

A FRAMEWORK FOR BUILDING CUSTOMIZED ADAPTATION PROXIES *

Hana K. S. Rubinsztein, Markus Endler and Noemi Rodriguez

Departamento de Informatica, PUC-Rio

R. Marquês de São Vicente 225

22453-900, Rio de Janeiro, Brazil

{hana,endler,noemi}@inf.puc-rio.br

Abstract This article presents a framework for the development of adaptive proxies for context-aware mobile applications. The framework is in charge of collecting clients' current context (device and network), and trigger the appropriate adaptations. MoCA's *ProxyFramework* offers mechanisms for cache management, as well as for adaptation management. Developers need only create their application-specific adaptations (developing *adapters* modules) and define trigger conditions, priorities and selectors. This is done by specifying rules in XML format, which define the actions to be applied at the moment of a context change. The other extension point of the *ProxyFramework* is the caching policy to be used.

Keywords: Mobile Computing, Context-awareness, Proxy, Framework

1. Introduction

A common element in the architecture of distributed applications for mobile networks is a proxy [3, 4], which intercepts the messages exchanged between the mobile clients and servers, and which is in charge of executing a number of transformations, adaptations or management functions on behalf of one or several clients, such as content adaptation, protocol translation, caching, personalization, user authentication, handover management, etc. The main advantage of using such an intermediary is to bridge the *wired-wireless gap*, and make all mobility, connectivity and context-dependent issues transparent to the application developer.

Although each distributed application for such networks has specific adaptation and transformation requirements, there are a number of common and recurrent components and interaction patterns used for implementing usual

*This project is partially funded by CNPq, Grants 55.2068/02-2 and 47.9824/04-5.

adaptation and management functions. As a means of supporting the development of proxies for several applications for mobile networks, and enhance reuse of code, we are developing an object-oriented framework that can be extended and customized to produce concrete proxy instances according to the specific application requirements.

This work is part of a wider project, where we are implementing a middleware called *Mobile Collaboration Architecture MoCA* [8], consisting of APIs and services for context-provisioning and -processing, location inference, as well as mechanisms for notifying context changes to applications. Within MoCA, the framework will be used to generate instances of customized proxies for different context- and location-aware applications. Since most of the adaptations performed by a proxy are determined by the current execution context of a mobile client, e.g. the current wireless network or Access Point being used, the quality of the wireless link, or the availability of its local resources, the *ProxyFramework* includes functions to subscribe to MoCA's context services and mechanisms to trigger adaptations according to received notifications of context changes. The current focus is on content adaptation (e.g. distillation and transcoding) and caching, which are two central issues when developing adaptive applications for mobile devices and wireless networks.

2. The MoCA Middleware

MoCA [8] is a middleware architecture for the development of context-aware collaborative applications for mobile computing. It was designed for infra-structured wireless networks, and its current prototype works with an 802.11 wireless network.

MoCA offers client and server APIs which hide from the application developer most of the details concerning the use of the services provided by the architecture (see below). The *ProxyFramework* proposed in this paper is an element of MoCA. It is a white-box framework for developing and customizing proxies according to the specific needs of the application. It facilitates the programming of distributed, self-adaptive applications for mobile networks, where adaptations should be triggered by context-change events. The proxy not only intermediates the communication between the application server and its mobile clients, but also it serves as the interface with MoCA services, as Context Information Service (CIS).

The following MoCA services are in charge of collecting and distributing context information:

- *Monitor*: this is a daemon executing on each mobile device and is in charge of collecting data concerning the device's execution state/environment, and sending this data to the CIS (*Context Information Service*) executing on the wired network.

- *Context Information Service (CIS)*: This is a distributed service where each CIS server receives and processes devices' state information sent by the corresponding *Monitors*. It also receives requests for notifications (aka subscriptions) from application Proxies, and generates and delivers events to a proxy whenever a change in a device's state is of interest to this proxy. An example of proxy's request is given by the following *Interest Expression*, {FreeMem < 15% OR roaming=True}. The *Interest Expression* is defined as an SQL expression using some tags, as for example, EnergyLevel, CPU, OnLine, etc.
- *Location Inference Service (LIS)*: infers the approximate *symbolic* location of a device, using a specific context information of this device collected by CIS: the pattern of RF signal strengths received from all nearby Access Points.

3. Overview of MoCA's ProxyFramework

MoCA's *ProxyFramework* is being designed to accommodate a number of basic management and adaptation functions that an application proxy might be required to execute on behalf of each of its mobile clients. In fact, the *ProxyFramework* defines only abstract interfaces of proxy components and templates describing how these components interact. In order to implement application-specific adaptation and management functions, these components have to be extended or specialized by the application developer.

3.1 Main Components

The main envisioned components for *ProxyFramework* are described as follows:

- *Handover Management*: handles the tasks related to the migration of a client to a new network domain, such as, pre-allocation of resources at the new proxy, transfer of the client's (communication) session state, or of cached objects, to a new proxy, etc.
- *Caching Management*: is responsible for storing application-specific data, messages and user preferences of each client. This component incorporates the caching strategy, the concurrency and consistency strategy and memory management strategy (LRU, FIFO). The application developer can use a pre-defined set of management strategies, or customize some of them according to the specific needs of her application.
- *Adaptations*: implements any kind of adaptation (data compression, transcoding, summarization) of the application-specific data being transferred

from the server to the client, and vice-versa, according to the client's context.

- *Message Filtering*: is responsible for filtering of messages/data to be delivered to the clients according to their context and their profile.
- *Protocol Translation*: performs the transcoding from the specific wired protocol used by the application server to any of the possible wireless protocols used for interaction between the Proxy and the client.
- *Context Management*: performs the application-specific processing of the context information, such as: subscription for notifications from MoCA's CIS, analysis of context change notifications, diffusion of context information to other proxy components, etc.
- *Service Discovery*: is responsible for finding new services, users or data, according to the user profile. The lookup function will typically access some directory services, or receive some notifications from third-party "match-making" services.

3.2 Basic Steps to Use the *ProxyFramework*

In order to instantiate a proxy from the *ProxyFramework* the application developer has to follow two main steps: first, he has to implement the adaptation actions according to the specific needs of his application; and second, he needs to create trigger rules which define when (e.g. at which context condition) these actions are to be applied.

Defining Adaptive Actions. The *ProxyFramework* allows to condition the execution of certain proxy actions to specific states of the application client it represents. Since these actions are specific for each application, the proxy developer must implement them.

The actions are defined by the base class *Action*, which provides some common methods, as for retrieving action parameters. Essentially, there exist two types of actions: *adapters*, which modify a message, and *listeners*, which modify some state of the proxy related to a client.

Adapter actions are executed at the moment when a message is forwarded to the client, and depending on its current context. In order to implement a specific adaptation function, the developer has to extend method *execute* of the abstract class *Adapter*. This method gets the addressee of the message to be adapted and the message *per se*, and returns the modified message, or *null*. In the second case, the original message has been discarded and consequently the flow of adaptations is interrupted.

The actions of type *listener* react to changes in the state of clients. To implement a concrete listener, it suffices to extend the base class *StateListener*,

which has two abstract methods: `matches` e `unmatches`. The first is always executed when the corresponding state changes from OFF to ON, while the second is executed when it changes from ON to OFF.

Configuring Trigger Rules. The *ProxyFramework* uses a rule-based approach for determining which actions (adaptations) are needed in order to provide a better service according to the different environment conditions (context). The rule configuration should be done manually by the system administrator. With this configuration, the administrator can specify the proxy configuration for all environment conditions that the server wishes to support. The administrator can define the sequence of adaptations to apply to data and thus control the service composition, using any type of service.

The decision rules are composed by states (or contexts), that must be monitored; as well as actions which may be applied for each state. The states (or contexts) and the actions must be defined through a XML file.

```

<ProxyConf>
  <State>
    <Expression> <![CDATA[ OnLine = false AND DeltaT > 3000 ]]> </Expression>
    <Action class="proxy.listeners.DefaultCacheListener">
      <Parameter name="cacheClassName"> proxy.cache.FIFOCacher </Parameter>
    </Action>
  </State>
  <State>
    <Expression>
      <![CDATA[ CPU > 60 AND FreeMemory < 10000 ]]>
    </Expression>
    <Rule priority="1">
      <Filter>
        <!-- message data type -->
        <StartWith>
          <FieldValue> <Literal>datatype</Literal> </FieldValue>
          <Literal>image</Literal>
        </StartWith>
      </Filter>
      <Action class="proxy.adapters.ScaleImageAdapter">
        <Parameter name="factor">0.5</Parameter>
      </Action>
    </Rule>
  </State>
</ProxyConf>

```

Figure 1: Trigger Rules Configuration - XML file

Figure 1 shows an example of a *ProxyFramework* configuration file. In this example, element `State` represents a monitored state and has a single element `Expression`, which corresponds to the context *Interest Expression* that will be registered at CIS for periodic monitoring and delivery of corresponding notifications, whenever the expression switches from true to false, and vice-versa. When a change happens in either direction, the corresponding

customized *listener* action will be executed. Its configuration is done through element *Action*, where it is possible to indicate the class which implements the desired action, as for example, caching with FIFO policy. Each state may have several elements of type *Rule*, which aggregate several adapters which will be executed if the state for which they were registered is ON, and a certain condition related to the message (type) or the addressee is satisfied. The condition is determined through element *Filter*, which can be configured through the use of a number of logic and other operators, such as (AND, OR, NOT, EQUAL, STARTWITH) and available selectors such as (datatype, protocol, client, communicationmode, subject). Once the filter has accepted a message, the series of adapters registered for this rule will be executed. Adapters must also be registered with a rule, using the element *Action*.

It is possible to provide parameters both to the listeners and to the adapters, and this is done using element *Parameter* (each of which has a name and a value), as shown in the example.

4. Current Prototype of the *ProxyFramework*

The *ProxyFramework* consists of a set of basic functions and mechanisms for customizing, activating and combining adaptations, for the development of application proxies. Moreover, it provides the application developer with a simple means of accessing the client's context and defining context-dependent adaptations. The *ProxyFramework* was implemented in Java and offers these facilities through the structural reuse of components that are common to all application proxies, for example those for processing context notifications.

The framework is composed of a set of concrete components (*frozen-spots*), which implement utility functions for the proxies; and interfaces of abstract components (*hot-spots*), which can be implemented according to the specific need of each application. The frozen-spots include Communication, Caching and Adaptation Management and Selectors, while hot-spots are the Cache-Policies, Adapters and Listeners, and Context Configuration.

Essentially, the *ProxyFramework* is composed of two parts: the communication sub-system and the proxy *core*. While the first implements the protocols for synchronous and asynchronous communication with clients and servers, the second is responsible for collecting the context notifications regarding the clients and managing the execution of the adaptations according to the rules specified by the application developer (c.f. Section 3.2). Due to limitations of space, in the following we will further detail only the core.

4.1 Proxy Core

In order to achieve loose coupling among the different components of a proxy, and allow for their concurrent execution, the core architecture has been

structured as a set of independent elements called *Managers*, and a singular manager called *Dispatcher*, which intermediates the interaction between any pair of Managers, such as those described in Section 3.1. This way, a manager does not need a reference to all other managers it interacts with. This decoupling also facilitates the inclusion of new managers. Each manager has a private queue of messages, which are processed in FIFO order. The components of the proxy core are the following:

AdapterManager. It manages the message adapters, inspecting and modifying messages according to the specific states of the corresponding destination client. Once the states to be monitored have been defined, the proxy starts to trace the status of each state, for each client. This way, it is possible to establish a set of adaptation strategies to be applied to each message, for each client. The implementation of the specific adapters (c.f. section 3.2), the order of their execution, and the criteria for their application on each message type, are all customization points of the framework, which have to be defined/implemented by the application developer.

ContextManager. This component subscribes to MoCA's CIS according to the expression defined in the XML file (c.f. sec. 3.2) and receives messages from this context service about the current state of every client registered with the proxy. The ContextManager receives notifications from CIS (i.e. a CISMessage), whenever the interest expression (which defines a client state) flips between true and false. Essentially, a CISMessage contains three pieces of information: the client whose context changed; an identifier of the changed state; and the type of transition (i.e. ON, for a transition from off to on, and OFF, for a switch from on to off). Using this information, the state of the corresponding client is updated in the proxy. In this case, i.e. at the moment of this transition, it is possible to execute some specific actions of type listener, which modify the behavior of the proxy for the following message addressed to this client.

CacheManager. It is responsible for checking if according to the current state of a client, the messages addressed to it should be cached. This may be necessary when either the client gets (temporarily) disconnected, or the bandwidth of its wireless link falls below a given threshold. When a message for a client arrives, it verifies the state of the addressee, and then either records it in the cache, or forwards it to the AdapterManager.

The framework provides a special listener action for caching. This action is implemented through class `DefaultCacheListener`, which just activates or de-activates a given cache policy, which is passed as a parameter to this class and hence can be customized by the application developer.

The framework makes available a simple default caching policy, FIFO-Casher, which stores messages in FIFO order.

Sender. The Sender is responsible for delivering the intercepted messages to the corresponding addressee. This component implements a mechanism which ensures the ordered delivery of messages to each client.

Figure 2 depicts the logic relationship between the managers, and the message flow within the proxy core, from the moment it is received from the server until it is forwarded to the corresponding client.

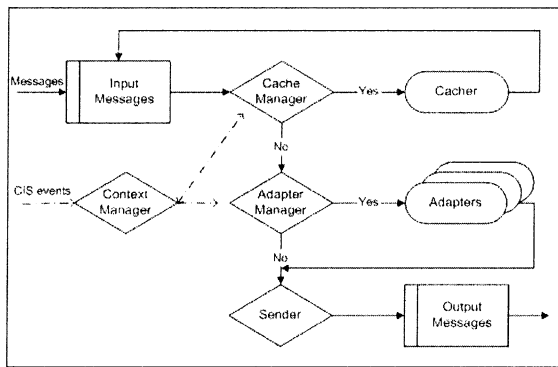


Figure 2: Message Logic Flow

Every incoming message is first inserted in the Input Message queue, and is then retrieved by the CacheManager, which checks if the message should be cached, or if it can be directly sent to the client. At the next stage, the message is sent to the AdapterManager which verifies which adaptations are to be applied to the message. After all adaptations, if any, have been applied the messages are enqueued in Output Messages, and are sent to the corresponding client in FCFS.

When caching is required, the messages are cached according to the caching policy defined by the developer. When the client's context changes, all of its cached messages return to the input queue, as if they were arriving at this moment. This is necessary due to the possibility that while some of these messages are being processed, the client's state changes, and some messages need to be cached again.

Our decision to implement the check for caching before the check for adaptation in the proxy's message flow was based on the understanding that the processing-intensive adaptations should be done according to the current client state, and only immediately before the message is sent to the client. Otherwise, the adaptations would not be effective, and hence useless.

5. A First Instantiation for Image Adaptation

The first instantiation of *ProxyFramework* was for an application that transfers and adapts images sent from a server to clients. The development of this context-aware proxy was simple and required only the implementation of some image adapters and the definition of trigger rules in the configuration XML file (c.f Section 3.2). The implemented adapters were for transforming color images into grayscale, for converting any image into JPEG with a compression quality, for scaling and for cropping.

Using our first proxy instance, we made some initial tests (using AspectJ) to evaluate the overhead introduced by the proxy. This overhead takes into account only the message management and queueing, the matching of the client state and the selection of the adaptation to be performed. It does not include the time spent on the adaptation *per se*.

In our experiments the scenario was composed by one server (source of images), one proxy and a set of clients, in which we varied the number of clients from 10 to 100. The proxy was configured with five states of interest and received images for adaptation at a rate of 2 messages per second. Each message for clients was of size 100 KB. For each set of parameters, we made 20 executions and calculated the mean value of proxy overhead. For these tests we did not cache the messages, but all the messages passed through the CacheManager, which did not act upon the messages. We executed the proxy on a 2.4 GHz Pentium 4 with 512 MB RAM.

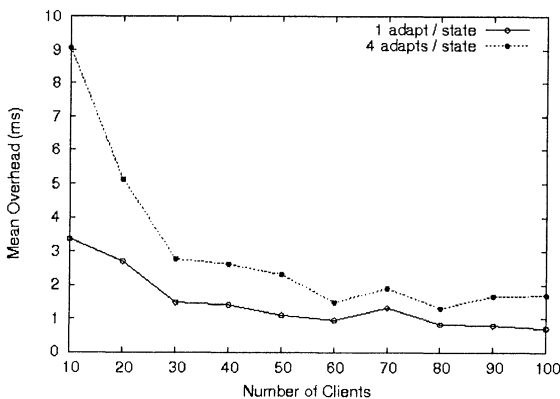


Figure 3: Number of clients x Overhead (msec)

Figure 3 shows the results of our measurements. As expected, the number of applied adaptations affects the mean latency within the proxy, since the messages stay more time in the queues waiting to be adapted.

In all curves the values for small number of clients are quite high, but this is caused by the fact that the initial Java class loading overhead is proportionally greater for fewer messages (due to fewer clients) than it is for a greater number of messages.

6. Related Work

Several other efforts have been made to develop generic proxy architectures, or *proxy frameworks*, that can be customized or extended to solve a particular problem, for example, Mobiware [1], RAPIDware [7], Web Intermediaries (WBI) [3, 6], MARCH [2] and TACC [4].

The decision of which adapters to use and when to use them can be defined in two ways: via programmable interfaces, as in Mobiware and TACC; or via rule-based configuration, as MoCA's *ProxyFramework*, MARCH and WBI. Rule-based systems are easily configured and less error prone (defining a model) than the ones based on programmable interfaces; besides there is no need to deal with intrinsic details of the framework. Furthermore, only the content provider can decide which adaptation is acceptable under different contexts, and thus, by using rules, may define the sequence of adaptations to apply to data, better controlling their composition, which is a very complex task to automate.

Comparing the two most common approaches for loading adapters, the dynamic loading of adapters, as in MARCH and RAPIDware, supports on-demand loading of adapters from an adapter repository, and provides more flexibility to the system. However, statically configurable proxies support verification of a consistent combination/configuration of adapters. In these proxies, the adapters are defined at proxy deployment time, like in WBI and *ProxyFramework*. In addition, dynamic (down)loading of adapters can be time consuming. Therefore, it is more suited for systems where context changes are not very frequent.

Comparing the systems, all of them support content adaptation, while some of them also implement caching management. Handover management is provided only by Mobiware. Concerning communication capabilities, only MoCA's *ProxyFramework* supports asynchronous (publish/subscribe) communication, which is very useful for mobile computing [5]. Context awareness is also supported by most of the frameworks (i.e. except WBI), but only MARCH and MoCA's *ProxyFramework* consider also the state of the client's devices. Our framework is the only one that supports connectivity-aware caching, where caching is automatically activated as soon as client's connectivity state changes.

7. Conclusion

As the number of applications for mobile networks increases, and their services become more complex and personalized, proxies will be used for an increasing number of specialized functions. Although each (type of) application will have specific demands for proxy based functions, we have identified a common and recurrent set of functions in proxy implementations which shall be used as the basis for developing proxies for specific needs. Based on our experience in developing some context-aware application prototypes, we felt that there is an increasing demand for flexible and extensible tools and frameworks for the rapid development and customization of proxy-based architectures.

In this paper we have presented a framework for the development of proxies for mobile computing. Our first prototype includes caching, message filtering and context-aware adaptations, since these form the core functionalities of a proxy. Our future work includes the design and development of components responsible for handover, authentication and translation for different mobile protocols. Another feature is the interaction with Location Services (as MoCA's LIS) in order to be able to implement location-based adaptations.

References

- [1] O. Angin, A.T. Campbell, M.E. Kounavis, and R.R.-F Liao. The Mobiware Toolkit: Programmable Support for Adaptive Mobile Networking. *IEEE Personal Communications Magazine, Special Issue on Adapting to Network and Client Variability*, August 1998.
- [2] S. Ardon, P. Gunningberg, B. LandFeldt, M. Portmann Y. Ismailov, and A. Seneviratne. March: a distributed content adaptation architecture. *International Journal of Communication Systems, Special Issue: Wireless Access to the Global Internet: Mobile Radio Networks and Satellite Systems.*, 16(1), 2003.
- [3] R. Barrett and P. P. Maglio. Intermediaries: An approach to manipulating information streams. *IBM Systems Journal* 38, IBM, 1999.
- [4] E. Brewer and et al. A network architecture for heterogeneous mobile computing. *IEEE Personal Communications Magazine*, October 1998.
- [5] Gianpaolo Cugola and H.-Arno Jacobsen. Using publish/subscribe middleware for mobile systems. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):25–33, 2002.
- [6] Steven C. Ihde, Paul P. Maglio, Jörg Meyer, and Rob Barrett. Intermediary-based transcoding framework. In *Ninth International World Wide Web Conference*, Amsterdam, The Netherlands, 2000.
- [7] Philip K. McKinley, Udiyan I. Padmanabhan, Nandagopal Ancha, and Seyed Masoud Sadjadi. Composable proxy services to support collaboration on the mobile internet. *IEEE TRANSACTIONS ON COMPUTERS*, 52(6):713–726, June 2003.
- [8] V. Sacramento, M. Endler, H.K. Rubinsztein, L.S. Lima, K. Gonçalves, and F.N.do Nascimento. MoCA: A Middleware for Developing Collaborative Applications for Mobile Users. *IEEE Distributed Systems Online*, 5(10), October 2004.