Introducing Distribution into Applications: a Reflective Approach for Transparency and Dynamic Fine-Grained Object Allocation

Antonella Di Stefano[1] Giuseppe Pappalardo[2]
Emiliano Tramontana[2]
[1]Dipartimento di Ingegneria Informatica e delle Telecomunicazioni
[2]Dipartimento di Matematica e Informatica
Università di Catania, Viale A. Doria, 6 - 95125 Catania, Italy
adistefa@diit.unict.it, {pappalardo, tramontana}@dmi.unict.it

Abstract

Developing distributed software systems is a complex activity that involves facing not only the problems of a specific application, but also those typical of distribution. Computational reflection supplies a means to handle different concerns with distinct components and a framework in which the latter can interact smoothly. We propose a reflective software architecture that encapsulates distribution concerns within components that are separated from and independent of those addressing functional concerns. The proposed architecture achieves a thorough management of distribution and in particular provides a means to dynamically adapt allocation policies to the characteristics of application objects, available hosts and changes of the distributed environment. The proposed approach is helpful for achieving the incremental development of easy to evolve software systems. In particular, we discuss the benefits of applying it to existing web and e-commerce applications.

1. Introduction

Distributed software systems often intertwine the handling of application functionalities with distribution related issues, which typically include (possibly) location transparent access to services, workload allocation among different hosts, adaptation to changing conditions of the network, etc. This intertwining makes development of distributed applications more complex, and their reuse and evolution more difficult [6]; on the contrary the key to improving this kind of system design is to enforce separation between application and distribution concerns.

It should be emphasized, however, that such a separation is not so easy to achieve, at least as far as distributed allocation is concerned. Indeed, allocation policies aiming to provide the most appropriate environment to application objects cannot ignore the characteristics of the latter, nor those of the hosts and the distributed environment. As a result it would be desirable to have the ability to tailor allocation policies to specific groups of objects. However, this goal may well conflict with the noted one of separating application from distribution issues.

In the last decade it has been shown that *computational reflection* is helpful in separating application concerns from others, such as fault-tolerance [12], synchronisation [16], etc. Reflection provides the means for a software system to observe some of its own parts and perform operations on them [7]. A reflective object-oriented system usually consists of two, or more, levels; objects at the first or base-level (in our case application objects) are transparently observed and influenced with higher level objects, called *metaobjects*. For reflective systems the cost of integrating an application with other concerns (such as distribution) is very low: for the programmer it simply consists of making some application objects reflective, whereas at run time it amounts to the cost of jumping to the metalevel. However, existing reflective approaches that deal with distribution [8, 10, 11] lack both the means to adopt specific allocation policies, tailored to the characteristics of application objects, and the mechanisms that allow run time adaptation to changes of the distributed environment. A detailed analysis of these approaches is given in Section 6.

Such limitations are not inherent to reflective approaches, but were already present in object-based [9], and even more traditional ones, like PVM [14]. Furthermore, they leave entirely to the application programmer the burden to cope with the adaptation of the application to a distributed environment.

Yet different approaches rely on language support, typically in the form of directives, to let the programmer specify the allocation of objects. This does allow allocation policies to be tailored to the characteristics of application objects. In [3] such a language includes some high-level directives to drive object-allocation decisions at run time.

The present work proposes a software architecture that, through the use of reflection, provides support for remote communications, determines how to allocate objects based on their characteristics, and adapts to run time network faults. The proposed architecture enforces *separation* and *independence* between functional and distribution concerns, thus simplifying the development and evolution of distributed software systems. Distribution concerns are handled by intercepting operations of application objects (i.e. object instantiations and method invocations), thus hiding any statements concerned with distribution from those addressing functionalities. A further advantage of the advocated approach is that metalevel components can be potentially reused for various applications. In our view, a metalevel like that described in the following could even become a general framework suitable for introducing distribution into a wide class of object-oriented applications.

Furthermore, the proposed reflective architecture permits *incremental development*, in that it can be employed to make distributed applications that were not thought as such initially. This additional benefit is instead unavailable to solutions based on language level allocation directives.

The outline of the paper is as follows. Section 2 presents computational reflective systems. Section 3 describes how distribution can be dealt with by means of a reflective software architecture, whose merits are discussed in Section 4. Section 5 provides an overview of how the proposed architecture can be helpful for handling distribution issues for e-commerce web applications. Section 6 examines other distribution related reflective software architectures. Finally, the conclusions are drawn in Section 7.

2. Reflective Systems

A software system is reflective when it represents some of its own aspects through structures which enable it to observe and act on itself [7]. Typically, a reflective object-oriented system consists of a baselevel and a metalevel, where objects and metalevel objects are respectively hosted. The most widespread reflective approach is the *metaobject model* [5], which is the one adopted for the proposed architecture. This model operates on objects and uses *metaobjects* (i.e. instances of a special class at the metalevel) to intercept events of the baselevel. In particular, a metaobject associated with an object is able to inspect the object's state and to trap control before such an object is, e.g., instantiated or invoked. Once control is trapped, metaobjects can perform some computation and decide whether to give control back to the objects.

The association of an object with a metaobject, which is what makes it reflective, relies on bytecode manipulation or injection of keywords into the source code of objects. The former approach is used for Java extensions such as Kava [17] and Javassist [2]. The latter is adopted by languages such as OpenC++ [1] and OpenJava [15].

3. Reflective Software Architecture for Distribution

3.1. The Architecture

This Section proposes a reflective software architecture aimed at introducing distribution into an existing object oriented application. The only a priori architectural constraint on these applications is that concurrent activities, if any, should only interact through methods of monitor objects. This guarantees that distribution introduced with our approach does not interfere with or jeopardize existing synchronisation mechanisms of the application.

The proposed architecture places the application at the baselevel and the support that handles distribution at the metalevel. The latter distributes application objects according to their characteristics, while keeping them unaware of distribution. Moreover, the metalevel is able to re-locate objects in order to meet run time changes within the network due, e.g., to faults or unexpected overloads of relevant duration affecting either hosts or network links. More specifically, the activities entrusted to the metalevel of the architecture are the following.

- Providing location transparency for the interaction between objects that have been spread to different hosts by the allocation mechanisms.
- Measuring host workloads in view of balancing them.
- Monitoring network efficiency (through selected parameters), and the reachability of remote objects before actually invoking their methods, in order to handle potential communication problems.
- Controlling the creation of application objects in order to select the best available host for their allocation.
- Handling migration of application objects to adapt to network faults and load balancing goals.

The activities of the metalevel are started by trapping control when events of the application, such as creation of a new instance and method calls from objects, are detected. Both events can be intercepted in the reflective language Kava [17], which has been employed for experiments.

Reflection is a helpful mechanism to detect at run time distribution-related events generated by an application, and to provide support to insert at run time activities extending applications with distribution. Compared with other mechanisms, reflection provides the ability to intercept events without actually changing the application objects and to inspect these in order to be aware of their state when handling distribution. Thus, the reflective metalevel allows additional capabilities, related to distribution and fault-tolerance, to be added to an existing application in a transparent way.

The metalevel of the proposed architecture is built on top of the existing classes. Thus, once the classes of an application have been developed in a centralised version, they are transformed into reflective ones in order to allow the application to take advantage of a distributed environment. Classes are transformed into reflective versions by associating them with metaobject classes called *Interceptors*[1]. This association makes it possible to bring control within the metaobject both before method invocations, which allows remote communication to be handled transparently, and before the creation of new instances, which permits objects to be allocated remotely. Different interceptor classes are available, each implementing a different allocation policy. For each application class, the interceptor class associated with it at the metalevel, hence the relevant allocation policy, is selected according to its specific characteristics. In general, this enables allocation policies to be tailored to classes, without making them even aware of distribution.

Besides interceptors, the metalevel hosts other classes in order to provide its services. Class `Adapter` is responsible to monitor the state of the network and hosts, and to adapt the application to changing conditions by relocating objects. Class `Locator` maintains a table storing the location of each object of the application and the last available state of the object. Classes `Communicator` and `ServerProxy` are responsible for handling communication between hosts at client and server side, respectively.

3.2. Case Study: a Multiuser Electronic Diary

To illustrate the above concepts, we consider as a case study a shared electronic diary that handles the appointments of its users. In such a software system, each user is characterised by a name and an email address, and an appointment is characterised by the date and time when it takes place, its location and the users engaged. As shown in figure 1, the electronic diary consists of classes `Appointments`, `AddressBook`, and `FrontEnd`. The first and second class handle information about a list of appointments and users, respectively. The third class handles interaction with users by exploiting the methods provided by the previous two classes.

Suppose now that objects of class `Appointments` need a relevant amount of memory and CPU "power" to find a free time frame for all the invited users; method `findSpace()` is available for this purpose. `Appointments` is therefore associated with interceptor `InterceptorCPU` which tends to allocate instances on hosts possessing enough resources. On the other hand, suppose that class `AddressBook` needs access to a particular file system storing user data, which is only available on certain hosts. For this reason, `AddressBook` is associated with a different interceptor, i.e. `InterceptorFileSys`. Finally, class `FrontEnd` may need to run on the local host for faster/easier communications with a user; it is therefore associated with a suitable interceptor, called `InterceptorLocal`, which does not perform remote allocation.

Figure 2 shows, using the UML notation, the distributed reflective software architecture of the electronic diary, in a scenario where objects have been instantiated on two different hosts. The first host holds an object <u>`:FrontEnd`</u> with its metaobject <u>`:InterceptorLocal`</u>; the second host holds object <u>`:Appointments`</u> with its metaobject <u>`:InterceptorCPU`</u>.

At run time the invocation of method `find-`

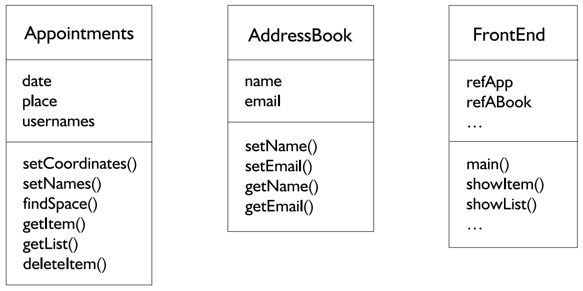

Figure 1. Class diagram of the electronic diary

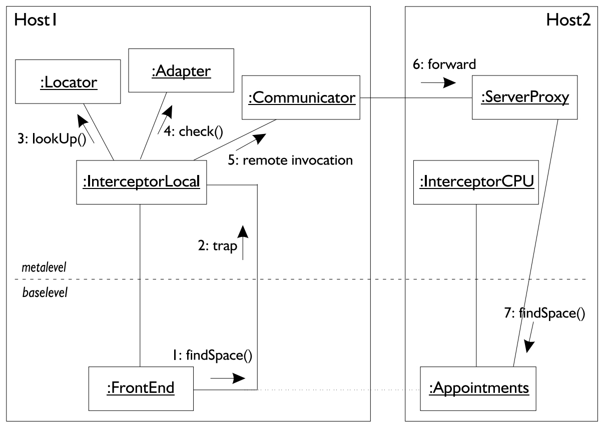

Figure 2. Distributed reflective software architecture

[1]The association is specified by a configuration file, with Kava, or a class, with Javassist, and then a transformer has to be invoked.

`Space()` by `:FrontEnd` (see (1) in Figure 2) is trapped at the metalevel (2) by the associated metaobject `:InterceptorLocal`. This metaobject checks whether the invocation is for a remote object, by looking up its position using metalevel object `:Locator` (3). If the object is local, control is given back to the baselevel, otherwise the availability of the connection is checked, using metalevel object `:Adapter` (4) and if possible the remote invocation is handled by metalevel object `:Communicator` (5). The latter metalevel object communicates the request to a proxy on the remote host, i.e. `:ServerProxy` (6), which then invokes the appropriate method of object `:Appointments` (7). The return value of the invocation is handled back from object `:Appointments` to the caller, by means of the metalevel objects `:ServerProxy`, `:Communicator` and `:InterceptorLocal`.

The metalevel objects introduced in Section 3.1 are described in the following Sections.

3.3. Interceptor

Interceptors are metaobjects that are associated with the classes of an application in order to handle the allocation of their instances and its interactions in the distributed environment. When an object is about to be instantiated, the interceptor traps such an operation and, according to its allocation policy, decides where the object has to be created. Moreover, it intercepts all the outgoing messages, so as to handle remote communications transparently, and state variables changes, so as to take a snapshot of the state of the associated object. This is periodically communicated to remote hosts, which will need it when the object must be re-located.

Our architecture provides three different types of interceptors (but of course others can be introduced, as necessary). The first, named `InterceptorLocal` is used for objects that need to stay on the local host, so it does not implement any remote allocation policy of its associated object (of course it must still check whether the creation of objects is compatible with the current workload of the local host). The second type of interceptor, named `InterceptorCPU`, tries to allocate objects on hosts that have an adequate amount of memory and CPU “power”. The third type of interceptor, called `InterceptorFileSys`, aims at allocating objects on hosts that provide access to resources that can replace those the application would expect on the local host. These could be I/O devices, software libraries, or file systems which are needed at run time. It is worth noting that `InterceptorCPU` and `InterceptorFileSys` seek to achieve load balancing among those hosts that provide interchangeable resources.

The specific kind of interceptor employed is chosen when the association between objects and metaobjects is performed. This allows allocation policies to be tailored to the varying needs of application classes. This is a configuration choice to be performed when the baselevel application is deployed by integrating it with the metalevel. Thus, at this stage, the programmer should be aware that some communication costs could arise for accessing certain objects. This implies that in some cases `InterceptorLocal` could be preferred to avoid incurring into excessive communication penalties.

In order to perform the allocation of objects, each type of interceptor holds the information shown in table 1 about the hosts in the distributed system.

Table 1. Interceptor table for the characteristics of hosts

hostname	type	workload
⋮	⋮	⋮
hostname	type	workload

For each available host there is a table entry providing the `hostname`, the `type`, i.e. an indication of its resources (CPU `power`, file systems, I/O devices etc.), and `workload`, a measure of its workload (as determined by the `Adapter` component, see Section 3.4). Interceptors compute the maximum of the ratio `power / workload` over a set of hosts which includes all of them for `InterceptorCPU`, or those providing the same resources as the local host for `InterceptorFileSys`. The remote host maximising the ratio is selected for the object to be instantiated. The description of the network in terms of hosts available and their characteristics (i.e. power of each host, resources) is stored in a configuration file that interceptors read when they are created.

Figure 3 shows the operations that are performed when an object is instantiated and the associated interceptor decides to create the object on a remote host.

- The instantiation of an object (1) is trapped by the associated interceptor (2), which starts some activities to place the object on a remote host.
- The interceptor checks the state of the network and hosts by querying object `:Adapter` and updates its table of available host workloads (3).
- The interceptor selects the best host according to its allocation policy, and updates a table, held by object `:Locator`, maintaining the position of each remote object (4).
- The interceptor sends the class of the object to be instantiated remotely to the remote host (5,6); the

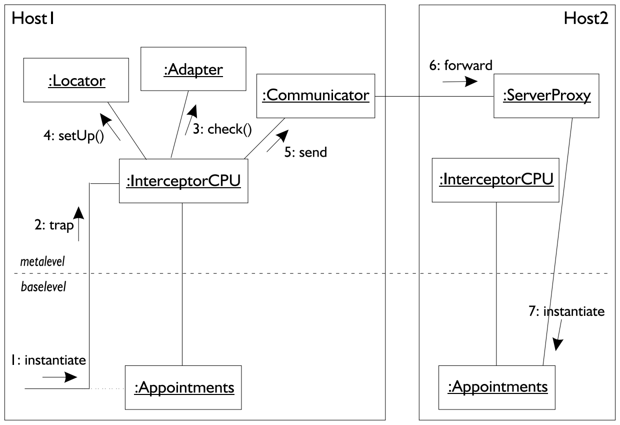

Figure 3. Creating a remote object

transmission is handled by `:Communicator` and `:ServerProxy`. Object `:ServerProxy`, on the remote host, listens for any incoming communication. The class can now be instantiated at the remote host (7). (The first time this happens, the class bytecode must have been shipped in advance by the `:Communicator`).

3.4. Adapter

An adapter is responsible to detect changes in the state of the network (i.e. hosts and connections) and to handle such changes, thus adapting the application to different network configurations dynamically. When a network connection or a host becomes unavailable or their performance degrade excessively, some objects will become effectively unusable. The adaptation then consists in moving the objects to different hosts. This is particularly useful in networks whose condition can change remarkably at run time.

An adapter periodically measures the workload of each host and the performance of the network connection. Since a host may become unavailable because of unexpected crashes or performance changes, we need to measure workload rather than estimate it. The workloads of hosts are measured by running a test on each host, and then communicating the results periodically to peer adapters on other hosts. The workload is measured using the concept of *relaxation* [4], which consists in calculating the execution time of a specific sample process. By using various resource types, such as I/O and CPU, its execution time provides a measure of the workload of the whole host. Thus, a relaxed host is one that makes processes evolve slowly, and the measurement of its relaxation gives a measurement of its workload. This measurement has proved to be richer of information and more effective than operating system workload data, such as the number of processes, etc.

Analogously, connecting to the host may be difficult for a certain time frame due to network interruption or traffic. The state of the network connection is estimated by noting the time frame necessary to establish a connection with a remote host; when this frame exceeds a threshold the host is considered to be unavailable for allocations of new objects.

An adapter runs as a thread that periodically checks the workload of the local host, sends results to peer remote adapters, and assesses network connections based on the results received. For this aim, an adapter holds the information shown in table 2 about the state of the hosts. Each table entry provides, for each host, the `hostname`, its `workload`, `tcheck`, the time when the host measured the workload with its own clock, `tdown`, the duration of the time frame over which a host has been down, and `ok`, a flag indicating whether the host is currently available. The flag is set when the `tdown` value exceeds a threshold.

Table 2. Adapter table with the state of hosts

hostname	workload	tcheck	tdown	ok
⋮	⋮	⋮	⋮	⋮
hostname	workload	tcheck	tdown	ok

If a remote host cannot be reached, new objects that replace those become unreachable have to be instantiated on other hosts. One of the hosts still alive is selected to reallocate objects and it is responsible to instantiate the missing objects. The instantiation is captured by the associated interceptor that decides where they will be created. The last observed state available of the missing objects is used to set the state of these newly created ones.

The adapter and locator tables are sent to remote hosts whenever an object is sent for allocation, or when a timeout has expired for the purpose of updating the global state.

3.5. Communicator

On each host, a unique communicator dispatches all invocations for remote objects to a server proxy, and handles the delivery of a class that has to be instantiated remotely. When delivering a class for remote allocation, the communicator also sends the bytecode of the metalevel classes of the architecture and the local locator tables, since the remote host needs to handle method calls and instantiations of objects, as described earlier.

The communicator connects with a server proxy that first instantiates the object and then invokes its methods, as directed by the interceptor. Moreover, it communicates the results of these operations back to the caller, or a failure indication (through a predefined return value/raising an exception) if a communication time-out expires.

The workload introduced by a communicator is very limited, since its tasks are communication- rather than computation-oriented.

3.6. Server Proxy

For each host, a server proxy is used to receive, from a remote host, invocations to be propagated to the local object, and to send invocation results back to the caller. It continually executes a thread that listens whether some host wishes to communicate. When this happens a server proxy will establish a connection, receive invocations and pass them to the proper local object, send results back, check whether the communication has been successful, and close the connection.

The workload introduced by a server proxy on its host is very limited, since its task only consists of communicating through the network with its peer, and, locally, invoking operations on the object it represents.

3.7. Locator

Class `Locator` is responsible to store information about the location of each object of an application, and to keep a copy of state variables of objects on its host. The location of objects is useful to decide where to redirect method invocations. The copy of state variables is used when relocating objects, in case of network failure.

4. Discussion

Within the proposed reflective framework for handling distribution, objects constituting a software system are clearly separated from, and independent of, distribution concerns, which are addressed only by means of metalevel objects. Separation and independence allow both functional and distribution concerns to be developed and evolved without interfering with each other.

Thus, the development of a distributed software system is facilitated since it is achieved by separately developing support for distribution and application. When the development of an application is complete, it is then possible to transform it into a distributed version simply by associating metaobjects with objects, without any need to modify any of these. The association provides ways to tailor a specific allocation policy to the need of each class of the application.

The proposed reflective architecture can be expanded to address more general needs for distribution than those we have discussed here. For example, new types of interceptors can be derived to decide the allocation according to the other resources needed or specific constraints about an object (i.e. network performance, geographic position, security issues, etc.). The task of choosing the appropriate type of allocation policy, which is currently delegated to the programmer, could be also supported by a tool that carries out an analysis of the source code of classes and proposes/decides association of objects and metaobjects.

In terms of performance, the overhead introduced by the proposed software architecture is only that of jumping to the metalevel. Other costs due to the operations activated from the metalevel should not be considered as an overhead, since they have to be paid anyway if distribution support is desired. Regarding the cost of jumping to the metalevel, in order to evaluate whether its introduction is sensible, we have to compare it with the cost of the operations carried out by a system implementing the same functionalities as the baselevel and metalevel but without reflective mechanisms (i.e. in a more efficient way). When the operations of such a system are computationally costly or time consuming, as remote communication is, the cost of jumping to the metalevel is only a small percentage, thus making the overhead due to reflection not critical.

We are currently using the reflective version of Java, Kava [17], to implement a more complex case study based on the reflective software architecture presented in this paper. With Kava, the association between objects and metaobjects does not require modifying any source code, but is specified by a configuration file. A transformer tool is then invoked that modifies selected application bytecodes to make objects reflective. As a result no injection takes place of any statements into the entities addressing application functionalities, thus achieving a strict separation of concerns. Using OpenJava [15], instead, the programmer is forced to add some keywords to the baselevel code, in order to provide the connection with metaobjects. Moreover, besides the metalevel classes, the metaprogrammer has to write instructions that are used to translate the keywords inserted into baselevel classes. This activity is much more complex than simply writing metalevel classes.

5. An Application to Web and E-Commerce

The proposed reflective methodology can be employed for transparently customising or evolving the service offered in reply to user requests by a web application. Typically, the Java Servlet technology is employed to develop these applications, which therefore consist of various Java classes, including some implementing the *`Servlet`* interface [13]. Following our approach, the objects composing the web application represent the baselevel; of these, only objects instantiating servlet classes need to be associated with suitable metaobjects, intended to handle issues such as request balancing or service customisation. Thus, in keeping with the philosophy of this paper, such concerns can be dealt with transparently for the original web application.

Metaobjects of a class called `InterceptorRequest` are introduced to: (i) intercept and analyse incoming HTTP requests before they are processed by servlet objects, and (ii) decide where requests should be directed. For instance, a request coming from a specific country could be redirected to the "nearest" or less loaded web server. It is also worth noting how this scheme allows new servers to be seamlessly integrated into existing server-side topology (possibly centralised one).

In the specific case of e-commerce web applications, redirection can be useful to better satisfy user needs. E.g. a request intended for a servlet can be intercepted by a metaobject capable of detecting the place where the request comes from and redirecting it in suitable ways. Requests from a certain country could be redirected to a servlet that handles the proper language and currency, or offers goods that are easily shipped to, or appreciated in, that country.

Thus, additional servlets can be developed, each for a specific scenario or need, independently from each other and separately from the base web application. It is up to the metaobject associated with a base servlet, upon intercepting a request for it, to decide which of the available servlets (base or additional) can best handle the request and redirect it accordingly.

6. Related Work

CodA is an architecture that uses metalevel objects to alter the behaviour of baselevel application objects in order to provide a distributed computation model with remote references, replication, migration. CodA defines a set of seven small components: `Send`, `Accept`, `Queue`, `Receive`, `Protocol`, `Execution` and `State` that are present for all application objects. The components are used for an object to, respectively, define how it handles outgoing messages; implement the interface for incoming messages; provide a queue for accepted but not yet received messages; define the operations for receiving messages; map messages and methods to execute; describe the processing resources used for executing methods; and finally to define how a state is stored and retrieved. A message sent from an object to another object is handled at the metalevel by these seven components that provide support for handling distribution for objects that were not intended for a distributed environment [8].

Compared with our architecture, CodA has a similar aim, however it deals mainly with communication issues (message sending, acceptance execution, etc.), whereas our architecture focuses on global management of distributed systems, such as the means to determine how to distribute objects and to adapt to a changing network. Furthermore we focus on facilitating the way in which a centralised application is transformed into a distributed one, making distribution transparent to the application programmer. CodA instead forces the application programmer to know the metalevel objects, since s/he has to explicitly call them.

The Correlate metaobject protocol offers a set of building blocks that represent reified information of the baselevel. Correlate defines two metalevels, each consisting of a set of metalevel objects. A metalevel in Correlate is an autonomous concurrent system that observes the baselevel and acts when appropriate, for example handling interactions between objects. The baselevel is seen as a set of passive objects whose operations can be suspended. One metalevel reifies baselevel objects and the other reifies baselevel interactions. The first metalevel can be used to deal with physical distribution of objects by implementing a metaobject that acts as a proxy for the real metaobject on the remote host, so that a Correlate application can be transformed into a distributed version [11].

Correlate provides support for distributed systems by using some metaobjects that behave as proxies to interconnect objects on different hosts. For its implementation, Correlate requires the insertion of special keywords that extend an existing language and specify structure and interactions of objects. A translator transforms a Correlate program into one that a standard compiler can handle. Correlate simplifies the programming of distributed systems, only in that it supports remote communication; however there is no support for facilitating the transformation of a centralised application into a distributed one, since the programmer has to specify how objects have to be distributed and which parts of the baselevel need to be controlled by the metalevel. Finally, it does not provide any means to adapt to run time changes of the network, nor to choose specific allocation policies for classes.

An AL-1/D system consists of a compiler and a virtual machine that interprets bytecode, running under a Unix-based Operating System. AL-1/D is a distributed reflective programming system that provides means for separating application programs from the handling of remote message sending, migration and allocation policies. It also uses for the purpose of allocating objects [10], run time information on the executing objects, such as the number of remote messages. Like our approach, it favours reuse and evolution of both application programs and allocation policies by enforcing a strict separation of concerns. However, it does not take into account how to adapt the application to run time network changes. Moreover, although allocation policies do take into account run time information, by observing the behaviour of objects, they are not tailored to specific classes, but one policy is adopted for all classes, since the metaobject handling allocation is shared among all the objects.

7. Conclusions

This paper presented a reflective software architecture that provides support for distribution by using metalevel classes that control communication between objects, allocate objects to hosts, enable checking at run time the access to distributed objects and re-locating them. The reflective architecture aims at favouring reuse, evolution, and incremental development of software systems by following the principle of separating concerns. Distribution issues are addressed only by metalevel classes, which are separated from, and independent of, the components addressing functional concerns. A set of suitable allocation policies have been encapsulated into metaobjects, each employed for the specific needs or constraints of classes.

The proposed approach does not require any special compiler or language, except for the reflective ability that has to be provided.

As a future work we are investigating the means to make the association between application objects and different types of interceptors an automated task, driven by an analysis of application objects. Such an analysis can be performed at run time but should not cause noticeable delays, since computational costs can be fragmented and paid only when necessary.

References

[1] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, pages 285–299, 1995.

[2] S. Chiba. Load-time Structural Reflection in Java. In *Proceedings of the ECOOP 2000*, volume 1850 of *Lecture Notes in Computer Science*, 2000.

[3] A. Corradi, L. Leonardi, and F. Zambonelli. High-Level Directives to Drive the Allocation of Parallel Object-Oriented Applications. In *Proceedings of the HIPS'97*, Geneve, Swiss, April 1997.

[4] A. Di Stefano, L. Lo Bello, and E. Tramontana. Factors Affecting the Design of Load Balancing Algorithms in Distributed Systems. *The Journal of Systems and Software. Elsevier*, 48:105–117, 1999.

[5] J. Ferber. Computational Reflection in Class Based Object Oriented Languages. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*, volume 24 of *Sigplan Notices*, pages 317–326, New York, NY, 1989.

[6] W. L. Hürsh and C. V. Lopes. Separation of Concerns. Technical Report NU-CCS-95-03, Northeastern University, 1995.

[7] P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, volume 22 (12) of *Sigplan Notices*, pages 147–155, Orlando, FA, 1987.

[8] J. McAffer. Meta-level Architecture Support for Distributed Objects. In *Proceedings of International Workshop on Object-Orientation in Operating Systems (IWOOOS'95)*, 1995.

[9] M. Nuttal. A brief survey of systems providing process or object migration facilities. *Operating Systems Review*, 28(4), October 1994.

[10] H. Okamura and Y. Ishikawa. Object location control using meta-level programming. In *Proceedings of the 8th European Conference on Object-Oriented Programming*, volume 821 of *Lecture Notes in Computer Science*, pages 299–319. Springer-Verlag, 1994.

[11] B. Robben, W. Joosen, F. Matthijs, B. Vanhaute, and P. Verbaeten. A Metaobject Protocol for Correlate. In *Proceedings of the Workshop on Reflective Object-Oriented Programming Systems at the European Conference on Object-Oriented Programming (ECOOP'98)*, 1998.

[12] R. J. Stroud and Z. Wu. Using Metaobject Protocols to Satisfy Non-Functional Requirements. In C. Zimmermann, editor, *Advances in Object-Oriented Metalevel Architectures and Reflection*. CRC Press, 1996.

[13] SUN Microsystems. Java Servlet Technology. WWW - Internet publication, 2002. White paper. http://java.sun.com.

[14] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[15] M. Tatsubori, S. Chiba, K. Itano, and M.-O. Killijian. OpenJava: A Class-Based Macro System for Java. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*. Springer-Verlag, June 2000.

[16] E. Tramontana. Managing Evolution Using Cooperative Designs and a Reflective Architecture. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*. Springer-Verlag, June 2000.

[17] I. Welch and R. J. Stroud. Kava - A Reflective Java Based on Bytecode Rewriting. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*. Springer-Verlag, June 2000.