Dynamically Programmable and Reconfigurable Middleware Services

Manuel Roman and Nayeem Islam

DoCoMo Communications Labs
181 Metro Drive
San Jose, CA 95110
{roman,islam}@docomolabs-usa.com

Abstract. The increasing software complexity and proliferation of distributed applications for cell phones demand the introduction of middleware services to assist in the development of advanced applications. However, from the user perspective, it is essential that these new phones provide a smooth error-free experience. Despite of the complexity underlying a cell phone, placing a phone call remains a simple task that can be performed by most users regardless of their technical background. Furthermore, cell phones rarely crash (especially compared to PCs) and carriers are able to correct certain problems remotely without user intervention.

We advocate for a middleware infrastructure that allows carriers and developers to correct middleware behavior, configure it, and upgrade it, without requiring user intervention and without stopping the execution of applications. We introduce a new technique we refer to as externalization. This technique explicitly externalizes the state, the logic, and the internal component structure of middleware services. As a result, carriers and developers have full control over these middleware services. They can access, inspect, and modify the state, logic, and structure of middleware services at runtime while preserving the execution of existing applications and providing an error-free experience to users. We claim that externalization is the key for the future evolution of cell phones' middleware infrastructure.

1 Introduction

Cell phone functionality has evolved tremendously over the last 10 years. First, there was just voice transmission. Then, short messages (SMS) and web browsing (WAP and iMode) were added. Later, interactions with vending machines (cMode [1]) and multimedia messaging (MMS) became available. Most recently, video conferencing, Internet access, and interaction with the surrounding physical environment (iArea [2]) became possible. The evolution of cell phones and wireless-enabled handheld devices as well as the increasing proliferation of wireless networks are changing our traditional understanding of computers. The notion of desktop computing is slowly evolving into a more dynamic model. Cell phones do not sit on a desktop, are not disconnected from the surrounding environment, and are not immobile anymore. These devices are capable of connecting to wireless networks, they have enough processing power to perform tasks previously reserved for servers and workstations, and they are carried by users on a regular basis. Phones are poised to replace our keys [3], identification cards, and money with digital counterparts. Furthermore, increasing data

H.-A. Jacobsen (Ed.): Middleware 2004, LNCS 3231, pp. 372–396, 2004.

transmission rates (2Mbps with UMTS or 14.4Mbps with Japan's FOMA networks[4]) enable the development of applications that allow cell phones to interact with distributed services (e.g., Web Services) and access and share rich multimedia contents.

The increasing number and sophistication of cell phone applications demands a cell phone middleware infrastructure to assist in the development and execution of applications. Examples of middleware are: RPC, discovery, security, QoS, group management, event distribution, and publish subscriber.

Due to the cell phone constraints (for example, limited resources and reliability) these services must meet the following requirements:

1. They must be configurable, both statically and dynamically, to accommodate the requirements of applications (e.g., transactions, QoS, and security) and to meet the requirements of heterogeneous devices and execution environments[5].
2. They must be dynamically updateable to correct errors and therefore provide users with an error-free and zero maintenance execution model [6]. According to existing studies [7], 10% cell phones are returned due to software problems. With over 1200 million subscribers worldwide, it means that over 120 million phones are returned every year. Requesting cell phone users to take their device to a customer support center to correct the software errors is too costly for carriers and frustrating for cell phone users.
3. They must provide support for run-time upgrades to incorporate new functionality (for example, new interfaces, new protocols, and new policies)[8].

We refer to the previous issues as configurability (1), updateability (2), and upgradeability (3). Reflective middleware services[9, 10] provide functionality for configurability. They support replacement and assembly of certain components to adapt to changes and create certain device dependent configurations. However, most reflective systems assume a basic skeleton where only certain pre-defined changes and configurations are allowed. We seek a mechanism that allows modifying every aspect of the system (including the static skeleton), and enables fine-grained customizations.

Bitfone[7], Redbend [11], and DoOnGo[12] support updateability and upgradeability. They provide functionality to update the cell phone's firmware at runtime. They calculate the binary differences between the new and old images and update the differences. However, none of these products allows updating the software without stopping the system. They require restarting the devices, require user intervention, and do not provide fine-grained updating capabilities, that is, it is not possible to change certain logic or structural properties. The whole software image has to be replaced. Our goal is to avoid or minimize user intervention and preserve the normal execution of the system.

In this paper, we present a new middleware construction approach that assists in the development of configurable, updateable, and upgradeable middleware services. These services are assembled dynamically from small execution units (micro building blocks) and can be reconfigured at runtime. Our approach externalizes three key middleware execution elements: state, structure, and logic. As a result, we have fine-grained control over running middleware services in terms of configurability, updateability, and upgradeability. We have used the construction technique to build an efficient communication middleware service that we can configure, update, and upgrade at runtime. Despite of this flexibility, the service provides performance equivalent to non-reconfigurable services.

The paper is structured as follows: Section 2 motivates middleware externalization as a key technique for middleware configuration, updating, and upgrading. Section 3 describes Dynamically Programmable and Reconfigurable Software (DPRS), our approach to construct flexible middleware services. Section 4 describes in detail a communication middleware service (ExORB) that we have built using a Java prototype of DPRS. Section 5 provides a performance evaluation of ExORB. We present related work in Section 6 and conclude in Section 7.

2 Motivation for Middleware Architecture Externalization

Middleware architecture externalization relies on three key aspects: state externalization, structure externalization, and logic externalization. State externalization exports the internal middleware state attributes so they can be inspected and modified. Structure externalization exports the list of components that compose the middleware service and supports inspection and modification. Finally, logic externalization exports the interaction rules among the structural components (logic of the middleware service), thus providing support to inspect and modify the logic.

The main benefit of architecture externalization is the ability to learn, reason, and modify every aspect of a middleware service. The notion of architecture externalization is similar to computational reflection[13], which is a technique that allows a system to maintain information about itself (meta-information) and use this information to change its behavior (adapt). However, the key difference between computational reflection and architecture externalization is the scope of information maintained by the software, and the scope of the changes allowed. Existing computational reflection middleware services [14, 15], explicitly define the internal aspects they export, and the changes they accept. However, middleware services based on architecture externalization export every detail in terms of structure, logic, and state, and accept arbitrary changes in any of the three categories. Building an externalized middleware service requires identifying the functional units of the service (we call them micro building blocks) and using the techniques described in Section 3 to define their composition and interaction rules. The resulting middleware service can be inspected and modified. Figure 1 depicts an architecture browser tool we have built that connects to a device and extracts the structure, logic, and state of the software it hosts. The figure illustrates the structure and part of the logic of the externalized communication middleware service running in the device (RemoteInvocationSupport).

Externalized middleware services are assembled at runtime using an architecture descriptor that contains information about the components that compose the system (structure), the interaction rules for these components (logic), and a descriptor with detailed information about each structural component (input parameters, output parameters, and state attributes). The collection of all state attributes corresponds to the global middleware service state. These descriptors are the service blueprints and provide the information required to assemble the service at runtime. We use the descriptors to configure middleware services to different devices. Furthermore, these blueprints constitute a valuable formalism to understand the composition and behavior of existing middleware services. Developers can access these descriptors (or extract them directly from a running system), understand the internal details of the system, and introduce changes to customize the service without reading a single line of source code.

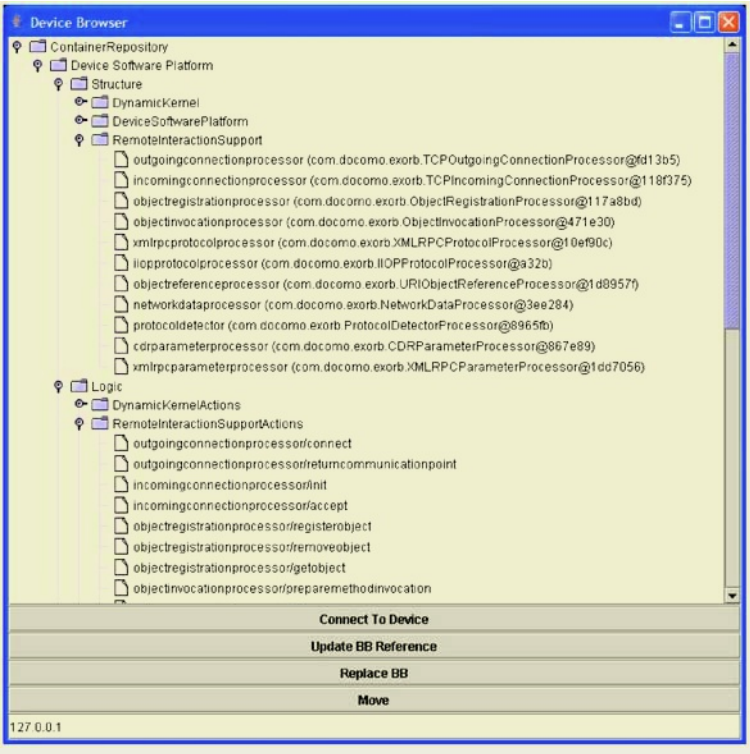

Fig. 1. Middleware architecture browser.

Another benefit of architecture externalization is that it exports the execution state of the system, which includes information about the currently executed internal component. This information becomes essential in determining safe reconfiguration points, which correspond to execution states where it is safe to replace components, modify the logic, and modify state attributes. The system can determine these safe points without requiring any support from the software developers.

Finally, a benefit of architecture externalization is the ability to virtualize the software infrastructure and create snapshots of the running system. This functionality is particularly useful to suspend, resume, and migrate software automatically. Furthermore, heterogeneous systems can exchange architecture definitions and reconfigure themselves to enable interoperability.

We have built a software construction mechanism that relies on architecture externalization. We refer to this type of software as Dynamically Programmable and Reconfigurable Software (DPRS). Programmable because similarly to hardware FPGAs (Field Programmable Gate-Arrays) that allow programming the behavior of the hardware, our technique allows programming the behavior of the software by defining the structure and logic of the software. Reconfigurable, because it is possible to access and alter the structure, logic, and state of the middleware. Finally, the adverb dynamically specifies that changes to the middleware architecture can be performed at runtime. In this paper we describe our experience using DPRS to build middleware services.

2.1 Configurability, Updateability, and Upgradeability: Requirements and Examples

In this subsection we concentrate on configurability, updateability, and upgradeability. We define each term, describe the requirements to obtain such functionality, and finally present some examples.

We refer to configurability as the ability to select the functional components that compose a middleware service to accommodate changes in the execution conditions and to accommodate heterogeneous devices. With updateability we denote the functionality to replace software components at runtime, as well as the functionality to modify the behavior of the middleware (that is, the execution logic), also at runtime. Finally, we use the term upgradeability to refer to the ability to add new functionality to existing middleware services at runtime.

Configurability, updateability and upgradeability require functionality to modify the structure and the logic of the middleware. To modify the structure, the middleware must export its internal component composition, thus allowing external entities to inspect it and modify it. Furthermore, in order to modify the logic, middleware must export information about its internal component interaction rules, and must provide functionality to modify them. Finally, updateability requires also access to the state of the components, so it can automate component replacement (no need to transfer the state from old to new components).

To motivate the relevance of configurable, updateable, and upgradeable middleware services we use a middleware service we have built (ExORB), and explain how we leverage the architecture externalization technique. ExORB is an object request broker communication middleware service that provides functionality to invoke and receive RPC requests using different protocols such as IIOP and XMLRPC. As a DPRS-enabled service, ExORB can be configured, updated, and upgraded. We present an example for each feature next and leave the detailed description of the design of ExORB for Section 4.

As an example of **configurability** assume the following two cases:

- Customizing ExORB for client-side only functionality (sending requests only) or client and server side functionality (sending and receiving requests) depending on the role of the devices and available resources
- Replacing an existing protocol encoding component with a new one that implements an algorithm optimized to the new execution conditions (for example, reduced bandwidth).

Regarding **updateability**, consider the two following examples:

- Assume that the component that encodes IIOP requests has an error and adds wrong information to the request header. Developers can correct the component code and replace the existing component with the new one without stopping the system and without requiring user intervention.
- Consider the case where developers find a more efficient way to send requests. The new approach invokes ExORB's internal components in a different order (connects first to the remote object and generates the IIOP messages only upon successfully connecting) and reduces the time required to send remote invocations. Due to the logic externalization property, we can dynamically modify the component invocation order.

Finally, regarding **upgradeability**, we describe four examples next:

- The first example is about interceptors, which allow adding functionality to object request brokers. An interceptor is an object that is invoked before and after sending a request to customize the behavior of the request broker. Interceptors are used, for example, to encrypt and decrypt the buffer before it is sent. Building interceptors in ExORB is simple. We create a new component, register it with ExORB's externalized structure, and modify the logic of ExORB so it invokes the new component before sending the request data over the network. The key difference with traditional interceptors is that with ExORB we can leverage the structure and logic externalization to insert interceptors in arbitrary positions. With existing ORBs, interceptors are installed at predefined points. Furthermore, most existing ORBs do not accept installing or modifying interceptors at runtime.
- The second example extends ExORB with functionality to broadcast information about its registered objects, which is functionality commonly implemented by discovery services. In order to provide the functionality, we create and register a new component that accesses the state attribute that stores the list of registered objects and broadcasts their references. Then, we modify the logic of ExORB so we invoke this new component periodically.
- The third example adds a new protocol to ExORB at runtime (for example, SOAP). We develop components to marshal and demarshal SOAP parameters, encode and decode SOAP messages, and leverage the rest of ExORB's infrastructure. To add the new functionality, we leverage structure externalization to register the new components, and the logic externalization to modify the logic of the system. Once the functionality is installed, existing applications can receive and send requests over SOAP without any change in their code.
- Finally, the last example adds new functionality to ExORB to support object migration. The new migration functionality removes the target object from the source ExORB, instantiates a copy of the object in the remote location, transfers the state, and registers the object with the target ExORB. Furthermore, we insert a component that maintains a list of migrated objects so it can redirect incoming requests. We can install the new functionality at runtime without affecting the execution of the existing ExORB. Adding the migration functionality leverages the logic externalization to add new logic to ExORB, structure externalization to add new components that implement the migration functionality, and state externalization to access and modify the list of registered objects both at the source and target ExORBs.

Based on our experience building ExORB and an event distribution middleware service, architecture externalization is a key mechanism to achieve configurability, updateability, and upgradeability. Externalization gives full control over running middleware services and allows tuning and correcting their behavior without requiring user intervention. Furthermore, with architecture externalization, the scope of reconfiguration of middleware services is open ended. The ability to manipulate the structure, logic, and state, allows programming the behavior of the middleware services at runtime.

3 Dynamically Programmable and Reconfigurable Software (DPRS)

This section provides a detailed description of DPRS. The description includes information about the abstractions and the execution model. Furthermore, in this section we explain how the different components of DPRS contribute to the middleware configurability, updateability, and upgradeability.

3.1 DPRS Abstractions

DPRS relies on three abstractions to build dynamically reconfigurable software: Micro Building Block (MBB), Action, and Domain. An MBB is the smallest functional unit in the system, an action defines the logic of the system, and a domain represents a collection of related MBBs. DPRS relies on name and value tuples as an indirection mechanism to address system entities. This indirection simplifies the introduction of changes in the system at runtime and has proved effective to build decoupled and manageable systems.

3.1.1 Micro-building Block

An MBB is the smallest addressable functional unit in the system. An MBB receives a collection of input parameters, executes an action that might affect its state, and generates a collection of output parameters. An example of an MBB is "registerObject", which receives two input parameters, a name and an object reference, updates a list (its state) with the new entry, and returns the number of registered objects.

MBBs store their state attributes as name and value tuples in a system provided storage area. This mechanism avoids implementing state transfer protocols to replace MBBs. Replacing an MBB requires registering the new MBB instance and providing it with a pointer to the existing state storage area. Accessing the attributes by name enables also external attribute manipulation. External services operate on the existing state and the MBBs obtain the new values when they resolve them by name during the execution of their algorithm. Furthermore, storing the state as a collection of name and value tuples simplifies state suspension, resumption, and migration. We provide services that implement this functionality transparent to MBBs. Figure 2 illustrates the structure of an MBB.

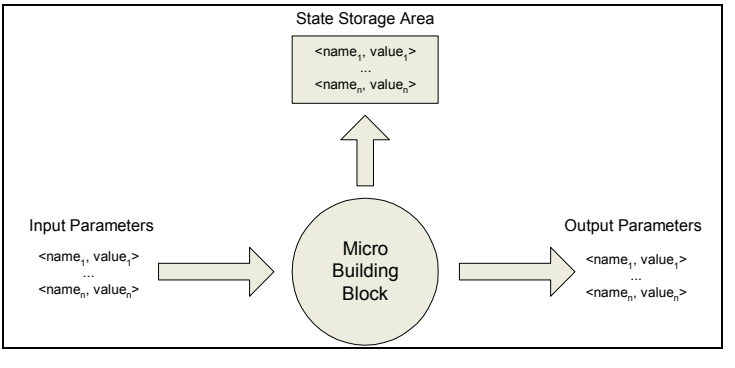

Fig. 2. Micro Building Block structure.

The execution model of DPRS invokes a collection of MBBs in a specific order (the exact mechanism is described in section 3.2). However, MBBs do not store references to the next MBB in the chain. This mechanism implies that no MBB in the system stores references to any other MBB. This approach allows replacing MBBs easily. There is no need to notify any MBB about the replacement because no MBB knows about any other MBB.

3.1.2 Action

Actions specify the MBB execution order and therefore define the logic of the system. DPRS defines two types of actions: interpreted actions, and compiled actions.

An **interpreted action** is a deterministic directed graph where nodes are MBBs that denote execution states, and edges define the transition order. Every edge has an associated conditional statement that is evaluated at runtime to determine the next transition. Conditional statements can refer to parameters generated by MBBs (output parameters). Finally, for nodes with multiple out edges, only one of edge can evaluate true at runtime (deterministic graph). By default, the value of this conditional statement is true. Action graphs have one start node, intermediate nodes, and one end node. The start and end nodes (the end node denotes the action graph terminates) are part of every graph traversal. The intermediate nodes depend on the traversal of the graph according to the conditional statements assigned to the edges and their runtime evaluation. Action graphs include additional nodes and edges that specify the transitions in case of errors. That is, if no errors are detected, the system uses the default action graph (for example, the one depicted in Figure 3). However, if execution errors are detected, then the system uses the error nodes and edges. For example, Figure 3 has an additional edge that goes from each node to the end state (not included in the figure). That is, if an error is detected, the action simply terminates. Note, however, that it is possible to define more sophisticated behaviors. Action graphs allow cycles to support loop statements, such as “while”, “for”, and “repeat”.

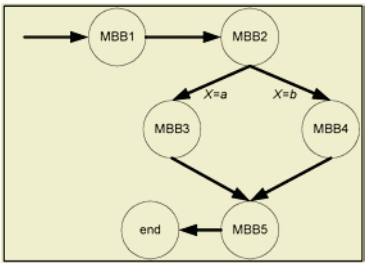

Fig. 3. Interpreted Action example.

Executing an interpreted action corresponds to traversing the graph. Figure 3 depicts an action example where MBB1 is the start node. The action starts with the invocation of MBB1, continues with the invocation of MBB2, then, depending on the value of ‘X’ it invokes MBB3 or MBB4, and finally, it invokes MBB5. The value of the variable ‘X’ is either provided by the client invoking the action or it is an output parameter generated by MBB1 or MBB2. This value is stored as part of the action execution state, which is described in detail in the execution model (Section 3.2).

Interpreted actions provide reflection at the execution level by exporting information about the current execution state, and by providing support to modify the action graph at runtime. Furthermore, the explicit representation simplifies reasoning about the logic of the system, supports static analysis, and allows third parties to modify the behavior of the system by adding or removing states and configuring the graph.

A **compiled action** is a code fragment that specifies the MBB invocation order. Compiled actions invoke MBBs using a DPRS library. This library receives an MBB name and a collection of input tuples, and invokes the specified MBB with the provided input parameters. This mechanism allows DPRS to take control over MBB invocation, which allows DPRS to replace MBBs safely. Figure 4 illustrates an example of a compiled action, which corresponds to the interpreted action depicted in Figure 3.

```
action Test
{
    outputParams = InvokeMBB(MBB1, inputParams);
    outputParams = InvokeMBB(MBB2, outputParams);
    char X = outputParams.get("X");
    if (X=='a')
      outputParams = InvokeMBB(MBB3, outputParams);
    if (X=='b')
      outputParams = InvokeMBB(MBB4,outputParams);
    outputParams = InvokeMBB(MBB5, outputParams);
}
```

Fig. 4. Compiled action example.

The compiled actions' code is provided as an MBB that is registered with the system. Therefore, invoking the action corresponds to invoking the MBB. This approach allows us to replace action definitions at runtime.

The key difference between interpreted and compiled actions is the runtime manipulation granularity. Compiled actions cannot be modified at runtime, that is, it is not possible to add, remove, or modify transition states. Changing their behavior requires replacing their associated MBB, that is, replacing the action code. Furthermore, it is not possible to inspect compiled actions at runtime, and therefore it is not possible to learn about the current execution state, or learn about the action behavior. With interpreted actions, the graph provides enough information to learn about the behavior of the action. The benefit of compiled actions is that they execute faster than interpreted actions because they do not require an interpreter to drive their execution. Furthermore, a compiled action gives more control to the programmer over the programming of the software behavior.

Both interpreted and compiled actions contribute to MBB replacement. One of the key requirements to automate runtime MBB replacement is detecting the system has reached a safe execution state. With DPRS actions, these safe states can be determined automatically. The safe reconfiguration states correspond to MBB invocations. With interpreted actions, the interpreter explicitly invokes the MBBs. Compiled actions use a DPRS library to invoke MBBs. In both cases, the supporting system controls MBB invocation and therefore can safely replace MBBs.

Finally, both interpreted and compiled actions contribute to updateability and upgradeability of the systems. Updating an action corresponds to replacing an existing

action, or in the case of interpreted actions, modifying the execution graph. Upgrading the system implies adding new actions, or in the case of interpreted actions, modifying the action graph to incorporate or modify states.

3.1.3 Domain

A domain is an abstraction that aggregates collections of related MBBs. It provides a storage area to store the structure of the domain (list of MBBs), the logic of the domain (list of actions), and the state of the domain (MBBs state attributes and execution state values). Domains can be composed hierarchically, and they provide a useful mechanism to manipulate collections of MBBs as a single unit (for example, move, suspend, and resume).

Figure 5 illustrates the components and structure of a domain. All three memories (structure, logic, and state) store name and value tuples. The **structure memory** maintains a collection of tuples that correspond to MBBs registered in the domain. The tuple name refers to the name of the MBB (every MBB is assigned a name at registration time), and the value stores the reference to the MBB. Note that the reference can be a local pointer or a pointer to a remote MBB. The DPRS execution model makes local or remote invocation transparent to developers. The **logic memory** stores a list of actions exported by the domain. Similarly to the structure memory, the logic memory refers to actions by name. Finally, the **state memory** stores the state attributes for the MBBs registered in the domain. During the MBB registration, the system assigns a pointer to the state memory to the MBB. MBBs belonging to the same domain share the same state memory. We refer to the three memories as the *domain memory*.

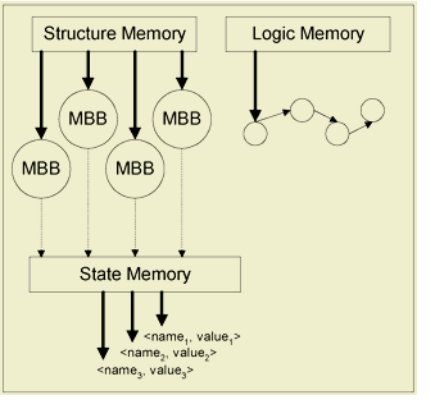

Fig. 5. Domain components.

Domains can be composed hierarchically, which provides a useful mechanism to organize large collections of MBBs. Domain memories store a reference (name and value tuple) to the domain memories of the registered sub-domains, and they also store a reference to the root domain memory. Figure 6 illustrates an example of a hierarchical composition of domains. Root domain has two sub-domains (domain 1 and domain 2) and domain 1 has three sub-domains (domain 3, domain 4, and domain 5).

The default visibility policies dictate that a domain has access to the sub-domain memories. For example, the root domain has access to all the domain memories of the system (that is, domains 1, 2, 3, 4, and 5), while domain 5 has access to its own domain memory only. However, it is possible to modify the visibility policies and allow sub-domains to access their parents or siblings' domain memories.

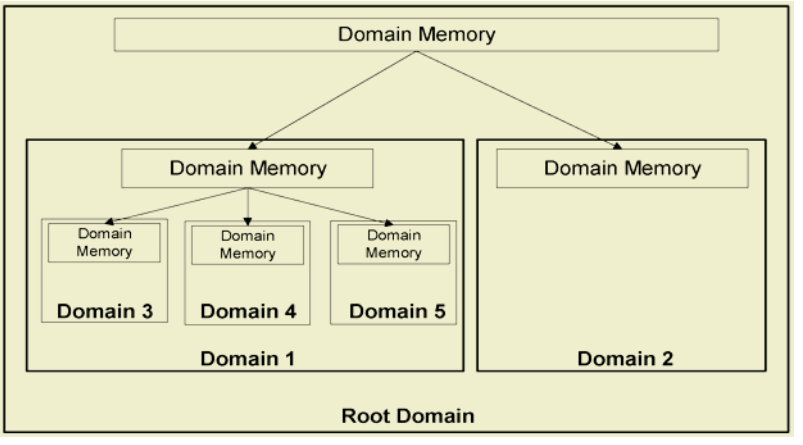

Fig. 6. Hierarchical composition of domains.

Domains provide a useful mechanism to organize complex systems consisting of a large number of MBBs. The recursive composition of domains contributes to the static configuration of middleware services. Developers can provide different domain compositions for different devices or execution requirements.

3.2 Instantiation Model

DPRS-based systems are assembled at runtime using a "blueprint" we refer to as *architecture descriptor*. This descriptor contains information about the domain hierarchy. Each domain entry in the architecture descriptor points to two additional descriptors, structure and logic descriptors, which specify the MBBs and actions registered in the domain. Finally, the structure descriptor points to the MBB descriptors that correspond to the MBBs that compose the structure. Figure 7 illustrates a descriptor diagram for the example depicted in Figure 6 (we only include the root domain descriptors' hierarchy for clarity).

DPRS relies on a runtime infrastructure that provides functionality to parse the architecture descriptor, instantiate the required MBBs, introduce changes at runtime, and parse interpreted actions. Figure 8 illustrates the runtime infrastructure. It consists of two key components: Static Kernel and Dynamic Kernel. The static kernel provides functionality to parse the architecture descriptor, functionality to generate an architecture descriptor for the running system, and functionality to instantiate micro building blocks (for example, Java objects, .NET object, or DLLs). The static kernel is the minimum functionality required to assemble a system based on micro building blocks and it is the only non-reconfigurable non-MBB based component in the system. The

layers on top of the static kernel are recursively built using micro building blocks. The dynamic kernel consists of a domain called Domain Manager, which provides micro building blocks to manage the domain. The dynamic kernel provides also the Micro Building Block Scheduler, which provides functionality to execute interpreted actions. Finally, on top of the dynamic kernel are the dynamically programmable and reconfigurable middleware services.

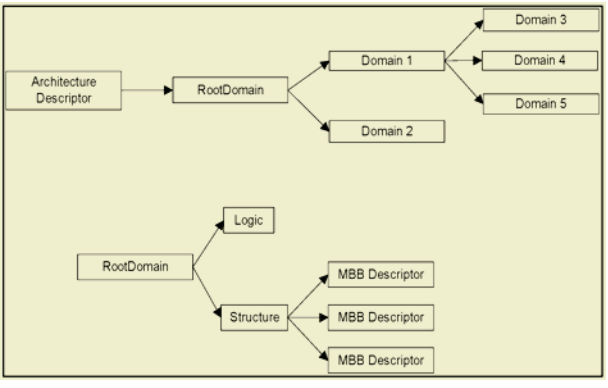

Fig. 7. DPRS architecture description.

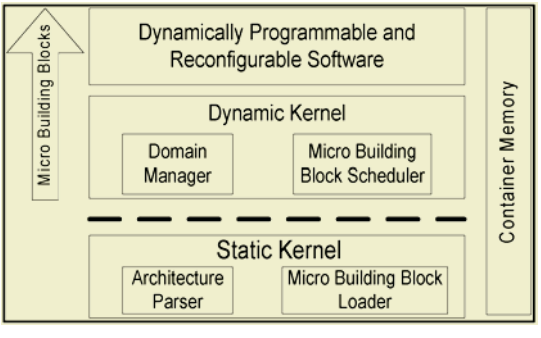

Fig. 8. DPRS Runtime Infrastructure.

3.3 DPRS Execution Model

DPRS interpreted actions externalize the logic of the system. They provide information about the MBB invocation sequence required to execute a functional aspect of the system. The DPRS execution model relies on a component called *MBB scheduler*, which drives the execution of the system using the action's graph as an MBB invocation schedule. The MBB scheduler maintains and exports information about the execution state of the system. This information consists of:

1. Currently executed action
2. Currently executed MBB
3. Action associated parameters, that is, parameters provided by the action invoker, plus parameters generated by the action's MBBs.

The MBB scheduler is implemented as an MBB. Therefore, its state is accessible, and it can be modified at runtime as any other MBB. The ability to replace the MBB scheduler allows developers to provide different execution semantics. For example, they can choose an MBB scheduler that supports transparent local or remote MBB invocation, therefore simplifying runtime software partitioning. Furthermore, they can choose an MBB scheduler that checkpoints the parameters and state after every MBB invocation therefore providing fault tolerant semantics. Also, they can select a real time MBB scheduler that defines action execution time boundaries therefore providing guarantees on the action execution times. The ability to select a specific MBB scheduler combined with dynamic software replacement capabilities simplifies the construction of adaptive systems. That is, systems that can modify their execution model according to the execution conditions and external requirements.

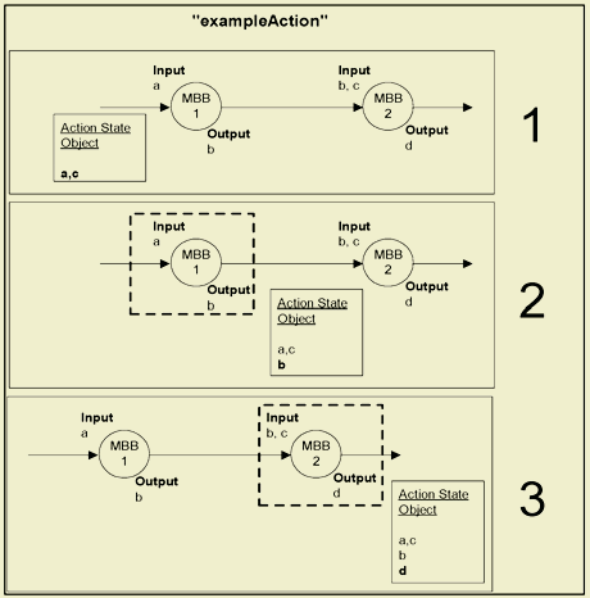

Fig. 9. Action execution example.

The DPRS execution model associates an object called *action state object* to each action execution. Actions use this object to store the input and output parameters associated to the action execution. Parameters are provided by the clients invoking the action and are also generated by MBBs as the result of their invocation. MBBs consume parameters stored in the action state object to implement their algorithm. Saving the parameters generated during the action invocation and synchronizing the MBB access to their state attributes allows clients to invoke actions concurrently. Figure 9 illustrates an interpreted action execution example. The name of the action is "exampleAction" and it consists of two MBBs. To simplify the explanation we assume an action with no conditional transitions or loops. The execution model remains the same. The difference is that the MBB scheduler evaluates an expression to obtain the name of the next state. We describe the execution model algorithm next:

1. The MBB scheduler receives a request to execute an action called "exampleAction". The request includes an action state object that contains two parameters, *a*, and *c* (Step 1 in Figure 9).
2. The MBB scheduler uses the action name to access the logic memory and obtains a pointer to the action graph's first node.
3. The MBB scheduler obtains the name of the MBB from the action graph's node, and uses the name (MBB1) to resolve the MBB from the structure memory.
4. After resolving MBB1, the MBB scheduler invokes the MBB passing the action state object. MBB1 requires an input parameter named *a*, which it obtains from the action state object. MBB1 executes its algorithm and generates an output parameter called *b*, which it stores in the action state object (Step 2 in Figure 9).
5. Next, the MBB scheduler obtains the name of the next state from the current actions graph's node, obtains the name of the MBB (MBB2), and resolves MBB2 from the structure memory.
6. The MBB scheduler invokes MBB2 with the action state object as a parameter. MBB2 requires two parameters, *b* and *c*, which it obtains from the action state object. MBB2 executes its algorithm, generates an output parameter called *d*, and stores the parameter in the action state object (Step 3 in Figure 9).
7. Finally, the MBB scheduler returns the action state object to the caller.

The main contribution of the DPRS' execution model is the ability to detect safe software reconfiguration points automatically. The basic rule is that the system allows reconfiguring the system between MBB invocations only. MBBs are allowed to access and modify the externalized structure, logic, and state. Therefore, modifying these parameters might affect the execution of the MBBs and could lead to an inconsistent software state. The system waits until the MBB completes its execution to avoid undesirable results. Note that this behavior applies to both interpreted and compiled actions. Compiled actions use a DPRS library to invoke MBBs and therefore give control to the system to implement reconfiguration.

The main concern about the interpreted execution model is performance. Invoking an action requires accessing the domain memory to resolve the nodes of the action graph, the MBBs, and accessing the parameters stored in the action state object. In Section 5, we present experimental results that illustrate the performance penalty incurred by DPRS. The results indicate that the penalty can be considered negligible in most cases.

4 ExORB: A Dynamically Reconfigurable Communication Middleware Service

In this section we present a multi-protocol Object Request Broker (ORB) communication middleware service that we have built using DPRS. The service provides client and server functionality independently of wire protocols. That is, the server object's methods can be invoked over different protocols, such as IIOP or XML-RPC. Similarly, client requests use the same interface and semantics regardless the underlying protocol. Although our implementation supports IIOP and XML-RPC, it is possible to add additional protocols by developing and deploying additional micro building blocks at runtime. As a DPRS system, ExORB's architecture (state, structure, and

logic) is externalized, and therefore, it is possible to inspect it and manipulate it at runtime. ExORB has been built using a Java implementation of the DPRS supporting infrastructure, and we use it extensively as a basic component of our infrastructure to enable transparent remote MBB invocation. In the following section, we present the architecture of ExORB including a list of micro building blocks, domains, and actions.

4.1 Structure of ExORB

ExORB is composed of 28 micro building blocks grouped into 11 domains. Figure 10 depicts the structure of ExORB. Next we explain the functional goal of each domain and the MBBs that compose each of the domains.

The CDR Parameter Management domain provides functionality to marshal and demarshal parameters according to the Common Data Representation (CDR) format (CORBA default representation). It contains two MBBs: CDR Marshal Parameters and CDR Demarshal Parameters.

The XMLRPC Parameter Management domain is similar to the CDR Parameter Management Domain but provides functionality to marshal and demarshal parameters encoded according to the XMLRPC protocol.

The IIOP Protocol Processing domain aggregates micro building blocks that export functionality to encode and decode messages that conform to the IIOP protocol. It contains five MBBs: IIOP Encode Request, IIOP Decode Request, IIOP Encode Reply, IIOP Decode Reply, and IIOP Decode Header.

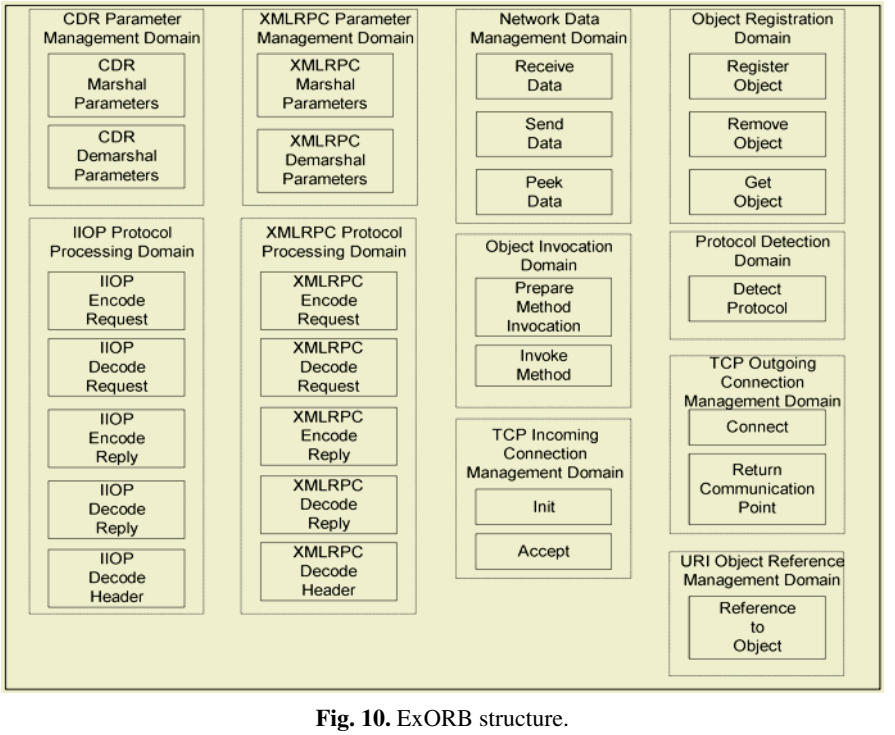

Fig. 10. ExORB structure.

The XMLRPC Protocol Processing domain is equivalent to the IIOP Protocol Processing Domain and provides functionality to handle XMLRPC requests and replies.

The Network Data Management domain is responsible for handling incoming and outgoing network traffic. It is composed of three micro building blocks: Send Data, Receive Data, and Peek Data.

The Object Invocation domain contains two micro building blocks: Prepare Method Invocation and Invoke Method. These MBBs automate server method invocation using the Java language reflection capabilities. Developers do not need to build skeletons for their server objects; they simply register them and the system automatically obtains all the information it requires.

The TCP Incoming Connection Management domain provides functionality to handle incoming TCP network connections. It exports two MBBs: Init and Accept.

The TCP Outgoing Connection Management domain handles TCP connection establishment with remote peers. The domain includes two micro building blocks: Connect and Return Communication Point.

The Object Registration domain is responsible for the management of server objects. It contains three MBBs: Register Object, Remove Object, and Get Object.

Table 1. ExORB size.

Domain	**Size**
CDR Parameter Management	16KB
XMLRPC Parameter Management	20KB
IIOP Protocol Processing	7KB
XMLRPC Protocol Processing	8KB
Network Data Management	3KB
Object Invocation	2KB
TCP Incoming Connection Management	5KB
TCP Outgoing Connection Management	4KB
Object Registration	2KB
Protocol Detection	1KB
URI Object Reference Management	2KB

The Protocol Detection domain exports functionality to identify the communication middleware protocol of incoming requests. This functionality is required to support the multi-protocol behavior of ExORB. It exports one MBB only: Detect Protocol. Current implementation of the MBB detects two types of protocols: XMLPRC and IIOP.

Finally, the URI Object Reference Management domain provides functionality to parse a remote object URI reference and extract all required information to send requests to the remote object. This domain contains a single micro building block called Reference to Object, which receives a URI and a protocol type, and returns a host name, a port number, and the object id.

Table 1 lists the size of each ExORB domain (Java version). The total size, without debugging information is 70KB. For the current implementation, each domain statically aggregates the micro building blocks. That is, micro building blocks are not installed individually but as a group. The numbers in Table 1 correspond to the size of the code of the domain.

4.2 Logic of ExORB

ExORB exports four actions: send request, receive request, init, and register object. The first one is intended for client-side functionality, while the remaining three (receive request, init, and register object) are intended for server-side functionality. Init and register object are single node interpreted actions, which simply invoke the init MBB and register object MBB described in section 4.1. In this section we provide a detailed description of send request.

Figure 11 illustrates the action graph for the *send request* action. To simplify the figure, we have removed the error states. When the client object invokes the action, it provides an action state object (the one storing the parameters generated during the execution of the action) containing the name of the action, the remote object's reference, the method to invoke, the required parameters, and the protocol to use (that is, XMLRPC or IIOP). The action starts invoking the *reference to object* micro building block, which parses the remote object's reference and extracts the hostname, object id, and port. These parameters are stored in the action state object.

Next, the action invokes *connect*, which obtains the hostname and port from the action state object, establishes a connection with the remote host (or reuses an existing connection), and stores an object that encapsulates the TCP socket (TCP Communication Point) in the action state object. The transition to the next state is conditional. It depends on the value of the "protocol" variable stored in the action state object. If the value of the variable is "iiop", the action invokes *CDR Marshal Parameters* to marshal the parameters, and then it invokes *IIOP Encode Request* micro building block to create the request message. If the value of the variable is "xmlrpc", the action invokes *XMLRPC Marshal Parameters* and then *XMLRPC Encode Request*. Both *IIOP Encode Request* and *XMLRPC Encode Request* micro building blocks generate a byte buffer with the request formatted according to the appropriate protocol. The next state in the action graph is *Send Data*, which retrieves the buffer from the action state object and sends it to the remote object using the TCP Communication Point object stored in the action state object. After invoking *Send Data*, the action retrieves a tuple named "oneway" from the action state object. If the value is "true", the action invokes *Return Communication Point*, which disposes the TCP communication point object from the action state object, and finishes, returning the action state object to the action invoker.

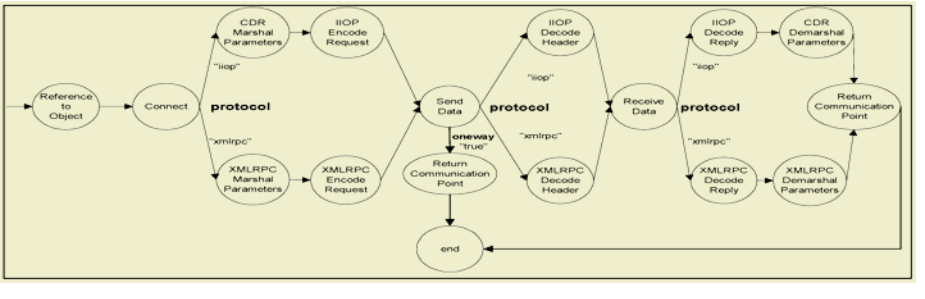

Fig. 11. Send Request action graph.

If the value of "oneway" is "false", the action continues with the decoding of the reply. First, depending on the value of the "protocol" tuple, the action decodes an

IIOP header, or an XMLRPC header. Both micro building blocks parse the message header and store information about the request in the action state object. One compulsory field for both micro building blocks is the length of the remaining of the reply. The action invokes *Receive Data*, which requires the length tuple to determine the amount of data that it has to read from the network. Next, the action proceeds with the decoding of the reply and the demarshaling of the parameters. Again, the action interpreter uses the value of "protocol" to decide what path to follow in the graph. Finally, the action invokes the *Return Communication Point* micro building block (disposes the TCP communication point) and terminates, returning the actions state object to the action invoker. The action state object contains the result parameters.

4.3 State of ExORB

A key feature of DPRS is the ability to manipulate the software state as a first class object. Every micro building block explicitly specifies its state dependencies, which are defined in terms of name and value pairs. These tuples are stored in a storage area provided by the micro building block domain. The state of the software is the union of all the micro building blocks' state attributes. The state of ExORB consists of all the state attributes defined by the 28 micro building blocks. Table 2 lists the state attributes associated to ExORB. The table includes the name of the attribute, its purpose, and the name of the domain that stores it.

Table 2. ExORB state attributes.

	Domain	**Purpose**
Sent Data (long)	Network Data Management	Stores the total amount of bytes sent by ExORB.
Received Data (long)	Network Data Management	Stores the total amount of bytes received by ExORB.
Send Timeout (long)	Network Data Management	Value in milliseconds the send MBB waits before timing out.
Receive Timeout (long)	Network Data Management	Value in milliseconds the receive MBB waits before timing out.
Server Object Registry (hash table)	Object Registration	Stores the list of registered server objects.
Server Communication Point Cache (list)	TCP Incoming Connection Management	Stores a list of connected communication points.
Client Communication Point Cache (list)	TCP Outgoing Connection Management	Stores a list of connected communication points.

5 DPRS Evaluation

In this section, we use ExORB to provide a quantitative and a qualitative evaluation of DPRS. For the quantitative evaluation, we present performance numbers and compare ExORB with a non-reconfigurable communication middleware. For the qualitative evaluation, we explain how we have successfully configured, updated, and upgraded ExORB using the functionality provided by DPRMS.

5.1 Quantitative Evaluation

The goal of this section is to examine the overhead incurred by DPRS. According to our experiments, this overhead is mostly due to domain memory accesses (currently implemented as a hash table). When an action is invoked, the MBB Scheduler parses the graph and accesses the logic memory to resolve each of the nodes of the graph. For each graph node, the scheduler obtains the MBB from the structure memory, and finally, during the MBB execution, the MBB might access the state memory and action state object to obtain and store state variables and input and output parameters. Note that the MBB execution is atomic and the MBB resolves any required state variable and input and output parameters at the beginning of its execution, and stores the values before completing its execution. That is, the MBB does not resolve the state variables and input and output parameters each time it needs to use them during an invocation; it caches their references until the end of its execution. For every action state, there is one access to the logic memory to obtain the action graph node, and another access to the structure memory to resolve the MBB. Then, each MBB accesses input (*input)* and output (*output)* parameters from the action state object, and state (*state*) variables from the state memory. Equation 1 illustrates the total number of memory accesses for an action with "n" states.

$$MemoryAccesses = 2 * n + \sum_{i=n}^{i=1} (input_{\_i} + output_{\_i} + state_i) \quad (1)$$

To measure the performance overhead of the Java implementation of DPRS (Java 1.4), we built a static version of the IIOP configuration of ExORB. We took the IIOP related MBBs' code, modified it, and created a collection of non-MBB Java objects. These objects have internal state (they do not access a hash table), use standard interfaces (instead of a generic "process" method), keep references to other objects, and are assembled statically. As a result, the new IIOP-based ORB does not incur any of the DPRS overhead but it cannot be reconfigured). We used this ORB as the performance baseline for our experiments. For the experiments, we created two objects that communicate using IIOP, a server that receives an integer, calculates its cube, and returns the result, and a client that invokes the remote method 10000 times and outputs the average requests per second value. We repeated the test 10 times and generated an average value, as well as the standard deviation. For the experiment, we used two machines connected to a 100Mbps Ethernet LAN. The server was a Pentium IV at 2.2GHz, with 512MB of RAM. The client was a Pentium M at 1.7GHz with 1GB of RAM. We run the experiment using the static ExORB implementation first, and then we repeated the experiment using the DPRS version of ExORB, followed by three optimized DPRS versions (we explain these optimizations next). Figure 12 illustrates the results of the experiments.

The left-most bar corresponds to the static version of ExORB with 4260 requests per second. The next bar to the right illustrates the performance of the unoptimized version of DPRS ExORB, which handles 1937 requests per second (45% of the static version's performance). The unoptimized version uses interpreted actions, and a hash table to implement the state memory and the action state object. The next bars on Figure 12 correspond to the performance of DPRS ExORB with a number of optimizations. The third bar from the left shows a version of DPRS ExORB that uses com-

piled actions. With compiled actions, we do not need to obtain each action graph node from the logic memory and therefore, for an action of "n" states we eliminate "n" logic memory accesses. As illustrated in figure 12, the improvement is not too significant (around 300 requests per second more, or 52.4% of the static ExORB performance). For the next optimization, we replace the state memory and the action state object hash table with an array of references, and use an index to access each variable. For this optimization, we need to process the logic and MBB descriptors to assign an index to each variable. We are currently creating a tool that parses the descriptors and generates the additional information automatically. This approach maintains the full flexibility of DPRS ExORB (we can reconfigure every aspect of the system at runtime) but requires additional steps when installing, removing, and reconfiguring the system (which can be automated). With this optimization we obtain 3126 requests per second, which corresponds to 73.3% of the performance of the static ExORB implementation. When using indices, we do not reduce the number of accesses to the domain memory; instead, we reduce the lookup time by avoiding the hash table. The last optimization uses indices and replaces the interpreted actions with compiled actions. As before, the improvement is not significant, we get an additional 112 requests per second.

Although the optimizations presented in this section are still work in progress initial results are promising. The architecture externalizing technique provides detailed information about the system that we leverage to build optimization tools that reduce the overhead of the system.

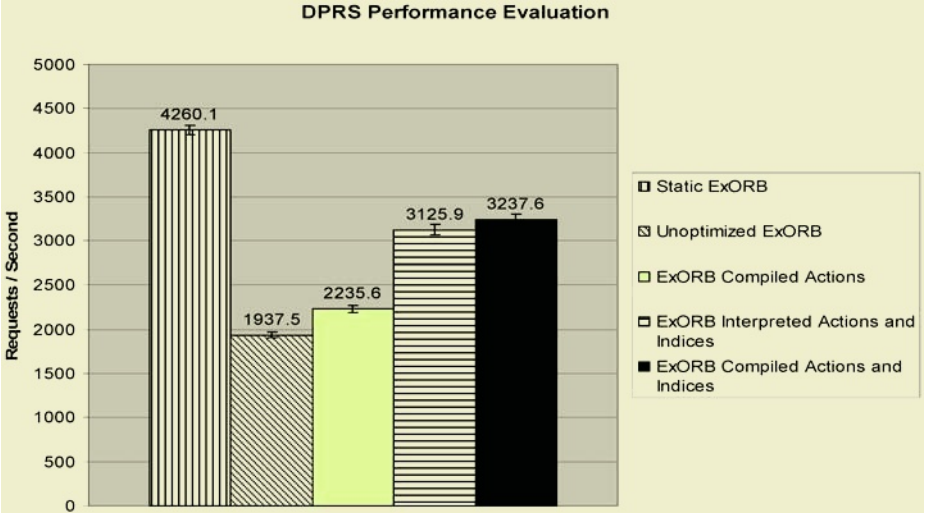

Fig. 12. DPRS Performance Evaluation.

5.2 Qualitative Evaluation

In this section, we show examples of ExORB configurability, updateability, and upgradeability. These examples leverage the basic functionality provided by DPRS and sustain the claims made in Section 2.1.

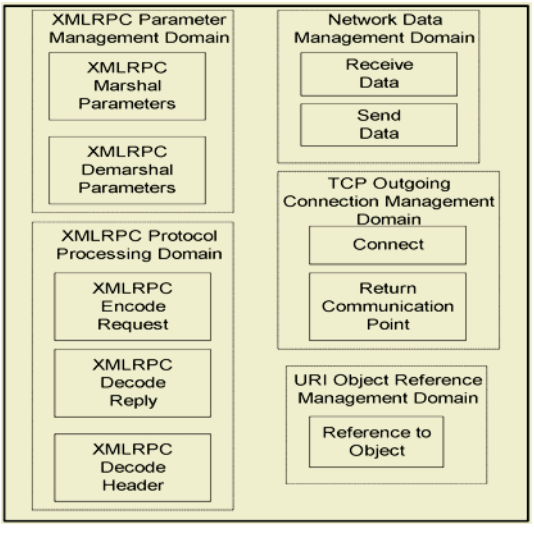

Fig. 13. ExORB configuration for client-side and XMLRPC only functionality.

For the **configurability** evaluation, we modify ExORB to provide client-side functionality and support for the XMLRPC protocol only. This configuration is particularly useful for resource constrained devices (for example, a sensor) that connect to a server periodically and send certain information (for example, temperature or pressure readings). Figure 13 illustrates the new configuration of ExORB, which removes the domains for CDR Parameter Marshaling, IIOP Protocol Processing, Object Invocation, TCP Incoming Connection, Object Registration, and Protocol Detection. Furthermore, we remove the Peek Data MBB from the Network Data Management Domain, and the Decode Request and Encode Reply MBBs from the XMLRPC Protocol Processing Domain. We edit the architecture descriptor to reflect these changes: modify the structure descriptor to remove the non-required MBBs, and modify the logic descriptor to remove the receive request and init actions. This configurability flexibility is the result of the micro building block construction model. The size of this configuration is around 43KB.

DPRS provides default support for **updateability**. We can replace any ExORB's MBB simply by interacting with the Domain Management domain, which provides functionality that guarantees the safe replacement of MBBs at runtime. Furthermore, the Domain Manager provides functionality also to modify existing actions. Figure 14 illustrates a modified version of the send request action (Figure 11), where we marshal the parameters first and connect to the remote object later.

Finally, **upgradeability** is also an integral part of DPRS. We have upgraded ExORB with functionality to encrypt and decrypt the data buffer before sending and receiving it. The upgrade requires adding an encrypting MBB, a decrypting MBB, and modifying the send request and receive request actions. The new actions invoke the encrypting/decrypting MBBs before and after sending data over the network. Figure 15 illustrates the changes to the send request action. The dashed circle corresponds to the encryption MBB, which is invoked after coding the request and before sending it over the network. Another example of upgradeability corresponds to the evolution of

ExORB. Our initial implementation provided IIOP functionality only. Later, we added XMLRPC capabilities by introducing new MBBs and modifying the existing actions.

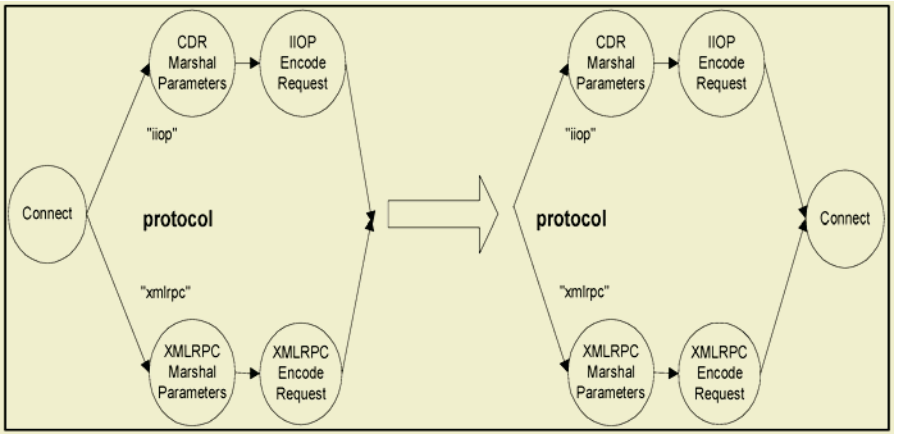

Fig. 14. Updating an interpreted action.

6 Related Work

For the related work, we compare DPRS with reflective middleware systems and dynamic software updating systems.

Reflective middleware [14, 15] refers to the capability of a service to reason about and act upon itself. Reflective middleware services provide a representation of their own behavior that can be inspected and modified at runtime. This representation is known as causally connected self representation (CCSR). Causally-connected implies that changes in the representation affect the underlying system and vice-versa. DPRS is a fully reflective (structural and behavioral) system. It provides a methodology to construct fully reflective middleware services. This approach contrasts with existing reflective middleware services that have to be designed having reflection in mind. That is, developers must decide beforehand those aspects of their services they plan to make reflective. DPRS supports structural reflection (supported interfaces) by means of action listing (actions correspond to interfaces). DPRS also provides architectural behavior by exporting the list of components and their interactions rules. Finally, DPRS provides behavioral reflection; it provides information about invocations' arrivals, and provides functionality to modify the behavior.

There is abundant work in the area of dynamic software updating systems [16]. These projects support executable code replacement at runtime. For example Hicks et al [6] describe a mechanism that relies on the OS linker to introduce code changes at runtime. Their work does not require a special software construction mechanism but they require developers to specify when it is safe to replace code. DPRS does not require developers to specify when it is safe to replace code. Its execution model can detect safe reconfiguration states automatically. Furthermore, DPRS provides information about the composition and execution state of the system. None of the traditional updating systems provides such functionality. Bitfone[7], Redbend [11], and

DoOnGo [12] are commercial products that support over-the-air cell phone firmware updates. Their approach is different from previous work on dynamic updates because they do not support partial image updates. They replace the whole cell phone firmware. Their algorithms calculate the differences between new and old images and transmit the differences to the phone update agent.

Finally, systems such as Ensemble[17] , Cactus[18], and the Dynamically Loadable Protocol Stacks[19] provide functionality to create updateable network protocols using state machines. The latter generates the protocol stack on the fly from a formal definition. The other systems assemble the stack from existing components but allow for changes in the stack (updates and upgrades). DPRS is similar to these systems in terms of dynamic composition and the use of state machines. However, DPRS allows inspecting the internal architecture and supports the construction of arbitrary software.

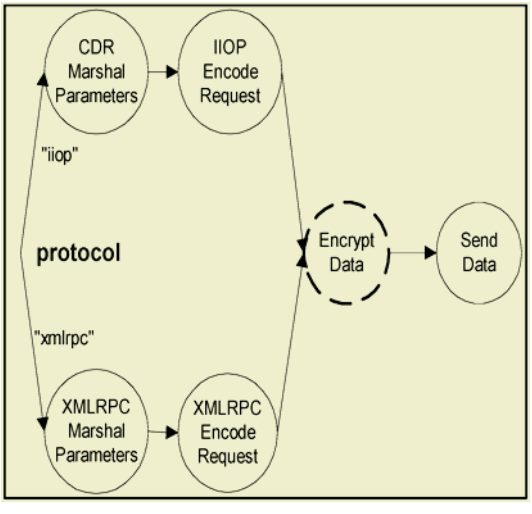

Fig. 15. Upgraded send request action that uses encryption and decryption.

Petri Nets [20] are a formalism to model concurrent asynchronous processes. Petri Nets consist of places (conditions), transitions (events or processes), arcs connecting places and transitions, and markings consisting of a number of tokens associated to each place. Unlike our system, Petri Nets do not directly address state, and structure externalization. Furthermore, Petri Nets do not define mechanisms for safe replacement of components. Finally, our system uses a scheduler that drives the execution of the system. With Petri Nets, state transitions are modeled asynchronously based on the firing of conditions. Note however, that we can leverage Petri Nets theory to model our systems.

7 Conclusions

In this paper, we present a technique to build dynamically programmable and reconfigurable middleware services. This technique relies on three abstractions: micro building blocks, actions, and domains. A micro building block (MBB) is the smallest

functional unit in the system that can be composed with additional MBBs to implement software functionality. An MBB receives input parameters, implements an algorithm that affects state attributes, and generates output parameters. An action is responsible for the coordination of MBBs and defines the logic of the system. Finally, a domain aggregates related MBBs and provides a storage area to save the state, the structure, and the logic of the system. DPRS supports the construction of configurable, updateable, and upgradeable middleware services that suit the requirements of next generation mobile handsets.

The architecture externalization technique gives software administrators and developers full control over middleware services. This supports phone evolution by allowing different software configurations, runtime updates, and runtime upgrades. The result is remotely managed cell phones that minimize or eliminate crashes and maximize user satisfaction by avoiding users from participating in maintenance tasks.

Our key assumption is that the individuals that benefit from architecture externalization are experts that know well their domain. For example, we assume that someone modifying ExORB will not insert an MP3 decoder micro building block between the parameter marshaler and the connector micro building blocks. Using the different system descriptors (logic, structure, and state) we implement static analysis to detect syntactic errors, such as mismatching number or type of input and output parameters. However, we do not provide any functionality to check for structural and logic semantic errors. Parlavantzas et al. [5] use component frameworks to address this issue. We leave the topic as future research work.

The architecture externalization technique has proven useful not only for configuring, updating, and upgrading software, but also for simplifying the suspension, resumption, migration, and partitioning of software. Accessing the externalized architecture allows services to automate these tasks without requiring any code from the original software developer.

One of the main concerns about DPRMS is the programming model. MBBs can be built using existing languages such as C, C++, Java, or C#. However, software development requires developers to think in terms of MBBs and actions. A solution to this problem is to provide tools that hide these extra steps. For example, it is possible to provide an IDE environment where users can define the actions visually and the system generates their XML representation (or even compiled code) automatically.

Finally, DPRS introduces a performance overhead, which we have been able to reduce to 25% with initial optimizations. We are currently working on additional optimizations and expect to reduce the current overhead even further.

References

1. http://www.nttdocomo.com/corebiz/imode/alliances/cmode.html.
2. http://www.nttdocomo.com/corebiz/imode/services/iarea.html.
3. A. Beaufour and P. Bonnet, "Personal Servers as Digital Keys," presented at International Conference on Pervasive Computing and Communications, Orlando, Florida, 2004.
4. http://www.3gnewsroom.com/3g_news/dec_02/news_2861.shtml.
5. N. Parlavantzas, G. Blair, and G. Coulson, "An Approach to Building Reflective Component-Based Middleware Platforms," presented at MSRC Summer Research Workshop, Cambridge, U.K., 2002.

6. M. Hicks, J. T. Moore, and S. Nettles, "Dynamic Software Updating," presented at (SIGPLAN) Conference on Programming Language Design and Implementation, Snowbird, Utah, United States, 2001.
7. Biftone, "http://www.bitfone.com/usa/index.html."
8. A. Andersen, G. Blair, V. Goebel, R. Karlsen, T. Stabell-Kulo, and W. Yu, "Artic Beans: Configurable and Reconfigurable Enterprise Component Architectures," *IEEE Distributed Systems Online*, 2001.
9. F. Kon, F. Costa, G. Blair, and R. H. Campbell, "The Case for Reflective Middleware," *Communications of the ACM*, vol. 45, pp. 33-38, 2002.
10. L. Capra, G. Blair, C. Mascolo, and W. Emmerich, "Exploiting Reflection in Mobile Computing Middleware," *Mobile Computing and Communications Review*, vol. 6, pp. 34-44, 2002.
11. Redbend, "http://www.redbend.com/."
12. DoOnGo, "http://www.doongo.com/us_web/."
13. P. Maes, "Concepts and Experiments in Computational Reflection," presented at Conference on Object Oriented Programming Systems Languages and Applications, Orlando, Florida, USA, 1987.
14. M. Roman, F. Kon, and R. H. Campbell, "Design and Implementation of Runtime Reflection in Communication Middleware: the dynamicTAO case," presented at ICDCS, Austin, Texas, 1999.
15. G. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. Saikoski, "The Design and Implementation of Open ORB v2," *IEEE Distributed Systems Online. Special Issue on Reflective Middleware*, vol. 2, 2001.
16. M. E. Segal and O. Frieder, "On-the-fly Program Modification: Systems for Dynamic Updating," in *IEEE Software*, vol. 10: IEEE, 1993, pp. 53-65.
17. R. v. Renesse, K. P. Birman, M. Hayden, A. Vaysburd, and D. Karr, "Building Adaptive Systems Using Ensemble," *Software - Practice and Experience*, vol. 28, pp. 963-979, 1998.
18. M. A. Hiltunen, R. D. Schlichting, C. A. Ugarte, and G. T. Wong, "Survivability through Customization and Adaptability: The Cactus Approach," presented at DARPA Information Survivability Conference and Exposition (DISCEX 2000), 2000.
19. S. K. Tan, Y. Ge, K. S. Tan, C. W. Ang, and N. Ghosh, "Dynamically Loadable Protocol Stacks. A Message Parser-Generator Implementation," in *IEEE Internet Computing*, vol. 8, 2004, pp. 19-25.
20. C.A. Petri, "Kommunikation mit Automaten" Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962