ISSN 0103-9741

Monografias em Ciência da Computação
n° 22/05

A Framework for Building Customized Adaptation Proxies for Mobile Computing

Hana Karina Salles Rubinsztejn

Markus Endler

Noemi Rodriguez

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 22/05 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena June, 2005

A Framework for Building Customized Adaptation Proxies for Mobile Computing

Hana Rubinsztejn, Markus Endler and Noemi Rodriguez

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

Abstract. This article presents a framework for the development of adaptive proxies. 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. It is possible to specify priorities and associate selectors to each adaptation. It is also possible to start an action at the moment of a context change. Developers need only to create their application-specific adaptations (developing *adapters* modules) and define trigger conditions, priorities and selectors, in the form of rules in XML format. In addition to the adapters, another extension point is the caching policy to be used.

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

Resumo. Este artigo apresenta um framework para o desenvolvimento de proxies adaptativos. O framework é responsável por coletar o contexto atual (rede e dispositivo) dos clientes e ativar as adaptações apropriadas. MoCA's *ProxyFramework* oferece mecanismos para gerenciamento de cache, bem como para gerenciamento de adaptações. Pode-se especificar prioridades e associar seletores para cada adaptação. Também é possível iniciar uma ação no momento da mudança de contexto. O desenvolvedor apenas precisa criar as adaptações específicas para sua aplicação (implementando módulos *adapters*) e definir as condições para adaptação, prioridades e seletores, na forma de regras em formato XML. Além dos adaptadores, outro ponto de extensão é a política de caching a ser usada.

Palavras-chave: Computação Móvel, Percepção de Contexto, Proxy, Framework

In charge for publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3114-1516 Fax: +55 21 3114-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

1 Introduction

A common element in the architecture of distributed applications for mobile networks is a proxy [4, 5], 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 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 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. its current Access Point, 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.

2 The MoCA Middleware

MoCA (Mobile Collaboration Architecture) 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).

In addition, the architecture offers the following core services which support the development of context-aware applications:

- *Monitor*: This is a daemon executing on each mobile device and is in charge of

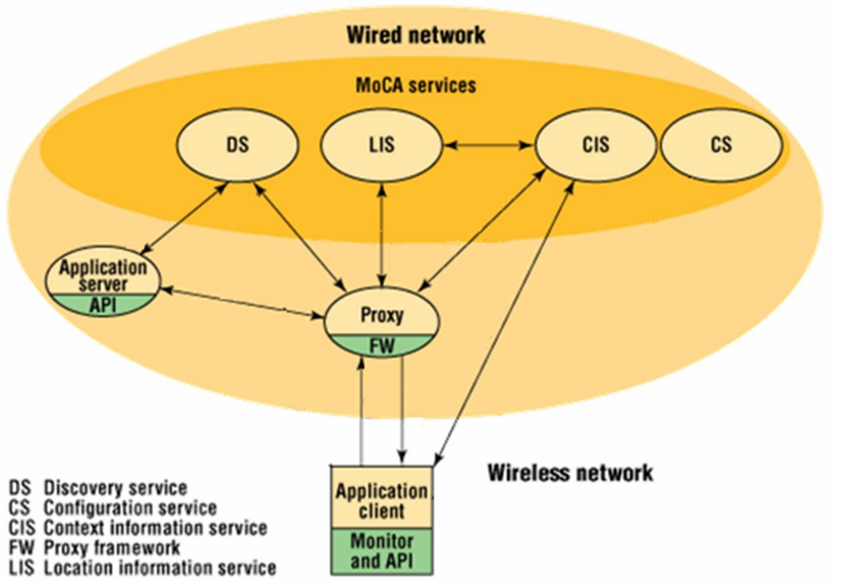

Figure 1: MoCA's core services

collecting data concerning the device's execution state/environment, and sending this data to the CIS (*Context Information Service*) executing on the wired network.

- *Configuration Service (CS)*: This service is in charge of storing and managing configuration information for all mobile devices, so that these can use MoCA's core services, such as CIS and *Discovery Service (DS)*.

- *Discovery Service (DS)*: This is a service in charge of storing information, such as name, properties, addresses, etc., of any application or any service registered with the MoCA middleware.

- *Context Information Service (CIS)*: This is a distributed service where each CIS server receives and processes devices' state information sent by the corresponding *Monitor*s. 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`}. Now, whenever the CIS receives a device's state information (from the corresponding Monitor), it checks whether this state change evaluates any *Interest Expression* to true. In this case, CIS generates a notification event and sends it to all Proxies which have registered interest in such change of the devices state. The *Interest Expression* is defined as an SQL expression using the tags listed in the Table 1.

- *Location Inference Service (LIS)*: This service 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.

Tag	Description
CPU	CPU usage (0 to 100%)
EnergyLevel	Energy level available (0 to 100%)
AdvertisementPeriodicity	Monitor's periodicity of sending of notifications (s)
APMacAddress	Current Access Point's Mac Address
FreeMemory	Total of available memory in kbytes
DeltaT	Context information freshness (ms)
OnLine	True if the mobile device is on-line
IPChange	True if the mobile device changes its IP
APChange	True if the mobile device changes of AP
Roaming	True if the mobile device is in roaming (same as "APChange" tag)

Table 1: CIS Tags

3 Framework Overview

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.

Figure 2 shows a high-level view of the architecture we envision for *ProxyFramework* , with its main components, which will be described in section 3.1.

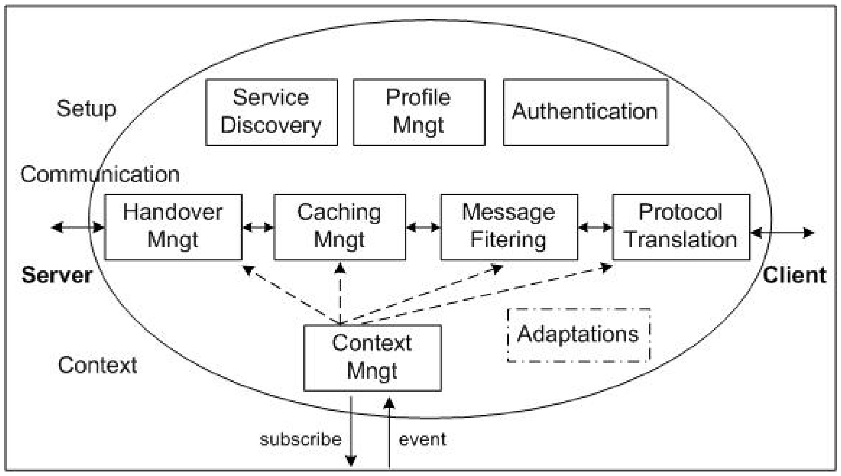

Figure 2: Proxy overview

3.1 Main Components

- *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, (when/what to cache) the concurrency and consistency strategy and (detection, invalidation and resolution of conflicts), 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. Commonly, this component is used for the detection of application-relevant complex context changes and the triggering of dynamic changes.
- *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.
- *Profile Management:* maintains the user's profile (e.g. her interests, skills, preferences) and interacts with a *Profile Matching Service* being developed to search for other users, data or applications witch match the user's profile.

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.

3.2.1 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.

```
public abstract class Adapter extends Action {

   public abstract Message execute(ClientInfo clientInfo, Message msg)
          throws AdaptationException;

}
```

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`. framework provides the identity of the client who suffered the state change, as well as information enabling the listener to forward messages to other proxy components.

```
public abstract class StateListener extends Action {

    public abstract void matches(ClientInfo info, Dispatcher disp)
           throws StateListenerException;

    public abstract void unmatches(ClientInfo info, Dispatcher disp)
           throws StateListenerException;
}
```

3.2.2 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 3: Trigger Rules Configuration - XML file

Figure 3 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 Architecture of the Current Prototype

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. In the following, we give an overview of these components:

Frozen-Spots

- Communication: implements both synchronous and asynchronous communication with the server and with the clients, using different communication protocols
- Caching Management: implements the cache management policy of the application, aiming the support for intermittent connectivity;
- Adaptation Management: is responsible for managing the application-specific adapters of messages. Adapters are activated according to the current context of each client, and to selectors related to the message type and destination.
- Selectors: responsible for evaluating conditions related to the client and the message, in order to determine the applicability of an adaptation;
- Context Manager: in charge of collecting and processing notifications of context changes for each client;

Hot-Spots

- Cache-Policies: The developer of the proxy can choose a specific cache management policy provided by this component, and extend it;
- Adapters and Listeners: implement the adaptations of the message contents;
- Context Configuration: used to describe the relevant context of the clients and the rules that trigger the adaptations (via a XML file).

Essentially, the *ProxyFramework* is composed of two parts: the communication subsystem 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.

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. 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, 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, as explained in Section 3.2.

ContextManager This component receives messages from MoCA's CIS 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. The CacheManager receives both internal control messages from other proxy components, such as from a CacheListener, and normal messages from clients registered at the Proxy. When a message from 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, FIFOCasher, 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 4 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.

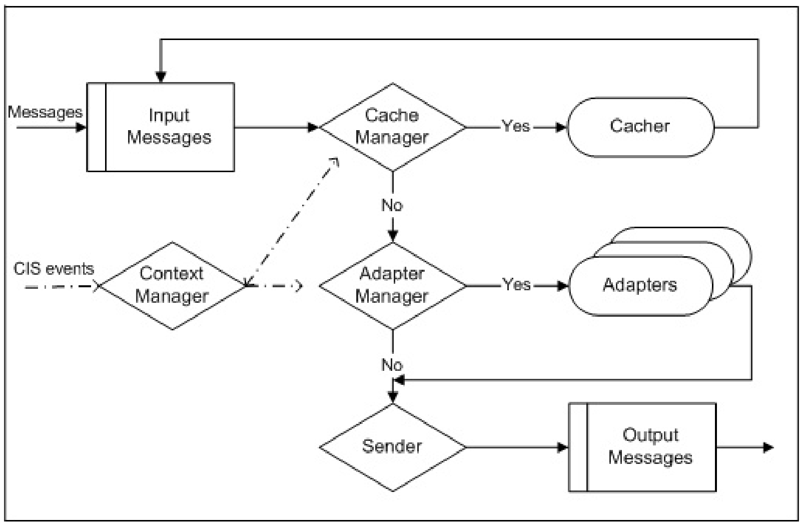

Figure 4: Message Logic Flow

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.

Selector	Description	Values
communicationMode	Client communication mode	SYNCHRONOUS ASYNCHRONOUS
protocol	The communication protocol used by the client	TCP, UDP
dataType	Message data type, MIME-like format	"image/jpeg"
client	Client identifier	
subject	Subject of a notification (pub/sub)	

Table 2: Currently provided selectors

4.2 Currently Provided Selectors

As explained in Section 3.2, selectors can be used to specify the condition used to select the messages on which adaptations are to be applied, according to the current client context. Selectors are specified for messages (and their content types) and/or for their destinations. Table 2 shows the current available selectors.

There are binary and unary operators to specify selectors:

- **Equals**: Binary operator that receives two strings as operands. It returns *true* if the operands are equal, and *false* otherwise;
- **StartWith**: Binary operator that receives two strings as operands. It verifies if the second operand is a prefix of the first one;
- **Literal**: Unary operator used to specify constant value (string);
- **FieldValue**: Unary operator that specifies the selector which will be used to evaluate the message. It receives a Literal as argument, to indicate the selectors name.

The following is an example of a filter specification, for image files, as it would appear in the XML file.

```
<Filter>
   <!--  message data type -->
   <StartWith>
      <FieldValue> <Literal>datatype</Literal> </FieldValue>
      <Literal>image/</Literal>
   </StartWith>
</Filter>
```

Selectors can also be combined through the use of logic operators, such as AND, OR and NOT, allowing for the definition of complex expressions for message selection.

4.3 Communication

The framework offers support for synchronous and asynchronous communication for both the client-proxy and the server-proxy communication. So far, only TCP and UDP can be chosen for both communication types, but we intend to extend the *ProxyFramework* 's support also for other protocols, specilly for the communication with mobile clients.

After receiving the message, a proxy instantiation, should decide if the message will be retransmitted, in which case, it should use one of the several methods of sending of messages provided by the framework.These methods make a distinction among types of communication and also messages for customers or for servers. Messages for the servers are just forwarded according to the configured protocol, while messages for clients may be adapted according to the current client's state and to the addressee's communication protocol.

For asynchronous communication, the framework is based on MoCA 's Event Service ECI, but some modifications were necessary for the framework to be able to intermediate the messages between the server (acting as a publisher), and clients (in a subscriber role).

Unlike in conventional pub/sub communication, where a message published on a certain topic is sent to all subscribed hosts, the framework needs to intercept the message and do specific adaptations for each client individually, according to the client's current context. Hence, the proxy must act as a subscriber to the server, and as an event publisher for all the clients.

Therefore, the proxy instantiation must subscribe to the server for the topics of interest of its clients, and when receiving a message, publish it to the interested clients. The framework verifies which clients have subscribed to the topic of the message and then creates copies of the message for each client, such that adaptations (specific for each client) can be applied to each copy. Finally it sends to each client the corresponding adapted message.

5 A first instantiation for Image Adaptation

The first proxy instantiation was for an application that transfers and adapts images sent from a server to clients. The development of this proxy was simple and required only the implementation of some image adapters, which are described in the section 5.1. In section 5.2 we then present preliminary performance results for the proxy, and the overhead incurred on message delivery.

5.1 Developed Adapters

In the following we enumerate the image adapter classes we have developed, their functionality and their parameters.

Class *ColorToGrayAdapter* converts a color image into grayscale, maintaining the image type.

Class *ConvertToJPEGAdapter* converts any image into JPEG format with a predefined compression quality. This quality is determined by parameter *compressionQuality* expressed in the XML proxy configuration, as shown below. The parameter, which accepts values in the range [0,1], defines the quality of the compressed image as well as the compression rate.

```
<Action class="proxy.adapters.ConvertToJPEGAdapter">
   <Parameter name="compressionQuality">0.4</Parameter>
</Action>
```

Class *CropCenterAdapter* is an adapter which chops off the borders of an image, creating a new image that contains only the central rectangle of the original image. The parameters *height* X *width* define the size of the rectangle.

Class *ScaleImageAdapter* scales an image by a pre-defined factor, making it larger (e.g. $factor > 1$), or smaller (e.g. $0 \leq factor < 1$)[1].

5.2 Performance Results

Using our first proxy instance, we made some tests (using AspectJ [7]) 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*, nor the network latency.

In our experiments the proxy was configured with one and five states of interest and received images for adaptation at a rate of 2 messages per second. Each message was of size 100 KB, and we varied the number of clients from 1 to 100. For each set of parameters, we made 20 executions and calculated the mean value of the proxy overhead. For these tests we did not use caching the messages. However, 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.

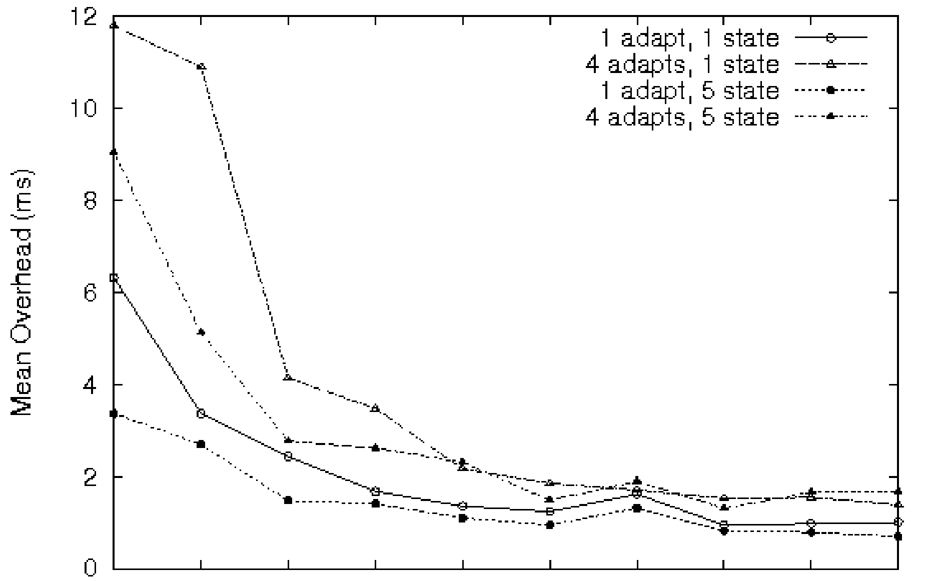

Figure 5: Number of clients x Overhead (msec)

[1]If *factor* is negative, the image will be reduced by 10%

Figure 5 shows the results of our measurements. As expected, the number of applied adaptations affects the mean latency of memory management within the proxy, since the messages stay more time in the queues waiting to be adapted. Contrary of our expectations, the proxy performs just a little better when there are 5 states of interest, rather than with 1 state.

In all curves the values for small number of clients happens to be 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 clients.

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], Web Intermediaries (WBI) [4, 6], MARCH [2] and TACC [5] .

The main customization point of a proxy framework is the *adapter*[2], a module responsible for implementing the adaptation functionality. In some contexts, more than one adapter can be selected for adapting a message. Therefore, some frameworks, as MARCH, Mobiware, *ProxyFramework* support the definition of priorities, ordering, and/or composition of adapters.

Table 3 summarizes the main characteristics of the above systems.

	Mobiware	MARCH	TACC	WBI	MoCA Framework
Purpose	Multimedia, QoS	General	General	Web	General
Level	Middleware	Application	Middleware	Application	Middleware
Dynamic Adapter Loading	Yes	Yes	No	No	No
Adaptation Selection	Programmable	Trigger-Rules Configuration	Programmable	Trigger-Rules Configuration	Trigger-Rules Configuration
Funcionalities	Content Adaptation, Handover Mngt	Content Adaptation	Caching, Content Adaptation	Caching, Content Adaptation	Caching, Content Adaptation
Communication	Synchronous	Synchronous	Synchronous	Synchronous, Asynchronous	Synchronous, Asynchronous
Context Awareness	wireless link	device & wireless link	wireless link	-	device & wireless link

Table 3: Comparison table of extensible proxy approaches

Comparing the systems, all of them support content adaptation, while some of them also implement caching management, and handover management is provided only by Mobiware. Concerning communication capabilities, only MoCA's Framework and WBI support asynchronous (publish/subscribe) communication, which has been recognized as best suited for mobile computing. Context awareness is also supported by most of the frameworks (i.e. except WBI), but only MARCH and the MoCA Framework consider also the state of the client's devices.

[2]Some publications use different names for the adapter, such as *filter* [3], *transcoder* [4] and *worker* [5].

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 *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 Mobiware, 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.

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 Netwoking. *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] A. Balachandran, A.T. Campbell, and M.E. Kounavis. Active filters: Delivering scalable media to mobile devices. In *Seventh Intl. Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV), St Louis*, May 1997.

[4] R. Barrett and P. P. Maglio. Intermediaries: An approach to manipulating information streams. IBM Systems Journal 38, 1999.

[5] E. Brewer and et al. A network architecture for heterogeneous mobile computing. *IEEE Personal Communications Magazine*, October 1998.

[6] Steven C. Ihde, Paul P. Maglio, Jrg Meyer, and Rob Barrett. Intermediary-based transcoding framework. In *Ninth International World Wide Web Conference*, Amsterdam, The Netherlands, 2000.

[7] Ramnivas Laddad. *AspectJ in Action - Practical Aspect-Oriented Programming*. Manning Publications Co., 2003.

[8] V. Sacramento, M. Endler, H.K. Rubinsztejn, L.S. Lima, K. Gonalves, and F.N.do Nascimento. MoCA: A Middleware for Developing Collaborative Applications for Mobile Users. *IEEE Distributed Systems Online*, 5(10), October 2004.