Domint: A Platform for Weak Connectivity and Disconnected CORBA Objects on Hand-Held Devices

Denis Conan, Sophie Chabridon, Olivier Villin, and Guy Bernard

GET /INT

CNRS Samovar 5157

9, rue Charles Fourier

91011 Évry cedex, France

{Denis.Conan,Sophie.Chabridon,Olivier.Villin,Guy.Bernard}@int-evry.fr

May 2003

Abstract

The intrinsic features of wireless communications remain the key factor that still limits the performance of mobile applications. It is therefore crucial to provide enabling middleware technology to mask wireless communications limitations and to ease mobile and recent pervasive, applications development. This article describes a framework, called Domint, which adapts legacy CORBA applications so that they can keep working when weakly connected or even disconnected. A proxy object representing the remote server object called "disconnected object", is deployed automatically within the client execution unit. Connectivity management relies on a reconfigurable hysteresis mechanism to avoid too frequent state transfers and switchings between the disconnected object and the remote server object. Application-transparent switching is provided at the middleware level through the use of portable interceptors. We show performance results of a prototype on both PC (with Windows 2000 and Redhat Linux) and an iPAQ PDA (with Windows CE and Linux Familiar).

1 Introduction

Since the early 90's, the field of mobile computing has witnessed tremendous research and technological advances. With wireless communications and mobile hand-held or wearable devices becoming a reality, new applications where users can have access to information anytime, anywhere are made possible. In the future IT society, mobility will be the rule and no longer the exception. The emergence of the new field of pervasive computing as a successor to both distributed systems and mobile computing enforces

this vision: environments will be "*saturated with computing and communication capability, yet gracefully integrated with human users*" [24].

Today, distributed applications development is facilitated by the use of standard middleware technology providing services such as identification, directory, security or event notification. In 2001, the OMG adopted a specification for Wireless Access and Terminal Mobility in CORBA [18]. This specification provides a support for mobility of users at the middleware level. The key features of the specification include the mobile Interoperable Object Reference (IOR), which hides the mobility of invoked objects hosted on mobile terminals, and the GIOP (General Inter-ORB Protocol) tunnelling protocol, which enables hand-off and message forwarding in a transport-independent fashion. Wireless CORBA allows applications, both client and server, to reside on a mobile terminal that does not have a fixed network access point. This technology is already available. Our work relies on Wireless CORBA for handling transient network disconnections; however additional mechanisms are still required to deal with long-time disconnections and these will be described in this article.We propose to benefit from state-of-the-art middleware technology and validate our approach by demonstrating the feasibility of running an Object Request Broker (ORB) compliant with CORBA 2.4 on a Personal Digital Assistant. In this article, we focus on the client mobile terminal and demonstrate that some current hand-held devices and a fortiori future mobile devices can embed our framework, which uses a full-length CORBA implementation.

Our main contribution is to propose a framework, called Domint, which adapts legacy CORBA applications so that they can keep working even when weakly connected or disconnected. Weak connectivity results from intermittent communication, low-bandwidth, high-latency or expensive networks [13]. We distinguish between two kinds of disconnections: voluntary disconnections when the user decides to work on their own to save battery life or communication costs, or when radio transmissions are prohibited, e.g. aboard a plane; and involuntary disconnections due to physical wireless communication breakdowns such as when the user has moved out of reach of a base station. Obviously, connectivity semantics in a mobile environment is different from what is expected in a connected environment. We consider that this difference is acceptable as long as the user always knows what the connectivity level is, via a graphical user interface for instance. The primary focus of Domint is on the mechanisms allowing work continuity however the connectivity level. We do not address data consistency and conflict resolution in this article.

The key ideas supported by the Domint framework are the following. A proxy object representing the remote server object and called a disconnected object, is deployed automatically within the client execution unit. Being an object means that both data and code are present thus the benefits of the mobile code technology can be valuable [5]. Connectivity management relies on an hysteresis mechanism to avoid too frequent state transfers and switchings between the disconnected object and the remote server object. For this purpose we introduce a first-class partially connected mode, in addition to the connected and disconnected modes. Application-transparent switching is provided at the middleware level (rather than at the operating system level like in Odyssey [14]). This is in line with the end-to-end argument [22] stating that a functionality is often best implemented at a higher

layer at an end system to match application's specific requirements [26]. This comes with all the advantages of language, operating system and network interoperability of CORBA middleware technology.

The remainder of this article presents the Domint architecture, implementation and prototype performance. Section 2 develops the design rationale and overviews the architecture. The connectivity management mechanism is detailed in Section 3. Section 4 describes the way transparent switching between the different modes is performed, while Section 5 is dedicated to the design of disconnected object and the interactions with the logging mechanisms. Section 6 evaluates the performance of a prototype using an email browser as application example. In Section 7 we survey related work, and we conclude in Section 8.

2 Design Rationale and Architecture

In a classical distributed application with strong connectivity, the graphical user interface is loaded on the mobile terminal and the server objects are hosted on machines of the wired network. Keeping working while disconnected implies transferring some elements from the servers to the mobile terminal before losing connectivity, logging operations or state changes during the disconnection, and re-integrating when re-connecting. This section first presents the design rationale and then the architecture of Domint.

2.1 Design Rationale

The weak connectivity of mobile environments in conjunction with the relative resource poverty of hand-held devices leads to two main trade-offs: hardware-dependent non-interoperable system support with limited services *versus* full-service general interoperable system support; and autonomous applications *versus* inter-dependent distributed applications.

Even in the CORBA world that provides interoperability intrinsically, the first trade-off is exemplified by the existence of the minimum CORBA specification [16] as a subset of the full-length CORBA specification including numerous services, facilities and domain specifications. In our work, CORBA is chosen for its ability to be used in multiple domains and for providing extensibility mechanisms such as portable interceptors to build application-transparent services (*cf*. Section 4). The prototype presented in Section 6 demonstrates that some current hand-held devices and a fortiori future mobile devices can embed our framework, that includes a complete ORB. In addition, in order to make the functionality of application server objects available even when disconnected, proxy objects that we call disconnected objects are created on the mobile terminal. A disconnected object is a CORBA object which is similar in design and implementation to the remote object, but specifically built to cope with disconnection and weak connectivity. It is the application designer's responsibility to balance between a straightforward design and a more complex one that adapts better to connectivity variations. Rover [9] makes a distinction between

per-application caches of Relocatable Dynamic Object (RDO) and a global cache. Disconnected objects, being CORBA objects, are accessible from anywhere, so from all the applications of the mobile terminal. Section 5 gives patterns for the development of such disconnected objects, even if this is not the main focus of this article. Another rationale for letting the disconnected object being a CORBA object is that it can use standard CORBA services such as naming, event notifications or transactions independently of the Domint framework.

The trade-off between autonomous applications and interdependent distributed applications is well explained in [23] where the range of adaptation strategies results in three design alternatives: no system support (*laissez-faire* strategy); collaboration between the applications and the system (*application-aware* strategy); and no change to the applications (*application-transparent* strategy). Previous work [9, 12, 14, 19, 28] has demonstrated the possibility that a system can provide good performance even when the network bandwidth varies over several orders of magnitude, but shows also the need for application intervention to improve agility (speed and accuracy) in reaction to changes in resource availability and to specify fidelity in terms of data consistency.

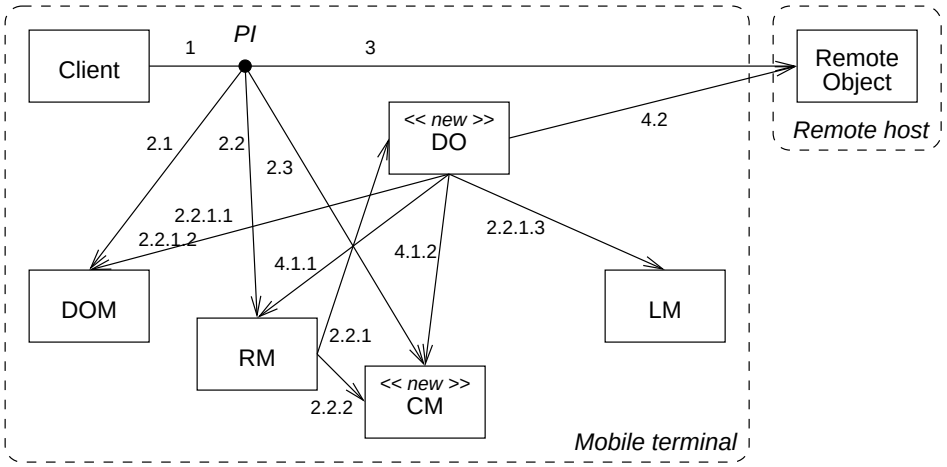
In our work, in order to deal with multiple applications concurrently, some parts of resource management and log management (respectively the management of resources such as network bandwidth and the propagation of logged requests) are centralised and application-transparent. In addition, these services are achieved by CORBA objects and accept requests from the application for better adaptation. The application-aware resource management service abstracts to applications in the CORBA world connectivity information provided by the operating system. More precisely, it accepts requests to modify the per-application perception of which resources and resource levels correspond to bad, weak or strong connectivity, thus improving agility. The application-aware log management service is run by a CORBA object able to act as a representative of the disconnected objects during the re-transmission of the logged requests. It also accepts some code from the latter objects in the form of CORBA Objects By Value (OBV) to interpret the logged requests and to perform for example, log compaction, hence improving fidelity.

2.2 Architecture

The architecture of Domint is depicted in Figure 1. The two sub-figures present UML-like collaboration diagrams of the client sending the first request to a remote object when the connectivity is strong and then sending a request in the case of weak connectivity, respectively. In the rest of the section, the different entities of the architecture are presented and then the collaborations are explained.

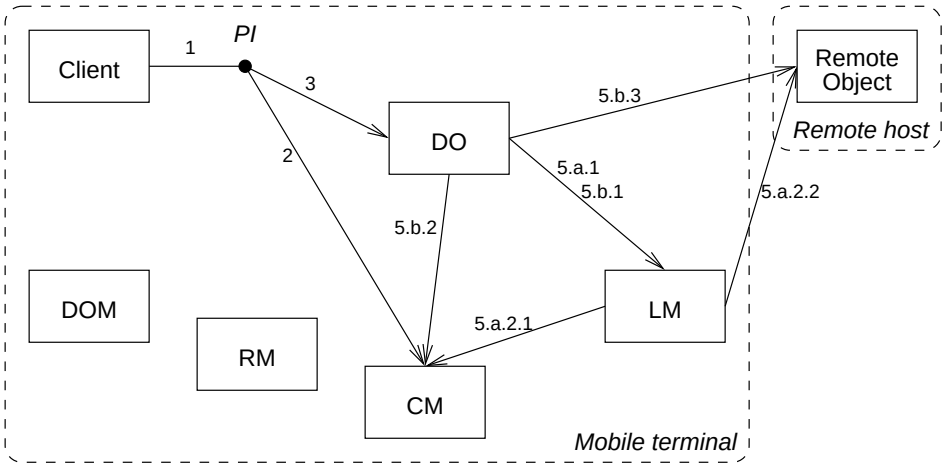
2.2.1 Role of Domint entities

All the rectangles in Figure 1 represent CORBA objects. The *portable interceptor PI* is also a CORBA object but a local one, that is to say, it cannot be called outside of its con-



1: Client's request before interception
2.1: findRM() *2.2: findCM()* *2.2.1: createDO()* *2.2.1.1: findRM()*
2.2.1.2: findLM() *2.2.1.3: addInterpretLogOBV()* *2.2.2: createCM()* 2.3: getCMInfo()
3: Client's request after interception
4.1.1: findCM() *4.1.2: getCMInfo()* 4.2: <<periodic>> incrementalStateTransfer()

(a)



1: Client request before interception
2: getCMInfo()
3: Client request after interception
4: getCMInfo()
5.a.1: addLog() 5.a.2.1: << periodic >> getCMInfo() 5.a.2.2: DO request
5.b.1: treatLog() 5.b.2: getCMInfo() 5.b.3: DO request

(b)

Figure 1: The Domint architecture: *(a)* interactions during the first call to a remote object in the case of strong connectivity, corresponding to the connected mode; the next client's requests sent in the connected mode do not generate the requests italicised; *(b)* interactions during a call to the same remote object in the case of weak connectivity, corresponding to the partially connected mode. The remote object is hosted on a machine of the wired network while all the other entities are executed on the mobile terminal.

taining execution entity[1]. All the requests from and the replies to the *client* are intercepted by the PI. On request sending, the PI acts as a switch between the *disconnected object DO*, and the *remote object RO*. On response reception, the PI detects possible communication failures between the sending of the request and the reception of the response. The client user interface (Client in the diagrams) and the PI belong to the same execution entity whereas the *disconnected objects manager DOM*, the *resource manager RM*, the *connectivity manager CM*, the DO and the *log manager LM* are grouped in another execution entity, also on the mobile terminal. On the other hand, in our use case, the execution entity comprising the remote object is hosted on a machine of the wired network. The execution entity of the managers is launched before that of the client and can handle multiple applications. Except the connectivity manager, the managers are single objects.

Either the client is a legacy application to which disconnected objects are associated via application-transparent portable interceptors or it obtains the reference of the DOM for example from a file stored on the mobile terminal. The DOM is the entry point to find the other managers. The RM is a factory of CMs. A CM accomplishes the abstraction of connectivity information related to one resource. The policy currently implemented associates a CM per logical link between a client and a remote object; it is the finest granularity at the middleware level. We can easily imagine other policies such as one CM per application or one CM per remote host. To simplify the first design, the DO and the corresponding CM are created on PI's demand during the first call to the remote object and stay alive till the end of the client's execution. When to create DOs and for which remote objects is an open issue not treated in this article.

2.2.2 First call handling in connected mode

Figure 1-a shows the interactions during the first call to a remote object in the case of strong connectivity, in the connected mode. In the diagram, each numbered line with a forward arrow is a CORBA request, that is a synchronous call "à la" RPC[2]. The requests 1 and 3 are particular cases: request 1 is intercepted by the PI that does not interpret it but lets the ORB transmit it as request 3 to the remote object. Between these two requests, the following happens.

The PI searches for the reference of the CM in its internal map and concludes that it is the first call to this remote object by this client. The PI asks the DOM (2.1) the reference of the RM, from which it will get the reference of the CM that is going to manage the connectivity (2.2). The RM searches for the reference of the CM in its internal map and concludes that it is the first call to this remote object by this client. The RM creates the DO (2.2.1) and the associated CM (2.2.2). During its construction, the DO gets the references of the RM and the LM by calling the DOM (2.2.1.1 and 2.2.1.2) and provides the LM (2.2.1.3) with data and code in the form of a CORBA OBV to interpret future DO requests drawn in Figure 1-b. Having the references of the CM, the PI decides where the client's request must be issued (*cf*. Section 4 for the decision table). In this scenario

[1] In the article, an execution entity is an address space used by several threads and containing a single ORB instance.

where the connectivity is strong, PI lets the mobile terminal access the remote object for this client's request (3).

In the connected mode, like in Coda [13], we "*don't punish strongly-connected clients*": they experience no more than the delay of the interceptions on the round-trip-time of their calls. As a result, the DO cannot keep up to date with the latest requests. Therefore, the DO periodically calls the remote object for an incremental state transfer (4.2). Of course, the DO tests the connectivity before by calling the CM (4.1.2). If the DO doesn't know the CM reference, it asks for it first (4.1.1). The next client's requests sent in connected mode do not generate the requests italicised in Figure 1-a.

2.2.3 Weak and null connectivity handling

When the connectivity becomes weak or null, forcing the client to enter into the partially connected or disconnected modes respectively, the PI indicates to the ORB to transmit the client's requests to the DO as in Figure 1-b. The client's request is intercepted by the PI (1). The PI obtains the connectivity information from the CM (2) and lets the ORB transmit the client's request to the DO (3). The requests that follow are application-dependent because the DO is built by the application designer. For the sake of clarity, we give two possible ends to this scenario: 5.a and 5.b (*cf*. Section 5 for a more insightful discussion about the role of the DO and the LM).

If the client's request is interpreted as providing information to the remote object, case 5.a, the DO updates its state and prepares a new request, called a DO request, for the remote object. The simplest case is when the DO request is equivalent in parameters' content and operation name to the client's request. Next, the DO encodes the DO request in a data container called a CORBA `Any` and sends the `Any` to the LM (5.a.1). Periodically, the LM decodes the logged request with a method of the OBV provided previously by the DO in Figure 1-a, request 2.3, tests the connectivity (5.a.2.1) and then if possible —*i.e.* partially connected mode—, forwards the DO request to the remote object.

The second case 5.b happens for example when the client's request is interpreted by the DO as a request that is going to return information to the client. If the size of the log given by the LM (5.b.1) is null and the connectivity information obtained from the CM (5.b.2) permits wireless communication with the remote object —*i.e.* partially connected mode—, the DO first tries to load the information requested by the client from the remote object (5.b.3), and next, updates its state before returning the information to the client. Otherwise, the DO returns to the client the information which it got during the last incremental state transfer and which it is keeping up to date with the client's last requests.

3 Connectivity Managers

Connectivity managers are entities dedicated to the estimation of network connectivity. Domint follows the end-to-end argument [22] in taking the "highest" entity —*i.e.* CORBA objects— that can do the estimation without doing redundant work in several entities. A connectivity manager handles a logical connection between a client on the mobile terminal

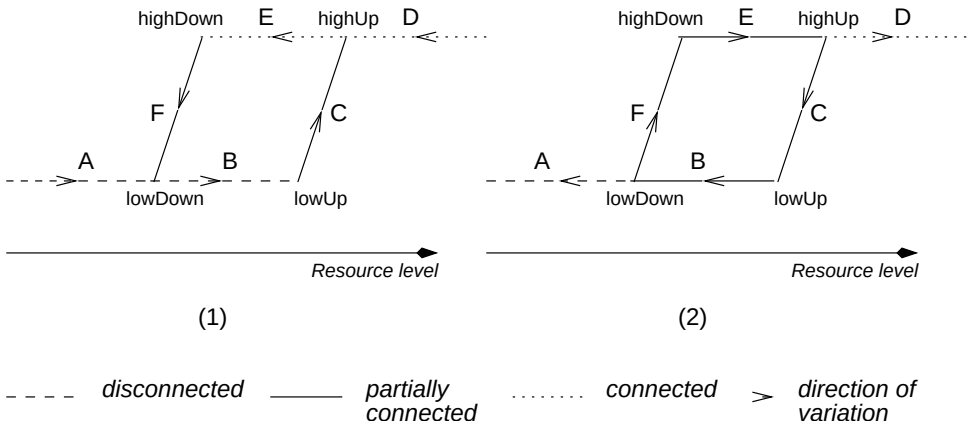
and a remote object on the wired network, however many the number of wireless physical connections that link the two objects at a given time and regardless of whether the logical connection corresponds to different wireless physical connections over time. Connectivity managers rely on network monitoring entities that effectively measure the resource levels: network activity, available bandwidth, transmission cost, round-trip time... The monitoring can be provided by non-CORBA entities, and preferably, by the operating system. Connection monitors can be organised, for instance, in hierarchical structures controlled by a resource manager; the Domint's resource manager could be the representative in the CORBA world of this resource manager.

The design rationale of connectivity managers is many-fold. Firstly, as already explained in Section 2, in order not to "*punish strongly-connected clients*" [13], while strongly connected, client's requests go directly to the remote object. Secondly, in order to "*insulate applications from insignificant variations in resource level*" [14], an hysteresis mechanism is designed; it is detailed in the next paragraph. Thirdly, in order to "*expose network connectivity to applications and permit applications*" and users "*to be involved in connectivity related decisions*" [9], users' interfaces can set the parameters of the hysteresis mechanism and can obtain and display connectivity information. Of course, users can disconnect or re-connect voluntarily by invoking operations `disconnect()` or `reconnect()`; these calls are not addressed to the connectivity managers, but to the remote objects, and are intercepted and treated by the PI (*cf*. Section 4).

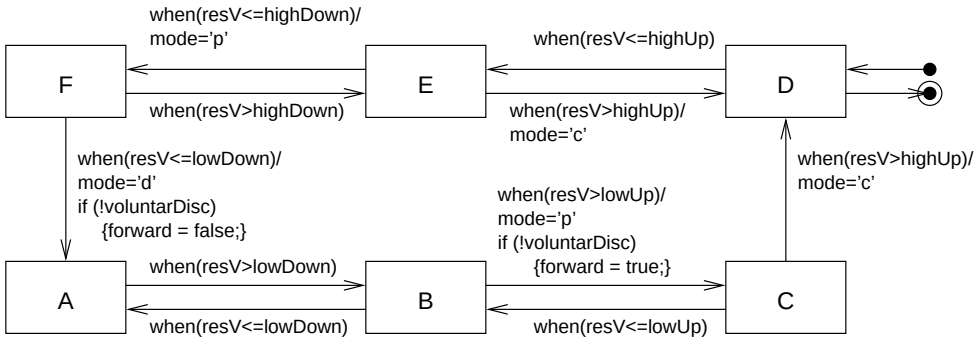
3.1 Hysteresis mechanism

The connectivity managers rely on a hysteresis mechanism for smoothing variations in network resource availability (*cf.* Figure 2-a). The hysteresis defines three modes: *disconnected* when the request is only performed by the disconnected object on the mobile terminal; *connected* when the request is only performed by the remote object on the wired host; *partially connected* when the request is performed by the disconnected object which also transmits the call to the remote object. Since the disconnected object does not perform the operations when the client is (directly) connected to the remote object, it will become out-of-date. So, when going from connected to partially connected, a state transfer is necessary. When the client is either disconnected or partially connected, the local copy processes all the operations and is up-to-date, except for messages that were recently received by the remote object. Then, when going from partially connected to connected, a flushing of the log is necessary.

What we define as the "ping-pong effect" occurs when small variations around a resource level imply back and forth state transfers or log flushings. In [7], Harbus points out a similar effect, the "boomerang effect", in the context of process migration. The common solution to the latter problem is solved using two thresholds. In our case, since the situation occurs in two cases —*i.e.* state transfer and log flushing—, an hysteresis mechanism, which includes four thresholds, is necessary. On the diagram 2-a.1, when the resource level increases and is lower than `lowUp` (*resp.* `highUp`), the mobile terminal is disconnected (*resp.* partially connected). When the resource level decreases and is higher than `highDown` (*resp.* `lowDown`), the mobile terminal is connected (*resp.* partially



(a)



"resV" stands for "resource value"
"c", "p" and "d" stand for "connected", "partially connected", and "disconnected", respectively

(b)

Figure 2: The hysteresis of the connectivity management: *(a)* the hysteresis defines three modes and without the two diagrams there still exists a risk of the "ping-pong effect" around the `highDown` value and around the `lowUp` value; *(b)* the UML state diagram of the hysteresis shows the assignment of the mode whether or not the client asks for a voluntary disconnection and the assignment of the `forward` boolean variable which indicates whether the disconnected object and the log manager can transmit the DO requests to the remote object.

connected). Without the diagram 2-a.2, observe that there would still exist a risk of the "ping-pong effect" around the `highDown` value and another one around the `lowUp` value. Thus, when the connection arrives in state `F` from state `E` (*resp.* in state `B` from state `C`) and the resource level becomes higher than `highDown` (*resp.* lower than `lowUp`), the mode remains "partially connected" up to the `highUp` value (*resp.* down to the `lowDown` value).

The UML state diagram of the hysteresis is drawn in Figure 2-b. The state which immediately follows the initial state and which immediately precedes the final state is state `D`, so that each client's first request to a remote object and the end of the application —*i.e.* corresponding to a `destroy()` call on the ORB instance— occur while the mobile terminal is strongly connected. The hysteresis keeps evolving whether or not the client

asks for voluntary disconnection. On the contrary, whether the disconnected object and the log manager can transmit the DO requests to the remote object depends on voluntary disconnection. This is indicated by the `forward` boolean variable.

3.2 Reconfigurability of the hysteresis

The versatility of the hysteresis mechanism allows to cater for different kinds of applications and for various environment characteristics. The choice of the value of the different thresholds (`lowDown`, `lowUp`, `highUp`, `highDown`) must be done carefully and depends on the type of resource. Let us take as an example the available bandwidth; threshold values then represent a percentage of this bandwidth. Small threshold values (shifting the hysteresis to the left) correspond to a pessimistic case where the connected mode is rarely used, while high values (shift to the right) represent an optimistic assumption where partially connected and disconnected modes are not often visited. We can also configure a large hysteresis (small values for `lowDown` and `lowUp` and large values for `highDown` and `highUp`), priveledging the partially connected mode to deal with an instable environment where weak connectivity is the rule. On the opposite, having a narrow hysteresis with very close threshold values is not recommended. This would mean that the partially connected mode is traversed very rapidly, at the risk of a ping-pong oscillation between the two extreme modes. Moreover, the agility of Domint and its ability to adapt to environment changes such as a sudden unexpected disconnection is limited by the rapidity of the underlying ORB in detecting a communication failure.

Finally, the hysteresis is reconfigurable meaning that thresholds' value can be changed dynamically by the application, or on a decision of the user via a graphical user interface. This enables Domint to adapt easily to various operating scenarios such as transparent network roaming and long disconnection management as presented in [4].

4 Transparent Switching between Modes

Portable interceptors is the CORBA mechanism used by CORBA services to transparently add extra-functional services to applications. Section 4.1 gives a short introduction to CORBA portable interceptors and Section 4.2 develops the use of portable interceptors in the Domint framework, which adopts this approach of application-transparent service support.

4.1 Portable Interceptors Use

"*Portable interceptors are hooks into the ORB through which ORB services can intercept the normal flow of execution of the ORB*" [15]. Portable interceptors are instantiated and registered to an ORB during the creation of the ORB instance by invoking the method `ORB.init()`.

Client-side interceptors introduce five interception points in a request and reply sequence on the client side. Two of them take place before a request is sent by the ORB.

The last three allow the reply to be parsed: normal (successful) or exceptional, or other than normal and exceptional, 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, client-side interceptors can parse the IOR profile of the target object, find a specific component, read the data included in the component, and apply special treatments based on this information, hence for instance distinguishing between requests to the remote object (*e.g.* request 1 of Figure 1-b) and requests to the managers or the disconnected objects (*e.g.* request 2 of Figure 1-b). In addition, client-side interceptors can know the identity of the object adapter which manages the sender, thus for instance distinguishing the client's request (*e.g.* request 1 of Figure 1-b) from requests sent, for example, from the interception points (*e.g.* request 3 of Figure 1-b).

4.2 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 ORB. Therefore, some or all of the server's objects can have the "disconnected mode" policy; this is up to the application. When the user's graphical user interface (GUI) starts, an IOR interceptor and a client-side interceptor are registered at the creation of the ORB. The IOR interceptor of the client is the same as the server's. In order to treat every request to disconnected objects, all the disconnected objects created on the mobile terminal possess the "disconnected mode" policy. Depending on the state changes of the connection, the client-side request interceptor can build a CORBA `ForwardRequest` exception indicating the change of request target IOR and raise 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 disconnected object and *vice versa*.

Table 1 gives the decision table contained in the PI for the transparent switching between modes. Events activating the decision are client's requests. The first two inputs are boolean variables: `voluntary` and `direct`, indicating whether the user asks for a voluntary disconnection and whether the ORB sent the previous client's request to the remote object, respectively. The next two inputs are the mode and the name of the operation (`opName` in the table). The actions to perform are on the right side of the table: reversing the two boolean variables `voluntary` and `direct`, and raising of `ForwardRequest` exceptions (`Forward` in the table). Raising a `ForwardRequest` exception means re-transmitting the same client's request, with the variables `voluntary` and `direct` that may have changed, and also intercepting the latter client's request. When actions include a throw action, it is the last one. In order to disconnect (*resp.* re-connect) voluntarily, the application calls the `disconnect()` (*resp.* `reconnect()`) operation on the remote object. These calls are both issued on the disconnected object: the switch is done before the execution of the `disconnect()` operation and the `reconnect()` operation is executed before the switch —*i.e.* the switch is performed during the next call. Like the other RPC-like calls, these operations return to the client, meaning that the disconnection or re-connection is effective.

voluntary	direct	mode	opName	reverse voluntary	reverse direct	throw Forward (DO)	throw Forward (RO)
true	false	d	disconnect				
			reconnect	+			
			other				
		p	disconnect				
			reconnect	+			
			other				
		c	disconnect				
			reconnect (*1)	+			
			other				
false	true	d	disconnect	+	+	+	
			reconnect				
			other		+	+	
		p	disconnect	+	+	+	
			reconnect				
			other (*2)		+	+	
		c	disconnect	+	+	+	
			reconnect				
			other				
	false	d	disconnect	+			
			reconnect				
			other				
		p	disconnect	+			
			reconnect				
			other				
		c	disconnect	+			
			reconnect				
			other (*1)		+		+

Table 1: The decision table of the transparent switching in the PI: 'd', 'p' and 'c' stands for 'disconnected mode', 'partially connected mode' and 'connected mode', respectively; (*1): the `reconnect()` operation is called on the DO so that it tries to flush the log pessimistically and the switch will occur during next call if the mode permits it; (*2): the PI asks the DO either to perform an incremental state transfer or to switch silently without contacting the remote object (RO in table).

In addition, the decision table allows the client to send any sequence of `disconnect()` and `reconnect()` operations.

5 Disconnected Objects and the Log Manager

The two previous sections, Section 3 and Section 4, developed basic application-independent mechanisms. On the contrary, as already stated in Section 2, the design

of disconnected objects is highly application-dependent. This section sketches succinctly patterns for, and open issues (left as future work) on, the design of disconnected objects and remote server objects in the three different modes, and summarises the role of the log manager.

5.1 Connected mode

In the connected mode,client's requests are directly sent to the remote object. Hence, the state of the disconnected object on the mobile terminal does not evolve. The advantage of this mode is that there is no indirection and the state of the disconnected object can be empty, thus saving memory. When the mobile terminal becomes partially connected, the client-side request interceptor calls `disconnect()` on the disconnected object which in turn calls `disconnect()` on the remote object to transfer the state if the mode permits it. The remote object encodes its state in a CORBA `Any` data container. In order to support rapid transitions from the connected mode to the disconnected mode, disconnected objects periodically call `disconnect()` on the remote object. We could imagine other hoarding policies such as application-triggered or system-triggered. These state transfers can of course be incremental. In addition, note that disconnected objects always try to inform remote objects before disconnecting, hence allowing them to adapt concurrency of accesses while some clients are disconnected and to prepare future reconciliations.

5.2 Partially connected mode

In the partially connected mode, the operations are executed both locally and remotely, in an order depending on the parameters semantics. If the prototype of the operation contains only `in` parameters, the operation is executed locally first and then remotely so that the disconnected object keeps up to date. If the prototype contains only `out` parameters and/or a return type, the operation is executed remotely first and then locally. The consequence is that the disconnected object keeps up to date with the data loaded from the remote object before it responds to the client. Regardless the prototype of the operation, before forwarding a request, the disconnected object calls the connectivity manager to know if a recent disconnection has occurred, in which case, the request is logged and the operation performed locally only. The mixing of `in`, `inout`, and `out` parameters and of a return value is left as an open issue in our first study. Another open issue is the support of exceptions raised by the servers and sent as responses to the clients. While partially disconnected, in order to be up-to-date, disconnected objects may keep periodically asking remote objects for new state changes.

5.3 Disconnected mode

In the disconnected mode, the operations are executed only locally and possibly logged. If the prototype of the operation contains only `in` parameters, the operation is logged. If

the prototype contains only `out` parameters and/or a return type, whether or not the operation is logged depends on it changing the state of the target object changes. The mixing of parameters (`in`, `inout` and `out`) and of a return value, and the raising of exceptions raises the same difficulties as mentioned in the partially connected mode. In addition, the transition between the disconnected mode and the partially connected mode leads to the replay of the operations logged by the log manager. Clearly, the execution when disconnected may not be equivalent to an execution while connected. This is acceptable provided that the connectivity information is visualised by an iconic image in the user's GUI. In case of voluntary disconnection, the `reconnect()` operation is called on the disconnected object so that it tries to flush the log pessimistically, that is in an atomic action; trickle re-integration "à la" Odyssey [23] is still an open issue in object-oriented systems.

5.4 Log Management

In the partially and disconnected modes, we log and propagate operations instead of state contents like in [19]. This facilitates the reconciliation of the disconnected and remote objects at reconnection time. Moreover, we consider that the remote object cannot be accessed concurrently by other clients while the current client is disconnected. Work is in progress to remove this assumption. In this study, the reconciliation algorithm is kept simple since the transition between the disconnected mode and the partially connected mode corresponds to the replay of the operations logged by the local copy. We are currently working on designing reconciliation algorithms well-suited to mobile environments for ensuring data consistency. The Domint framework will allow us to test and evaluate various approaches such as optimistic replication [11] or operation transforms [29]. Checkpointing and recovery techniques are well suited to deal with voluntary disconnections. But it is still an open issue for unexpected disconnections [3]. The intermittent errors that may frequently occur in wireless networks require failure detection mechanisms different from that in traditional networks. Likewise, the checkpointing mechanism in itself should deal with both disconnection types.

The interpretation of the logged requests (DO requests in Section 2) is application-dependent. The code that can parse and forward the logged requests is provided at the beginning of the execution by disconnected objects via object by values (OBV) that we name DO request interpreters. Disconnected objects and DO request interpreters can log and propagate either operations or state changes. The concept of OBV was introduced in the CORBA 2.3 standard. It enables the passing of an object by value rather than by reference. The log manager receives as an `in` parameter a DO request interpreter, that is a description of the state and the code of the object responsible for the interpretation of future logged requests, and a new instance is automatically created in the execution entity of the log manager. Provided that all the DO request interpreters inherit the same abstract interface, the log manager is application-independent. In addition to operations for the parsing and the forwarding of logged requests, the abstract interface could include operations like log compaction "à la" Coda [13]: Log compaction in object-oriented systems is an open issue. Finally, an important limitation of the current design is the lack of support of return values for logged operations. A solution, similar to the QPRC

call-backs in Rover [10], may be based on CORBA call-backs. We have not focused on call-backs for now because we considered the client-side applications as legacy parts.

6 Performance Results

We have conducted performance measurements on different software and hardware combinations: laptop PC running GNU/Linux RedHat 7.2 or Microsoft Windows2000, and iPAQ PDA (206 Mhz, 16 MB of ROM and 32 MB of RAM) running Microsoft WindowsCE or GNU/Linux Familiar 0.6. For wireless communications, a Compaq IEEE 802.11b WL110 card at 11Mbps was plugged in all devices and we used a Compaq IEEE 802.11b WL100 + WL300 software base station. Each test was run 1000 times in order to obtain meaningful averages with confidence intervals computed at 0.99. A garbage collection occurred before each run on the client and server sides in order to have no interference with previous operations. All the programs were written in Java and we used IBM J9 v1.2.2[2]. Measurements were also performed with Classic VM Blackdown-1.3.1-RC1 on Familiar, and the latter virtual machine proved to be less efficient for small-size requests (up to 70%) but much more efficient for medium-size and large-size requests (up to 270%).

Firstly, we present an evaluation of basic processing times independently of Domint. We then analyze the performance results of Domint and extract the costs of its different constituents using linear regression.

6.1 Basic measurements

We have measured basic times to process simple requests[3] using TCP sockets, and ORBacus 4.1.0 without interceptors and with interceptors doing nothing (*cf.* Table 2). For these experiments, the Domint software is not involved.

As expected, it appears that local processing is in general better than remote processing, except on WindowsCE for small-size requests of 1B and 128B. When comparing results for WindowsCE and Familiar, there is a difference according to the request size and to the use of an ORB or not. For local TCP requests, Familiar is significantly better (from 80% to 90%). Using an ORB, WindowsCE gives smaller response times for messages of 1B and 128B while Familiar is better for large-size requests, the difference ranging from 40% to 75%. The cost of using an ORB can be evaluated by comparing TCP and ORB (without interceptors) figures. The impact of the ORB is smaller under Familiar.

Another interesting measurement concerns the overhead of portable interceptors. Comparing the results without interceptors (lines 3–4 and 9–10) and with interceptors (lines 5–6 and 11–12, respectively), we observe an overhead ranging from 7% to 13% on Familiar and ranging from 1% to 33% on Windows, the maximum overhead being obtained for small messages. Although not negligible, the cost of interceptors appears reasonable with respect to the message size. The overhead we obtain is smaller than in [30]; in this

[2]It works with no difficulties on all these software and hardware combinations.

[3]For more clarity, only times with '`out`' parameters for iPAQ are shown.

iPAQ - MS WindowsCE

Experiment	1 B	128 B	16 KB	128 KB	512 KB	1 MB
1. TCP–L	7.0 ±0.0	7.1 ±0.0	35.6 ±0.1	243.6 ±0.3	951.8 ±0.6	1904.0 ±0.5
2. TCP–R	3.7 ±0.1	4.2 ±0.1	54.1 ±3.1	446.5 ±10.0	1751.1 ±17.9	3487.5 ±23.2
3. WOPI–L	17.9 ±0.2	17.8 ±0.2	48.6 ±0.2	296.8 ±0.3	1241.3 ±0.4	2306.3 ±0.5
4. WOPI–R	12.1 ±0.1	13.8 ±2.0	65.4 ±3.4	456.2 ±6.6	1870.0 ±17.1	3707.3 ±21.6
5. PI–L	21.6 ±0.2	21.9 ±0.2	53.3 ±0.2	300.9 ±0.3	1270.0 ±0.4	2341.6 ±0.5
6. PI–R	16.3 ±1.9	17.4 ± 1.9	68.1 ±2.9	462.8 ±7.3	1898.7 ±11.6	3707.8 ±14.9

iPAQ - GNU/Linux Familiar

Experiment	1 B	128 B	16 KB	128 KB	512 KB	1 MB
7. TCP–L	0.9 ±0.0	1.0 ±0.1	3.7 ±0.1	28.2 ±0.7	105.7 ±1.7	223.2 ±1.7
8. TCP–R	4.6 ±0.0	5.1 ±0.0	62.3 ±3.3	436.7 ±5.7	1737.5 ±5.7	3494.6 ±23.2
9. WOPI–L	21.5 ±0.6	20.8 ±0.5	29.6 ±0.5	83.9 ±0.2	269.4 ±0.2	531.4 ±0.2
10. WOPI–R	49.5 ±0.7	49.5 ±0.6	68.9 ±6.4	461.7 ±3.7	1810.1 ±5.2	3702.5 ±7.5
11. PI–L	24.5 ±0.5	23.1 ±0.4	32.5 ±0.4	87.8 ±0.2	282.1 ±0.2	554.8 ±0.5
12. PI–R	53.1 ±0.9	56.1 ±0.6	69.4 ±3.2	464.2 ±3.7	1818.7 ±5.4	3712.3 ±7.5

Table 2: Basic times (ms) for processing local (client and server collocated on the iPAQ) and remote (server on a remote wired laptop) requests of different sizes (bytes) on iPAQ running Microsoft WindowsCE *(lines 1–6)* or GNU/Linux Familiar *(lines 7–12)*; '*TCP*', '*WOPI*', and '*PI*' stands for 'TCP sockets', 'ORB without client-side interception', 'ORB with client-side interception doing nothing', respectively; '*L*' and '*R*' stands for 'client and server collocated on the iPAQ', 'server on a remote wired host', respectively.

paper, the authors conducted experiments using two dual-processor UltraSPARC-2 workstations running SunOS 5.7 and showed a performance penalty of 26% with interceptors doing nothing.

6.2 Domint measurements

We now analyze the performance results of Domint. We do not detail all the measures obtained as our objective is to identify the overhead of each Domint constituent. We performed a linear regression analysis for this purpose.

According to the collaboration diagrams depicted in Figure 1, the performance figures presented in Table 3 and depicted in Figure 3 can be partly deduced from the ones given in Table 2 using the following equations:

$$t_{conn-O} = t_{conn-I} = t_{PI-L-short} + t_{PI-R} + t_{conn\_mngt}$$
$$t_{part-O} = t_{conn-O} + t_{PI-L} + 2 * t_{request\_intra\_VM}$$
$$t_{part-I} = t_{part-O} + t_{logging}$$
$$t_{disc-O} = t_{part-O} - t_{PI}$$

$t_{disc-I} = t_{part-I} - t_{PI}$

where:

- $t_{PI-L-short}$ is the time for issuing small-size local request using the ORB
- t_{conn_mngt} is the computation time for the management of the hysteresis of the connectivity management
- $t_{request\_intra\_VM}$ is the time for performing a small-size request between CORBA objects collocated in the same execution unit (JVM)
- $t_{logging}$ is the computation time for packaging a request in a CORBA `Any` and logging it.

For both WindowsCE and Familiar, the linear regression computed shows that both the coefficients of correlation and of determination are very good (more than 0.99) for TCP figures. The coefficients are still very good for ORB figures for WindowsCE, but less for Familiar (around 0.6 and 0.4 respectively). However, when using the formulae to compare the values deduced by these formulae and the ones obtained by experimentation, the differences are much more important for WindowsCE than for Familiar.

We now analyze the overhead due to connectivity management and logging. For WindowsCE, the overheads for t_{conn-O},t_{part-O}, t_{disc-O} range from 70% to 2%, from 52% to 21%, and from 77% to 5%, respectively, the worst values being for small-size requests. For Familiar, the overheads range from 14% to 1%, from 50% to 6%, and from 20% to 6%, the worst values being also for small-size requests. This suggests that WindowsCE is more sensitive to the addition of the threaded daemon which comprises the managers : DOM, RM, CM, and LM of Figure 1. It also implies that the cost of logging is not negligible. Finally, the overheads induced by Domint *w.r.t.* `PI` can be computed with raw (real) data. For WindowsCE, the overheads for conn–`O`, part–`O`, disc–`O` range from 71% to $\approx$0%, from 80% to 40%, and from 77% to 5%, respectively, the worst values being for small-size requests. For Familiar, the overheads range from 14% to 1%, from 59% to 6%, and between 40% and 13%, the worst values being also for small-size requests.

7 Related Work

A number of research projects deal with adapting existing applications for wireless access. In this section we distinguish pioneering projects that are not based on standard middleware and more recent projects relying on advanced services provided by CORBA middleware. For a broad survey on client-server computing in mobile environments, the reader can refer to [8].

Coda is the first research effort to deal with disconnected operations [12, 13, 25]. It implements application-transparent adaptation in the context of a distributed file system. Several Coda key concepts were of interest in our design such as weak connectivity and the hoarding mechanism. As a successor to Coda, Odyssey supports application-aware

iPAQ - MS WindowsCE

Experiment	1 B	128 B	16 KB	128 KB	512 KB	1 MB
1. conn–O	56.5 ±2.6	56.2 ±2.6	106.6 ±4.6	497.8 ±12.9	1891.7 ±14.7	3648.9 ±26.8
2. part–O	79.6 ±2.3	81.5 ±3.1	180.6 ±5.5	782.3 ±8.8	3241.3 ±19.5	6183.0 ±98.3
3. part–I	140.5 ±27.7	130.2 ±4.1	241.9 ±28.3	935.7 ±38.7	3538.7 ±77.3	6520.4 ±101.0
4. disc–O	92.8 ±0.5	93.1 ±0.5	123.4 ±0.5	403.1 ±7.6	1404.2 ±7.7	2459.4 ±6.8

iPAQ - GNU/Linux Familiar

Experiment	1 B	128 B	16 KB	128 KB	512 KB	1 MB
5. conn–O	61.8 ±4.1	64.3 ±6.0	80.12 ±20.3	521.4 ±11.0	1878.1 ±19.8	3733.1 ±33.9
6. part–O	61.6 ±5.0	62.5 ±5.7	123.9 ±12.4	619.6 ±22.1	2332.5 ±34.3	4810.6 ±28.4
7. part–I	87.1 ±7.1	90.6 ±5.9	132.1 ±12.5	692.7 ±15.1	2176.8 ±24.2	4246.6 ±371.1
8. disc–O	52.6 ±3.1	56.0 ±4.0	66.0 ±8.6	186.4 ±9.8	326.8 ±5.1	591.1 ±5.9

Table 3: Times (ms) for processing of remote requests of different sizes (bytes) on iPAQ running Microsoft WindowsCE *(lines 1–4)* and GNU/Linux Familiar *(lines 5–8)* in the three modes of connectivity: '*conn*', '*part*', and '*disc*' stand for 'connected', 'partially connected', and 'disconnected', respectively; '*O*' and '*I*' stand for '`out` arguments' and '`in` arguments', respectively.

adaptation that is better suited to multimedia data such as speech and video [14]. 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 CORBA applications and implemented through portable interceptors in the Domint prototype. Coda and Odyssey are based on a file model; this rather low-level semantic only allows to deal with coarse-grain entities. We believe that the object concept provides more flexibility in the manipulation of applications. Coda and Odyssey are now part of Aura. This new project started in 2000, focuses on distraction-free ubiquitous computing, stating that the most precious resource in computing is human attention [24]. Aura manipulates higher level abstractions such as tasks which can capture the user's intent; this concept of task-driven computing seems very promising in modelling adaptation policies [27]. In contrast to the scenarios presented in [27] where the infrastructure will "*allow users to move their computational tesks easily from one environment to another*", in our scenarios, we focus on keeping working while disconnected.

The *Rover* [9, 10] project provides a framework to handle resource variations and disconnections between a mobile terminal and fixed servers. The two key concepts are 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. Rover manages two kinds of caches: a global cache and per-application caches of RDOs. In our framework, disconnected ob-

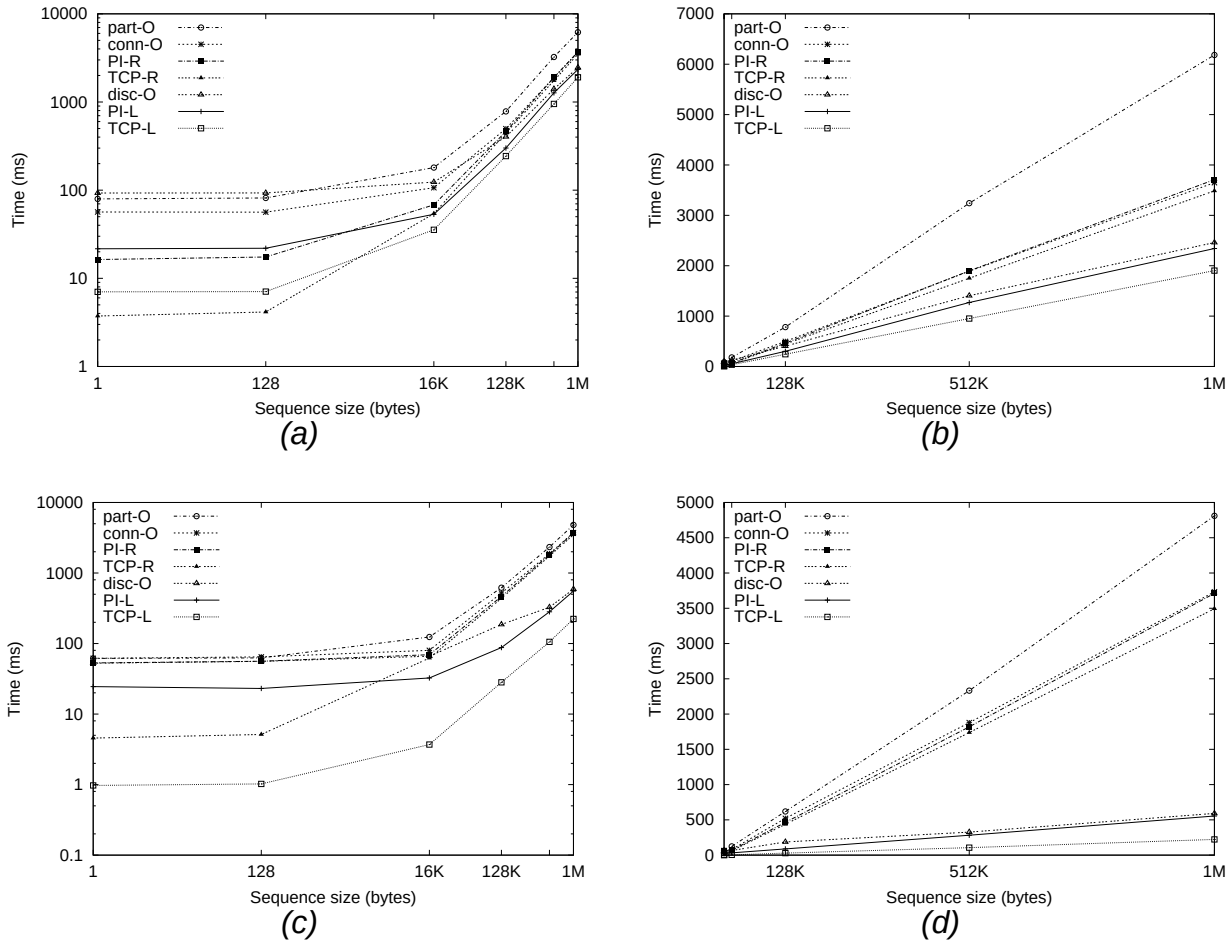


Figure 3: Display of some values presented in Table 2 and 3: *(a)* and *(b)* iPAQ running Micorsoft WindowsCE, *(c)* and *(d)* running GNU/Linux Familiar: the same abbreviations as in Tables 2 and 3 are used.

jects, being CORBA objects, are accessible from everywhere, that is to say from all the applications of the mobile terminal. The Rover 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 voluntary disconnections to be made.

In the CORBA context, the CORBA Messaging Service [17] provides programming language stubs that support "asynchronous" or "time-independent" invocations using extensions to the GIOP protocol that can handle the storing and the forwarding of requests and responses. The former mechanism allows a client to issue a request without block-

ing for the response. Later, the client receives the reponse either by a callback from the ORB or by polling. The later mechanism permits the client to make a request, disconnect from the network, and then reconnect later and get the response. The specification presents a prototypical architecture with routers: client's and server's routers and more routers between them. This architecture is in a sense similar to Rover QRPC. In contrary, we advocate that as long as users visualise the connectivity information, if they continue working while disconnected, they implicitely want "immediate" responses —*i.e.*, continuity of service—, even if they depart from "normal" responses while connected. Therefore, what is logged on the mobile terminal is what will be necessary when reconciling: not all requests need logging and the logs may be treated (compaction...).

Also in the CORBA context, Π^2 [20, 21] and *ALICE* [6, 1] focus on the management of the hand-off due to terminal mobility during the execution of a distributed application, and thus address the problem of dealing with short-time disconnections. Π^2 introduces two proxies (one on the terminal and one running in the wired network) to make involuntary disconnections transparent to the user. *ALICE* uses a proxy too, for handling terminal mobility and provides a mechanism for supporting both involuntary and voluntary disconnections. Our design does not imply any proxy installation in the wired network. In addition, the level at which disconnections are handled in Domint 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 clients; in our approach on the other hand, disconnection events are trapped at the ORB level through the portable interceptor mechanism, so that the appropriate code is included in the portable interceptors, leaving the legacy application code unchanged.

8 Conclusion

In this article, we have shown how current standard middleware technology can enable legacy applications to work unchanged on the client side (mobile devices) and with minor modifications on the server side. Even though the implementation is CORBA specific, the presented concepts can be applied to other middleware as well, provided that they possess some interception and object by value mechanisms.

An appropriate service configuration can take care of showing the state of the connection on a special user's GUI and offering the application the means to disconnect voluntarily. Involuntary disconnection being handled entirely at the middleware level, no modification at all is made in the application code. The state transfer and the caching and logging mechanisms can be derived from the application code without having to modify the server or the client legacy code in any way. As it is now possible to embed a full-service off-the-shelf ORB product compliant with the CORBA 2.4 specification on PDAs like iPAQ, we can take advantage of the CORBA portable interceptor mechanism to implement application-transparent adaptation policies.

Our framework is open and enables application-specific adaptation; for instance, the log manager can make use of objects by value to dynamically determine which information should be logged. In addition, we have proposed an hysteresis mechanism to deal with

back and forth switching which may occur when the connectivity level oscillates around the median value. We have shown the first performance results of a CORBA prototype. Tests were run on both a laptop PC and an iPAQ PDA. The performance results show that the overhead introduced by the Domint framework is acceptable by the end user. The Domint framework is available as an Open Source project under a GNU GPL license at https://picolibre.int-evry.fr/.

References

[1] G. Biegel, V. Cahill, and M. Haahr. A Dynamic Proxy Based Architecture to Support Distributed Java Objects in a Mobile Environment . In *Proc. Int. Symposium on Distributed Objects and Applications (DOA'02)*, pages 809–826, Oct. 2002.

[2] A. Birrell and B. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1), February 1984.

[3] G. Cao and M. Singhal. Mutable Checkpoints: A New Checkpointing Approach for Mobile Computing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 12(2):157–172, Feb. 2001.

[4] D. Conan, S. Chabridon, O. Villin, G. Bernard, A. Kotchanov, and T. Saridakis. Handling Network Roaming and Long Disconnections at Middleware Level. In *Workshop on Software Infrastructures for Component-Based Applications on Consumer Devices (in conjonction with EDOC 2002)*, Lausanne, Switzerland, Sept. 2002.

[5] A. Fuggetta, G. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5), May 1998.

[6] M. Haahr, R. Cunningham, and V. Cahill. Supporting CORBA Applications in a Mobile Environment. In *Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking (MobiCom'99)*, Seattle, Washington, USA, 1999.

[7] R. Harbus. Dynamic process migration: To migrate or not to migrate. Technical report csri-42, University of Toronto, Toronto, Canada, July 1986.

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

[9] A. Joseph, J. Tauber, and F. Kaashoek. Mobile Computing with the Rover Toolkit. *IEEE Transactions on Computers*, 46(3), 1997.

[10] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proc. 15th ACM Symposium on Operating Systems Principles*, pages 156–171, Dec. 1995.

[11] A. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The icecube approach to the reconciliation of divergent replicas. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC 2001)*, Newport, Rhode Island (USA), 26-29 August 2001.

[12] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[13] L. Mummert, M. Ebling, and M. Satyanarayanan. Exploiting Weak Connectivity for Mobile File Access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, Copper Mountain resort,CO, December 1995.

[14] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP'97)*, 1997.

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

[16] OMG. *Minimum CORBA*, OMG Document formal/01-02-01 Chapter 23. In [17], February 2001.

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

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

[19] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proc. 16th ACM Symposium on Operating Systems Principles*, pages 288–301, Saint Malo, France, Oct. 1997.

[20] R. Ruggaber and J. Seitz. Using CORBA Applications in Nomadic Environments. In *Proc. 3d IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'2000)*, Monterey, CA (USA), Dec. 2000.

[21] 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.

[22] J. Saltzer, D. Reed, and D. Clark. End-to-end Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4), November 1984.

[23] M. Satyanarayanan. Mobile Information Access. *IEEE Personal Communications*, 3(1):26–33, Feb. 1996.

[24] M. Satyanarayanan. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications*, August 2001.

[25] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computer*, 39(4), April 1990.

[26] A. Snoeren and H. Balakrishnan. An End-to-End Approach to Host Mobility. In *Proceedings of the 6th ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'00)*, Boston, MA, 2000.

[27] J. Sousa and D. Garlan. Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environment. In *Proc. 3rd Working IEEE/IFIP Conference on Software Architecture*, Montreal (Canada), Aug. 2002.

[28] D. B. Terry, M. M. Theimer, K. Petersen, and A. J. Demers. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proc. 15th ACM Symposium on Operating Systems Principles*, pages 172–183, Dec. 1995.

[29] N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Proc. ACM CSCW*, Dec. 2000.

[30] N. Wang, K. Parameswaran, and D. Schmidt. The Design and Performance of Meta-Programming Mechanisms for Object Request Broker Middleware. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, San Antonio, TX (USA), Jan. 2001.