Dynamic Binding in Mobile Applications

A Middleware Approach

By separating binding concerns from application logic, the Colomba approach exploits metadata to let mobile applications adapt to dynamic environments.

Paolo Bellavista, Antonio Corradi, and Rebecca Montanari
Università di Bologna

Cesare Stefanelli
Università di Ferrara

Service development and deployment face new challenges from the wide availability of Internet points of attachment, increasing deployment of wireless networks, and growing portable-device market. To provision traditional Internet services to mobile clients, service providers must address the possibility of roaming during service sessions and the heterogeneity of access-terminal hardware and software. In addition, mobile computing encourages the development of location-dependent services.

Traditional middleware solutions are not designed to handle mobile users, their frequent temporary disconnection, or the wide range of access devices they employ. Novel middleware components should extend the fixed Internet infrastructure at service provisioning time when necessary to accommodate user and device mobility.[1] The middleware should also propagate to the service-level system information such as user location, preference profiles, and device characteristics.

The mobility of users, terminals, and service components requires novel middleware solutions to handle the set of bindings to needed resources. We have developed a middleware approach for binding management in mobile applications that addresses several of these issues. The Context- and Location-based Middleware for Binding Adaptation (Colomba) automatically updates mobile client references to needed resources whenever a client moves and dynamically selects and enforces the most suitable binding strategy. Colomba operates according to dynamic environmental conditions and various metadata, ranging from administrator management requirements to user, terminal, resource profiles, and resource co-locality constraints — forcing two resources to reside close to each other and eventually move together, for instance. Our middleware lets service providers express binding strategies at a high level of abstraction in terms of declarative directives that are cleanly separated from the service code; changes in binding

strategies thus require no intervention in the application logic. This separation of concerns is crucial to leveraging mobile applications in statically unknown usage scenarios.

Binding Management in Mobile Applications

The bindings to needed resources must be properly rearranged to maintain resource accessibility when users, terminals, and service components migrate to new locations. To introduce the binding management problem, consider a news service that lets mobile users download articles from newspaper data resources that are available over the fixed network from a station at an airport boarding gate. After the plane lands, the users' PDAs should reconnect automatically to another station at the arrival gate to download updated news as well as local information, such as weather and traffic reports.

This scenario exemplifies some of the possible resource-binding strategies that can be applied when a mobile entity (ME) moves in the network. In the following, an ME is a component — either a user or a terminal — that can change its physical location. In principle, four general strategies can rule resource bindings[2,3]:

- *Resource movement*. This strategy transfers bounded resources along with the ME when it moves. This type of binding is possible only if the resource transfer is technically and semantically possible; a database could not move to a different location, for instance, because it is already in use for queries and there is no way to move copies of it.
- *Copy movement*. This strategy copies bounded resources and transfers them along with the migrating ME. The resource copy must be technically and semantically possible, and conflicts might arise from concurrent modifications to multiple resource copies.
- *Remote reference*. Rather than moving resources, this strategy modifies ME resource bindings after migration to refer to the resources remotely. This strategy requires network communication with the remote execution environments hosting the resources.
- *Rebinding.* This strategy binds the ME to equivalent resources available in the new locality. It typically applies to bindings defined by type[3] and is fundamental anytime an ME accesses resource instances that provide service contents based on the instance location.

The choice of the proper binding strategy depends on several factors, from runtime conditions and access-device properties to management requirements and user preferences. A mobile device with no strict constraints on memory resources and computing capabilities could move or copy the needed resources and work on them locally, whereas a resource-limited wireless device might want to move resource copies to fixed hosts in the network currently providing connectivity. However, the language adopted for service design usually determines the binding strategy. Moreover, the strategy is typically embedded within the service code, thus limiting binding-management flexibility.[3]

Mobile applications require greater binding flexibility than is provided by conventional approaches, in which developers hard code strategies into the service logic. Moreover, such approaches can complicate service design and deployment, requiring the programmer to generate multiple statically specified versions of a mobile application, for example, to accommodate different deployment scenarios.

Discovery Services

The wide variety of implementation mechanisms available for resource retrieval, access, and usage further complicates the service developers' work. Researchers in both industry and academia are thus investigating mechanisms for dynamically retrieving and binding to the service components available in a locality for which the client has incomplete knowledge. These solutions are usually identified as *discovery services.*

Among commercial proposals, the Bluetooth service discovery protocol (www.bluetooth.com) and the Universal Plug and Play simple service discovery protocol (UPnP; www.upnp.org) are representative of simple solutions that work at low levels of abstraction. The IETF's service location protocol (SLP; RFC2608) and the Salutation suite (www.salutation.org) are examples of more complex and articulated middleware with APIs for attribute-based resource queries. Jini, on the other hand, lets servers advertise and clients discover service components. It also lets interface proxy objects dynamically migrate toward the clients to distribute code and information about how to access and use service components.[4] However, Jini requires clients to run a Java Virtual Machine or to be associated with fixed hosts running a JVM on their behalf. The Surrogate research project (http://surrogate.jini.org) tries to fill the gap between Jini and wireless devices that cannot run

a JVM by specifying surrogate proxies that non-Java-based portable devices can dynamically retrieve via low-level Bluetooth discovery.

Service Description

Several heterogeneous implementation mechanisms are also available for describing service components and for local or remote interactions with them. For instance, remote method invocation (RMI) assumes ubiquitous JVM support and exploits the Java type system to expose the interfaces of available resources in a repository. Because it imposes static heavy restrictions on resource binding, RMI allows only the default copy movement strategy and — when the bounded resource implements a specific interface and extends a specific base class — the remote reference strategy.

From the Web Services Description Language (WSDL) and SOAP to more traditional and less mobile-oriented architectures such as DCOM and Corba, several other description formats exist for resource advertising, as well as several mechanisms for resource access. Each makes trade-offs between flexibility and efficiency, and expressive power and limited footprint. Typically, they are all used as partial mechanisms for implementing a remote reference strategy.

Dynamic Binding

These considerations all point out the potential for a new middleware approach to designing mobile applications with dynamic binding capabilities. Past research in process migration, which faced similar issues with processes rebinding to needed resources after movement, provides a significant seminal basis for our own work.[5] Addressing the binding issues specific to mobile applications requires significant rethinking, however, to solve such challenges as location-aware support infrastructures for provisioning location-dependent services.

Few recent proposals in mobile code research suggest initial solutions for dynamic binding management.[6,7] They all demonstrate the emerging interest in binding issues by following the principle of clean separation between application logic and binding strategy, but they support binding strategy decisions only at the start of the mobile application execution.

The Colomba Framework

Colomba separates service logic from binding management. This permits developers to code, change, and reuse service components and binding strategies independently of each other. Administrators can express binding strategies at a high level of abstraction in terms of declarative policies. In particular, Colomba supports a dynamic binding management that requires:

- *Context awareness* is the knowledge of application-specific attributes, such as user preferences, level of trust, subscribed services, and access device characteristics. Users can refer to a set of resources determined by context information.
- *Location awareness* is the knowledge of the physical position of the user or device connection to the network infrastructure. Available resources depend on location information.

In the mobile news service (MNS) scenario described earlier, context and location awareness are crucial for choosing the most suitable binding strategy. The service should mainly offer news that can be specific to the ME's current locality. News should be tailored to user interests and represented in a suitable format for a given device's hardware and software characteristics. Location awareness enables rebinding anytime the ME changes locality: after migration, the ME transparently reconnects to the locally available news resources. Context information drives the choice between different instances of news resources in the locality. In fact, the binding strategy can identify local resources on the basis not only of simple description attributes, such as type identifiers, but also of complex application-specific information and device characteristics.

Colomba recognizes two components in a mobile-application deployment scenario:

- *Users* are the principals that request services and access resources from heterogeneous and possibly portable access devices; they might change device or location during service provisioning.
- *Resources* identify both physical devices (printers, disks, and so on) and logical information (files, profiles, service-specific software components, and such).

To support dynamic binding, Colomba exploits *metadata* and provides *middleware facilities*. Metadata provide information about users, devices, and resources and about the preferred binding strategies; middleware facilities perform runtime binding management actions based on metadata and context and location visibility.
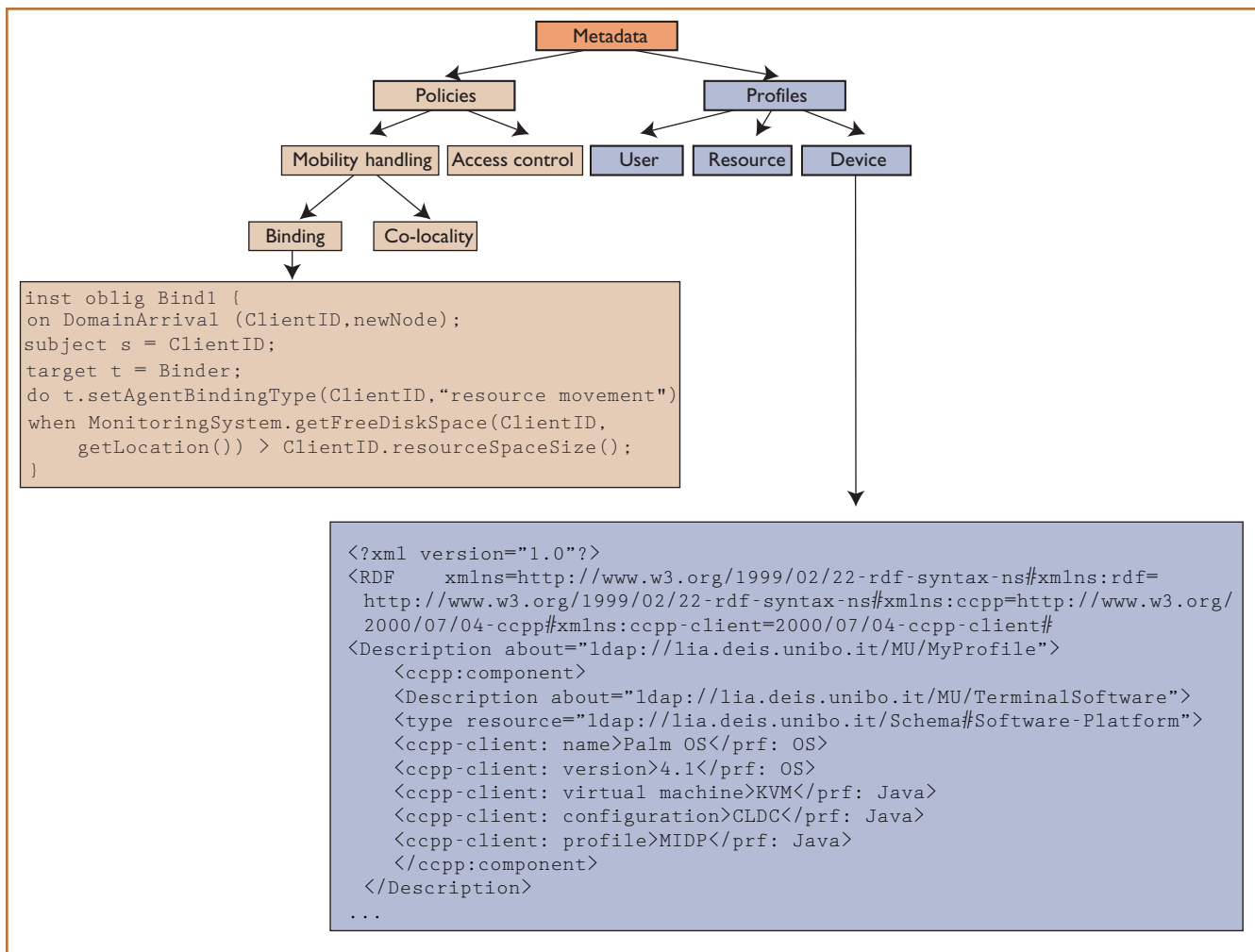
```
inst oblig Bind1 {
on DomainArrival (ClientID,newNode);
subject s = ClientID;
target t = Binder;
do t.setAgentBindingType(ClientID,"resource movement")
when MonitoringSystem.getFreeDiskSpace(ClientID,
    getLocation()) > ClientID.resourceSpaceSize();
}
```

```
<?xml version="1.0"?>
<RDF    xmlns=http://www.w3.org/1999/02/22-rdf-syntax-ns#xmlns:rdf=
 http://www.w3.org/1999/02/22-rdf-syntax-ns#xmlns:ccpp=http://www.w3.org/
 2000/07/04-ccpp#xmlns:ccpp-client=2000/07/04-ccpp-client#
<Description about="ldap://lia.deis.unibo.it/MU/MyProfile">
    <ccpp:component>
    <Description about="ldap://lia.deis.unibo.it/MU/TerminalSoftware">
    <type resource="ldap://lia.deis.unibo.it/Schema#Software-Platform">
    <ccpp-client: name>Palm OS</prf: OS>
    <ccpp-client: version>4.1</prf: OS>
    <ccpp-client: virtual machine>KVM</prf: Java>
    <ccpp-client: configuration>CLDC</prf: Java>
    <ccpp-client: profile>MIDP</prf: Java>
    </ccpp:component>
</Description>
...
```

Figure 1. Colomba metadata taxonomy. The `Bind1` *binding policy commands a resource movement strategy only if there is enough space in the device disk. The device profile refers to a PalmOS portable device.*

Colomba Metadata

As Figure 1 illustrates, Colomba exploits multiple types of metadata. *Policies* manage and control choices in resource-binding strategies. *Profiles* describe user requirements and resource properties. In particular, *user profiles* include information related to a user's context, such as personal preferences, required security level, and subscribed services. *Device profiles* report access devices' hardware and software characteristics. *Resource profiles* describe resource interfaces as well as the properties that could be useful for binding decisions, such as whether the resource can be copied or migrated over the network.

We use XML-based standard formats for profile representation. Colomba adopts the World Wide Web Consortium's Composite Capability/Preference Profiles (www.w3.org/Mobile/CCPP) for specifying user and device profiles, and the Resource Description Framework (RDF) for the interoperable descrip-

tion of resource components. (Figure 1 shows the CC/PP-compliant profile for a PalmOS device hosting the Java Kernel Virtual Machine/Connected Limited Device Configuration/Mobile Information Device Profile software suite).[8]

Policies are high-level directives that define choices in binding management by specifying the management tasks to perform rather than how to implement them.[9] The policy adoption helps in separating management strategies from service implementation, thus increasing flexibility and adaptability.

Access control policies ensure secure resource usage by specifying the actions a principal can perform, depending on various conditions such as the principal's identity or the resource's status.

Mobility handling policies guide binding-related management operations after ME migration. They include *binding* policies that define which binding strategy to apply, and when, and *co-locality* poli-

cies that describe when it is convenient to allocate a set of resources within a single node. Suppose, for instance, that a mobile subscriber to the MNS wants to compare political news in two opposing tabloids. The moving of both copies of the tabloid data resources in the user locality (as required by a co-locality policy) might conveniently improve performance and increase overall accessibility in the case of network partitioning, which could compromise the availability of master tabloid information.

Colomba adopts the Ponder language for policy specification.[10] In particular, we use Ponder obligation policy types for defining both binding and co-locality policies and Ponder authorization policy types for specifying access control. In this article, we focus only on obligation policies, which are essential for dynamic binding, but readers can find further details about Ponder authorization policies elsewhere.[11]

Administrators express obligation policies as declarative event-action-condition rules that define the binding management operations to perform when specific events occur. Figure 1 shows an example of a Ponder-based binding policy that selects a resource-movement strategy after the ME's migration. `Bind1` states that when the *ClientID* ME arrives at a new execution node (`on` clause), *ClientID* (`subject` clause) should command a Colomba middleware facility called *Binder* (`target` clause) to activate resource movement (`do` clause) if the new node has enough free space on disk to host the needed resource, as observed by the underlying Colomba monitoring facility (`when` clause) at runtime.

Colomba Middleware Facilities

Colomba supports mobile applications primarily via distributed deployment of active middleware proxies over the fixed network. The system provides any portable device with a companion middleware proxy — a *shadow* proxy — that autonomously acts on its behalf. Shadow proxies are designed to be able to negotiate services tailored to fit user and device characteristics, to operate asynchronously if the user or device gets disconnected, and to follow user movements among network localities by maintaining session state and ensuring binding updates.

Using mobile agents (MAs) to implement shadow proxies lets us achieve the crucial properties of mobility, autonomy, and asynchronicity. Because MAs are typically location-aware, they can exploit the current visibility of execution environments, for example, the set of locally available resources, to adapt their actions — primarily their migration — to the position of needed resources.[3] Colomba is built on top of the Secure and Open Mobile Agent (http://lia.deis.unibo.it/Research/SOMA) platform, which provides proxies with execution environments, called *places*, that typically model nodes.[1] Places can be grouped into domains that correspond to network localities, such as Ethernet-based LANs or IEEE 802.11b-based wireless LANs. Colomba implements each shadow proxy via a SOMA agent running on a place in the portable device's current domain.

Figure 2a shows Colomba's layered architecture. The upper-level facilities exploit lower-level functions for identification, discovery, directory, monitoring, and events. The *binder manager* dynamically and transparently readjusts the bindings between shadow proxies and needed resources, and the *policy manager* triggers binding rearrangement according to the specified metadata.

Binder manager. The binder manager (BM) mediates the shadow proxies' access to resources and dynamically adjusts binding strategies according to the specified policies. At startup, shadow proxies can refer only to the BM because they have no direct access to resources. In response to the first resource request, the BM sends the proxy a *resource descriptor* — an object with the same methods and constructor interface as the requested resource. Shadow proxies then operate on resources directly via the resource descriptors received.

Figure 2b depicts the internal structure of the proxy resource bindings. When a shadow proxy is first instantiated at one place, the local BM associates the proxy with a *resource table* that records all active resource descriptors for it. Each table entry includes the resource identifier (`ResID`), the corresponding resource descriptor (`ResDes`), a binding strategy identifier (`BindStratID`), and a reference object. The `BindStratID` tag keeps track of which binding strategy to apply when the proxy migrates. Colomba supports all four of the binding strategies we've described in this article. The reference object encapsulates the specific mechanism for implementing the binding strategy indicated by `BindStratID`.

The BM plays a crucial role in the proxy bindings' initial arrangement and dynamic readjustment after migration. When a proxy is created, the BM sets the `BindStratID` for each resource table entry according to the proxy-specific binding policies enabled. It then instantiates one reference object for each resource and puts it in the proxy resource table.

When a proxy arrives at a new place, the local BM checks the `BindStratIDs`' validity and transparently updates them as needed to reflect possible changes in binding choices. We assume that the proxy-specific binding strategy cannot change during proxy execution at one place; this gives us an effective implementation with limited overhead. Similar considerations guided the design of the JDK 1.2 security architecture in which permissions are generally granted to classes before they are defined in the Java runtime environment.[12]

The first time a proxy accesses a resource at a new place, the BM checks the proxy reference object's conformity to the set `BindStratID`. This avoids the need to check and change the binding strategy implementation mechanism for resources the proxy does not use in the new place. If the enabled binding strategy has changed, the BM instantiates and adds a new reference object to the resource table entry. For a remote reference strategy, the BM triggers the creation of the reference object that uses Java RMI.

In the case of resource or copy movement strategies, the BM forces the instantiation of a reference object that exploits Java serialization mechanisms: either the resource object or its copy are serialized, transmitted to the new place, and there deserialized. In choosing a rebinding strategy, the BM installs a reference object that exploits Jini to obtain information about resource availability and location. If several instances of the same resource type are locally available, the reference object for a rebinding strategy selects and binds to one instance based on retrieved user and device profiles.

Policy manager. The policy manager (PM) supports policy specification, dynamic installation, and enforcement. To enforce binding and co-locality policies, the PM detects changes in the operating environment that are relevant for binding management. It then notifies policy subjects about
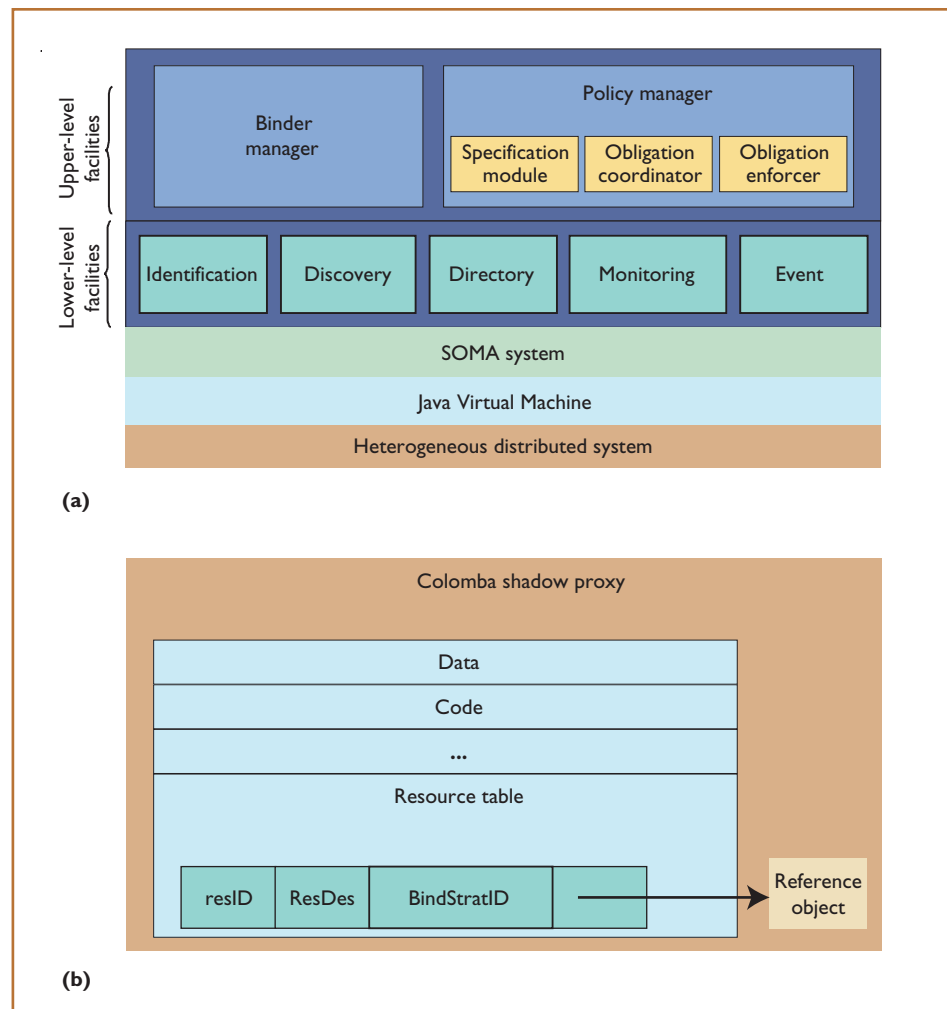

Figure 2. Colomba architecture. (a) Middleware facilities in two layers; the binder and policy managers sit on top of low-level components. (b) The proxy resource table contains the binding information to refer to resources.

event occurrence and interprets policy specifications to activate appropriate low-level management actions. As shown in Figure 2a, the PM consists of three modules:

- the specification module (SM)
- the obligation coordinator (OC)
- the obligation enforcer (OE)

The SM exploits the tools developed within the Ponder project for editing, updating, removing, and browsing binding and co-locality policies.[10] There are also tools for transforming high-level policy specifications into low-level policy representations that the Colomba middleware can interpret. In particular, the SM generates individual Java policy objects for each Ponder obligation policy. After creating a new Java policy object, the SM stores it in the Colomba directory and distributes it to the

interested policy subjects, the shadow proxies.

The OC module coordinates policy enforcement. It retrieves newly instantiated Java policy objects from the directory and parses them to retrieve relevant information: events, subjects, targets, and actions. It then registers the significant events with the Colomba event service on behalf of the policy subjects. At event occurrence, the system dispatches the events to all interested policy subjects, even if they have migrated to new localities. The event service coordinates with the naming service that keeps track of the policy subjects' current locations.

The OE module enforces the policies. In particular, event service notifies a subject of policy event occurrences, and the subject delegates the OE to interpret the triggered policy specifications. Policy interpretation comprises parsing policies, controlling the dynamic conditions for policy applicability, extracting the policy actions, and commanding the BM to activate the corresponding operations in the system. Note that the OE enforces policies sequentially — that is, only after completing the actions triggered by preceding events.

Colomba-Based Mobile News Service

To show Colomba at work during service provisioning, we implemented a context- and location-dependent MNS. When a user with a mobile terminal (MT) requests to start an MNS session, the system assigns a dedicated shadow proxy `Proxy-ID`. The instantiated `ProxyID` runs on the MT if it can host a full JVM and a SOMA place on top of it. Otherwise, `ProxyID` executes on a neighbor host on the fixed network in the locality to which the MT is currently attached. In this case, `Prox-yID` automatically follows the MT's movements to maintain co-locality with its mobile client.[8]

Figure 3a shows an excerpt from the simple, reusable, and binding-transparent code of the `MNSProxy`. Each shadow proxy executes the `init()` method when it is created to initialize a reference to the information resource called `news-paper`. `ColombaDataResource.get()` invokes the BM that adds a new entry in the `ProxyID` resource table, sets the resource binding type to the default value, `remote reference`, and returns a `resourceID` descriptor to the shadow proxy to access the resource. As long as the MT stays connected to the fixed network (`isConnected` set), `ProxyID` repeats a user-specified query on `resourceID` at regular time intervals to visualize the newly obtained results. Other `ProxyID` threads (not shown in the code excerpt)

let the user browse the results and insert new queries.

Without modifying the `MNSProxy` implementation, we can adapt MNS to work in different operating scenarios. For instance, service providers can enable the shadow proxy to access needed resources, even when disconnected, by simply specifying the `Pol1` policy at service deployment time (see Figure 3b). When the `ProxyID` needs to disconnect from the current attachment point (`DisconnectRequest` event), `Pol1` commands the BM (named `Binder`) to move needed resources to the proxy device if the device has enough disk space to host them.

In particular, `ProxyID` first opens a message window asking the user to stay connected until the resource management process is complete. The MNS prototype requires the MT not to disconnect before the completion of the resource movement phase. Next, the BM sets `isConnected` to false and the binding type to `resource movement` for any referenced resource in the `ProxyID` resource table. The BM `updateReference-Object()` method checks the binding strategy type for any entry in the `ProxyID` resource table and updates the reference objects accordingly. If a newspaper resource R1 moves, for example, the event service notifies a `ResourceMovement` event to the shadow proxy that triggers the evaluation of the related `Pol2` co-locality policy, shown in Figure 3c. In `Pol2` the shadow proxy commands the BM to move a copy of the newspaper data resource R2 in the same location where R1 (or one of its copies) migrates. If the R2 profile specifies that R2 can be copied and transferred, the BM adds a new line for R2 in the `ProxyID` resource table (`setAgentBindingType()` method) at `Pol2` enforcement and forces the copy movement by updating the corresponding reference object (`updateReferenceObject()` method).

In the current implementation, the BM moves a resource copy even if other shadow proxies refer to it as well. Because the MNS is based on read-only operations, no inconsistencies will arise because of this. However, we must extend the BM with functions for merging updates and resolving conflicts in order to handle resource copies in more general mobile applications. Only these enhancements can allow concurrent modification of distributed copies; we are currently implementing a BM version that supports the consistency restoration schema presented elsewhere.[13]

Other dynamic conflicts could occur when two roaming MTs request to move the same resource

```
class MNSProxy extends ShadowProxy {
...
void init() {
... DataResource resourceID =
    COLOMBA.DataResource.get ("newspaper");
... )
void run() {
... for (;;) {
    if (isConnected==true) results =
        resourceID.query(search);
            visualizer(results);
            sleep(pollingInterval);
    } ... }
... }
(a)
```

```
inst oblig Pol1 {
on DisconnectRequest (proxyID);
subject s = ProxyID;
target t1 = Binder, t2 = ProxyID;
do t2.showWaitingMessage() ||
    t1.setIsConnected(ProxyID,false) ->
    t1.setAgentbindingType(ProxyID,"resource
        movement") ->
    t1.updateReferenceObject(ProxyID) ->
    t2.removeWaitingMessage();
when MonitoringSystem.getFreeDiskSpace
    (ProxyID.getLocation())
        > ClientID.resourceSpaceSize();
}
(b)
```

```
inst oblig Pol2 {
on ResourceMovement(ProxyID,R1);
subject s = ProxyID;
target t = Binder;
do t.setAgentBindingType(ProxyID,R2,"copy
movement") ||
    t.updateReferenceObject(ProxyID,R2);
}
(c)
```

```
inst oblig Pol3 {
on domainArrival(ProxyID,newNode);
subject s = ProxyID;
target t = Binder;
do t.setAgentBindingType(ProxyID,"rebinding") ||
    t.setIsConnected(ProxyID,true);
when MonitoringSystem.isDiscoveryAlive() == true;
}
(d)
```

Figure 3. Programming MNS on top of Colomba. (a) This code excerpt from MNSProxy *allows the* ProxyID *to access needed resources and browse requested news. (b)* Pol1 *facilitates disconnected operations. (c)* Pol2 *expresses the co-locality constraints related to* Pol1. *(d)* Pol3 *allows the* ProxyID *to deploy the context- and location-dependent MNS.*

to different localities (directly or as a result of an enforced co-locality policy). Colomba solves this conflict through the OE's sequential policy enforcement. Moreover, to avoid continuous resource migrations with no useful work on it, Colomba disables a resource's ability to be moved for a time after it migrates. The resource profile specifies this time interval that is based on both resource movement costs and expected mobility patterns in the deployment scenario.

Colomba facilitates the deployment of the context- and location-dependent version of the MNS simply by specifying the Pol3 policy shown in Figure 3d: When ProxyID enters a new locality (DomainArrival event), Pol3 forces the BM to set both the binding strategy type to rebinding for any MNS resource in the ProxyID resource table and isConnected to true. In the example, Colomba performs the Pol3 actions only if the Colomba discovery is working in the new MT locality.

Unlike Pol1, Pol3 does not include any BM action to adjust the reference objects in the resource table. In fact, the BM automatically per- forms the update at the first use of one ProxyID resource reference in a new locality to avoid wasting time with requalifying resource links that were not used during the ProxyID's execution.

To rebind MT to the most suitable MNS resource instance in the new domain, the Jini-based reference object considers the user and device profiles when ProxyID first requests resource usage. The reference object retrieves profiles from the globally available Colomba directory before interrogating the Colomba discovery service for its current location to obtain a list of compatible local resources. Our discovery solution exploits an SLP-compliant protocol to register and deregister resources that are typically visible in one Colomba domain.[1]

If the list of suitable resources includes several items, the reference object exploits the profile metadata to choose which to rebind to. In the current implementation, it performs simple parsing and processing operations on the attribute-value pairs in the profiles, and increments a score counter for each resource profile attribute that is compatible with the corresponding user or device profile.

The reference object then chooses the resource with the maximum score and updates the dynamically downloaded Jini-based client stub for that resource.

Ongoing Work

The complexity of developing and deploying mobile applications over the Internet dictates a separation of concerns between resource-binding strategies and application logic implementations. Only this clean isolation permits the necessary flexibility and reusability of middleware and service components. Novel and programmable middleware solutions, integrated with different types of high-level metadata, can provide management configurability while hiding low-level mechanisms and implementation details from service developers and system administrators.

First experiences with Colomba have shown that the middleware can simplify service design and implementation in different deployment scenarios. We are currently extending the prototype and developing other mobile applications on top of it. In particular, we are designing a set of middleware components for dynamic QoS adaptation (filtering, downscaling, transcoding, and so on) of video-on-demand flows, based on user and device profiles (www.lia.deis.unibo.it/Research/ubiQoS/). We are also using Colomba to specify dynamic binding strategies in non-mobile usage scenarios for reducing network traffic and for balancing the load of different resource copies.　⬚

References

1. P. Bellavista, A. Corradi, and C. Stefanelli, "Mobile Agent Middleware for Mobile Computing," *Computer*, vol. 34, no. 3, 2001, pp. 73–81.
2. L. Cardelli, "Mobile Computation," *Mobile Object Systems: Towards the Programmable Internet*, LNCS 1222, J. Vitek and C. Tschudin, eds., Springer-Verlag, 1997, pp. 3–6.
3. A. Fuggetta, G.P. Picco, and G. Vigna, "Understanding Code Mobility," *IEEE Trans. Software Eng.*, vol. 24, no. 5, 1998, pp. 342–361.
4. G.G. Richard III, "Service Advertisement and Discovery: Enabling Universal Device Cooperation," *IEEE Internet Computing*, vol. 4, no. 5, 2000, pp. 18–26.
5. D.S. Milojicic et al., "Process Migration," *ACM Computing Surveys*, vol. 32, no. 3, 2000, pp. 241–299.
6. E. Tanter and J. Piquer, "Managing References upon Object Migration: Applying Separation of Concerns," *21st Int'l Conf. Chilean Computer Science Soc.* (SCCC'01), IEEE Press, 2001, pp. 264–272.
7. O. Holder, I. Ben-Shaul, and H. Gazit, "Dynamic Layout of Distributed Applications in FarGo," *21st Int'l Conf. Software Eng.* (ICSE'99), ACM Press, 1999, pp. 163–173.
8. P. Bellavista, A. Corradi, and C. Stefanelli, "The Ubiquitous Provisioning of Internet Services to Portable Devices," *IEEE Pervasive Computing*, vol. 1, no. 3, 2002, pp. 81–87.
9. S. Wright, R. Chadha, and G. Lapiotis, eds., Special Issue on Policy Based Networking, *IEEE Network*, vol. 16, no. 2, 2002, pp. 8-56.
10. N. Damianou et al., "The Ponder Policy Specification Language," *2nd Int'l Workshop Policies for Distributed Systems and Networks* (Policy'01), LNCS 1995, Springer-Verlag, 2001, pp. 18–38.
11. A. Corradi et al., "A Flexible Access Control Service for Java Mobile Code," *16th Ann. Computer Security Applications Conf.* (ACSAC'00), IEEE Press, 2000, pp. 356–365.
12. L. Gong, *Inside Java 2 Platform Security*, Addison-Wesley, 1999.
13. E. Pitoura and B. Bhargava, "Data Consistency in Intermittently Connected Distributed Systems," *IEEE Trans. Knowledge and Data Eng.*, vol. 11, no. 6, 1999, pp. 896–915.

Paolo Bellavista is a research associate of computer engineering at the University of Bologna. His interests include mobile agents, pervasive computing, systems and service management, location and context-aware services, and adaptive multimedia. Bellavista received a PhD in computer science engineering from the University of Bologna. Contact him at pbellavista@deis.unibo.it.

Antonio Corradi is a professor of computer engineering at the University of Bologna. His research interests include distributed systems, object and agent systems, network management, and distributed and parallel architectures. Corradi received an MS in electrical engineering from Cornell University. Contact him at acorradi@deis.unibo.it.

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. Contact her at rmontanari@deis.unibo.it.

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