOP'2000*, LNCS. Springer Verlag, 2000.

[CKV98] D. Caromel, W. Klauser, and J. Vayssire. Towards Seamless Computing and Metacomputing in Java. *Concurrency Practice and Experience*, pages 1043–1061, September–November 1998.

[Coi99] P. Cointe, editor. *Proceedings of the 2nd International Conference on Reflection*, volume 1616 of *LNCS*. Springer Verlag, 1999.

[FPV98] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[Fue98] S. Fuenfrocken. Transparent migration of Java-based mobile agents: Capturing and reestablishing the state of Java programs. In *Second International Conference on Mobile Agents*, volume 1477. Springer Verlag, 1998.

[GGM96] R. Guerraoui, B. Garbinato, and K. Mazouni. Lessons from designing and implementing GARF. In *Object-Oriented Parallel and Distributed Computing*, LNCS. Springer-Verlag, 1996.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[HL95] W. Hürsch and C. Videira Lopes. Separation of concerns. Technical Report NU-CCS-95-03, Northeastern University, 1995.

[HL98] D. Hagimont and D. Louvegnies. Javanaise: distributed shared objects for internet cooperative applications. In *Proceedings of the International Conference on Middleware*, 1998.

[KdB91] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, Cambridge (MA), USA, 1991.

[Led99] T. Ledoux. OpenCorba: a Reflective Open Broker. In *Reflection'99*, volume 1616 of *LNCS*. Springer Verlag, 1999.

[Mae87] Patricia Maes. *Computational Reflection*. PhD thesis, Vrije Universiteit, Brussel, 1987. also Technical Report 87-2.

[McA95] J. McAffer. *A Meta-level Architecture for Prototyping Object Systems*. PhD thesis, University of Tokyo, Japan, September 1995.

[OI94] H. Okamura and Y. Ishikawa. Object Location Control Using Meta-level Programming. In *Proceedings of ECOOP*, pages 299–319, 1994.

[Smi84] B. C. Smith. Reflection and semantics in LISP. Technical Report P84-00030, Xerox Palto Alto Research Center, Palo Alto, 1984.

[SMY99] T. Sekiguchi, H. Masuhara, and A. Yonezawa. A simple extension of java language for controllable transparent migration and its portable implementation. In *Proc. of the International Conference on Coordination*, 1999.