Disconnected Operations in Mobile Environments

Denis Conan, Sophie Chabridon and Guy Bernard
Institut National des Télécommunications
9, rue Charles Fourier
91011 Évry cedex, France
{Denis.Conan|Sophie.Chabridon|Guy.Bernard}@int-evry.fr
Phone: Denis Conan: (+33|0) 1 60 76 45 34, Fax: (+33|0) 1 60 76 47 80

Abstract

The execution of distributed applications involving mobile terminals and fixed servers connected by wireless links raises the need for handling network disconnections, both involuntary during unexpected connection breakdowns, and voluntary when the user wants to save money and energy. In this paper, we investigate how standard CORBA mechanisms (Objects By Value and Portable Interceptors) can be used for enhancing legacy CORBA-based distributed applications in order to support voluntary and involuntary disconnections. We show that the first mechanism can be used for handling easily voluntary disconnections by copying on the terminal all the data necessary for running the application in a disconnected mode. The second mechanism allows also to handle involuntary disconnections ; a switch between connected mode and disconnected mode can be performed transparently to the user.

1 Introduction

Wireless communications and distributed services will be of vital strategic importance in the near future. Right now, mobile handheld devices such as Personal Digital Assistants (PDA) are attracting great attention and the availability of wireless telecommunication networks provide new prospects for client-server applications and their interactions. However, software capacities of mobile handheld devices are rather modest, the connectivity to services via the Internet is limited and the number of available services is very low compared to the possibilities of home and personal computers connected to the Internet via wired links.

The work presented in this paper is part of the ITEA Vivian project [2] concerning the opening of mobile platforms for the development of component-based applications. The consortium is composed of major european industries such as Nokia and Philips, research institutes, namely HUT in Finland and INRIA and INT in France, and providers of services on mobile handheld devices. The objective of Vivian is to provide an adequate platform for mobile handheld devices which will enable the production of third-party software applications. In that context, we focus on providing CORBA compliant software support for the execution of client-server applications in a disconnected manner in wireless environments. Two kinds of disconnections are taken into account: voluntary disconnections, decided by the user of the wireless handheld, and involuntary disconnections, as a result of physical wireless communication breakdowns. From the user point of view, the expected benefits of voluntary disconnections are: money and energy savings by reducing the duration of wireless communications, possibility to run the application when no wireless connection is available (*e.g.* in a plane), and minimization of the probability of unexpected disconnections.

The goal of this paper is to investigate how standard CORBA mechanisms can be used for enhancing legacy CORBA-based distributed applications in order to support both voluntary and involuntary disconnections in mobile environments. For illustration and evaluation purpose, we chose an email browser application. The structure of the paper is the following. In Section 2 we present the email browser application and the extented functionalities needed for supporting disconnections in a wireless environment. In Section 3 we discuss how standard CORBA mechanisms (Objects By Value and Portable Interceptors) can be used for supporting these functionalities. Section 4 compares our approach with related research work, and finally conclusions and future research directions are drawn in Section 5. We developed a prototype using the ORBacus ORB [1] on Portable PCs under Linux and Windows using `Java`, namely the `jdk 1.3.1` virtual machine, and on iPAQ using also Java but with the `VAME` virtual machine called `j9`. All the machines are equipped with wireless IEEE 802.11

communication cards.

2 Example application: Wireless Email Browser

This section illustrates the adaptation of an email browser for wireless environments. Our email browser offers the basic functionalities of well-known software such as Netscape Messenger or Microsoft IE. The user handles messages composed of a body and a header, itself divided into an identifier, the names of the sender and the receiver, a subject, the date of sending, and a status (read or unread). The main functionalities provided by the graphical user interface (GUI) are sending, replying to, forwarding, receiving, and deleting a message. The adaptation to disconnected operations is the subject of the next sections.

In the first version of the email browser named “centralized”, the GUI is executed into the same execution entity as the user mailbox object. The mailbox object plays two roles: *(1)* the mailbox object stores received emails ; *(2)* for sending a message, the GUI sends the message to the mailbox object, the latter gets the receiver’s address from the mailbox manager object and forwards the message. A second execution entity contains the mailbox manager that is responsible for creating, deleting, and localizing the mailbox objects.

The second version of the email browser named “distributed” is obtained by separating the GUI and the user mailbox object into different execution entities (*cf.* figure 1). The GUI is launched by the user in the mobile terminal and communicates with the corresponding mailbox object via wireless links. Since some of the data that are copies of the mailbox object’s data are locally stored within the GUI, the distribution leads to the separation of GUI’s operations into two groups: the operations that only impact local (GUI’s) data and the ones that are carried over the mailbox object straight after being executed by the GUI. A typical operation of the first group is the operation `changeStatus()` of the GUI saying that a message has been read or unread. In the centralized version, the operation is synchronously performed on the mailbox object. Now, it is applied and logged by the GUI in order to avoid generating too many requests on the wireless network. At the next remote operation execution, for instance a call to `receiveMessage()`, the log of local operations is transmitted as an argument and applied to the mailbox object before the processing of the remote operation. So, the effects of `changeStatus()` are seen before the effects of `receiveMessage()`, as in the centralized version. Therefore, the GUI logs all its local operations since the last remote operation that do not need to be processed remotely in a synchronous way. Another consequence is that all the remote operations have as their first argument an array containing the list of local operations. This design pattern is rather simple and can apply to distributed applications that are piece-wise deterministic[1] [5].

However the quality of the wireless link, in order to load data from the mailbox object at a convenient rate and quantity, messages are read in two steps: first the header, next the content. The user browses the set of headers and loads only the desired contents. The reasons for this distinction are that contents are usually much larger than headers, and being optimistic, users will not read every content whereas they read all the headers. Another way to adapt to the wireless link is the addition of “collective” operations that read and delete groups of messages: *e.g.* all the read/unread messages, all the messages.

The rest of the paper is on the study of how to handle disconnections using CORBA mechanisms, namely Objects By Value and Portable Interceptors. Figures 2 and 3 sketch the solutions designed in the following sections. Before the disconnection, a copy of the mailbox object is instantiated on the mobile terminal. During the disconnection, the requests are locally served and logged by the local copy. The received messages are also stored in the remote mailbox object for future delivery. When reconnecting, the logged requests are forwarded to the remote mailbox object and the received messages are loaded if the user asks for it. The GUI is augmented with an iconic image stating the current mode: connected or disconnected.

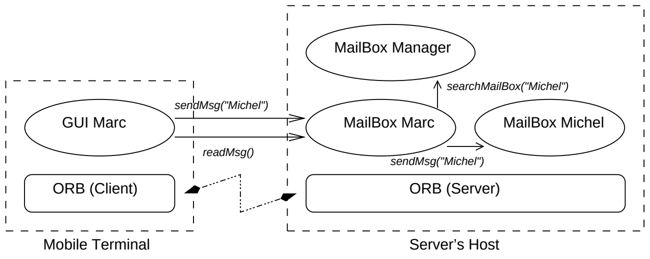

Figure 1. The email browser in the connected mode.

[1]The execution of each execution entity is divided into separate intervals by the messages the entity receives. Each interval is a deterministic sequence of execution, started by the reception of a message and ended by the reception of the next message. The execution within a single interval is completely determined by the state at the time the message is received and by the content of the message. The entity may send any number of messages to other entities during the state interval.[5]

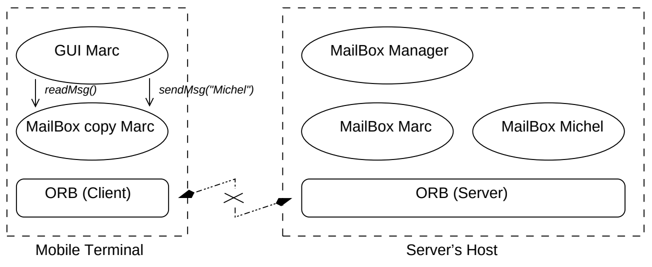

Figure 2. The email browser in the disconnected mode.

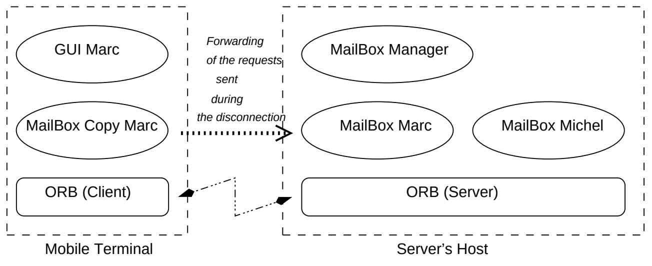

Figure 3. The email browser during a reconnection.

3 Handling disconnections

In the first two versions of the application wireless email browser, the GUI cannot continue its execution when the user closes the connection, for instance to save money or when the connection is broken. The GUI should access a local object that should be a copy containing all the state of the original one that remains on the wired network or an empty default-state copy. There are two ways to proceed: either the user asks the GUI to make a local copy of the remote object and then disconnects voluntarily or the application periodically updates a local copy and switches to it in case of connection failure. In the following, we develop the two cases in two distinguished solutions using two different CORBA mechanisms, namely Objects By Value (OBV) and Portable Interceptors (PI). The first solution with OBV is developed in Section 3.1, the main issue being the state transfer of the remote object. The second solution is divided into the following issues: the state transfer of the remote object (*cf.* Section 3.1), the monitoring of the connection to detect disconnections (*cf.* Section 3.2), the transparent switching between modes (*cf.* Section 3.3), and the design of the local copy of the remote object (*cf.* Section 3.4).

3.1 State transfer

Before presenting the solution using OBV in Section 3.1.2, Section 3.1.1 gives a short overview of this CORBA mechanism.

3.1.1 Short overview of OBV

The concept of OBV was introduced in the CORBA 2.3 standard [10]. It enables the passing of an object by value rather than by reference. The semantics of passing an object by value is similar to that of standard programming languages. The client receives as a return value a description of the state of the object and a new instance is automatically instantiated with that state which has a separate identity from that on the server side.

CORBA 2.3 introduces new keywords to the Interface Definition Language (IDL): `valuetype`, `ValueBase`, `custom`, `abstract`, `supports`... We will present only part of them here. A valuetype is a new IDL construct that can be thought of as a `struct` with inheritance and methods. The valuetype is passed by value, like the `struct`, but can contain operations, like the `interface`. An essential property of valuetypes is that their implementation is local and their use does not involve the Object Requests Broker (ORB). They have no Interoperable Object Reference (IOR), which is the way CORBA identifies remote objects. An `abstract interface` allows to determine at runtime whether an object is to be passed by value or by reference. An `abstract interface` can not be instantiated directly. It must either be inherited by a regular interface or supported by a valuetype (using the `supports` keyword).

3.1.2 Use of OBV for state transfer

The impact of the addition of the wireless mode to a distributed legacy application is not negligible but we payed particular attention to limit it as much as possible in the IDL and the client code.

The abstract interface functionality makes it very easy to add in the client code a voluntary switch between a remote object present on the server side and a valuetype copy on the client side. Therefore, an abstract interface named `AbstractMailBox` is defined in the IDL with all the necessary methods to manipulate messages. This abstract interface is then supported by a valuetype named `ValueMailBox` and is inherited by the regular interface of the mailbox object. The new interface provides two new methods with respect to the legacy interface: `disconnect()` is called by the user for initiating state transfer and `reconnect()` allows to apply to the remote object the modifications that occurred locally on the client side during disconnection. In the client code of the legacy application, the reference to the remote object must be replaced by a reference to the abstract interface, which contains a reference to a remote object and to a valuetype instance in connected and disconnected modes, respectively.

We now detail the way we perform state transfer from a remote object to its valuetype copy. In [7], the main drawback found to the use of OBV for disconnected operations was the many changes required on client and server sides. A stateful valuetype must define in the IDL a public data field for each state variable (public data fields and attributes) present in the supported interface (see [7] p. 66) and then implement the corresponding accessor and modifier methods. To overcome this drawback, the custom marshalling facility of OBV requires the valuetype to implement the `marshal()` and `unmarshal()` methods and to be tagged `custom`.

In conclusion, the prototype we developed with OBV satisfies the requirements for voluntary disconnection with limited modifications in the IDL and a very simple use in the client code. Nevertheless, the interface must have operations that checkpoint and restore the state of the object. In the case of OBV, this is the operations `marshal()` and `unmarshal()`. In addition, it is not well adapted to involuntary disconnections where it is necessary to switch automatically between two copies of the same object, one that is on the server and one that is local to the client. This is due to the fact that a valuetype is not addressable via an IOR since it is not a CORBA object. These aspects are considered in Section 3.3 where we investigate a solution with portable interceptors.

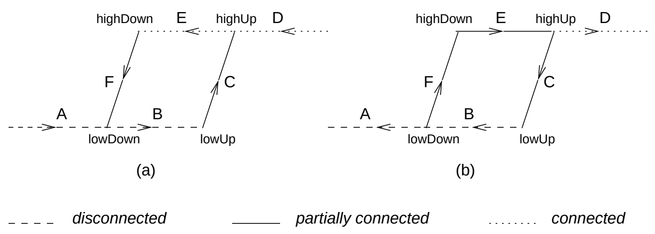

Figure 4. The hysteresis of the monitoring of the connection state.

3.2 Connection monitoring

Since no entity provides the information yet, we design a connection monitor to simulate the quality of the wireless

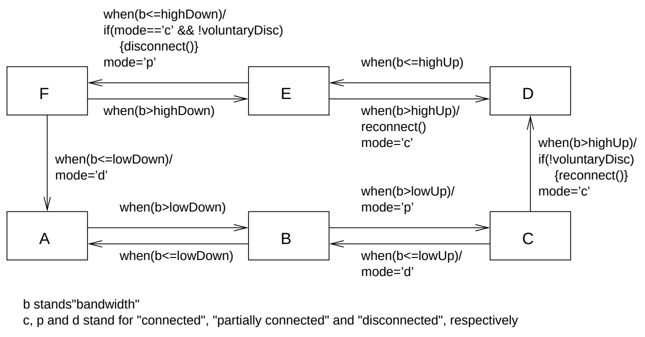

Figure 5. The state diagram of the monitoring of the connection state.

connection. This monitoring involves a connection bandwidth monitor and a connection state monitor on the mobile terminal. We expect hardware and software manufacturers providing in some time a system call giving the real bandwidth or signal noise ratio.

The connection state monitor implements an hysteresis mechanism for smoothing bandwidth variations (*cf.* figure 4 and figure 5). The hysteresis defines three modes: “disconnected” when the request is only performed by the local copy on the terminal ; “connected” when the request is only performed by the object on the wired host ; “partially connected” when the request is performed by the local copy that transmits the call to the remote object. Since the local copy does not perform the operations when the GUI is (directly) connected to the remote object, the former is not “up to date”, and when going from connected to partially connected, a state transfer is necessary. When the GUI is disconnected or partially connected, the local copy performs the operations and is “up to date”, except for messages that were recently received by the remote object. The user must explicitly call a receive operation to know if there are new messages received . Of course, we can adapt the GUI to do that periodically. Therefore, the third mode named “partially connected” is introduced to avoid the “ping-pong effect” between the two other modes. What we define as the “ping-pong effect” occurs when small variations around a value of the bandwidth imply successive state transfers.

On diagram 4.a, when the value of the bandwidth increases and is lower than `lowUp` (*resp.* `highUp`), the mobile terminal is disconnected (*resp.* partially connected). When the value of the bandwidth decreases and is higher than `highDown` (*resp.* `lowDown`), the mobile terminal is connected (*resp.* partially connected). Without diagram 4.b, observe that there still exists a risk of “ping-pong effect” around the value `highDown`. Thus, when the connection is in state `F` and the bandwidth becomes higher than `highDown`, the connection stays being partially connected up to the value `highUp` in case the bandwidth rapidly decreases again.

3.3 Transparent switching between modes

Before developing the solution using portable interceptors in Section 3.3.2, Section 3.3.1 gives a short introduction to CORBA portable interceptors.

3.3.1 Short overview of portable interceptors

"*Portable interceptors are hooks into the ORB through which ORB services can intercept the normal flow of execution of the ORB*" [9]. Three types of interceptors provide eleven interception points. Firstly, IOR interceptors give the application an opportunity to modify IORs, more precisely IOR profiles, before they are exported and seen by clients. When registered in an execution entity, an IOR interceptor acts on every object creation. In our work, this kind of interceptor serves to add a component to all profiles. The presence of such a component in a profile indicates that the object belongs to an application supporting the disconnected mode. In addition, the component contains a CORBA policy object that itself includes a boolean value stating whether the object can be copied on the mobile terminal. Secondly, client-side interceptors introduce five interception points in a request and reply sequence on the client side. Two of them intervene before a synchronous or asynchronous request is sent by the ORB. The last three allow to parse the reply: normal (successful) or exception, or other than normal and exception, before the control returns to the client. When registered in an execution entity, a client-side interceptor acts on every request and reply regardless of which types of components the IORs have. Nevertheless, when the IOR profile possesses the component "disconnected mode", special treatments are applied (*cf.* Section 3.3.2). Finally, server-side interceptors define five other interception points, two on the reception of requests and three on the sending of replies. The server-side interceptors are presented here for the sake of completeness but are not used in the prototype.

Portable interceptors is the CORBA mechanism used by CORBA services to transparently add extra-functional services to applications. In fact, portable interceptors are instantiated and registered to an ORB during the creation of the ORB when invoking the method `ORB.init()`. This operation can be transparent —*i.e* the source code of the application does not need to modified. In the following section, the solution follows the same approach.

3.3.2 Transparent switching between modes

We now detail the way we perform the transparent switching between modes using IOR and client-side interceptors. When the server on the wired host starts, an IOR interceptor is registered at the creation of the ORBs. As a consequence, all the remote objects have by default the policy "disconnected mode". When the GUI starts, an IOR interceptor and a client-side interceptor are registered at the creation of the ORBs. The IOR interceptor of the client is the same as the server's one. As a result, local copies on the mobile terminal also possess the policy "disconnected mode". When the GUI sends its first request to an object whose IOR possesses the policy "disconnected mode", a local copy is instantiated on the mobile terminal. Likewise, every request is intercepted and the client-side request interceptor calls the connection state monitor that itself calls the connection bandwidth monitor (*cf.* 3.2). Depending on the state changes of the connection, the client-side request interceptor can build a CORBA `ForwardRequest` exception indicating the change of target IOR and throw that exception. The exception is automatically managed by the ORB. The effect is a transparent switching of target object: from the remote object to the copy on the mobile terminal and *vice versa*.

In conclusion, the prototype we developed with Portable Interceptors satisfies the requirements for transparent switching between connected mode and disconnected mode with no modifications of the IDL and no modifications of the client. In addition, the current prototype uses interceptors dedicated to the example application. No much work needs to be done to obtain generic interceptors by using the introspection mechanism of the Java programming language. Likewise, because the switching is done by the ORB through the use of CORBA `ForwardRequest` exceptions, the local copy cannot be a valuetype, since a valuetype is not managed by the ORB and does not have any IOR. The last but not the least issue we have to address in the next section is the design of the local copy launched on the mobile terminal.

3.4 Design of the local copy

In the connected mode, the requests of the GUI are directly sent to the remote object. Hence, the state of the local copy on the mobile terminal does not evolve. The advantage of this mode is that there is no indirection and the state of the local copy can be empty, thus saving memory. When the GUI becomes partially connected, the client-side request interceptor calls an operation `disconnect()` on the local copy which in turn calls an operation `disconnect()` on the remote object. This operation is similar to the operation `disconnect()` in the OBV solution (*cf.* Section 3.1.2):

it transfers the state. The only difference is that, now, the `out` parameter is not a valuetype but an `Any`.

In the partially connected mode, the operations are executed locally and remotely. If the prototype of the operation contains only `in` parameters, the operation is executed locally first and then remotely so that the local copy is "up to date". If the prototype contains only `out` parameters and a return type, the operation is executed remotely first and then locally. The consequence is that the local copy gets "up to date" with the data loaded from the remote object before it responds to the GUI. Regardless of the prototype of the operation, before forwarding a request, the local copy calls the connection state monitor to know if a recent disconnection has occurred, in which case, the request is logged and the operation performed locally. The mixing of `in`, `inout`, and `out` parameters and a return value is let as an open issue in our first study. Another open issue is the support of CORBA exceptions thrown by the servers and sent as responses to the clients.

In the disconnected mode, the operations are executed only locally. If the prototype of the operation contains only `in` parameters, the operation is logged. If the prototype contains only `out` parameters and a return type, whether or not the operation is logged depends on whether the state of the target object changes. The mixing of `in`, `inout`, and `out` parameters and a return value and the use of CORBA exceptions raises the same difficulties as mentioned previously. In addition, recall from Section 2 that every operation has as its first argument an array representing a log of operations that were local to the GUI. This log is also added to the log of the local copy. Of course, this first argument is an `in` parameter but does not take part in the previous discussions. Finally, an important hypothesis of this study is that the remote object cannot be accessed concurrently by other clients while the current client is disconnected. Thus, the reconciliation is eased and kept simple. The transition between the disconnected mode and the partially connected mode corresponds to the replay of the operations logged by the local copy. Clearly, the execution when disconnected is not equivalent to an execution while connected. This is acceptable provided that the state of the connection is visualized by an iconic image in the GUI.

4 Related work

A number of research projects deal with adapting existing applications for wireless access. For a broad survey on client-server computing in mobile environments, the reader can refer to [4].

Outside the CORBA context, *Odyssey*[8] focuses on dynamic adaptation of client-side applications to bandwidth variations, especially for data-intensive applications (image viewer or video player). The system monitors the available bandwidth and notifies the application when a significant change occurs. The application can then switch to a suitable "*fidelity*" (image quality) level. This concept of fidelity is attractive and could be re-used for stream-based Vivian applications and implemented through portable interceptors in the Vivian prototype. However, Odyssey does not support disconnected mode, and requests modifications to the underlying operating system (addition of new signals and system calls). Again outside the CORBA context, *Rover*[6] provides a framework to handle resource variations and disconnections between a mobile terminal and fixed servers. The two key concepts are relocatable dynamic object (RDO) and queued remote procedure call (QRPC). A RDO is a piece of code and data that can be loaded (copied) from a server to the terminal and *vice versa*. The applications control the location of RDOs, thus enabling adaptation to available resources (*e.g.* bandwidth or processing power) and disconnected operations. The QRPC mechanism handles the terminal-servers communications in a transparent way —*i.e.* remote procedure calls or replies are transmitted only when the network is up. This framework is very flexible and generic. However, programmers must design and code in terms of RDOs, either for new applications or for specific proxies in order to support legacy applications. In our approach, the programmers' task is much reduced, since the functional code of legacy CORBA applications remains unchanged with only a few additions for supporting disconnections to be made.

In the CORBA context, the Wireless CORBA specification[11], Π^2[12] and *ALICE*[7] focus on the management of terminal mobility during the execution of a distributed application, and thus address the problem of handling short-time disconnections. The goal of Wireless CORBA is to make the mobility of a wireless terminal transparent to the programmer. To this end, the transport layer provides mechanisms for detecting network disconnections and message retransmission provided a new network connection can be set up within some (short) delay, typically the handover delay in a cellular infrastructure. Otherwise, an exception is raised that can be caught by the application. The Vivian prototype will benefit from the features of Wireless CORBA for handling transient network disconnections due to terminal mobility. However, long-time disconnections (either voluntary or involuntary) are not handled by Wireless CORBA. Π^2 introduces two proxies (one on the terminal and one running in the wired network) for making involuntary disconnections transparent to the user. By contrast, our design does not imply any proxy installation in the wired network, and we provide a support for voluntary disconnected mode. *ALICE* uses a proxy too for handling terminal mobility, and provides a mechanism for supporting both involuntary and voluntary disconnections. Server objects can be duplicated on the terminal by the way of the Object By Value mechanism. As previously explained, we use the Object By Value mechanism too, but with less modifications in the client code, because by using a custom marshalling no state variables appear in the IDL. In addition, the level at which disconnections are handled is different: in *ALICE*, when a disconnection occurs, an exception is sent by the ORB to the client, so that the appropriate code for switching to disconnected mode has to be included in the client ; in our approach, disconnection events are trapped at the ORB level through the interceptor mechanism, so that the appropriate code is included in the interceptors, leaving the legacy application code unchanged —*i.e.* only a few additions have to be made.

In the software components context, *SIRAC*[3] proposes a mechanism for duplicating *beans* from an Enterprise Java Beans server to the terminal. Client stubs are extended so that the application can switch between standard mode (accessing an original *bean* on its server) and local mode (accessing the local copy), according to the value of the current access mode (standard *vs.* local). Voluntary and involuntary disconnections are handled through two methods (`setAccessMode()` and `getAccessMode()`) inherited by any client stub. Bean copies are created through a generic `load()` method which achieves serialization and transmission. This approach is very similar to ours (stub extensions play the same role as interceptors, `load()` plays the same role as the Object by Value mechanism) but in a different context: Enterprise Java Beans *vs.* CORBA.

5 Conclusion and perspectives

We have shown is this paper that standard CORBA mechanisms can be used for enhancing legacy CORBA-based distributed applications in order to support both voluntary and involuntary disconnections in wireless environments. The Objects By Value mechanism helps at answering the requirements of voluntary disconnections, through a very simple use in the client code and limited modifications to the IDL of the legacy application. The Portable Interceptors mechanism removes the limitations encountered with OBV. It allows to handle both voluntary and involuntary disconnections. Moreover the latter mode is supported transparently to the user, with no modification needed in the legacy code on the client side and with minor modifications on the server side. As an illustration of how to use these mechanisms, we have implemented a simple wireless email browser by adapting a CORBA-based web browser application.

Whereas these first results appear as promising, the amount of work needed for adapting a legacy application for supporting disconnected modes is still important. We are currently working on further minimizing the modifications needed to the legacy code. Future work will focus on a more detailed evaluation of the Object By Value and Portable Interceptor mechanisms, with respect to the following criteria: legacy code preservation, transparency towards the application, genericity and performance comparison focusing on the induced overhead. Moreover, in the current implementation, the calculation of the bandwidth is simulated. Hardware and software manufacturers should provide in some time a system call giving the real bandwidth. Finally, the design of the local copy installed on the mobile terminal leaves some open issues ; Section 3.4 of the paper contains a list as a starting point for further research directions.

Acknowledgements

The authors thank Bruno Bretelle and Olivier Villin for their taking part in the implementation of the prototype and the referees for their insightful comments of a previous version of the paper.

References

[1] ORBacus for C++ and Java - Vesion 4.0.5. http://www.ooc.com.

[2] The ITEA Vivian project web site. http://www-nrc.nokia.com/Vivian.

[3] S. Chassande-Barrioz. Adaptation à la mobilité par la duplication de Bean dans un serveur EJB. 2001 samoa workshop, `http://sirac.imag.fr/SAMOA/samoa-workshop01/wkshp2001.html`, March 21-23, 2001.

[4] J. Jing, A. Helal, and A. Elmagarmid. Client-Server Computing in Mobile Environments. *ACM Computing Surveys*, 31(2), June 1999.

[5] D. Johnson and W. Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. *Journal of Algorithms*, 11, September 1990.

[6] A. Joseph, J. Tauber, and M. Kaashoek. Mobile computing with the Rover toolkit. *ACM Transactions on Computers*, 46(3), 1997.

[7] N. Lynch. Supporting Disconnected Operation in Mobile CORBA. M.sc. thesis, Trinity College Dublin, 1999.

[8] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile Application-Aware Adaptation for Mobility. In *Proc. Sixteenth ACM Symposium on Operating System Principles (SOSP97)*, Saint-Malo, France, October 5-8, 1997.

[9] OMG. Portable Interceptors. Interceptors Finalization Task Force published draft, Object Management Group, April 2000.

[10] OMG. The Common Object Request Broker - Architecture and Specifications. Revision 2.4.2. OMG Document formal/01-02-01, Object Management Group, February 2001.

[11] OMG. Wireless Access and Terminal Mobility in CORBA Specification. OMG Document dtc/01-06-02, Object Management Group, June 2001.

[12] R. Ruggaber, J. Seitz, and M. Knapp. π^2 - A Generic Proxy Platform for Wireless Access and Mobility. In *Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC'2000)*, Portland, Oregon, July 2000.