Policy-Based Dynamic Reconfiguration of Mobile-Code Applications

Mobility complicates application programming by requiring developers to define when and where to move which components under varying operating conditions. A policy-based implementation framework supports high-level reconfiguration strategies that separate mobility concerns from application functionality.

Rebecca Montanari
University of Bologna

Emil Lupu
Imperial College London

Cesare Stefanelli
University of Ferrara

Global wireless networks have opened the way to a ubiquitous Internet computing environment in which a variety of portable devices remain connected to the Internet fabric—even as their locations change continuously—and access to data and services is independent of both device type and location. In this environment, applications must be able to change the location of their execution. In addition, they must allow heterogeneous and resource-limited portable devices to download only the device-specific software components needed for execution.

Code mobility enables dynamic customization and configuration of ubiquitous Internet applications.[1] Mobile applications can transfer the execution of software components from one device to another depending on resource availability. They can also adapt functionality according to user needs and device characteristics.

Developers are using programming paradigms and techniques based on code mobility in several application domains, ranging from distributed information retrieval and computer-supported cooperative work to network management and mobile user and terminal support.[2-4] However, current approaches force application programmers to explicitly define and control the reconfiguration strategies that determine which software components to move, under what circumstances, and where. Typically, the strategies are embedded in the code in a tightly coupled way that complicates both the application's design and its runtime adaptation to changes in the execution environment.

We have developed a policy-based approach to mobility programming that expresses and controls reconfiguration strategies at a high level of abstraction, separate from the application's functionality. The Policy-Enabled Mobile Applications (Poema) framework provides an integrated environment for developing applications that can change both their functionality and layout at runtime in response to environment conditions. The software is available for download at www.lia.deis.unibo.it/Research/Poema/.

MOBILE-CODE PROGRAMMING

Current mobile-code programming models try to solve Internet-based application design and deployment issues by making location a first-class design concept that programmers can use to control application behavior and layout. The models differ in how they distribute application data and code:[1]

- *remote evaluation* models enable a computational component to send code to another node

Related Work in Dynamic Mobility Management

The following mobile-code programming models all share the principle of separating mobility from computational concerns as a key design requirement. However, the models differ on how to achieve this separation.

FarGo[1] provides methods for dividing the application code into mobile modules, called *complets*, and allows separate specification of how to relocate the modules. Programmers use a scripting language to encode complet layout strategies and bind them to the application at loading time.

The MAGE model[2] introduces the abstraction of mobility attributes to describe the mobility semantics of application components. As with FarGo, programmers bind the mobility attributes to the application components at loading time.

Another approach[3] focuses on the mobile agent paradigm and suggests separately encoding an application's function, mobility, and management.

XMILE[4] supports fine-grained mobility, allowing even individual lines of code to be sent across the network, and exploits XML to program mobility patterns.

Some other approaches focus on algorithms and mechanisms for carrying out the runtime reconfiguration of mobile-code applications. For instance, Dacia[5] proposes several mechanisms for consistently changing the application structure at runtime according to the reconfiguration strategies embedded into application-specific modules at design time.

Poema differs from all these approaches in its ability to specify mobility strategies at a higher level of abstraction and to support their modification even during application execution. In addition, Poema's infrastructure offers middleware services that provide support for the design, development, and deployment of adaptive applications in various scenarios.

References

1. O. Holder, I. Ben-Shaul, and H. Gazit, "Dynamic Layout of Distributed Applications in FarGo," *Proc. 21st Int'l Conf. Software Eng.*, ACM Press, 1999, pp. 163-173.
2. E. Barr, R. Pandey, and M. Haungs, "MAGE: A Distributed Programming Model," *Proc. 21st Int'l Conf. Distributed Computing*, IEEE CS Press, 2001, pp. 303-312.
3. K.J. Lauvset, D. Johansen, and K. Marzullo, "Factoring Mobile Agents," *Proc. Workshop on Eng. Computer-Based Systems*, IEEE CS Press, 2002, pp. 253-257.
4. C. Mascolo, L. Zanolin, and W. Emmerich, "XMILE: An XML-based Approach for Incremental Code Mobility and Update," *Automated Software Eng. J.*, Apr. 2002, pp. 151-165.
5. R. Litiu and A. Prakash, "Developing Adaptive Groupware Applications Using a Mobile Component Framework," *Proc. 2000 ACM Conf. Computer-Supported Cooperative Work*, ACM Press, 2000, pp. 107-116.

for remote execution,
- *code on demand* models allow a computational component to load some code from a remote repository when needed, and
- *mobile agent* models migrate an entire computational entity (code and execution state) to a different node.

Most existing mobile-code programming environments support only a single mobility model. As a result, designers can only partially exploit the power of code mobility. In addition, most environments offer only basic runtime support for code mobility and limited programming abstractions and tools for specifying reconfiguration requirements.

The "Related Work in Dynamic Mobility Management" sidebar describes some other programming development models that, like Poema, separate application logic and reconfiguration concerns. However, these models do not support high-level specification of mobility strategies or modification of the strategies during the application execution.

POLICY-BASED PROGRAMMING

Policies are declarative rules governing choices in a system's behavior.[5] They are increasingly popular in both academia and industry as a means for constraining system behavior. Network and system management tools have relied upon policy-based techniques for several years to add flexibility and adaptability to management infrastructures.

Our work extends policy-based techniques to dynamic reconfiguration of mobile-code applications. Developers and even application administrators can dynamically load or remove policies, thus reconfiguring the application at runtime without requiring application code changes. Moreover, policies favor application reusability in different deployment scenarios. Policy templates can be used to encode common reconfiguration strategies and thus facilitate their reuse.

A policy-based mobile-code programming environment requires a policy-specification model to express reconfiguration strategies. Multiple specification approaches are possible including formal policy languages, rule-based policy notations, and attribute table representations.[5]

For dynamic reconfiguration, we advocate using event-triggered declarative rules. Three main reasons underpin this choice:

- an event-driven model is a natural candidate for notifying distributed components of changes that occur in both application and system state;
- declarative policies let programmers directly express what reconfiguration is needed without having to specify how to achieve it, thus facilitating the definition of reconfiguration strategies; and
- declarative policy specifications are amenable to policy analysis and verification. Unlike policies buried in implementation code, declarative policies can be validated externally—for example, via theorem-provers—as sufficient and correct for the application reconfiguration.

(a) Generic policy type

```
inst oblig policyName “{”
on       event-specification;
subject  domain-Scope-Expression;
target   domain-Scope-Expression;
do       obligation-action-list;
when     constraint-Expression;
“}”
```

(b) Personal assistant policy instances

```
domain a = /PoemaOrganisationID/DomainID;
inst oblig UserMobility_P1 {
on       ChangeDevice(userName, deviceName);
subject  s = a/siteA/UserProxyID;
target   t = a/siteA/UserProxyID;
do       t.go((deviceName.getSite()).toString(), “run()”);
when     MonitoringSystem.checkResStatus(deviceName) = = true;
}
```

```
domain a = /PoemaOrganisationID/DomainID;
inst oblig LoadImageViewer_P1 {
on       MissingCode(“ImageViewer”);
subject  s = a/siteA/UserProxyID;
target   t = a/siteA/UserProxyID;
do       t.fetchCode(“/PoemaOrganisationID/DomainID/siteRepository/CodeProvider”,
         “ImageViewer”);
}
```

```
domain a = /PoemaOrganisationID/DomainID;
inst oblig BatteryShortage_P1 {
on       BatteryStatus(deviceName, BELOW_THRESHOLD);
subject  s = a/sitePDA/SYSTEMCC;
target   t = a/siteA/UserProxyID;
do       t.relocate(“/PoemaOrganisationID/DomainID/siteHostA”,
         “run()”);
}
```

(c) Personal assistant policy types

```
domain a = /PoemaOrganisationID/DomainID;
type oblig UserMobility (subject s, target t) {
on       ChangeDevice(userName, deviceName);
do       t.go((deviceName.getSite()).toString(), “run()”);
when     MonitoringSystem.checkResStatus(deviceName) = = true;
}
inst oblig UserMobility_P1 = UserMobility(a/siteA/UserProxyID, a/siteA/UserProxyID)
```

```
domain a = /PoemaOrganisationID/DomainID;
type oblig LoadImageViewer (subject s, target t) {
on       MissingCode(“ImageViewer”)
do       t.fetchCode(“/PoemaOrganisationID/DomainID/siteRepository/CodeProvider”,
         “ImageViewer”);
}
inst oblig LoadImageViewer_P1 = LoadImageViewer(a/siteA/UserProxyID, a/siteA/UserProxyID)
```

```
domain a = /PoemaOrganisationID/DomainID;
type oblig BatteryShortage (subject s, target t) {
on       BatteryStatus(deviceName, BELOW_THRESHOLD);
do       t.relocate(“/PoemaOrganisationID/DomainID/siteHostA”,
         “run()”);
}
inst oblig BatteryShortage_P1 = BatteryShortage(a/sitePDA/SystemCC, a/siteA/UserProxyID)
```

Figure 1. Obligation policies examples. (a) Generic policy type, (b) policy instances, and (c) policy types for the dynamic reconfiguration of the personal assistant application.

Using policies also requires middleware support for policy lifecycle management. In particular, the middleware must provide the management operations needed to compile high-level policies into enforceable representations, such as Java bytecode. It must also distribute policies to interested components and enforce them when an event occurs. Other middleware services should detect and resolve conflicts between policies when new policies are added or removed at runtime.

APPLICATION RECONFIGURATION WITH POEMA

The Poema policy-based framework evolved from our past work on dynamic control of mobile agents.[6] The framework permits the specification of different reconfiguration patterns at different granularities of code mobility, and it supports runtime application reconfiguration.

Programming reconfiguration policies

A Poema application is implemented in two phases. In the *application programming phase*, developers define and code only the application functionality without addressing reconfiguration issues. In a separate *policy specification phase*, they specify the desired reconfiguration strategies.

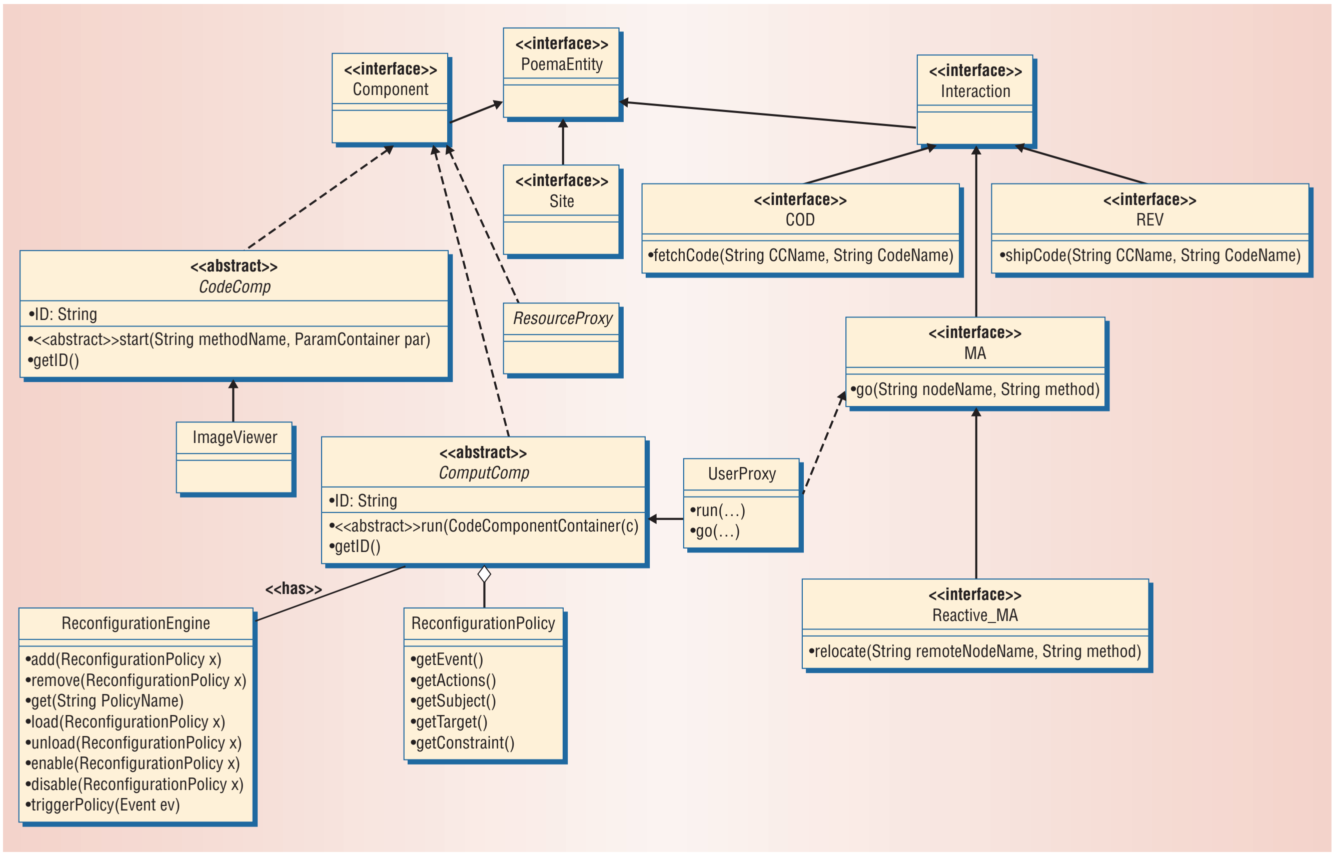
Poema specifies reconfiguration strategies in terms of Ponder *obligation policies*.[7] These policies are declarative event-condition-action rules that Poema uses to define the desired application reconfiguration when an event occurs.

To illustrate the use of Poema policies, we introduce a personal assistant application that supports the cooperative work of mobile users equipped with Internet-enabled portable devices. The application client module lets users view and update the shared information that the server module manages and maintains. Policies permit the specification of changes in the client and server module configurations and selection of the most appropriate mobility model—for example, code-on-demand to enable the client module to load a new image viewer.

Policy specification. Figure 1 shows the obligation policies for reconfiguring the personal assistant application. In the obligation policy type in Figure 1a, the `on` clause identifies the triggering event, which can be either *application-dependent*, generated by changes in application state, or *application-independent*, indicating a change in the execution environment. Event expressions can also combine events denoting the occurrence of specific event combinations.

The `subject` identifies the *computational components*—the autonomous units of execution that initiate a reconfiguration action. The `target` identifies the CCs to which the reconfiguration actions apply.

In the case of remote evaluation and code-on-demand, subject CCs ship or fetch code respectively to or from target CCs. In the case of mobile agents,



<<interface>> Component
<<interface>> PoemaEntity
<<interface>> Interaction
<<interface>> Site
<<interface>> COD
•fetchCode(String CCName, String CodeName)
<<interface>> REV
•shipCode(String CCName, String CodeName)
<<abstract>> CodeComp
•ID: String
•<<abstract>>start(String methodName, ParamContainer par)
•getID()
ResourceProxy
<<interface>> MA
•go(String nodeName, String method)
ImageViewer
<<abstract>> ComputComp
•ID: String
•<<abstract>>run(CodeComponentContainer(c)
•getID()
UserProxy
•run(…)
•go(…)
<<has>>
ReconfigurationEngine
•add(ReconfigurationPolicy x)
•remove(ReconfigurationPolicy x)
•get(String PolicyName)
•load(ReconfigurationPolicy x)
•unload(ReconfigurationPolicy x)
•enable(ReconfigurationPolicy x)
•disable(ReconfigurationPolicy x)
•triggerPolicy(Event ev)
ReconfigurationPolicy
•getEvent()
•getActions()
•getSubject()
•getTarget()
•getConstraint()
<<interface>> Reactive_MA
•relocate(String remoteNodeName, String method)

Figure 2. Unified Modeling Language diagram of the main Poema application architecture abstractions.

subject CCs either autonomously migrate or, if authorized, force target CCs to migrate.

The `do` clause identifies the reconfiguration action to perform when the event occurs. The Ponder language includes sequence and concurrency operators that support combinations of various reconfiguration actions.

The `when` clause defines the conditions that must hold for the policy to be applied. Developers can specify these conditions in terms of both application state and environment variables.

Figure 1b shows three policy instances used in the personal assistant application: UserMobility_P1, LoadImageViewer_P1, and BatteryShortage_P1. Note that policy subjects and targets are systemwide and identified by a globally unique name that complies with the Internet's lightweight directory access protocol. Names are a concatenation of the site ID where the subjects and targets were first created with a unique identifier within that site.

When the user changes the access device (the ChangeDevice() event), the UserMobility_P1 policy instance specifies that the client module (/siteA/UserProxyID) must migrate toward the new device if it has sufficient resources to support execution. An underlying monitoring system verifies resource availability. This is a *proactive* mobile agent strategy because the client module autonomously decides to move together with the current session state.

The LoadImageViewer_P1 policy instance in Figure 1b automatically loads a suitable image viewer in the client module from the code provider (/siteRepository/CodeProvider) as needed. The MissingCode() event triggers this policy that models a code-on-demand reconfiguration pattern.

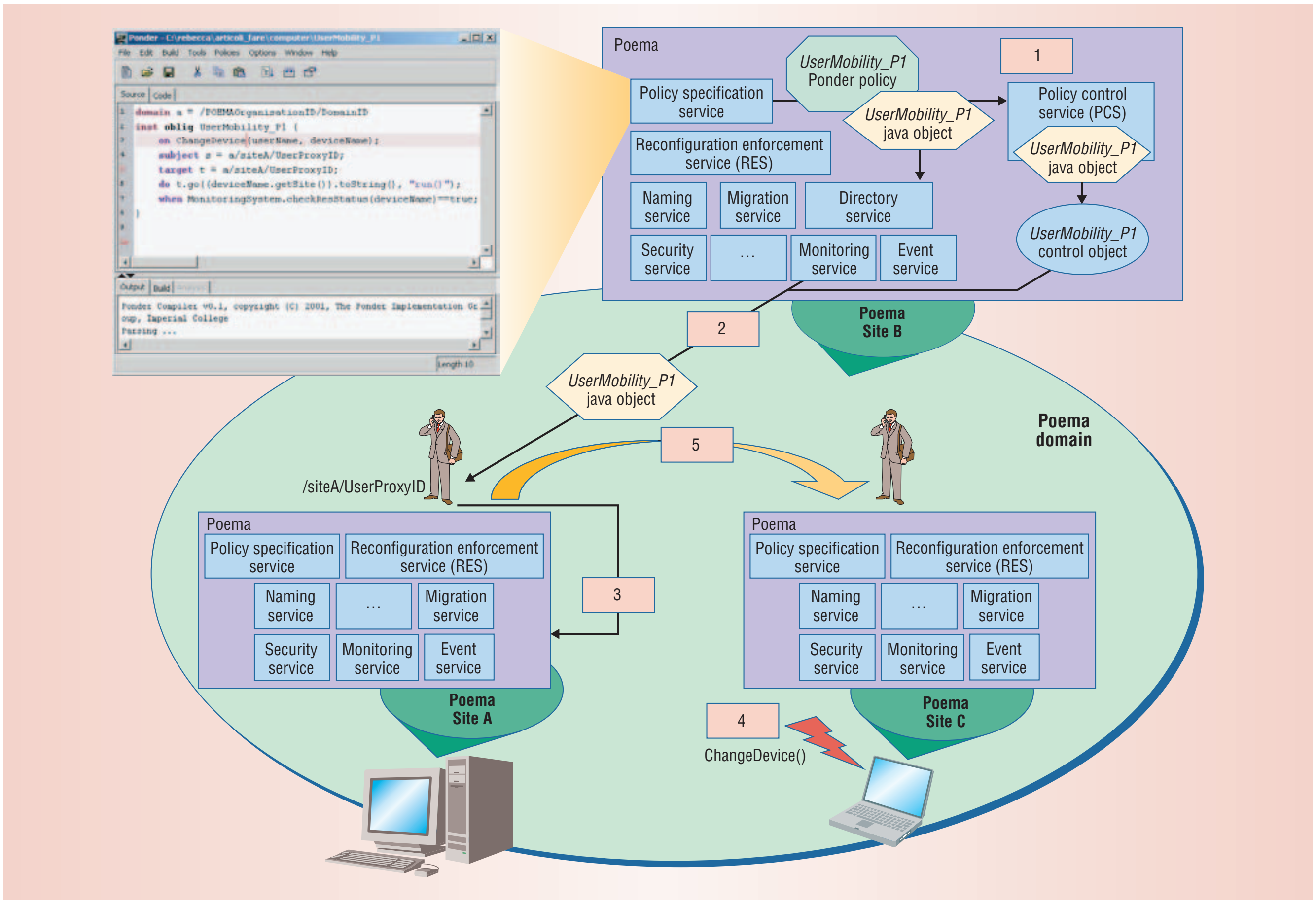
The BatteryShortage_P1 policy instance in this figure takes into account the possible battery power degradation in the portable device where the client module is currently executing (sitePDA).

In this case, a system-level computational component (SystemCC) on sitePDA relocates the client module to the node siteHostA, saving the current session state and permitting the client module to continue its execution. This policy implements a *reactive* mobile agent strategy in which an external entity forces the client module to move.

The Ponder language offers several benefits for reconfiguration. Its event-triggered obligation policies simplify the programming of CC reactions to changing conditions. Simply modifying the specification of either the event or the action causes CCs to react differently without changing their code.

In addition, the explicit distinction between policy subject and target permits modeling both proactive and reactive CC migrations. In proactive migration, the subject coincides with the target, as in the UserMobility_P1 policy. In reactive migration, the subject differs from the target, as the BatteryShortage_P1 policy.

Policy reuse. Ponder supports policy reusability in different application scenarios through the adoption of policy types. As Figure 1c shows, programmers


Ponder - C:\rebecca\articoli_fare\computer\UserMobility_P1
File Edit Build Tools Policies Options Window Help
Source Code
Output Build Analysis
Length 10
Poema
UserMobility_P1 Ponder policy
1
Policy specification service
UserMobility_P1 java object
Policy control service (PCS)
UserMobility_P1 java object
Reconfiguration enforcement service (RES)
Naming service
Migration service
Directory service
UserMobility_P1 control object
Security service
…
Monitoring service
Event service
Poema Site B
2
UserMobility_P1 java object
Poema domain
5
/siteA/UserProxyID
3
Poema Site A
4
ChangeDevice()
Poema Site C

```
domain a = /POEMAOrganisationID/DomainID
inst oblig UserMobility_P1 {
    on ChangeDevice(userName, deviceName);
    subject s = a/siteA/UserProxyID;
    target t = a/siteA/UserProxyID;
    do t.go((deviceName.getSite()).toString(), "run()");
    when MonitoringSystem.checkResStatus(deviceName)==true;
}
```

```
Ponder Compiler v0.1, copyright (C) 2001, The Ponder Implementation Gr
oup, Imperial College
Parsing ...
```

***Figure 3. Poema middleware.** Schematic shows the deployment of a policy-based personal assistant application in one domain.*

can exploit policy types to introduce specialized user-defined mobility patterns. They can parameterize policy types and create policy instances with application-specific parameters, such as subjects and targets.

Policy conflicts. Conflicts can arise because several policies may apply to the same CC. Previous work on the Ponder language distinguishes between *modality conflicts*—conflicts between permissions and prohibitions or between obligations and prohibitions—and *application-specific conflicts*.[8] More recent work on policy analysis in Ponder has focused on using an event-calculus-based policy representation together with abductive reasoning to identify the conditions and scenarios that lead to a conflict.[9]

Poema application architecture

Poema models an application in terms of components allocated in different sites and cooperating via interaction patterns. Figure 2 shows the main abstractions in the application architecture, which includes several *components*:

- CCs (ComputComp class) represent autonomous execution flows,
- code components (CodeComp class) define application-specific functionality, and
- resource components (ResourceProxy class) identify logical/physical resources.

Sites typically model network nodes that host components and provide execution environments for CCs. *Domains* are sites grouped to correspond to network localities, such as Ethernet-based LANs, and possibly organized in a hierarchy. Poema treats remote-evaluation, code-on-demand, and mobile-agent paradigms as specific *interaction* patterns. It provides the methods for sending and receiving code components as well as moving the entire CC either proactively or reactively (respectively, `shipCode()`, `fetchCode()`, `go()`/`relocate()`).

Developers encode any CC's application logic flow in the `run()` method and implement it as call sequences to the code components' `start()` methods. The type and number of code components that Poema executes depend on the application goals.

Note that the `run()` method does not contain reconfiguration directives, which are instead embedded into the reconfiguration policies (ReconfigurationPolicy class). Any CC contains the references to its policy objects. Poema provides all the needed policy management support—for example, policy load/unload and reconfiguration activation (ReconfigurationEngine class).

POEMA MIDDLEWARE

Figure 3 shows the two most important sets of reconfiguration support services that Poema provides:

- the *policy* set contains specific services for the runtime deployment of Ponder policies; and
- the *basic* set provides core support services, such as naming, migration, monitoring, event registration/dispatching, and security.

Poema exploits domain abstractions to facilitate policy lifecycle management and to improve scalability.

Policy services

The *policy specification service* provides tools for defining browsing and editing policies and for initializing reconfiguration policy objects. Initialization consists of generating a Java policy object that the runtime environment can load and interpret and storing both the Ponder and Java policy representations in a directory service.

The *policy control service* (PCS) is responsible for policy activation. It receives the Java representation of any new policy and generates a control object that keeps track of policy status transitions (initialized, activated, and removed). The control object interrogates the naming service to retrieve the current location of CCs interested in the policy and distributes the policy accordingly.

When the application first instantiates a CC, the PCS instructs the control objects to distribute the policies that apply to it. When a policy is deleted, the PCS coordinates with the responsible control object to inform the interested CCs to unload the policy and then removes the object. Note that Poema represents any policy change, such as policy creation or deletion, and any modification of the CC status, such as CC instantiation or termination, in terms of events that the event service communicates to the PCS.

The *reconfiguration enforcement service* (RES) provides the mechanisms for verifying policy applicability and enforcing reconfiguration policies at runtime. When an event occurs, RES retrieves all triggered policies, interprets them, and executes the needed reconfiguration actions according to policy specifications. If no policy applies, RES enforces specific exception-handling policies.

Reconfiguration can, however, interfere with a component's execution. For example, an asynchronous event might require CC migration at any execution point, possibly violating the consistency of resource bindings. In the current RES implementation, the CC reconfiguration can take place only outside sections the application programmer has explicitly labeled critical.

Basic service set

Figure 3 shows the most important infrastructure services that the Poema middleware provides. The *naming service* associates the CC with a globally unique identifier. CCs can exploit the *migration service* to either migrate among sites or to fetch or ship code respectively from or to other CCs.

The *monitoring service* and *event service* inform CCs about relevant state changes at both the application and operating environment levels. CCs can register their interest in one or more specific events. The event service dispatches event notifications to interested entities even when they have migrated to another location.

The *security service* enforces access controls according to authorization policy specifications. Poema exploits Ponder authorization policies for defining who can access, distribute, and enforce reconfiguration policies and under what circumstances.[6] For instance, in the personal assistant application, only trusted users with the required permissions can access the BatteryShortage_P1 policy stored in the directory. In addition, the /sitePDA/SystemCC component loads the BatteryShortage_P1 policy only if it trusts the policy supplier, and it enforces the client module migration only if the component has the necessary relocation privilege.

Poema integrates trust management solutions based on public-key infrastructure technologies to collect and verify the information required in trust decisions. In particular, Poema entities (programmers, administrators, the PCS, and so on) are associated with digital certificates for authenticating entities, signing policies, and securing communications.

Finally, Poema exploits domain abstractions to facilitate policy lifecycle management and to improve the scalability of policy-based mobile-code application deployment in large-scale environments. Domains define specific policy management boundaries: Each domain holds references to its CC members and the reconfiguration policies that currently apply to them. Applications deployed across several domains require tight interdomain coordination between Poema middleware services. Domains managed by different administrative authorities require a trust infrastructure.

PERSONAL ASSISTANT IMPLEMENTATION

In the personal assistant application in Figure 3, the administrator tool on site B defines the UserMobility_P1 policy. The tool provides predefined obligation policy templates to fill in application-specific data such as subjects, targets, and triggering events. This approach simplifies policy specification. The Ponder compiler, which is part of the administrator tool, generates the Java policy object—in this case, an instance of the ReconfigurationPolicy class from Figure 2.

The PCS then instantiates a specific UserMobility_P1 control object in charge of sending the Java policy object to the /siteA/UserProxyID client module. If the client module is temporarily unreachable, the control object suspends policy installation until the monitoring service detects its reconnection. The client module loads the received policy and registers the ChangeDevice() event with the event service. The client module can then migrate to any new access device where the user logs in.

Dynamic reconfiguration

At runtime, when the user connects to site C, the event service delivers the ChangeDevice() event to the client module. The client module then delegates the RES to perform the necessary reconfiguration actions via the triggerPolicy() method of the ReconfigurationEngine class shown in Figure 2. RES parses the UserMobility_P1 policy object to determine the subject, target, and other policy elements and enforces the client module migration to site C. If the new device is too limited to host the client module execution, RES handles the exception by sending the user a warning message that it cannot move the client module until the user connects to a more powerful device.

Poema also manages runtime policy removal. For instance, to delete the UserMobility_P1 policy, the PCS forces the UserMobility_P1 control object to contact the client module. The client module deletes the policy from its set of loaded policies, and the control object deregisters the ChangeDevice() event.

Poema provides several policy distribution strategies, depending on application requirements. At one extreme, the client module loads all the Java policy objects that control its reconfiguration. This favors prompt reaction to event notification but increases the client module size and migration time. At the other extreme, the client module loads only the references to its Java policy objects and retrieves the policies on demand from the directory when relevant events occur. Any intermediate solution is also possible.

Multiple concurrent events can lead to inconsistencies in the client module computation. To avoid this problem, RES adopts a sequential approach to policy enforcement based on the order in which the client module receives events. In addition, RES suspends reconfiguration while the client module is inside a critical section of code.

Poema provides several policy distribution strategies, depending on application requirements.

Security

To address policy-deployment security concerns, the personal assistant prototype trusts only the application administrator to define, modify, or remove policies. At policy specification time, the administrator digitally signs the policies. The client module loads only signed policies. The application's computational components—for example, the client and server modules—and the Poema middleware services—the PCS and event service—communicate through secure channels. Using digital signatures in conjunction with secure channels gives the client module evidence of the policy's author (the administrator) and ensures policy integrity.

Policy management

The Poema middleware configuration must define the initial service allocation on the domain's sites. For instance, Figure 3 features one directory service and one PCS per domain, while all other services, such as the RES, are allocated on every site.

A single directory simplifies policy analysis. To verify the consistency of any new policy, the Poema policy conflict checker analyzes all the policies stored in the directory. A single PCS centralizes policy activation, thus facilitating policy distribution and the management of concurrent and possibly conflicting policy activation and deletion requests. Allocating one RES per site allows CCs to make local reconfiguration decisions, which provides more scalable and efficient reconfiguration enforcement.

Finally, Poema can support alternative configuration solutions according to the desired tradeoffs among several requirements, such as scalability, fault tolerance, efficiency, and ease of management. For instance, a centralized directory facilitates policy storage and consistency checking but represents a single point of failure. A centralized RES could handle policy execution on behalf of resource-limited devices, but it might suffer from poor scalability and increase the network traffic due to the coordination between RES and the devices.

Limitations

The current Poema implementation has some limitations. Most significant is the middleware infrastructure's complexity, which might not be appropriate for devices with very limited resources.

The framework described here focuses on devices that can support a standard Java virtual machine.

We have compared the personal assistant application's performance in the Poema-based reconfiguration with the performance in traditional hard-coded reconfiguration directives. Specifically, we evaluated the application's policy response time to adapt to changes—that is, the time needed to complete a reconfiguration action after receipt of the related event.

The Poema-based reconfiguration exhibited a slightly higher response time—about a 10 percent increase—when compared with the hard-coded reconfiguration. The overhead comes from the additional time required to select the triggered policy and parse it for extracting policy applicability conditions and actions.[10] However, Poema's simplification of the runtime application configuration and its reusability counterbalance this difference.

Among emerging current approaches to separate mobility from computational concerns, Poema uniquely supports the specification of different mobility patterns—remote evaluation, code-on-demand, and mobile agent—in a single programming scheme. Application developers can modify reconfiguration choices at both compile/loading time and runtime without affecting the application implementation. Our future work will focus on new techniques for policy conflict detection and resolution and on experiments in other areas of ubiquitous computing, such as managing the resource bindings from software or device component migrations. ■

References

1. A. Fuggetta, G. Picco, and G. Vigna, "Understanding Code Mobility," *IEEE Trans. Software Eng.*, May 1998, pp. 352-361.
2. A. Tripathi et al., "Distributed Collaborations Using Network Mobile Agents," *Proc. 2nd Int'l Symp. Agent Systems and Applications*, LNCS 1882, Springer-Verlag, 2000, pp. 126-137.
3. R.H. Glitho and T. Magedanz, eds., special issue on Applicability of Mobile Agents to Telecommunications, *IEEE Network*, May/June 2002, pp. 6-40.
4. P. Bellavista, A. Corradi, and C. Stefanelli, "Mobile Agent Middleware for Mobile Computing," *Computer*, Mar. 2001, pp. 73-81.
5. S. Wright, R. Chadha, and G. Lapiotis, eds., special issue on Policy-Based Networking, *IEEE Network*, Mar. 2002, pp. 8-56.
6. R. Montanari, G. Tonti, and C. Stefanelli, "A Policy-Based Mobile Agent Infrastructure," *Proc. 3rd IEEE Int'l Symp. Applications and the Internet*, IEEE CS Press, 2003, pp. 370-379.
7. N. Damianou et al., "The Ponder Policy Specification Language," *Proc. 2nd Int'l Workshop Policies for Distributed Systems and Networks*, LNCS 1995, Springer-Verlag, 2001, pp. 18-38.
8. E. Lupu and M. Sloman, "Conflicts in Policy-Based Distributed Systems Management," *IEEE Trans. Software Eng.*, June 1999, pp. 852-869.
9. A.K Bandara, E. Lupu, and A. Russo, "Using Event Calculus to Formalise Policy Specification and Analysis," *Proc. 4th Int'l Workshop Policies for Distributed Systems*, IEEE CS Press, 2003, pp. 26-39.
10. R. Montanari, G. Tonti, and C. Stefanelli,, "Policy-Based Separation of Concerns for Dynamic Code Mobility Management," *Proc. 27th Int'l Conf. Computer Software and Applications,* IEEE CS Press, 2003, pp. 82-90.

***Rebecca Montanari** is a research associate of computer engineering at the University of Bologna. Her research focuses on Internet architectures, policy-based systems and service management, mobile agents, security in mobile agent systems, public-key infrastructures, and access-control models. Montanari received a PhD in computer science from the University of Bologna. She is a member of the IEEE and the Italian Association for Computing. Contact her at rmontanari@deis.unibo.it.*

***Emil Lupu** is a lecturer in the Department of Computing at Imperial College London, where he leads several projects on policy-based management of distributed systems and networks. His research interests include network and systems management and design, security, and adaptability issues in pervasive systems. Lupu received a PhD in computing from Imperial College London. Contact him at e.c.lupu@imperial.ac.uk.*

***Cesare Stefanelli** is an associate professor of computer engineering at the University of Ferrara. His research interests include distributed and mobile computing, mobile code, network and systems management, and network security. Stefanelli received a PhD in computer science from the University of Bologna. He is a member of the IEEE and the Italian Association for Computing. Contact him at cstefanelli@ing.unife.it.*