WEB MINDS

Wide-scalE, Broadband, MIddleware for Network Distributed Services

FIRB - Fondo per gli Investimenti della Ricerca di Base

The Esperanto Broker: a technical report

M. Cinque[1], D. Cotroneo[1], A. Migliaccio[1], and S. Russo[1,2]

1: Dipartimento di Informatica e Sistemistica
Università di Napoli Federico II
Via Claudio 21, 80125 - Napoli, Italy
2: Consorzio Inter-universitario Nazionale per l'Informatica (CINI)
Via Diocleziano 328, 80125 - Napoli, Italy
(macinque,cotroneo,armiglia,sterusso)@unina.it

Abstract

In the recent past, a plethora of mobile computing middleware are developed. Actually, most of them do not cope with terminals mobility at each level of the network protocols stack. We presents a new communication platform which implements a Broker for nomadic computing environments, named the Esperanto Broker. By using the proposed platform, developers model application components as a set of objects that are distributed over wireless domains and interact via remote method invocations. The Esperanto Broker is able to guarantee remote object interactions despite terminals movements and/or disconnections. We believe that the proposed platform acts as a building block for the realization of mobile-enabled middleware services. We describe the conceptual model behind the architecture, discuss implementation issues, and present preliminary experimental results.

WP	2
Numero	TR-WEBMINDS-37
Data	10/01/2005
Tipo di prodotto	Technical Report
Numero di pagine	16
Unità operativa	CINI - Napoli
Persona da contattare	Domenico Cotroneo Dipartimento di Informatica e Sistemistica Università degli Studi di Napoli Federico II Via Claudio 21, 80125 - Napoli, Italy cotroneo@unina.it

1 Introduction

Recent advantages achieved in wireless and in mobile terminals technologies are leading to new computing paradigms, which are generally described as mobile computing. Mobile Computing encompasses a variety of models which have been proposed in literature[1], depending on the specific class of mobile terminals (such as PDAs, laptops or sensors), and on the heterogeneity level of the network infrastructure. Nomadic computing, ad-hoc computing, and ubiquitous computing are different views of such models, depending on the characteristics they emphasize. In this work, focus is on Nomadic Computing (NC), which is a form of mobile computing where communication takes place over strongly heterogeneous network infrastructure, composed of several wireless domains, which are glued together by a fixed network infrastructure (the core network) [2]. A wireless domain may represent a building floor, a building room, or a certain zone of university campus. Several kinds of devices, with different characteristics in terms of resources, capabilities, and dimensions, may be dynamically connected to domains, and use provided services.
It is widely recognized that traditional middleware platforms (such as CORBA [3], DCOM [4], and JAVA-based technologies, such as JavaRMI [5]) appear inadequate to be used for the development of emerging applications for nomadic computing [6, 1, 7]. Indeed, since mobile computing applications operate under a broad range of networking conditions (including rapid QoS fluctuations) and scenarios (including terminals and users movements), mobility does not make the traditional middleware solutions suitable for supporting emerging nomadic applications. During past years, a great deal of research has been conducted on middleware for mobile settings, proposing enhancing versions or new models. Research efforts have been mainly progressed along the two following directions:

- Providing innovative mechanisms, such as context awareness, reconfiguration, dynamic and spontaneous discovery, and adaptation. Examples are solutions proposed in [8, 9, 10]. Novel strategies emerged from the mentioned works, which are based on new programming models, such as reflection, and aspect-orientation.

- Dealing with Quality of Service aspects of mobile computing infrastructures, such as security, network resources allocation, and fault-tolerant strategies. Examples are solutions proposed in [11, 12, 13]. In mobile computing environment, these issues become more challenging due to new characteristics of wireless network infrastructures (in terms of bandwidth, latency, topology, etc.) and mobile terminals (in terms of hardware and software capabilities).

While we recognize that these studies represent fundamental milestones for the pursuit of new middleware solutions for mobile/nomadic computing, most of them do not focus on mobile-enabled interaction mechanisms. Mobile computing environments are characterized by highly dynamic behavior: wireless networks have an unpredictable behavior, and users are able to move among different service domains. Therefore, the definition of a mobile-enabled interaction platform is a crucial step of the entire mobile computing middleware design process. Most of cited works delegate the mobility support to the underlying (network and transport) layers, by using protocols such as Mobile IP, Session Initiation Protocol (SIP), or a TCP tailor-made for mobile computing environments [14, 15, 16]. However, as stated in [17], mobility support is needed on each layer of the ISO/OSI stack, especially on the middleware layer. As for the interaction paradigms,

it is recognized that traditional remote procedure call does not accommodate itself to mobile/nomadic computing environments. However, although many different solutions have been proposed for adapting interaction paradigms to mobile settings (such as tuple space, and publish/subscribe [6]), they often result in a programming model which is often untyped, unstructured, and thus complex to program. Furthermore, the W3C has recently carried out a set of interaction paradigms, which have been assuming as standard *de facto* in the deployment of distributed and web-based services [18].
The above motivations conduct us to claim that a communication broker, which acts as a building block for the realization of enhanced services, is needed. This paper proposes a new communication platform which implements a Broker for nomadic computing environments, named the Esperanto Broker (EB), which has the following proprieties:

1. it provides a flexible interaction paradigm: a crucial requirement for such a paradigm is the decoupling of the interacting entities. EB provides an object oriented abstraction of paradigms proposed in WSDL specification [18]. It thus reduces the development effort, by preserving a structured and typed programming abstraction.

2. it supports terminal mobility by providing mechanisms to handle handovers and handoffs occurrences: by this way, interactions between involved parts are preserved despite terminal movements among different domains and/or terminal disconnections.

By using the proposed platform, developers can model application components as a set of objects that are distributed over wireless domains and interact via remote method invocations.
The rest of technical report is organized as follows. Section 2 describes the conceptual architecture of the overall system, including the adopted interaction model, the programming abstraction, and mobility support. Section 3 details the implementation issues. Section 4 presents preliminary experimental results. Section 5 concludes the paper with some final remarks about results achieved and direction of future work.

2 Overall Architecture

The Esperanto Broker is designed according to the layered architecture depicted in figure 1. As figure shows, such an architecture is composed of two main platforms: Device-side and Mediator-side. In particular, on each domain of the NC infrastructure runs a crucial component, named the Mediator, which is in charge of handling connections with terminals operating in its domain. Mediators are connected each other via the core network. The Device-side platform is in charge of connecting devices with the domain's Mediator, and of providing applications with distributed object view of services (i.e., services are requested as remote invocations on a target server object).
The Mediator-side decouples service interactions in order to cope with terminal disconnections and/or terminal handovers. To this aim, among the mechanisms aiming to achieve flexibility and decoupling of interactions, we adopt the tuple space model thanks to its simple characterization. In our approach, the virtually shared memory is conceived as a distributed space among mediators, and the Device-side platform provides functionalities to access to it. In the following, we describe both Mediator and Device side platforms, in order to highlight design issues have been addressed.

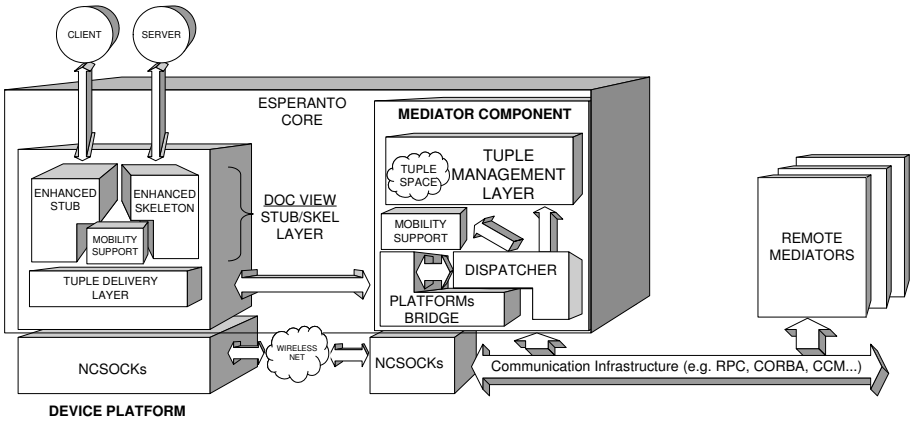

Figure 1: The Esperanto Broker layered architecture

2.1 Device-side platform

Three are the major issues which have been addressed on the device-side platform: i) providing mobile-enabled communication facilities, which also allows applications to interact each others regardless of wireless technologies being used (e.g., WiFi, Bluetooth, and IrDA); ii) incapsulating service method invocations into a list of tuples which enables the Mediator to keep the interaction decoupled; and iii) providing an enhanced IDL language which encompasses all the WSDL interaction types (*request/response, oneway, solicity/response, notify*). In the following, we describe the Device-side components, namely the NCSOCKS communication library, the Tuple Delivery Layer (TDL), and the Stub/Skeleton layer.

NCSOCKS Layer. It allows above layers to request an IP-based communication channel, irrespective of the communication technology being used. Nomadic Computing SOCKetS (NCSOCKS) library is organized into a set of APIs, which are similar to the Java socket APIs, which provides client application with a uniform communication, and with a location and connection awareness support. By using NCSOCKS, applications are able to monitor communication channel status, and, more important, the AP changes in order to trigger a lower level handover procedure. Details about such a library is given in the next section.

Tuple Delivery Layer. This layer provides an API for accessing to the tuple space. More precisely, three are the provided primitives: *write*, *read*, and *take*. The first accepts a tuple as input parameter, containing application-level information (e.g. the method signature to be invoked on the server side). The second (third) accepts as input a tuple template, containing parameters needed to retrieve (to remove) a tuple from the space. All the provided primitives are non-blocking, even though it is possible to specify for the *read* and the *take* primitives a timeout value, which indicates the maximum blocking time (i.e. partial blocking strategy). An asynchronous notify primitive is also provided, in order to reduce the overhead due to tuple retrieval operations. In other words, an application can subscribe itself to a tuple notification service, by means of a *subscribe* primitive. By this way, once a tuple is written into the space, it is forwarded to the subscribed applications directly. Similarly to JavaSpaces [5], a lease is associated to each tuple: a tuple will stay in the space until either it is taken, or its lease expires. This enables the platform to guarantee object interactions despite terminal disconnections. As an example, if a terminal is temporarily disconnected, the Mediator keeps its tuples until the terminal will be able to reconnect to the domain or its associated lease expires. Configurations of lease values should be done in accordance of the application and the network characteristics.

Stub/Skeleton Layer. This layer is in charge of providing applications with a distributed ob-

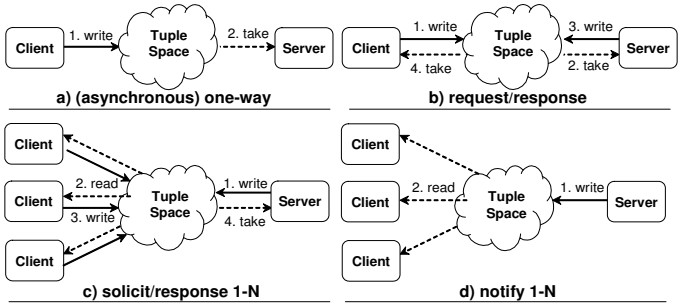

Figure 2: RMI to tuple-oriented implementation

ject paradigm, by translating and marshalling/unmarshalling remote invocation requests in terms of tuple-based operations. As stated by the Web Services standard, four distinct remote invocation strategies are provided: i) *request/response*; ii) *asynchronous one-way*; iii) *notify*; and iv) *solicit/response*. The *notify* implements a server-initiated one-way invocation mechanism which aims to send messages to one or more interested clients. The *solicit/response* strategy is comparable to the *request/response*, except that the request message is initiated by the server, and the response is sent by one or more clients. To support the defined invocation strategies, the standard OMG IDL is enriched. As an example, the following interface

```
interface MyService {
     oneway void fooA(in string name);
     reqres int fooB(in long op, out long result);
     solres void fooC(in string info, out string position);
     notify void fooD(in string weather);
 };
```

defines an interface `MyService` which includes all defined invocation strategies. Parameter-passing directions emphasize a server-side view, for *oneway* and *reqres*, and a client-side view for *solres* and *notify*. In particular, `name` and `op` are input parameters for server objects, but `info` and `weather` are input parameters for client objects. The marshaling procedures involves the following activities: i) extracting parameters from the method signature; ii) formatting them into a tuple; and iii) writing the tuple into the space. Remote invocation strategies are thus implemented using tuple-oriented services provided by the underlying layer. Implementations of such strategies are illustrated in figure 2.

2.2 The Mediator Component

As mentioned, the Mediator component plays a crucial role in a domain, in that it manages connections with terminals, providing applications with a centralized view of the space (i.e., it provides tuple location transparency). More precisely, the Mediator component is in charge i) of managing connections with terminals by keeping transparent communication technologies (since a domain may be composed of heterogeneous wireless networks); and ii) of providing tuple space abstractions. Mediators communicate each others by means of a CORBA ORB. We thus realize the distributed tuple space, as set of CORBA Objects, with all benefits stemming from the use of CORBA middleware (e.g. transparency of communication protocols, object oriented design, the possibility of introducing load balancing and/or fault tolerance techniques). The Mediator itself is a distributed component,

implemented as a set of distributed objects:
Bridge and Dispatcher Objects: since we did not mandate the use of a CORBA implementation on a mobile terminal, a Bridge object is needed, in order to map Protocol Data Units (PDUs, such as a tuple write request) coming from the Device side platform, into tuple-oriented operations. Such operations are implemented as method invocations on Dispatcher object. The Dispatcher object is in charge of providing tuple location transparency, allowing Device-side platform to interact only with the Mediator being currently connected. Moreover, the Dispatcher copes with the mobility of terminals, by implementing the protocols TDPAM and HOPAM, as described later.
Tuple Management Objects: they are in charge of providing tuple space primitives, namely *read, write, take, subscribe/unsubscribe*. The tuple management layer encapsulates a local shared memory providing services for accessing to it. It is worth noting that the tuple space is conceived as a distributed space where each Dispatcher handles a part of the space.
Mobility support: in order to manages terminals mobility, we adopted a solution that was inspired by cellular network. Each Mediator has two responsibilities: i) as *Home* Mediator, it keeps track of domain handovers (namely the domain changes) of devices which are registered with it; ii) as *Host* Mediator, it notifies *Home* Mediators when a roamer device (i.e. a device which is not currently in its home domain) is currently connected to it. To this aim, each Mediator has a Device Manager that is in charge of storing a list of terminals being connected to NC infrastructure. Furthermore, the Mediator side platform implements two protocols in order to guarantee the correctness of a tuple delivery process despite terminal movements and/or disconnections: the Tuple Delivery Protocol Among Mediators (TDPAM) and the HandOver Protocol Among Mediators (HOPAM). The former defines how a tuple is delivered among Mediators and mobile terminals ensuring "seamless" device migrations. [19]. The latter is in charge of managing domain changes as mobile terminals moves, preserving the correctness of the former. We briefly details such protocols in the following:
TDPAM: mainly, this protocol concurs to provide tuple location transparency to the Device side platform. Object interactions between a Source Endpoint (SE) and one or more Destination Endpoints (DEs) take place even though they reside on terminals located in distinct domains. TDPAM works as follows:

1. When an application (i.e. its related stub or skeleton) issues a write request, TDL encapsulates tuple in a Protocol Data Unit (named WRITE PDU), which is delivered to the current Mediator. Three are the possible situations:
   (a) DE is running on terminal which is hosted in the domain where TDL PDU is coming from: tuple is written into the space or notified to interested applications, directly.
   (b) DE is running on terminal which is not hosted in the domain where TDL PDU is coming from: tuple is forwarded to the Mediator which hosts the device. Due the terminal mobility, reference to *Host Mediator* may be obsolete. For this reason, current Mediator inquires terminal's *Home Mediator*, which is in charge of tracking terminal movements (in term of their actual *Host Domain*).
   (c) DE refers to group of application objects: tuple is forwarded to each Mediator of NC network infrastructure.

   Figure 3 depicts the collaboration diagram of TDPAM in the case (b).

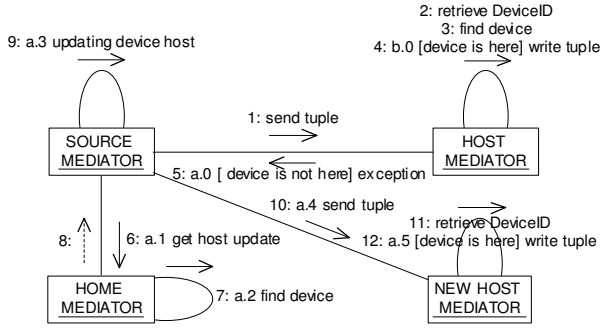

Figure 3: Collaboration diagram in TDPAM - case b)

2. When an application issues a read request, a tuple retrieval operation is invoked on the TDL. This causes the TDL to send the READ PDU on current Mediator, which returns the tuple.

HOPAM: the protocol aims to preserve TDPAM correctness despite device movements among different domains, namely when a device is in *roaming*. More precisely, HOPAM consists of procedures performed by mediators when a device handover occurs. It is described in the following:

- Each tuple addressed to a device which has moved to a new domain, is transferred from the old Mediator's domain to the new one;
- The device's *Home Mediator* is notified about the new *Host Domain*.

In order to trigger handover procedures, the Device side platform is in charge of sending greeting messages to the current Bridge.

3 Design and Implementation Details

In this section we describe the current implementation of our prototype.

3.1 Device-side platform

NCSOCKS Layer. The NCSOCKS provides C++ APIs for accessing to an IP-based communication channel. Such APIs are similar to Java sockets [5], indeed they implement interfaces and classes for providing TCP and UDP communication facilities as described in the following:

- *ServerSocket*, and *Socket* are classes provided to manage TCP connections: `ServerSocket` represents the socket on a server that waits and listens for requests for service from a client, whereas `Socket` represents the endpoints for communication between a server and a client.
- *DatagramPacket*, and *DatagramSocket* are classes provided to send and to receive datagrams using UDP: `DatagramPacket` represents a packet used for connectionless payload delivery, whereas, `DatagramSocket` is a socket used for sending and receiving datagram packets over a network via UDP.

However, NCSOCKS primitives differentiate from the Java network API for the followings characteristics: i) they provide transport level interface which is network technology independent (in the current prototype, NCSOCKS provides an IP abstraction of both Wi-Fi [20] and Bluetooth [21] wireless tecnologies); ii) they have a *mobility aware* behavior; and iii) they raise exceptions in accordance with the channel status. In such a case, the application is able to perform actions that depend on specific needs. As for the *mobility aware* behavior, the `send()` primitive works as follows: if the channel is established, it sends packets to the IP destination; if the connection establishment is in progress, it waits for a specified timeout; then it tries to send packets only if the channel reaches the *connected* status within the time interval. In the all other cases, an exception is raised. The `receive()` has a *mobility aware* behavior, as well: it receives packets only if the connection is established, otherwise raises an exception. However, applications can specify a timeout in order to wait for the reconnection.
NCSOCKS are used to allow TDL (that resides on mobile terminal) and the Bridge (that resides on the core network) to exchange PDUs. A mobile terminal (such as a PDA or a laptop), is connected to the fixed network infrastructure via an Access Point (AP) through services of the Connection and Location Manager (CLM, a software module placed between application and kernel space). The CLM component is in charge of implementing lower level handover procedures (on data-link and network layer).
Tuple Delivery Layer. In order to provide primitives to access to tuple space (i.e., *write, read, take, subscribe and unsubscribe*), the TDL is in charge of exchanging PDU between the Device side and Mediator side platforms using API provided by NCSOCKS layer. However, as stated in [22, 23], TCP is unsuitable for mobile computing environments (although it provides reliable delivery), while UDP appears inadequate for wireless links (although it introduces lower overhead). Therefore we adopt the NCSOCKS UDP network interface, but we implement TDL (and its counterpart on the Mediator side, the Bridge), integrating the necessary strategies in order to overcome the UDP disadvantages, such as the out of order packet delivery or the datagram packet loss. However, each TDL primitive raises exceptions in accordance with the status of related PDU process delivery, so that stub and skeleton classes can perform actions in order to recover from communication failure. In particular, stub and skeleton class are provided with primitives to detect domain changes and/or long terminal disconnections. In such cases, it may be possible that tuple delivery process does not complete correctly. In order to prevent such problems, stub and skeleton classes implement recovery strategies that take advantages of the presence of tuple space.
Enhanced Stub/Skeleton layer. The enhanced Stub/Skeleton layer is in charge of providing the DOC view to the developers. At time of this writing we are currently implementing an enhanced version of a pseudo-IDL compiler for the C++ language. However, we actually implemented a set of templates in order to generate stubs and skeletons by a semiautomatic process (with the developer intervention). Such a process produces the stub/skeleton classes `InterfaceStubC` and `InterfaceSkelC`, for the client-side and `InterfaceStubS` and `InterfaceSkelS` for the server side. `InterfaceStubC` and `InterfaceSkelS` classes implement the proxy pattern [24] in order to provide *request/response* and *oneway* methods; `InterfaceSkelC` and `InterfaceStubS` classes implement such a pattern in order to provide *solicit/response* and *notify* methods. To exemplify, in figure 4 is illustrated the result obtained by such a process in the case of a simple `reqres` method. In the figure, we emphasize the mapping process (implemented in the stub class) of a remote method invocation into *subscribe* and *write* tuple-oriented primitives. In order to avoid busy wait

Figure 4: Pseudo-IDL compilation results

on client and server sides, the stub and skeleton use callback interfaces. According to the implementation strategies depicted in figure 2, mapping of *request/response* into tuple-oriented primitives consists of the following operations: i) callback interface subscription in order to retrieve the response/request tuple (implemented in the stub/skeleton constructor by providing the tuple template); ii) tuple write operation; and iii) waiting for the notification. If the tuple can not be notified to the interested entities, due to temporarily disconnections, it is written in the tuple space. When the network is available again, interested objects (i.e. related stubs or skeletons) can retrieve the tuple via the *take* operation.

Mobility support: a daemon on Device platform periodically sends a `GREETINGS` PDU to current *Host* Mediator (i.e., its related Bridge). `GREETINGS PDU` contains the following information: Device HoSt Mediator Identifier (DHSI), Device HOme Mediator Identifier (DHOI), and DEvice Identifier (DEID). An handover procedure is triggered if DHSI $\neq$ CHSI, i.e. when reference to *Host Mediator* is different from Current Host Mediator Identifier. To this aim, the Device side platform has also the responsibility of keeping information about DHSI, DHOI, and DEID.

3.2 Mediator-side platform

Figure 5 depicts a detailed UML CORBA component diagram of the Mediator-side platform. As figure shows, the Mediator component consists of four CORBA servers: the first contains the Bridge and Dispatcher objects; the second, the *TupleManager*, implements the Mediator's tuple space; the third, the *TupleFactory*, acts as tuple factory server; and, the fourth, the *DeviceManager*, is in charge of handling information concerning mobile devices actually connected to the domain. As far as the implementation is concerned, we used TAO [25] as CORBA platform.

As depicted in figure 6, a tuple consists of a XML descriptor containing the following tags: i) *sender*, which is the Source Endpoint (SE) of the remote interaction, i.e. the object that sends messages; ii) *receiver*, which is the Destination Endpoint (DE) of the remote interaction, i.e. the object that receives messages (a DE may represent also a group of objects, so that more objects may receive messages from one SE); iii) *paramList*, which

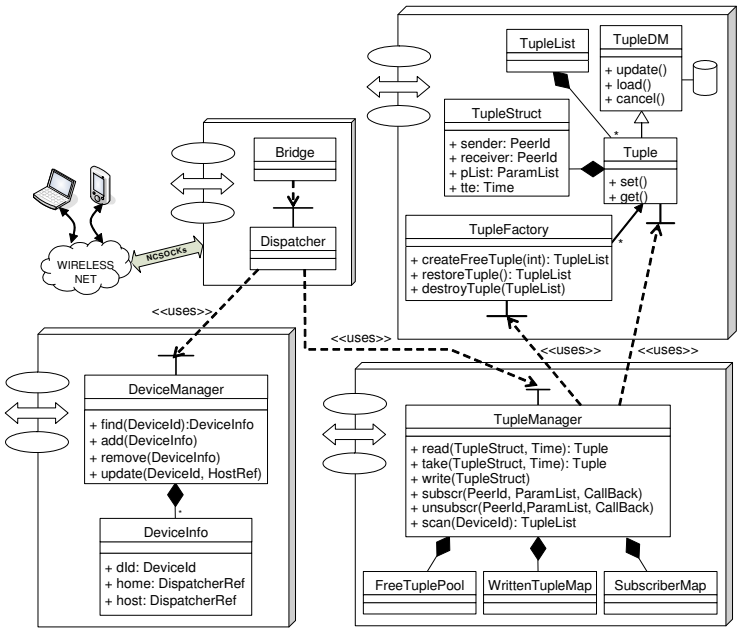

Figure 5: Detailed Component diagram of Mediator-Side platform

```
<tuple>

  <sender> </sender>
  <receiver> </receiver>

  <parameters>

    <parameter>
       <name> </name>
       <type> </type>
       <value> </value>
    <parameter>
          ...
  </parameters>


  <timeToExpiration> </timeToExpiration>

</tuple>
```

```
//module Mediator ...
struct DeviceId {
  DeviceIndex idx;
  DomainId homeId;
};
struct PeerId {
  PeerIndex  pI;
  DeviceId   dI;
};
struct Param {
  string type, name, value;
};
typedef sequence<Param> ParamList;
struct TupleStruct {
  PeerId sender, receiver;
  ParamList pList;
  Time tte;
};
interface Tuple {
  attribute TupleStruct ts;
};
```

Figure 6: Tuples description: XML view and their mapping into CORBA Objects

is a list of parameters, where each one has a name, an attribute, and a value; and iv) *tte* (Time To Expiration), which indicates the lease time of the tuple. As figure 6 shows, each communication endpoint is represented by a *PeerId*: a *PeerId* allows EB to locate an Esperanto object, despite its terminal movements. It is composed of a *PeerIndex*, which allows EB to dispatch requests to several objects that are running on the same terminal, and a *DeviceId*, which consists of a device identifier, and of terminal's home domain identifier. By this way, the EB is able to locate a terminal in the entire NC environment. In order to achieve persistence of XML descriptors, our solution mandates such responsibility to a CORBA object (i.e. *Tuple*, see figure 6). Such an approach allows to preserve the transparency of a specific database technology, encapsulating the persistence strategy (e.g. XML-native DB, RDBMS, file stream) in the CORBA servant implementation. In the current prototype, persistence is achieved by the `TupleDM` class, which implements a serialization of the *TupleStruct* attribute into a XML file.

Tuple Factory. It is a factory of *Tuple* servants. Servants can be created in two ways: i) factory creates ready-to run *Tuple* servants; ii) factory creates recovered instances of *Tuple*

servants, i.e., it creates tuples by XML tuple representations from mass storage. As far as the allocation strategy is concerned, since it is expensive to create a new tuple for an each incoming request, a pool of tuple servants is created at initialization time (the administrator sets the number of tuples which make up the pool). The allocation algorithm may be specialized for achieving load balancing among servers, or redundancy by replicating tuples over the available servers.

Tuple Manager. It is a crucial component, in that it offers tuple space primitives: *read, write, take, subscribe, unsubscribe, and scan*. The *read* and the *take* methods implement the template matching algorithm. Such an algorithm performs a template matching between a tuple template (i.e. a tuple with both formal and actual parameters) provided as input, and all the tuples residing on the Mediator's space. A template matches a tuple if the following conditions hold: i) the tuple and the template have the same XML schema or the template has a `null` schema (i.e. the parameters section is empty); ii) the tuple and the template have the same DE; iii) the tuple and the template have the same SE, if specified in the template, otherwise this condition will be ignored; and iv) the list of parameters matches, i.e. each parameter has the same name, type, and (if specified) the same value. The shared memory (that contains a limited number of tuples) is implemented in terms of two classes, the `WrittenTupleMAP` class, and the `FreeTuplePOOL` class. The former contains the IORs of *Tuple* servants related to tuples being written in the space, the latter contains the IORs of *Tuple* servants which are available in the space. When `TupleManager` stores a tuple in the shared memory, it pops an IOR from `FreeTuplePOOL`, sets the *TupleStruct* attribute, and then it pushes tuple into `WrittenTupleMAP`. In order to implement the asynchronous notification of a tuple to an Esperanto object, `TupleManager` has a `subscriberMAP` class. It is a Standard Template Library (STL) `multimap` which stores the relationship among *PeerID*, callback interface, and tuple template. It also keeps the association (1-to-n) between the groupID and all the belonging *PeerIDs*. By this way, it possible to manage 1-to-n communication. The *write* algorithm works as follow: i) it extracts the tuple structure, provided as input parameter, and in particular the *receiver* field; ii) it checks if the provided Destination Endpoint are already subscribed to the write event (by means of `subscribe()` method); iii) for all subscribed Esperanto objects, it simply forwards tuple to them; iv) if there are destinations which are not reachable (i.e., they may be temporarily disconnected) or not subscribed, it stores the tuple into the space. It should be emphasized, that such an algorithm has been designed trying to minimize the overhead in the case of all the endpoints are "alive and kicking". *Subscribe* and *unsubscribe* methods allows Esperanto objects to register and to unregister their callback interfaces. The *scan* method is in charge of retrieving all tuples addressed to a specific mobile terminal. It processes the whole shared space, returning to the caller the list of tuples whose *receiver* field contains the specified *DeviceId*. This method is crucial for HOPAM implementation because it allows tuples belonging to a roamer device to be moved from old *Host Mediator* to new *Host Mediator*.

Device Manager. It is in charge of keeping the information about devices being connected to the NC infrastructure. More precisely, for each connected device identified by its *DeviceId*, `DeviceManager` holds IOR to its actual *Host Mediator* and its *Home Mediator*.

Bridge and Dispatcher. The Bridge interprets (creates) PDUs coming from (directed to) devices, and the Dispatcher is charge of performing operations described in the PDU. PDU may be of the following types: `WRITE PDU`, to write a tuple, `READ/TAKE PDU`, to read/take a tuple, `UN/SUBSCRIBE PDU`, to register/unregister to a tuple notification, `GREETINGS PDU`,

```
module Mediator {
        ...
  interface TupleManager {
    void write(in TupleStruct ts);
        ...
  };
  interface DeviceManager {
    Device find(in DeviceID dID);
        ...
  };
  interface Dispatcher {
    write(in TupleStruct ts);
    // d is device, oh is old host
    DomainId greeting(in DeviceId d, in DomainId oh);
  };
  interface RemoteDispatcher {
    remoteWrite(in TupleStruct ts);
    void notify(in DomainId host, in DeviceId dev);
    // d is device, nh is new host
    TupleList moveTuples(in DeviceId d, in DomainId nh);
        ...
  };
};
```

Figure 7: Partial view of Mediator IDL Module

to notify mobile terminal status to the current Mediator[1]. To exemplify, let assume that a `WRITE` PDU, coming from a device, contains a tuple that an Esperanto object would write into the shared space. From this point on, i) the Bridge extracts the PDU from the tuple and invokes the *write* operation on the Dispatcher object; ii) the Dispatcher extracts *DeviceId* from the tuple and retrieves device's state from the `DeviceManager` (namely, its current host domain and its home domain); then iii) it checks if the tuple is addressed to a device residing in the same domain; iv) if it is, it invokes the *write* operation on the `TupleManager` object, otherwise, it invokes a remote *write* on the Dispatcher which is currently connected to destination device. In figure 7 we give a partial view of Mediator IDL module. When a `GEETINGS PDU` is delivered to the current Mediator, it potentially triggers an handover procedure. Once the handover has been triggered, the new Mediator's Dispatcher i) notifies the current device location to device's *Home Mediator* (by invoking the `notify()` method); and ii) transfers all the tuples concerning the terminal, from the old domain space to its local space (by invoking the `moveTuples()` on the old Dispatcher. The resolution of Dispatcher IORs is performed by means of distributed CORBA Naming Service structure, in which all Mediators are bound.

4 Preliminary experimental results

In this section we discuss the results obtained from performance experiments we have conducted, as compared with MIwCO, an open-source implementation of OMG Wireless CORBA [26]. The Wireless CORBA reference architecture is quite similar to the Esperanto Broker: it forces the presence of a special component on mobile terminal which is similar to ESPERANTO mobility support on Device side platform (named Terminal Bridge, TB), and two components on the network core (one named Home Location Agent, HLA, and one named Access Bridge, AB) which have the same role of the ESPERANTO Dispatcher and Device Manager respectively. However, the OMG Wireless CORBA does not offer all the paradigms standardized by the W3C, and method invocations are implemented by means of synchronous invocations. Moreover, network monitoring algorithms needed to trigger handover procedures are not specified in the Wireless CORBA, and they are not implemented in the current version of MIwCO. The Esperanto Broker addresses such issues in its architecture and implementation.
The main objective of our experiments is to evaluate the performance penalty due to i)

[1] while the former are exchanged by the Esperanto objects, the latter is a PDU for mobility support

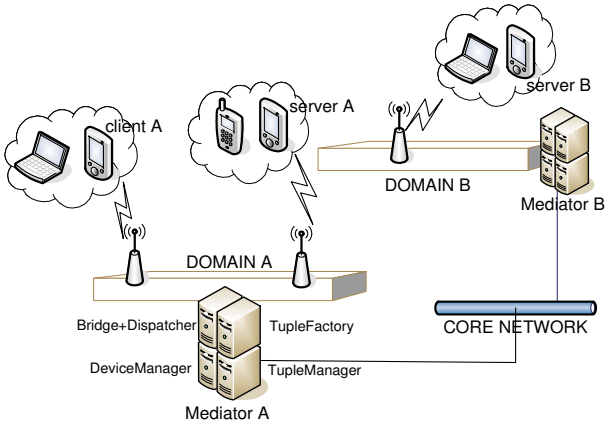

Figure 8: Experimental testbed

the adoption of a CORBA implementation for connecting Mediators; ii) the tuple space management (i.e. lease management, callback interfaces handling); iii) the decoupled interactions; and iv) the mobility management (i.e. the implementation of TDPAM/HOPAM), in order for reducing the introduced overhead.
All experiments were performed with following testbed (see figure 8): Mediators (Device Manager, Tuple Manager, Tuple Factory and Tuple, Bridge and Dispatcher) and AB+HLA are distributed on two 1.8Ghz CPU PIV computer with 1GB of RAM running on Linux 2.4.19. As far as mobile devices are concerned, we use Compaq IPAQ 3970, equipped with Linux familiar v0.7.1, Wi-Fi 802.11b and Bluetooth modules. As far as the core network is concerned, we used a Fast Ethernet switch which links all the servers where Mediators (ABs) are running. In order to interconnect Mediators, we used the CORBA TAO version 1.4.1. During tests, the external load was the normal background load of active services and applications. As for performance measurements, we aim to evaluate round-trip latency of a method invocations, according to guidelines given in [27]. More precisely, in order to compare ESPERANTO measurements with MIwCO ones, we evaluate the two-way *request/response* interaction paradigm. It should be emphasized that it does not represent a limitation, in that the round-trip latency, obtained by a *request/response* method invocation, can be used as a fine-grained evaluation to analyze the other interaction paradigms. As for MIwCO, we used both GTP over TCP and GTP over Bluetooth L2CAP protocols, but we experienced that obtained results differs only from a constant value (as function of parameter size). For this reason we used GTP (for MIwCO) and NCSOCKS (for ESPERANTO) over Bluetooth. As for the comparison, we referred to the following IDL and pseudo-IDL interfaces, which express the same service, namely `Measure`.

```
interface Measure {
   reqres long foo(in string op);
 };
```

```
interface Measure {
    reqres long foo(in string op);
};
```

The generated stub and skeleton contains the same signature of the `foo` method, both for MIwCO and for ESPERANTO. The performance penalty was measured as the the ratio of the ESPERANTO to the MIWCO round-trip latency time, i.e. $K = t_{ESPERANTO}/t_{MiWCO}$. We also measured the latency increment (calculated as $t_{i+1} - t_i$) for two subsequent invocations, as function of the dimension of the input parameter, namely `op`. Measurements are performed under two scenarios. In the first scenario, the client and the server objects are connected to the same domain (the same Mediator, for ESPERANTO, and the same AB, for MIwCO). Results are depicted in figure 9, which shows that the obtained performance

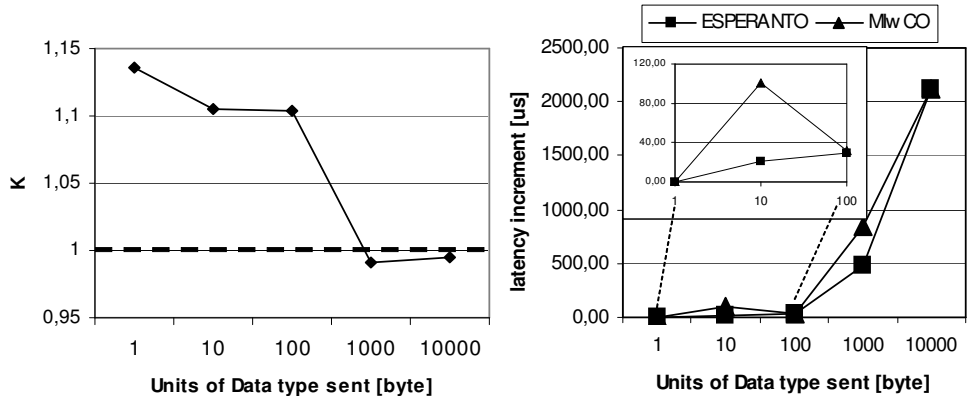

Figure 9: Client/Server interaction by means of same Mediator/Access Bridge

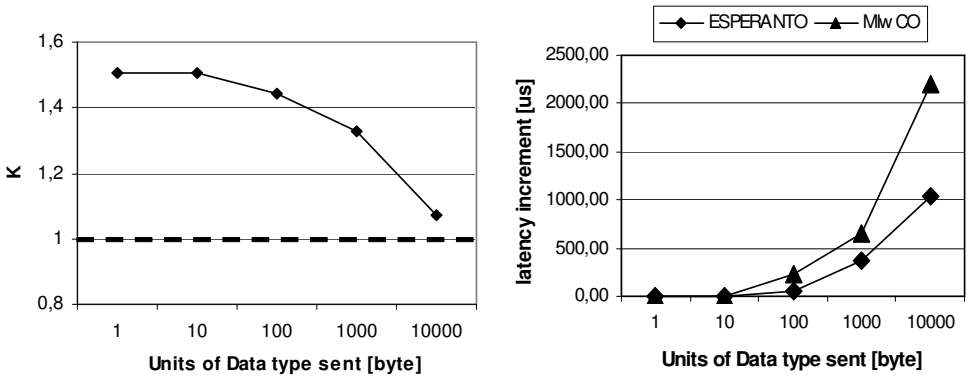

Figure 10: Client/Server interaction by means of different Mediators/Access Bridges

are quite similar.
In the second scenario, client and server objects are located in two distinct domains. As for MIwCO, client and server interactions take place as follows: i) client sends the CORBA Request, which encapsulates the Mobile IOR, to the TB; ii) the TB sends the request to the AB via the GTP; and iii) the AB contacts the HLA, first, to retrieve the IOR (by means of `LOCATION_FORWARD` message) of the AB where the server object is operating, and, for all successive invocations, it sends the request to the server's AB directly. Results are depicted in figure 10. As figure shows, ESPERANTO has a cost in terms of performance, due to the fact that it uses more complex interactions among Mediators. However, as stated in the next section, we are currently integrating real time facilities provided by TAO in order to improve performance.

5 Conclusions and Future Work

This paper has described the Esperanto Broker for Nomadic Computing, which provides application developers with an enhanced Distributed Object programming model, integrating mobility management support and adopting mobile-enabled interaction mechanisms. A prototype was developed and tested over distributed heterogeneous platform.
From a methodological point of view, this experience has shown that ESPERANTO pro-

vides an effective means for supporting applications running over a nomadic environment. From an experimental point of view, result demonstrated that the proposed architecture has a cost in term of performance. Our future work aims to integrate QoS-based mechanisms, and to support applications with requirements in terms of desired QoS. Indeed, although QoS issues may be not relevant to nomadic applications, we recognize that two kinds of application may run on a nomadic environment: applications specifically designed for mobile environments, e.g. Mobile News Services, which may tolerate frequent disconnections; and applications which are designed to operate in a stable environment but they are developed to be executed on mobile terminals, e.g. multimedia (streaming-based) smart guided. It is evident that the latter are less mobility tolerant than the former. For such applications, the middleware has to provide QoS-enabled mechanisms, in order to notify applications when the QoS requirements cannot be satisfied. In order to allow nomadic applications to specify the mobility tolerance, i.e. if the application may operate or not under frequent disconnections and QoS fluctuations, we are currently concentrating on real-time facilities provided by TAO, in term of its Real Time Scheduling Service (RTSS) and its RT Current ORB interface (which can be configured with different RT CORBA QoS Policies, in terms of *ThreadPool*, *Explicit Binding*, and *TCPProtocolProperties*). We are also concentrating QoS policies provided by the wireless network infrastructure in the case of Bluetooth connections. More precisely, NCSOCKS library allow to create Bluetooth Asynchronous ConnectionLess (ACL) connections, with different values of assigned temporal slots (DH1, DH3, DH5 modes [21]) achieving different levels of QoS. We claim that this approach will also result in a performance improvement.

References

[1] C. Mascolo, L. Capra, and W. Emmerich. Mobile Computing Middleware. *Lecture Notes In Computer Science, advanced lectures on networking*, pages 20 – 58, 2002.

[2] L. Kleinrock. Nomadicity: Anytime, Anywhere in a disconnected world. *Mobile Networks and Applications*, 1(1):pages 351 – 357, December 1996.

[3] OMG. Object Management Group: The Common Object Request Broker: Architecture and Specification 3.0.3 ed., 2004.

[4] Microsoft. DCOM tecnology, 2004. http://www.microsoft.com/com/tech/DCOM.asp.

[5] Sun Microsystems: Sun Microsystem Home Page, 2004. http://www.sun.com.

[6] A. Gaddah and T. Kunz. A Survey of Middleware Paradigms for Mobile Computing. Techical Report, July 2003. Carleton University and Computing Engineering.

[7] C. Mascolo, L. Capra, and W. Emmerich. An XML-based Middleware for Peer-to-Peer Computing. *In Proc. of 21st IEEE Int. Conf. on Distributed Computing Systems*, pages 69–74, 2001.

[8] A.T.S. Chan and S.N. Chuang. MobiPADS: a reflective middleware for context-aware mobile computing. *IEEE Transactions on Software Engineering*, 29(12):1072–1085, 2003.

[9] L. Capra, W. Emmerich, and C. Mascolo. CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications. *IEEE Transactions on Software Engineering*, 29(10):929–945, 2003.

[10] A. Popovici, A. Frei, and G.Alonso. A proactive middleware platform for mobile computing. *Proc. of the 4th ACM/IFIP/USENIX International Middleware Conference*, 2003.

[11] J. He, M.A. Hiltunen, M. Rajagopalan, and R.D. Schlichting. Providing QoS Customization in Distributed Object Systems. *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, LNCS 2218:351–372, 2001.

[12] J. Luo, P.T. Eugster, and J.P. Hubaux. Pilot: Probabilistic Lightweight Group Communication System for Ad Hoc Networks. *ACM transaction on mobile computing*, 3(2):164–179, April-June 2004.

[13] A. Montresor. The Jgroup Reliable Distributed Object Model. *In Proc. of the 2th IFIP WG 6.1 Int. Conf. on Distributed Applications and Interoperable Systems*, 1999.

[14] Network Working Group, IETF. *IP mobility support, RFC 2002*, 1996.

[15] H. Schulzrinne and E. Wedlund. Application-Layer Mobility Using SIP. *Mobile Computing and Communications Reviews*, 4(3):47–57, 1999.

[16] A. Bakre and B. Bradinah. I-TCP: Indirect TCP for mobile hosts. *in Proc. of 15th Int. Conf. on Distributed Computing Systems*, 1995.

[17] K. Raatikainen. Wireless Access and Terminal Mobility in CORBA. Technical Report, December 1997. OMG Technical Meeting, University of Helsinky.

[18] W3C. The World Wide Web Consortium. Web Services Description Language (WSDL) 1.1, 2004. http://www.w3.org/TR/wsdl.html.

[19] M. Endler and V. Nagamuta. General Approaches for Implementing Seamless Handover. *in Proc. of the 2nd ACM int. workshop on Principles of mobile computing*, pages 17–24, 2002.

[20] IEEE. *IEEE 802.11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications*, 1999.

[21] Bluetooth SIG. *Specification of the Bluetooth System - core and profiles v.1.1*, 2001.

[22] V. Cahill and T. Wall. Mobile RMI: Supporting Remote Access to Java Server Objects. *in Proc. of 3rd Int. Symp. on Distributed Objects and Applications (DOA)*, pages 41–51, 2001.

[23] A. Bakre and B. R. Badrinath. M-RPC: a remote procedure call service for mobile clients. *in Proc. of 1st Int. Conf. on Mobile Computing and Networking (MobiCom)*, pages 97–110, 1995.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[25] Douglas C. Schmidt, D. L. Levine, and S. Mungee. The Design of the TAO Real-Time Object Request Broker. *Computer Communications*, 21(4):pages 294–324, 1998.

[26] Mico project: MIWCO and Wireless CORBA home page, 2004. http://www.cs.helsinki.fi/u/jkangash/miwco/.

[27] A. S. Gokhale and D. C. Schmidt. Measuring and optimizing CORBA latency and scalability over high-speed networks. *IEEE Transactions on Computers*, 47(4):pages 391–413, 1998.