Chapter 15

Techniques for Dynamic Adaptation of Mobile Services

John Keeney, Vinny Cahill, Mads Haahr

Contents

Introduction

This chapter discusses the dynamic adaptation of software for mobile computing. The primary focus of the chapter is to discuss a number of techniques for adapting software as it runs, and managing the application of those adaptations. In a mobile computing environment the need for adaptation can often arise as a result of a spontaneous change in the context of the operating environment, ancillary software, or indeed the user. To exacerbate this problem, if that contextual change is in some way unanticipated, then the required adaptation may be itself unanticipated until the need for it arises. For this reason, this chapter is particularly concerned with supporting adaptations that are "completely unanticipated" [19]. The chapter discusses reflective and aspect-oriented techniques for dynamically adapting software for mobile computing. Policy-based management is then discussed as a mechanism to control such dynamic adaptation mechanisms. The chapter then introduces the Chisel dynamic adaptation framework, which supports completely unanticipated dynamic adaptation, and discusses a case study whereby Chisel is used with ALICE, a mobile middleware, to provide a flexible and adaptable middleware framework for mobile computing.

Issues in Dynamically Adaptable Mobile Applications and Middleware

The main difficulty with mobile computing is the inherent scarcity and variability of resources available for use by mobile computers as they move. The primary resource requirement of a mobile device when it is working as part of a distributed system is its network connection, often some form of wireless connection, which when used by a device that is physically moving, suffers from unanticipated and possibly prolonged disconnections [14]. The reason why this issue is such a major problem for mobile computing is that the applications currently being developed are being built as distributed systems applications that do not sufficiently account for these disconnections and reconnections [30].

Middleware for Mobile Computing

"Middleware can be viewed as a reusable, expandable set of services and functions that are commonly needed by many applications to function well in a networked environment" [1]. Traditional middleware systems provide abstractions and shelter applications from the complexities of the underlying environment, communication subsystems, and distribution mechanisms, thereby providing a single view of the underlying environment, as seen in traditional middleware systems such as COM+ [24] Java RMI [39] and CORBA [25].

A middleware system for mobile computing must be flexible in order to provide a homogeneous and stable programming model and interface to possibly erratic execution contexts. It is desirable that an adaptable middleware for mobile computing be open, allowing the application and the user to inspect the execution environment and manipulate the application and middleware in a mobile-aware manner, using application-specific and user-specific semantic knowledge.

Difficulties with Applications and Middleware for Mobile Computing

As environment conditions change, to values unknown and unprecedented by the application designer, the middleware that provides abstractions for these environmental resources must dynamically adapt to support the applications that run on top of that middleware. As stated, one of the primary services provided by the middleware is the ability to supply network communications services as these resources change. A key requirement for middleware for mobile computing is the ability to adapt to drastic changes in available resources, especially network connection availability [15]. The characteristics of the available connections can range from an inexpensive, very high-bandwidth, low-latency connection such as a high-speed wired LAN connection, to a very expensive, low-bandwidth, high-latency connection such as a GSM connection, where each communication protocol used may make use of different communication models and addressing modes.

Mobile computing applications should also be able to handle periods of disconnection, supported by the middleware underneath. The difficulties that are associated with such a range of connection characteristics are further compounded by the fact that these characteristics can change in an unanticipated manner. For example, these disconnections occur when the device moves out of range for wireless connections, or an interface device is suddenly disconnected, as seen when a user suddenly disconnects the device from a synchronization cradle or removes a networking device currently in use. A further issue with such a varied collection of communication technologies that can be leveraged for mobile computing is that the user may not wish to fully use the available resources in an eager or greedy manner to maintain data connectivity. For example, even if a GPRS connection is available, this connection is generally much more expensive than available wireless connections. A further example is the case where although currently disconnected, with connections available, the user may be aware that cheaper or more convenient connection resource will soon be available, i.e., something that cannot be anticipated in a generalized manner by the adaptable middleware platform. For these reasons, it is imperative that the added potential of the user's own resources, preferences, and intelligence are exploited.

Reflective Middleware

Principals and Key Ideas

A reflective computational system is one that reasons about its own computation. This is achieved by the system maintaining a representation (metadata) of itself that is causally connected to its own operation, so that if the system changes its representation of itself, the system adapts [22]. With behavioral reflection in an object-oriented system, the reflective system reasons about and adapts its own behavior by associating meta objects with the objects in the application, where the meta objects control or adapt the behavior of the application objects [12]. In a reflective system, the communications between the meta objects and base objects take place through a set of well-defined interfaces, referred to as that system's meta object protocol (MOP) [20].

Case Studies of Reflective Middleware

Although a number of reflective middleware frameworks are discussed in detail in previous chapters, this section discusses two additional reflective systems which target middleware for dynamic adaptation. In addition, a number of systems described later in this chapter make use of reflective techniques, but are discussed under a different category.

ACT

ACT [35, 36] is a generic adaptation framework for CORBA compliant [25] ORBs that supports unanticipated dynamic adaptation. When the ORB is started ACT is enabled by registering a specific portable request interceptor [25], intercepting every remote invocation request and handing them to a set of dynamically registered interceptors. These dynamically registered interceptors can be added in an unanticipated manner. Rule-based dynamic interceptors allow the request to be redirected to a different source or handed to either a number of local proxy components exporting the same interface as that of the destination server component [35] or to a generic local proxy component [36]. This generic proxy component can also be dynamically created in an unanticipated manner. This proxy in turn can request a rule-based decision making component, which can incorporate an event service, to either perform the invocation, or change parameters and forward the request to either its original destination or a different destination.

A prototype is described whereby the Quality Objects (QuO) framework [2], an aspect-oriented QoS adaptation framework for CORBA ORBs, was used with a CORBA-compliant ORB, to support completely unanticipated runtime aspect weaving in the ORB. A number of management interfaces were also provided to manage the runtime registration of new rule-based dynamic interceptors, and the addition of new rules to these interceptors.

Correlate

Presented by the DistriNet research group in Katholieke Universiteit Leuven, Correlate [16, 33, 34, 40], is a concurrent object-oriented language based on C++

(and later Java) to support mobile agents. It has a flexible runtime engine to support migration and location independent inter-object communication. Each agent object has an associated meta object that can intercept creation, deletion, and all invocation messages for the object. This system allows non-functional aspects of the application to be separated from the application object. The non-functional behaviors are designed to be largely application independent; however, independent policy objects can be defined to contain application-specific information to assist in the operation of these meta-level non-functional behaviors. The meta-level system was initially used to implement non-functional concerns such as real-time operation, load-balancing, security, and fault tolerance. Later this system was used to customize ORBs, using application-specific requirements, as an adaptable graph of meta-level components that could be extended or adapted at runtime.

The application-independent non-functional behaviors are implemented as meta object classes, which can interact with the base program to adapt its operation using a message-based MOP. These meta object classes define a set of possible property values in a policy template. Each application class has an associated singleton policy-class object, which is an instantiation of these templates and contains application-specific information. These singleton policy-class objects are consulted by the meta-level before performing the non-functional behaviors of the application, allowing the operation to be customized in an application-specific manner.

However, this policy system is limited since policy templates are imposed at the time the meta program is written. These templates, written in a declarative language, must fully define what possible customizations an application may require at a later stage. The policies, also written in the same declarative manner, select values for template properties according to the application class with which they are associated. These templates cannot be changed, so adaptation in response to unanticipated requirements cannot be fully handled. Policies are written before runtime by a system integrator, and these policies are then translated to code and compiled with the application and so cannot be changed at runtime. Unanticipated forms of dynamic adaptation cannot be achieved in this architecture as the meta-level programmer and template designer needs complete a priori knowledge of the possible changes in context values that may occur, and also the set of customizations from which the meta-level can choose is fixed at compile time.

Discussion

The use of reflective mechanisms for adaptable middleware is an old yet active research area. The main issue with reflection for the adaptation of middleware lies not with the use of reflection to adapt the structure, behavior, or architecture of middleware but rather with how the application of those adaptations are controlled and managed. This issue is of particular importance if the adaptation is required in response to an unanticipated change in the state, requirements, or context of the users, applications, or environment.

Aspect-Oriented Approaches to Dynamic Adaptation

Principals and Key Ideas

Aspect-oriented programming (AOP) [13, 21] is a programming methodology that allows cross-cutting concerns to be declared as "aspects". A cross-cutting concern is a property or function of a system that cannot be cleanly declared in terms of individual components, because the application of the cross-cutting concern must be scattered or distributed across otherwise unrelated components. AspectJ [42], the de-facto standard for AOP, introduced the concept of an aspect as a language construct, used to specify a modular unit to encapsulate a cross-cutting concern, which is then "woven" into the application code at compile time. An aspect is defined in terms of "pointcuts" (a collection of "join point" locations within the application code where the aspect should be "woven", and conditional contextual values at those join points), "advice" (code executed before, after, or around a join point when it is reached), and "introductions" (Java code to be introduced into base classes) [42].

AOP supports the production of these aspects in a manner that is separate or "oblivious" [13] to the application components, into which the aspects are later incorporated or woven at a specified or quantified set of join points. "Obliviousness", one of the key components of AOP, refers to the degree of separation between the aspects of the system and how they can be developed independently without preparation, cooperation, or anticipation. Most AOP systems support weaving before runtime, but newer dynamic AOP systems (e.g., Wool and PROSE) described in this section allow aspects to be woven at load-time or runtime, thereby allowing the incorporation of aspects into base programs to remain unanticipated until load-time or runtime.

Case Studies of Dynamic Aspect-Oriented Systems

Wool

Wool [38] is a dynamic AOP framework that uses a hybrid aspect weaving approach by using both the Java Platform Debugger Architecture (JPDA), and the Java HotSwap mechanism [39]. Since JPDA supports remote activation of breakpoints at runtime, join point hooks in the form of debugging breakpoints can be dynamically set from outside of the application. A pointcut may be made up of a number of these hooks. Each aspect specifies a pointcut, and a set of advices to be executed when one of the pointcut's join points (represented as breakpoints) is reached.

New aspects can be serialized and sent to the target JVM for weaving at any pointcut. In one approach, when a join point is encountered, the inserted breakpoint redirects the operation to the Wool runtime component in a manner similar to a debugger, where advices are then executed. The alternative approach allows the advice to be hotswapped into the application class thereby improving performance if the join point is encountered repeatedly. This is achieved by using Javassist [7] to rewrite the class, without access to its source code, and have the adapted class replace the original application class using the Java HotSwap mechanism. This also removes the breakpoint, so calls to the debugger are removed. However, this mechanism means that all objects of the woven class will have the adaptation incorporated, and so individual objects cannot be adapted. Currently the aspect programmer must specify in the aspect's source code whether the advice should be woven by the HotSwap mechanism or by the debug interface, so in order to achieve good performance, the aspect writer should anticipate the access patterns of the aspect's pointcut. Wool does not support adding introductions but a proposed solution is provided.

PROSE

PROSE [26, 29] is another dynamic AOP framework for Java that supports runtime aspect weaving. PROSE was originally intended as a framework for debugging or rapid prototyping of AOP systems, which could later be completed using compile-time or load-time aspect weaving [29]. This was mainly due to its use of the Java Virtual Machine Debug Interface (JVMDI) [39], which resulted in a large performance penalty. A later version of PROSE [26] was implemented by modifying an open source JVM, greatly improving its performance. In both versions, new aspects can be to dynamically woven, with support for these aspects to define new join points, for which new interception hooks are created at weave time, thereby allowing PROSE to be used to support dynamic adaptation by weaving additional non-functional behaviors into the code at runtime.A number of graphical user interfaces are included to manage the unanticipated weaving of new aspects at runtime. However, like Wool above, PROSE only supports weaving at a class level; therefore individual objects cannot be adapted individually.

MIDAS [27], implemented as a Spontaneous Container [28], is a middleware for the management of PROSE extensions which provides a distributed event-based system for the dissemination and management of aspects from a central server to mobile computers based on their location.

TRAP/J

TRAP/J [37] is a prototype unanticipated dynamic adaptation framework for Java. It combines compile-time aspect weaving using AspectJ [42] and unanticipated dynamic adaptation with wrapper classes and delegate classes. At compile time the programmer selects a subset of application classes that will be adaptable. The TRAP/J system then automatically creates AspectJ code to replace all instantiations of the selected classes with wrapper class instantiations. Java code for each wrapper class and a meta object class for that wrapper class is also automatically created. At runtime, each instantiated wrapper object has an instance of the original wrapped object and a meta object bound to it. These wrapper objects redirect all method calls to their meta object, which in turn act as placeholders for a set of delegate objects that may handle the invocation of the method, or adjust its parameters prior to execution by the original wrapped object. New, dynamically created delegates can be added or removed at runtime via an RMI [39] interface using a management console. These delegates can be added on a per object basis since the meta objects can supply a name for each instance and register it in an RMI registry.

This framework was used to demonstrate the dynamic adaptation of a network-enabled application by replacing instances of the *java.net.MulticastSocket* class with instances of an adaptable socket class *MetaSocket* [18]. The TRAP/J framework however does not support completely unanticipated dynamic adaptation. The adaptation, its intelligent and controlled dynamic application, and the timing of its application all remain unanticipated until runtime, but the possible locations for the adaptations are specified in the application source code, since the version of AspectJ used requires access to the application source code. Despite improving the performance of the TRAP/J framework, this restriction greatly limits the nature of the unanticipated dynamic adaptations that can be applied. No information is provided about whether the generated meta object class code can be modified prior to compilation and weaving.

In addition, TRAP/J seems to delegate the invocation of the method to only one delegate; the first one it finds implementing the method, but this ordering of delegates can be configured. This means that only one adaptation can be applied at a time since adaptation behaviors are not automatically composed. In addition, TRAP /J does not seem to allow the user to apply an easily recognizable name to the base object being adapted, and so may make it difficult for the user to identify the object to which adaptations should be dynamically applied. From the documentation TRAP/J does not seem to support applying dynamic adaptations via new delegates on a structured class-wide or interface-wide basis since RMI registry lookups are at a per meta object basis. Unlike Wool and PROSE above, which only support the adaptation of classes, TRAP/J only supports the adaptation of individual objects at any one time.

Discussion

Dynamic AOP technologies would appear to be a promising area of research for dynamically adaptable middleware. Not only can aspects be used to implement non-functional concerns within the middleware but also to adapt or augment the functional behavior of the middleware [21]. This ability to dynamically adapt functionality or inject new functionality at clearly defined join points is of particular importance to middleware for mobile computing since dynamic and possibly unanticipated adaptation requirements are typical for mobile computing. The "separation of concerns" model of aspects reduces the difficulty of incorporating adaptations into complex middleware frameworks since the introduced cross-cutting concerns can be targeted correctly to the location requiring adaptation.

However, current dynamic AOP methodologies such as Wool, PROSE, and TRAP/J are lacking a structured mechanism to dynamically specify these locations for dynamic adaptation, and how these adaptations should be applied, after the target software has started execution in a manner that incorporates user, application, and environmental context at runtime. Despite this, this area of dynamic AOP based dynamic adaptation of middleware is proving to be an active area of research and should quickly provide a number of solutions to this issue.

Policy-Based Management of Dynamic Adaptation

Principals and Key Ideas

Many traditional adaptable systems are composed of a single adaptation manager that is responsible for the entire adaptation process; i.e. monitoring, adaptation selection intelligence and performing the actual adaptation. Since the intelligence to select appropriate adaptations and the mechanism to perform these adaptations are embedded directly within the adaptation manager, this type of system becomes inflexible and inappropriate for general use. By decoupling the adaptation mechanism from the adaptation manager, and removing the intelligence mechanism that selects or triggers adaptations, the adaptation manager becomes more scalable and flexible. Policy specifications maintain a very clean separation of concerns between the adaptations available, the adaptation mechanism itself, and the decision process that determines when these adaptations are performed.

Policy specification documents are usually persistent text-based declarative representations of policy rules that ideally can be read, understood, and generated by users, programmers, and applications. A policy rule is defined as a rule governing the choices in behavior of a managed system [8]. Informally, a policy rule can be regarded as an instruction or authority for a manager to execute actions on a managed target to achieve an objective or execute a change.

An adaptation policy rule is usually made up of an event specification that triggers the rule, which is often fired as a result of a monitoring operation; an action to perform in response to the trigger; and a target object that is part of the managed system upon which that action is performed [8]. Many policies will also contain some restrictions or guards confining the rule action to appropriate occasions. This *event-condition-action* (ECA) format is standard for rule-based adaptation systems [4, 5, 6, 8, 9, 16, 19, 33, 34, 35, 36, 40], where an adaptation management system is responsible for monitoring these events, evaluating the conditions and initiating the management action on the targeted managed object. In a policy-based dynamic adaptation system it should be possible to edit the rule set and have them re-interpreted to support the dynamic addition of new rules or changes in policy.

Case Studies of Policy-Based Middleware

This section discusses two systems that employ policy-based management techniques to manage dynamic adaptation of middleware, but additionally the ACT, TRAP/J, and Correlate systems could also be described in terms of their use of policy rule-based techniques. A number of mechanisms discussed in other chapters could also be discussed in terms of their use of rule-based management mechanisms.

RAM

RAM (Reflection for Adaptable Mobility) [4, 9] from École des Mines de Nantes, takes the approach of completely separating functional and non-functional aspects of an application in a manner related to aspect oriented programming (AOP). Using this separation of concerns approach, only the core application functionality is inserted into the application code, with all middleware services represented as non-functional concerns. *Container* meta objects wrap each application, and supports the compositions of other meta object which implement these non-functional concerns. The wrapping of application objects with *Container*s occurs at either load-time using Javassist [7] in [4] or at compile-time using AspectJ [42] in [9]. These meta objects provide the middleware services by selecting appropriate *RoleProvider* objects for each service, i.e., the meta objects that provide the actual implementations of the services. Adaptation can occur by adding, removing, or reordering these *RoleProvider*s.

RAM also provides a resource manager, whereby the system maintains a tree of *MonitoredResource* objects, which describe a contextual resource or group of resources. These *MonitoredResource* objects are updated by *probe* objects that actively monitor the environment. *MonitoredResource* objects can be queried explicitly or alternatively by requesting change notifications to signal the adaptation engine when an interesting resource change occurs. The *Container* meta objects, that wrap each application component, can also expose the *MonitoredResource* interface, supporting queries of application context as resources, thereby exploiting application-specific knowledge in the adaptation process.

The set of meta objects (aspects) to use in each *Container* is adapted at runtime by means of an adaptation engine that uses both an application policy and a system policy, both written in a declarative Scheme-like language, and which are both passed to the adaptation engine when the application is started. The application policy defines pointcuts (a dynamic set of join points, i.e., *Container* objects) in the application, and the named non-functional aspects to be used at these pointcuts, in an application-aware but resource-independent manner. The set of rules that determine which join points make up a pointcut is also specified in the application policy, but these rules are dynamically evaluated, so this set of join points can change dynamically. The non-functional aspects woven at these pointcuts are defined in the system policy in an adaptive Condition-Action model, where sets of application-independent but resource-aware conditions are dynamically evaluated to decide which meta objects will implement the non-functional aspect. When the conditions are dynamically evaluated, the bindings of meta objects can be changed, in a manner similar to dynamic aspect weaving. Therefore, the set of join points that make up a pointcut, and the set of meta objects that implement an aspect can both be dynamically specified according to the rules in the policies. The current system does not support dynamic changes to the policies, and so cannot support unanticipated adaptation management logic; however this is planned for future versions. In most cases where AspectJ is used, access to the source code of the application is also required. A version of RAM suggests using a configuration file to specify the set of join points that can be used, and use AspectJ to create these join points at compile time rather than have *Container*s wrap every application object [11]. This means, however, that all possible locations for adaptation must be anticipated at compile time, and requiring access to the source code of the application. Preliminary designs for an adaptation framework extending RAM, which would possibly support completely unanticipated adaptation by allowing dynamic specification of policies and dynamic selection of adaptation locations, is presented in [10], but this system has yet to be implemented.

CARISMA

Research carried out at University College London on the CARISMA project [5, 6] presents a design for peer-to-peer middleware based on service provision. Each node can export services and possible different behaviors or implementations for those services. Services can be selected according to user and application context information, as specified in an "Application Profile", an XML policy document. Embedded in this application profile is the application-specific information that the middleware uses when binding to these services, e.g., which service behavior to use in response to changes in the execution context. The middleware is responsible for maintaining a view of the system environment by directly querying the underlying network-enabled operating system. Applications may request to view and change their profiles at runtime, thereby adapting the middleware as application-specific and user-specific requirements of change dynamically.

This system also provides the ability for the application to be informed by the middleware of changes in specific execution conditions, supporting the development of resource-aware applications. This system is based on the provision of multiple implementations of the same service with different behaviors, in a manner similar to the Strategy Design Pattern rather than adapting the service itself. The primary contribution of this work focuses on the identification and resolution of profile conflicts [6], and not on the actual provision of an adaptable middleware implementation. No information is provided about how the services are implemented, if they can be dynamically loaded, how they implement their different strategies, or if these strategies can be expanded at runtime. However, it should be noted that the application profile that controls how the system adapts, and the mechanism for profile conflicts, can both be adapted at runtime in an unanticipated manner. XMIDDLE [23], which appears to form the basis for CARISMA, is a peer-to-peer data sharing middleware for mobile computing. In XMIDDLE, data is replicated as XML trees pending disconnections, with these trees reconciled when possible in a policy-based manner according to application specific conflict resolution data embedded in the shared data structures.

Benefits of Policy-Based Management of Dynamic Adaptations

An adaptable system that has its adaptation logic encoded directly into it cannot operate in a general-purpose manner or adapt in response to unanticipated changes, as often arises with an enabling technology such as middleware operating in an environment where the operating context changes erratically, as seen in a mobile computing environment. The use of a policy-based control model allows the clean decoupling of adaptation logic from the adaptation mechanism used by the adaptation framework.

The control logic to manage the dynamic application of an adaptation must be capable of specifying what adaptation should be applied, where and when it should be applied, and conditions to restrict the application of the adaptation if necessary. Since many dynamic adaptations are necessarily required because some state, resource, or requirement has changed for the user, application, or execution environment, this dynamically specified control logic must also support the querying of this runtime context. Using dynamic loading and interpretation of policy directives can also be used to support the management of new unanticipated adaptations, by allowing those new adaptations to be referred to dynamically, along with where they should be applied and what management logic should be used to control how and when those adaptations are applied.

Chisel and ALICE: A Policy-Based Reflective Middleware for Mobile Computing

This section describes the Chisel Dynamic adaptation framework, and how it can be used with ALICE, a middleware for mobile computing, to create a dynamically adaptable middleware, which can be used to adapt a standard network application in an unanticipated manner to operate in a mobile computing environment.

Chisel

The Chisel dynamic adaptation framework [19], developed in Trinity College Dublin, supports the application of arbitrary completely unanticipated dynamic adaptations to compiled Java software, as it runs. An adaptation is "completely unanticipated" if the behavioral change contained in the adaptation, the location at which that adaptation is to be applied, the time when that adaptation will be applied, and the control logic that controls the application of the adaptation, can all remain unanticipated until after the target software has started execution [19].

The adaptations are achieved by dynamically associating Iguana/J metatypes [31, 32] with any application object or class and so changing their behavior on the fly, without regard to the type of the object or class, and indeed without access to its source code. The metatype of a class or object represents some coherent internal behavior change from its original source code behavior [31], i.e., a behavioral change associated with the class or object. In Iguana/J metatypes are implemented using custom MOPs, i.e., by deciding which parts of the object model to reify, writing a set of meta object classes for these reifications to implement the new metatype behavior, then associating that metatype implementation with an object or class. In the Iguana literature, the terms "metatype association" and "MOP selection" are similar and refer to this association of MOP implementations to objects and classes. This association mechanism is performed using runtime behavioral reflection techniques, whereby selected parts of application objects and classes are reified and intercepted, and the new metatype behavior inserted at this interception point. Iguana/J supplies the framework to instantiate these meta objects to reify the object model, and correctly order metatypes if more than one is selected. Iguana/J provides a mechanism to associate new metatypes with objects and classes at runtime, thereby changing the behavior to the system on the fly.

The execution of a new behavior embedded in the meta level can then occur alongside or around the original behavior of the target object, by wrapping the behavior of the target object and adapting or tailoring the intercepted operation, or by introducing the new behavior before, after, or instead of the intercepted operation. New metatypes can be defined at any time and compiled offline using the Iguana/J metatype compiler, even as a target application is running. In this way the adaptations to be applied can remain unprepared and unanticipated until it is needed. When a metatype is associated with a class the behaviors that are changed are both the "static" behaviors of the class, the behaviors of each current and future instance of the class, and the behavior of all subclasses and their current and future instances. Here static refers to the behavior and data embedded in a class, instead of in each of its instances. For example, static methods, static data fields, and class initialization procedures, implemented using the `static` keyword in Java and C++.

The dynamic associations of these metatypes are driven by a dynamically specified and interpreted policy script. Using this policy script, the user can specify which classes or named objects should be adapted, either in a proactive manner, or in a reactive event-based manner. The Chisel policy language, described in detail in [19], also supports the dynamic definition of new event types for use in reactive rules. In addition, the Chisel policy language allows events to be dynamically fired by other rules or in response to changes in dynamically specified contextual conditions. In this manner, the timing and control logic for any dynamic metatype association can remain unspecified until during runtime, and so remain unanticipated. By dynamically creating a new policy, specifying which class or object to adapt, and specifying which named metatype to associate, the location of the adaptation can also remain unanticipated until during runtime.

Together, this use of runtime behavioral reflection and runtime specification and interpretation of adaptation policies, allows the Chisel framework to support the completely unanticipated dynamic adaptation of any running Java application, without stopping it, and without access to its source code.

ALICE

ALICE [3, 15, 41], also developed in Trinity College Dublin, is an architectural middleware framework that supports network connectivity in a mobile computing environment by providing a range of client/server protocols (Figure 15.1). In ALICE, "mobile hosts" are mobile devices, which may interact with fixed computers

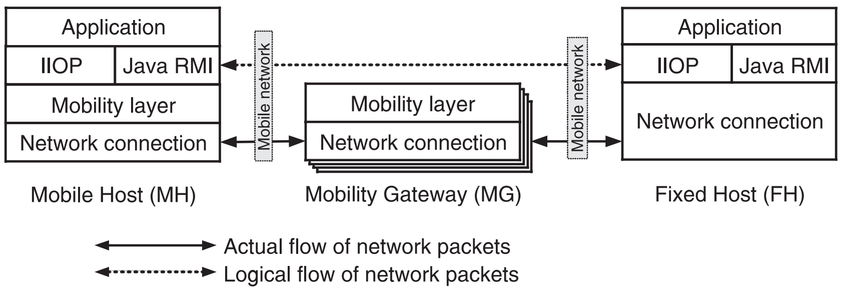

Figure 15.1 Overview of the ALICE middleware framework.

called "fixed hosts". These connections are tunneled through "mobility gateways", which are also fixed computers. The mobile host can become disconnected from a mobility gateway and later become reconnected to a different mobility gateway without interfering with the virtual connection to the fixed host.

The ALICE mobility layer handles communications between devices by overriding socket functions while hiding which communication interface is being used for the connection. The mobility layer tracks available connections and picks one using a reconfigurable selection algorithm. When a disconnection occurs, the ALICE mobility layer will synchronously queue unsent data between the mobile host and the mobility gateway until a connection is re-established.

For this case study a full Java implementation of the ALICE mobility layer was completed, based on the work presented in [41]. It provides a class `MASocket` that contains the ALICE connection behavior, which implements a socket interface similar to the standard Java socket class, `java.net.Socket.` When the `MASocket` class is used instead of the standard Java socket, all messages from a mobile host to a fixed host are redirected via a mobility gateway. When the connection between the mobile host and the mobility gateway breaks, all network data is cached at the mobile host and the mobility gateway for later reconnection. This disconnection and reconnection happens without the application being made aware of the disconnection.

Chisel and ALICE

To demonstrate the use of the Chisel dynamic adaptation framework, an off-the-shelf application, "The Java Telnet Application/Applet" [17], was adapted to operate in a mobile computing environment by dynamically adapting it to use the ALICE mobility layer, all without stopping the application and without changing or requiring access to the source code of the application in any way. The only initial assumption made about the internal programming of the application was that a standard Java socket, or a subclass of `java.net.Socket`, is used to connect the client and the telnet server, a reasonable assumption for any network enabled Java application.

A metatype, `DoAliceConnection` was developed to intercept the creation of the socket connection to the telnet server and replace the socket in use with an instance of the ALICE `MASocket.` The metatype definition below specifies that the reified creation of objects should be intercepted and handled by the `MetaObjectCreateALICEConn` metaobject class.

```
protocol DoAliceConnection {
reify Creation: MetaObjectCreateALICEConn();
}
```

This redirection behavior was embedded in the meta object class `MetaObjectCreateALICEConn` as shown below. This redirection behavior is achieved by intercepting the creation of all socket objects, and if the connection is a not localhost connection or one used by ALICE, then by the use of the Java reflective API the `java.net.Socket` constructor is replaced by the `MASocket` constructor. The application would be completely unaware of the change since the returned `MASocket` is extended from `java.net.Socket` and exposes the same interface.

```
class MetaObjectCreateALICEConn extends ie.tcd.iguana.MCreate {
  public Object create(Constructor cons, Object[] args) ... {
    if(/*not a localhost connection, or a connection used by
        ALICE */){
        //Change the constructor, from java.net.Socket to MASocket
        cons = (Class.forName ("MASocket")).getConstructor (...);
    }
    Object result = proceed (cons, args); /* create the socket */
    return result;// result is either a normal socket or an MASocket
  }
};
```

This adaptation was then applied to the telnet application in a number of ways using the Chisel policy language [19]. One method was to apply this adaptation in a context-aware manner, i.e., only perform the metatype association if the application was being used in a mobile computing environment, where the network connection was known to be intermittent. In the adaptation policy rules seen below, the `DoAliceConnection` metatype is only associated with the `java.net.Socket` class if the `UsingDodgyNet` event fires. When the connection moves to a stable network connection the `UsingGoodNet` event is fired, thereby re-enabling the use of standard Java sockets.

```
ON UsingDodgyNet java.net.Socket.DoAliceConnection
ON UsingGoodNet java.net.Socket.NullProtocol
```

The event `UsingDodgyNet` could be fired automatically by the Chisel event manager using an automatic rule definition and trigger rule, or by the Chisel context manager when a wireless connection was detected, or by the user using another event manipulation policy rule, etc. Similarly, the `UsingGoodNet` event could be fired when the network connection is deemed stable, or by another policy rule, by some network monitoring code, or by the context manager. In [19], a number of methods are presented to describe how these events could be defined and automatically triggered in an unanticipated manner.

Findings and Further Adaptations

This case study was fully implemented and functions as expected. This case study demonstrates the use of the Chisel dynamic adaptation framework to adapt an arbitrary application in a context-aware manner for use in a mobile computing environment, without accessing its source code. The telnet application was not prepared in any way to have the particular adaptation applied. Only when the adaptation was deemed necessary did the user need to create a set of adaptation rules, similar to the ones above, embedding any necessary context information. Only when these rules triggered the application of the adaptation would the adaptation be needed so it could be loaded and applied to the unprepared location deep inside the compiled application, without any requirement to change, interrupt, or restart the application. This case study also demonstrates how the operation of a complex compiled application was changed dynamically according to the environment and user's needs.

Using the Chisel framework further adaptations are also made possible, to both the application and the ALICE middleware framework. This mechanism of dynamically redirecting Java socket connections to ALICE `MASocket` socket connections could also be used to dynamically adapt the Java RMI middleware model similar to the approach discussed in [3, 15], but in an unanticipated manner. This possible approach could enable the adaptations described in [3], by intercepting the instantiation of both the `java.net.Socket` and `sun.rmi.server.UnicastRef classes.` An alternative approach could intercept the operations of the `java.rmi.server.RMISocketFactory` interface when it is requested to create the actual sockets used to perform remote object invocations, as described in [41].

Although a mobile computing scenario was chosen to demonstrate the Chisel dynamic adaptation framework, this case study equally applies to any environment or operation mode where unanticipated dynamic adaptation is required for satisfactory operation. A mobile computing environment was seen as a perfect example since the state, resources, and requirements of the application, the environment, and the user can all change to extreme values in an unanticipated manner.

Conclusions

This chapter has presented a discussion of dynamic adaptation for mobile middleware. The chapter began with a discussion of how unanticipated dynamic adaptation of applications and middleware are required in a mobile computing environment. A number of reflective and aspect-oriented techniques for dynamic adaptation were discussed, paying particular attention to support for unanticipated dynamic adaptation. The chapter then discussed the use of policy-based management to control unanticipated dynamic adaptation in a manner that was itself dynamically adaptable. The chapter then continued with an introduction to the Chisel dynamic adaptation framework. Chisel was then discussed in terms of how ALICE, a middleware for mobile computing, could be used to adapt an off-the-shelf network application to operate in a mobile computing environment in a completely unanticipated manner.

References

[1] Aiken, R., *et al.* Network Policy and Services: A Report of a Workshop on Middleware. (RFC 2768) (http://www.ietf.org/rfc/rfc2768.txt). 2000, Internet Engineering Task Force.

[2] BBN Technologies, Quality Objects (QuO) website (http://quo.bbn.com). 2002.

[3] Biegel, G., V. Cahill, and M. Haahr. A Dynamic Proxy-Based Architecture to Support Distributed Java Objects in Mobile Environments. International Symposium of Distributed Objects and Applications, (DOA 2002), (LNCS 2519). 2002. Irvine, CA.: Springer-Verlag.

[4] Bouraqadi-Saâdani, N., T. Ledoux, and M. Südholt. A Reflective Infrastructure for Coarse-Grained Strong Mobility and its Tool-Based Implementation (Technical Report 01-7-INFO). 2001, École des Mines de Nantes: Nantes, France.

[5] Capra, L. Reflective Mobile Middleware for Context-Aware Applications, Ph.D. Thesis, Department of Computer Science. 2003. University College London, University of London.

[6] Capra, L., W. Emmerich, and C. Mascolo. CARISMA: Context-Aware Reflective Middleware System for Mobile Applications. IEEE Transactions on Software Engineering, 2003. 29(10): p. 929-945.

[7] Chiba, S. Load-time Structural Reflection in Java. Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000) (LNCS 1850). 2000. Sophia Antipolis and Cannes, France: Springer-Verlag.

[8] Damianou, N., *et al.* The Ponder Specification Language. Workshop on Policies for Distributed Systems and Networks (Policy 2001). 2001. HP Labs, Bristol.

[9] David, P.-C. and T. Ledoux. An Infrastructure for Adaptable Middleware. Proceeding of the 2002 International Symposium on Distributed Objects and Applications (DOA 2002) (LNCS 2519). 2002. Irvine, California, USA: Springer-Verlag.

[10] David, P.-C. and T. Ledoux. Towards a Framework for Self-Adaptive Component-Based Applications. Proceedings of the 4th IFIP WG6.1 International Conference on Distributed Applications and Interoperable Systems 2003, (DAIS 2003) (LNCS 2893). 2003. Paris, France: Springer-Verlag.

[11] David, P.-C., T. Ledoux, and N.M. Bouraqadi-Saâdani. Two-step Weaving with Reflection using AspectJ. Proceedings of the Workshop on Advanced Separation of Concerns in Object-Oriented Systems, at OOPSLA 2001. 2001. Tampa Bay, USA.

[12] Ferber, J. Computational reflection in class based object-oriented languages. Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 1989). 1989. New Orleans, Louisiana, United States: ACM Press.

[13] Filman, R. and D. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. Workshop on Advanced Separation of Concerns, OOPSLA 2000. 2000. Minneapolis.

[14] Forman, G.H. and J. Zahorjan, The Challenges of Mobile Computing. 1994, University of Washington.

[15] Haahr, M. Supporting Mobile Computing in Object-Oriented Middleware Architectures, Ph.D. Thesis, Department of Computer Science. 2003. Trinity College Dublin: Dublin.

[16] Jørgensen, B.N., *et al.* Customization of Object Request Brokers by Application Specific Policies. Proceedings of Middleware 2000 (LNCS 1795). 2000. New York, USA: Springer-Verlag.

[17] Jugel, M.L. and M. Meißner. The Java Telnet Application/Applet (http://javatelnet.org): 2003.

[18] Kasten, E.P. and P.K. McKinley. Adaptive Java: Refractive and Transmutative Support for Adaptive Software. (Technical Report MSU-CSE-01-30). 2001, Department of Computer Science and Engineering, Michigan State University: East Lansing, Michigan, USA.

[19] Keeney, J. Completely Unanticipated Dynamic Adaptation of Software, Ph.D. Thesis, Department of Computer Science. 2004. Trinity College Dublin: Dublin.

[20] Kiczales, G., J. des Rivieres, and D. Bobrow, The Art of the Metaobject Protocol. 1991: MIT Press.

[21] Kiczales, G., *et al.* Aspect-Oriented Programming, Proceedings of 11th European Conference on Object-Oriented Programming. 1997, Springer-Verlag. p. 220-242.

[22] Maes, P. Computational Reflection (Tecnical Report VUB AI-Lab TR-87-02), PhD Thesis, Artificial Intelligence Laboratory. 1987. Vrije Universiteit: Brussels, Belgium.

[23] Mascolo, C., *et al.* XMIDDLE: A Data-Sharing Middleware for Mobile Computing. Personal and Wireless Communications, Kluwer, 2001.

[24] Microsoft Corporation, COM+ (http://www.microsoft.com/com/tech/COMPlus.asp). 1999.

[25] Object Management Group, Common Object Request Broker Architecture: Core Specification (OMG Document formal/02-12-06). 2002.

[26] Popovici, A., G. Alonso, and T. Gross. Just-in-time aspects: efficient dynamic weaving for Java. Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003). 2003. Boston, Massachusetts: ACM Press.

[27] Popovici, A., A. Frei, and G. Alonso. A proactive middleware platform for mobile computing. Proceedings of the 4th ACM/IFIP/USENIX International Middleware Conference (Middleware 2003). 2003. Rio de Janeiro, Brazil.

[28] Popovici, A., G. Alonso, and T. Gross. Spontaneous Container Services. Proceedings of the 17th European Conference for Object-Oriented Programming (ECOOP 2003) (LNCS 2743). 2003. Darmstadt, Germany.

[29] Popovici, A., T. Gross, and G. Alonso. Dynamic weaving for aspect oriented programming. Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002). 2002. Enschede, The Netherlands: ACM Press.

[30] Prakash:, R. Education: Mobile Computing. IEEE Distributed Systems Online, 2001. 2(6).

[31] Redmond, B. Supporting Unanticipated Dynamic Adaptation of Object-Oriented Software, Ph.D. Thesis, Department of Computer Science. 2003. Trinity College Dublin: Dublin.

[32] Redmond, B. and V. Cahill. Supporting Unanticipated Dynamic Adaptation of Application Behaviour. Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002) (LNCS 2374). 2002. Malaga, Spain: Springer-Verlag.

[33] Robben, B., *et al.* Building a Meta-level architecture for distributed applications (Technical Report CW 265). 1998, Department of Computer Science, Katholieke Universiteit Leuven: Leuven. p. 17.

[34] Robben, B., *et al.* Non-Functional Policies. Proceedings of the Second International Conference on Metalevel Architectures and Reflection. 1999. Saint-Malo, France: Springer-Verlag.

[35] Sadjadi, S.M. and P.K. McKinley. ACT: An adaptive CORBA template to support unanticipated adaptation. Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS'04). 2004. Tokyo, Japan.

[36] Sadjadi, S.M. and P.K. McKinley. Transparent self-optimization in existing CORBA applications. Proceedings of the International Conference on Autonomic Computing (ICAC'04). 2004. New York, NY.

[37] Sadjadi, S.M., *et al.* TRAP: Transparent reflective aspect programming. (Technical Report MSU-CSE-03-31). 2003, Computer Science and Engineering, Michigan State University: East Lansing, Michigan, USA.

[38] Sato, Y., S. Chiba, and M. Tatsubori. A Selective, Just-in-Time Aspect Weaver. Proceedings of the 2nd International Conference on Generative Programming and Component Engineering, (GPCE 2003), (LNCS 2830). 2003. Erfurt, Germany: Springer-Verlag.

[39] Sun Microsystems, Java 2 Platform, Standard Edition (J2SE) (http://java.sun.com/j2se/). 2002.

[40] Truyen, E., B. Vanhaute, and W. Joosen. Integrating flexible middleware solutions with applications through non-functional policies. Proceedings of OOPSLA Workshop on Reflection and Software Engineering (OORaSE '99). 1999. Denver, USA.

[41] Wall, T. Mobility and Java RMI, M.Sc. Thesis, Department of Computing Science. 2000. Trinity College Dublin: Dublin.

[42] Xerox PARC, The AspectJ Project (http://aspectj.org). 2004.