A. D. MCCXXIV

UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

Dottorato di Ricerca in Ingegneria Informatica ed Automatica

Comunità Europea
Fondo Sociale Europeo

THE DESIGN AND DEVELOPMENT OF A NOMADIC COMPUTING MIDDLEWARE: THE ESPERANTO BROKER

ARMANDO MIGLIACCIO

Tesi di Dottorato di Ricerca

Novembre 2006

Dipartimento di Informatica e Sistemistica

via Claudio, 21- I-80125 Napoli - *[#39]* (0)81 768 3813 - *[#39]* (0)81 768 3816

UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

Dottorato di Ricerca in Ingegneria Informatica ed Automatica

Comunità Europea
Fondo Sociale Europeo

THE DESIGN AND DEVELOPMENT OF A NOMADIC COMPUTING MIDDLEWARE: THE ESPERANTO BROKER

ARMANDO MIGLIACCIO

Tesi di Dottorato di Ricerca

(XIX Ciclo)

Novembre 2006

Il Tutore

Prof. Stefano Russo

Il Coordinatore del Dottorato

Prof. Luigi P. Cordella

Dipartimento di Informatica e Sistemistica

via Claudio, 21- I-80125 Napoli - *[#39]* (0)81 768 3813 - *[#39]* (0)81 768 3816

THE DESIGN AND DEVELOPMENT OF A NOMADIC COMPUTING MIDDLEWARE: THE ESPERANTO BROKER

By

Armando Migliaccio

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
"FEDERICO II" UNIVERSITY OF NAPLES
VIA CLAUDIO 21, 80125 - NAPOLI, ITALY
NOVEMBER 2006

"FEDERICO II" UNIVERSITY OF NAPLES
DEPARTMENT OF
COMPUTER AND SYSTEM ENGINEERING DEPARTMENT

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled "**The design and development of a Nomadic Computing middleware: The Esperanto Broker**" by **Armando Migliaccio** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Dated: November 2006

External Examiner: _____
Prof. Stefano Russo

Research Supervisor: _____
Prof. Stefano Russo

Examing Committee: _____

_____

"Alcuni si ritengono perfetti
unicamente perché sono meno esigenti nei propri confronti."
(Hermann Hesse, *"Aforismi"*)

Table of Contents

List of Tables

List of Figures

Acknowledgements

Solo tre anni mi separano dalla volta in cui mi sono trovato di fronte ad una pagina di ringraziamenti, la mia. Allora, la volontà di esperimere la gratitudine e l'amore verso chi mi ha aiutato, supportato, incoraggiato, guidato, aveva sprigionato in me una forte emozione che ancora oggi non riesco a contenere.

Tante cose sono cambiate, tante altre immutate, molti dubbi dissolti, alcune incertezze da sciogliere, diversi equilibri mutati, tanti desideri ancora da soddisfare. Ma di certo c'è che questa pagina sarà per tutti quelli che nel proprio modo di esprimere il loro affetto per me, mi sono stati vicino.

La scelta di scrivere questa pagina in italiano è per dedicarla principalmente a loro, alla mia famiglia, la cosa che ho più cara al mondo e per la quale nessun gesto di affetto vale l'espressione dell'amore che provo. Nonostante a volte sia in disarmonia con i loro modi di esternare le passioni, buone o cattive che siano, la mia comprensione di ciò è assoluta, ed il più profondo modo di esprimerlo non riuscirà mai ad essere racchiuso in queste righe. Grazie per tutto quello che sono, il tempo è ora dalla mia parte, sarà mio il compito di guidarvi ed esservi vicino qualsiasi distanza ci separi.

Alle persone care che nel corso di questi anni si sono avvicendate nella sfera dei miei affetti. Nonostante gli eventi passati e futuri, sono e rimarranno sempre nel mio cuore, come uno scrigno chiuso e nascosto nei luoghi ignoti del mio animo. Grazie.

Le persone con cui ho diviso la maggior parte delle mie giornate, colleghi, mentori ed amici. Grazie per avermi dato la possibilità di essere quello che sono, buono o

cattivo che sia, non importa. Il mio ringraziamento è di profonda riconoscenza per quanto mi è stato insegnato con ostinazione e perseveranza. Grazie.

Per aver vissuto anche questa esperienza insieme, grazie Marcello. In tutti questi anni sei stato per me allo stesso tempo amico, fratello, collega, guida, allievo, tutto. Non dimenticherò mai ciò che è stato tra noi, nonostante i nostri cammini possano prendere percorsi differenti e perdersi in orizzonti lontani.

Per quello che il tuo amore mi ha dato e quello che potrà darmi ancora, grazie Cristina. Sei ciò che in questi ultimi anni ha rappresentato uno scopo per i miei sforzi ad essere migliore, in tutto.

Naples, Italy Armando

November 30, 2006

Introduction

Recent advantages achieved in wireless and in low power consumer electronics have been leading to new computing paradigms, which are generally described as mobile computing. Nomadic Computing is a form of mobile computing where the communication may take place during users movements between different service locations such as their office, home, hotel, airport, car and so on [1, 2, 3]. Nowadays the interest in middleware for Nomadic Computing environments is still growing since such infrastructures are becoming wider and wider: suffice to say, telecom operators are competing to quickly interconnect different wireless networks (such as GSM, UMTS and Wi-Fi) to make the *Wireless Internet* the cutting-edge market where new services can be provided at a huge profit.

Most generally, middleware are set of abstractions, services and mechanisms to help applications use networked resources and services. They have emerged as a critical second level of the enterprise IT infrastructure, between the network and application levels. The need for middleware stems from the increasing growth in the number of applications, in the customizations within those applications, and in the number of locations in our environments. This provision of services eases application development, increases robustness, assists data management, and provides overall operating efficiencies.

When developing distributed applications, designers do not have to deal explicitly with problems related to distribution since middleware provide them with a higher level of abstraction. Existing middleware, such as object-oriented middleware, have been built trying to ease the development of distributed applications as much as possible [14]. To support designers building nomadic applications, research in the field of middleware systems has proliferated. Nomadic computing middleware aim at facilitating communication and coordination of distributed components, concealing complexity raised by mobility from application engineers as much as possible.

Focusing on distributed communication, nomadic computing middleware must deal with new challenging issues that are mainly due to inherent characteristics of wireless networks and mobile devices [4, 5]. Weak connectivity or battery power constraints may lead users to experience short periods of service unavailability. Moreover, users may interact with a service just before a travel from office to home. Even if the network connection is available both outdoor and indoor (by example, via GSM and Wi-Fi) switching from a technology to another may cause disconnections. Therefore temporary unavailability of counterparts is a rule rather than an exception. If traditional communication mechanisms required both counterparts to be available during the interaction, mobile computing requires mechanisms to let users communicate in a loosely coupled fashion.

From the software development perspective, developing next-generation mobile computing applications needs to rely on high-level abstractions and advanced services. Mobile infrastructures make possible new challenging scenarios: meeting people and exchanging information with them, receiving dynamic content or let the computation be location-aware are only some of the possible applications whose design and

implementation should be simplified by nomadic computing middleware.

It is widely recognized that traditional middleware (such as CORBA [6], DCOM [7], and JavaRMI [8]) appear inadequate to be used for nomadic computing environments [9, 10]. They do not provide any support in terms of both mobility management (i.e. handoff procedures for handling device mobility), and mobile-enabled paradigms (i.e. paradigms that are decoupled in space, time, and synchronism [11]). However, they offer a high-level computing model and a powerful programming interface which help developers to reduce the time to market.

In order to provide new solutions, during the last decade the research deal has been progressed along two distinct directions: i) extending traditional middleware implementations with primitive mobile-enabled capabilities (e.g. Wireless-CORBA [12]), and ii) proposing middleware which adopt mobile-enabled computing models (e.g. Lime [13]). While we recognize that these approaches have been leading to important results, both of them have drawbacks: the former adopts a more effective computing model [14], but does not effectively overcome the intrinsic limitation of the synchronous remote procedure call; the latter adopts decoupled interaction mechanisms, but fails in providing a high level and well understood computing model abstraction. We claim that a *unified* approach, both of adopting a powerful computing model and mobile-enabled communication mechanisms, should be adopted.

This thesis proposes the Esperanto Broker, a communication platform for nomadic computing environments which takes advantages of both the above mentioned approaches. The Esperanto Broker adopts the distributed objects computing model and enhances it to achieve the realization of new application scenarios. At the same

time, it adopts mobile-enabled interaction paradigms as the underlying infrastructure which applications will rely on. More precisely, the Esperanto Broker has the following features: i) it addresses mobility issues via a unified approach, i.e. both at data-link, network and middleware levels; ii) it adopts a tuple space as the underlying communication infrastructure; and iii) it provides Distributed Object Computing (i.e. DOC) model which is enhanced according to the communication paradigms standardized by the W3C [15].

The Esperanto Broker core is built using mobility aware mechanisms (such as hand-off procedures) to guarantee that communications successfully take place despite temporary disconnections due to device mobility. As for transport facilities, RMIs, used by Esperanto objects to interact, are built using the tuple space infrastructure. The Esperanto computing model provides both *request/response-oneway*, and *solicit/response-notify* paradigms which are consistent with the ones proposed by Web Services specification.

Using the Esperanto Broker, developers can model application components as a set of objects that can interact via *pull* and *push* models, in both one-to-one and one-to-many multiplicity. Built-in mechanisms to achieve group communication keep objects unaware of implementation details, and may greatly simplify the development and deployment of next-generation mobile computing application scenarios.

To test the effectiveness of the Esperanto Broker approach, we have employed it in educational projects of the basic distributed programming courses at the University of Naples, where several mobile applications have been successfully developed. Empirical experiments have been also conducted, proving the attractiveness of the proposed platform. Although our first prototype has a cost in terms of performance, it

shows a predictable behavior in presence of device mobility and high load situations. Invocation's latency remains basically stable despite how many objects are connected to the platform, whereas handoff procedures introduce predictable overhead.

Chapter 1

Nomadic computing

This chapter sheds some light on nomadic computing systems. Firstly, it provides the background of nomadic computing systems, describing scenarios, challenges, and limitations of traditional middleware in supporting nomadic computing applications. Secondly, it defines requirements and constraints that a nomadic computing middleware should deal with to represent an effective solution for such systems.

1.1 The nomadic computing paradigm

1.1.1 Nomadic computing systems

Nomadic computing systems are a compromise between totally fixed and totally mobile systems. Nomadic computing environments are usually composed of a set of mobile devices and a core infrastructure with permanent and wired nodes. In such systems mobile devices move among different locations, while maintaining a connection to the fixed network. Usually, the wireless network connects the edges of a fixed infrastructure to the mobile devices. Although nomadic computing and traditional distributed systems share a lot of similarities, functional requirements and systems

constraints greatly vary, thus demanding the design of suitable middleware solutions. In the following sections, motivating scenarios will lead the reader to understand why these environments are very challenging, and why the traditional approach in proposing middleware for distributed systems is nearly inadequate to let nomadic computing applications be a reality nowadays.

1.1.2 Nomadic computing scenarios

Lots of people require to be constantly on the move, by example, workers who oversee geographically dispersed operations, or other workers whose jobs simply require them to be on the go. Students are another type of user with demanding mobility needs. Constantly moving from class to class, meeting with professors and other students, and often working from various locations such as the classroom, library, hallway, or home. Nowadays, everyone may consider himself/herself a nomadic user, everyone needs, regardless of his/her specific job, to spend countless hours away from his/her desks, whether he/she is in meetings, talking to colleagues in hallways, working at other locations, or in transit. To support the user with the *Anytime*, *Anywhere Access* stated by Kleinrock in [16], lots of IT companies and Telecom operators have been struggling to deploy nomadic computing infrastructures, mixture of fixed and wireless network infrastructure where mobile and fixed devices may cooperate to provide/require services. Such infrastructures open up scenarios which have been unfeasible so far. Some of the scenarios detailed in [17, 18] are presented in the following:

- **The notebook PC User**: Liz is a notebook PC user. With a work schedule that typically consumes 70 to 80 hours a week, Liz has to be extremely efficient

in time management. From her office, Liz has to leave for a meeting at the company's research lab, located several miles across town. With grab-and-go hot docking, she is able to grab her notebook from the docking station at her desk and run without having to initiate a standby or hibernate sequence.

On the company shuttle bus, Liz opens her notebook and starts sending three urgent e-mail messages, one from the CEO and two from her peers. At the research lab, Liz spends about 90 minutes in a meeting with a new product team. With her PC, Liz takes extensive notes on the product, information that she wants to transfer to her sales and marketing division's headquarter.

At the meeting's close, Liz connects her Tablet PC to a network hub in the conference room. The notes she drafted are synchronized instantly when she connects to the company network. Her administrative assistant, located across town at headquarters, took Liz's notes, while she distributed the memos to the sales and marketing team leads.

- **The corridor warrior**: Garrett is a product development manager for a 300 person company that makes portable ultrasound devices for the medical industry. Garrett spends at least three or four hours away from his desk each day in meetings. Using his mobile personal digital assistant Garrett Garrett can spend even more time out of the office circulating among several different product teams, still maintaining seamless communication with wireless e-mail access, and taking advantage of the ability to work wherever he might be.

- **The student**: today's students are corridor warriors in training. They move

from class to class, from libraries to coffee shops, taking notes, and preparing papers, while constantly communicating with friends, faculty, and other students. Armed with productivity tools for capturing text, or Web content, preparing papers and reviewing documents, students are ready to take on their busy day. With support of wireless networks, a lightweight size, and a long battery life, a mobile device is the ideal solution for students who work from the classroom, hallway, library, or lab for to cooperating, sharing information with friends, or synchronizing their documents with their home PC.

- **The shopping mall warrior**: a smart mall is a special scenario made feasible by the nomadic computing systems: costumers are provided with enhanced services while they are walking among the courts. The shops and the mall itself are part of a fixed infrastructure backbone, while customers with personal mobile devices move around, interacting freely with a variety of shops and retailers. The mall scenario sees the following *actors*: the mall management, shops, and customers. These are all network capable entities. To offer their customers a greater level of service, the mall provides a network access throughout the entire mall. The mall has different logical regions with different network coverage. Any customer carrying an appropriate mobile device may connect to them. Shops may communicate to costumers to send them commercial advertisements and promotions, the mall management may suggest walking paths which depend on the current shopping areas where customers are located, and costumers may send inquiries to the mall management (e.g. mall maps, bus schedules, product information, etc.) or make reservations at food courts.

1.2 Nomadic computing and middleware

1.2.1 Challenges in nomadic computing

Nomadicity refers to the system support needed to provide a rich set of computing and communications capabilities and services, in a transparent and convenient form, to the nomad moving from place to place. This new paradigm is manifesting itself as users travel to many different locations with laptops, personal digital assistants, cellular telephones, pagers, and so on [16]. This section discusses some of the challenging issues arisen int the context of nomadic computing systems. Such issues may be classified in two main categories: *mobility*-related issues, and *application*-related issues:

- **mobility-related issues**: these issues are mainly due to inherent characteristics of wireless networks and mobile devices [4, 5]. Weak connectivity or battery power constraints may lead users to experience short periods of service unavailability. Moreover, users may initiate a communication just before a travel from office to home. Even if the network connection is available both outdoor and indoor (by example, via GSM and Wi-Fi) switching from a technology to another may cause disconnections. Therefore temporary unavailability of counterparts is a rule rather than an exception. If traditional communication paradigms required both counterparts to be available during the interaction, mobile computing requires mechanisms to let users communicate in a loosely coupled fashion. Such issues may be stated as follows:

  - *disconnectedness*: changes in network access points due to user's movements as well as mobile device's power constraints may cause disconnections;
  - *variable connectivity*: either voluntary changes (one travels, for example) or unpredictable changes (a noisy wireless connection) may cause changes in bandwidth, latency, reliability, error rate and delay of the network link.
  - *processing power*: network algorithms to support wireless access are far more complex than for the wired case. For instance, the details of tracking a user while moving in a nomadic environment add complexity and require rules for handover, roaming, etc;
  - *heterogeneity*: some terminals will also be able to use different access technologies either simultaneously or one at a time.

  These are the usual concerns for any computer communication environment, but what makes them of special interest for us is that the values of these parameters change dramatically (and sometimes suddenly) as the nomad moves from location to location.

  Furthermore, the *pervasiveness* of mobility is another crucial factor which makes it very challenging. With pervasiveness we mean that mobility affects all the layers of the ISO/OSI protocols stack, from the physical to the application layer.

- **application-related issues**: these issues are mainly due to the fact that nomadic computing systems are opening new application scenarios. Next-generation mobile computing applications will need to rely on high-level abstractions and advanced services. For instance, meeting people and exchanging

information with them, receiving dynamic content or let the computation be location-aware are only some of the possible applications whose design and implementation should be simplified by built-in mechanisms provided by nomadic computing middleware. More precisely, such issues may be stated as follows:

- *programming abstractions*: the most successful middleware solutions are usually based on object oriented programming and method invocations. The invocations are based on strongly typed interfaces that provide both compile and run time error checking. They also hide many implementation details. However, due to the violation of synchronization assumptions, the mobility may impact on communication paradigms on which programming abstractions rely on.
- *mobile-enabled middleware services*: the presence of mobility requires, for instance, the environment to become aware of the presence and location of nomads, as well as, the need for the user to become aware of the changing environment. Besides, the mobility itself allows users to rendezvous easily. This may impact the applicability of traditional communication paradigms that have typically one-to-one multiplicity.
- *tools for software design and development*: middleware should support fast service development and deployment. Designers have to worry about mobility issues if they rely on traditional service platforms. This represents an unsustainable burden without tools which help developers in designing and developing mobile computing applications.
- *interoperability*: it is highly unlikely that there will be, in a near future, a

single dominant middleware platform which would be good enough for different devices and purposes. This is especially true in nomadic computing environments due to diversity of network technologies and devices. The increasing diversity of devices terminals, network elements, and application needs imply that different middleware solutions will be in use.

1.2.2 Non-functional requirements

There are also other important systems issues. For instance, a primary issues is the security, which involves privacy as well as authentication. Yet another one is the interoperability of different service discovery and delivery solutions. Such matters are especially difficult in a nomadic environment since the nomad often finds that his computing and communication devices are outside the careful security walls of his home organization, or that he/she cannot exploit computing resources since they do not *speak the same language* of his/her mobile devices. Although this thesis partially deals with interoperability issues, interesting readings may be found in [19, 20]. This thesis is primarily focused on the above presented issues.

1.2.3 Limitations of traditional middleware

When developing distributed applications, designers do not have to explicitly deal with problems related to distribution, such as heterogeneity, scalability, resource sharing and fault tolerance. Middleware developed upon network operating systems provides application designers with a higher level of abstraction, hiding the complexity introduced by distribution. Existing middleware technologies have been built trying to hide distribution as much as possible, so that the system appears as a single

integrated computing facility. In other words, distribution becomes transparent.

These technologies have been designed and are successfully used for traditional distributed systems. However, as stated in the previous subsection, nomadic computing systems exacerbate issues and constraints of traditional distributed systems and pose new challenges. It is clear that some of the requirements introduced by nomadic computing systems, cannot be fulfilled by these existing traditional middleware.

As far as mobility-related issues are concerned, low-level mechanisms to implement traditional communication models (i.e. RPC) assume a stable, high bandwidth and constant connection between components. Furthermore, synchronous one-to-one communication supported by object-oriented middleware systems requires a rendezvous between the client asking for a service, and the server delivering that service.

On the contrary, in mobile systems unreachability is not exceptional and the connection may be unstable. Moreover, it is quite likely that client and server hosts are not connected at the same time, because of voluntary disconnections (e.g., to save battery power) or forced disconnection (e.g., loss of network coverage).

Moreover, mobility introduces higher degrees of heterogeneity in processing power and computing facilities than traditional distributed systems. Traditional middleware assume their components to run on powerful (and homogeneous in characteristics) devices. This may no longer be possible in nomadic computing systems. Finally, mobile hosts might have to support different communication protocols, according to the wireless links they are exploiting and require. Traditional middleware usually rely on the TCP/IP stack which provides complete transparency to upper layers.

As far as the software design and development is concerned, traditional middleware offer a high-level computing model and a powerful programming interface, which

help developers to reduce the product time to market. Due to different technologies and requirements, several middleware may be of use in nomadic computing systems. Traditional middleware lack in standard and proved mechanisms to achieve interoperability in such systems.

Besides, they lack in adequate tools to deal with mobility issues, such as tools to redistribute and reconfigure applications components. They also lack in internal mechanisms to adapt to changes in the execution and communication capabilities, as well as, strategies to use available communication resources efficiently.

Furthermore, traditional middleware usually do not provide high-level tools to help the designer to focus only on application-logic requirements, for instance by means of visual tools. This would be of help since design next-generation mobile computing application is more challenging due to novelty of scenarios.

The programming abstractions are of concern as well. Traditional middleware provide a powerful and high expressive set of programming abstractions (i.e. remote method invocations). However, such abstraction are not flexible enough to meet the need of mobile computing developers. If the developer wants to exploit different communication paradigms (e.g. using the *pull-* instead of the *push-* model), he/she has to implement all the abstractions by himself/herself.

Finally, traditional middleware lack in providing advanced services for mobile computing applications. Such services may be classified into three groups. First there are services designed to overcome common restrictions of mobile computing, which arise mainly from the slowness and instability of wireless lines utilized by the mobile user. Examples are connection management, caching or encryption services.

The second group of services handles the management and administration of mobile users moving around and connecting their portables to networks at different places. These mobility management services include tasks such as accounting and device's positioning. The last group of services are needed to adapt existing applications to mobile settings.

1.3 Nomadic computing middleware

From insights stated in section 1.2, it is clear that a nomadic computing middleware should be designed in accordance with the following subset of functional requirements as well as should be dealing with the following subset systems constraints:

1.3.1 Functional requirements

The design of a nomadic computing middleware has to face *mobility*-related issues, which mostly affect the adopted communication paradigm, and the mobile management procedures. Basically, to deal with such issues, a middleware designer should aim to answer these questions:

- how to accomplish the communication even in presence of possible disconnections due to users' travels? Users may start a communication just before a travel from office to home. Even if the network connection is available both outdoor and indoor (by example, via GPS and Wi-Fi), there might ever be a change of disconnections due to intrinsic limits of these technologies. Moreover, the technology migration (by example, when the user gets inside a building after being outside) needs efficient procedures to preserve the communication during the hand off.

- how to let users communicate even in presence of the counterpart unavailability? Users may experience temporary unavailability of counterparts. In mobile computing environments this represents the rule rather than the exception. If traditional communication mechanisms required both counterparts to be available during the interaction, mobile computing environments requires communication mechanisms which allow users to communicate in a loosely coupled fashion.

1.3.2 Systems constraints

Any solution to such issues has to take into account the following *application*-related issues, which mostly affect the provided abstractions and advanced services:

- the need of a proper solution (in terms of computational requirements) due to the presence of different kinds of mobile devices. Users can carry computers and communication devices which may have very different characteristics in terms of both computational, memory and power resources. Therefore, any solution to the above mentioned communication issues should take advantage of such variety letting powerful elements of the NC infrastructure accomplish complex tasks, while letting embedded devices accomplish their duties with moderate effort.

- the need by developers to rely on a high-level and powerful abstraction in the design of next-generation mobile computing applications. New scenarios are now made possible thanks to the mobility. For instance, the mobility makes possible users to physically join a group of persons and share information such as documents, music tracks, and so on. Group communication is rather difficult

to implement via traditional distributed communication mechanisms although they are pretty easy to use. A solution which proposes new communication mechanisms to easily achieve these scenarios without loosing the strength of the traditional ones will be a useful tool in developers' hands.

Other requirements and constraints should be taken into account as well (e.g. to design fully comprehensive context-aware infrastructure, or fully reconfigurable reflective infrastructure), however the proposed requirements and constraints are those which any middleware for nomadic computing must face in order to be an effective instrument's toolbox of mobile computing developers.

1.4 Contribution

During the last decade a great research deal has been done to propose middleware solutions for nomadic computing systems. Much effort has been striven along two distinct directions: i) extending traditional middleware implementations with primitive mobile-enabled capabilities (e.g. Wireless-CORBA [12]), and ii) proposing middleware which adopt mobile-enabled computing models (e.g. Lime [13]).

While we recognize that these approaches have been leading to important results, both of them have drawbacks: the former adopts a more effective computing model [14], but does not effectively overcome the intrinsic limitation of the synchronous remote procedure call; the latter adopts decoupled interaction mechanisms, but fails in providing a high level and well understood computing model abstraction. We claim that a *unified* approach, both of adopting a powerful computing model and mobile-enabled communication mechanisms, should be adopted.

This thesis proposes the Esperanto Broker (EB), a communication platform for

nomadic computing systems which take advantages of both the above mentioned approaches. EB adopts the distributed objects computing model and enhances it to achieve the realization of new application scenarios. At the same time, it adopts mobile-enabled interaction paradigms as the underlying infrastructure which applications will rely on. More precisely, EB has the following features: i) it addresses mobility issues via a unified approach, i.e. both at data-link, network and middleware levels; ii) it adopts a tuple space as the underlying communication infrastructure; and iii) it provides Distributed Object Computing (i.e. DOC) model which is enhanced according to the communication paradigms standardized by the W3C [15].

1.5 Communication paradigms

This section is meant to provide some background terminology and classification of communication paradigms adopted by distributed communication middleware to better clarify motivations behind the proposed approach and relate the Esperanto Broker with the state of art in the area of nomadic computing.

1.5.1 Properties

Depending upon how entities interact, exchange data, and synchronize themselves, several communication paradigms may be distinguished. In order to classify such paradigms, [21, 22] identify the following dimensions:

- **Decoupling**: it represents the strength that keeps client and server either tightly or loosely coupled to each other. Decoupling may be in space, time and synchronization: i) *space*: the interacting parties do not need to know each other. The clients do not usually hold references to their servers, neither

do they know how many of these clients are participating in the interaction. Similarly, clients do not usually hold references to the servers, neither do they know how many of these servers are participating in the interaction; ii) *time*: the interacting parties do not need to be actively participating in the interaction at the same time. In particular, the server might provide some services while the client is disconnected, and conversely, the client might get notified about the occurrence of some event while the original service provider is disconnected; iii) *synchronization*: servers are not blocked while providing service, and clients can get asynchronously notified (through a callback) of the occurrence of the service provision while performing some concurrent activity. The service delivery does not happen in the main flow of control of the servers and clients, and do not therefore happen in a synchronous manner.

- **Initiative**: the interaction may begin on both sides of the service delivery, either at the server-side or the client-side. Initiative may be pull-based or push-based: i) *client-pull*: the transfer of information from servers to clients is initiated by a client request. The pull-based interaction may be either *two-way* or *one-way*, that is, the client may either wait for a reply from the server or not; ii) *push-based*: data delivery involves sending information to a client population in advance of any specific request. With push-based delivery, the server initiates the transfer. The push-based interaction may be either *solicit/response* or *notify*, namely, the server may either wait for a reply from the client or not.

- **Multiplicity**: service delivery may occur between two or more parties, whether it is based on one-to-one or one-to-many communication: i) *one-to-one* communication: service is provided by one server to one client; ii) *one-to-many*

communication: it allows multiple clients to receive the service sent by one the server. It is worth noting that the communication may be also many-to-many, where more than one server provides the service to more than one client.

1.5.2 Models

According to the above mentioned dimensions, the following fundamental communication paradigms may be distinguished:

- **Remote Procedure Call**: One of the most widely used forms of distributed interaction is the remote invocation, an extension of the notion of "operation invocation" to a distributed context. This type of interaction was first proposed in the form of a remote procedure call (RPC) for procedural languages, and has been straightforwardly applied to object-oriented contexts in the form of remote method invocations. By making remote interactions appear the same way as local interactions, the RPC model and its derivatives make distributed programming very easy. This explains their tremendous popularity in distributed computing. Distribution cannot, however, be made completely transparent to the application, because it gives rise to further types of potential failures (e.g., communication failures) that have to be dealt with explicitly. The synchronous nature of RPC introduces a strong time, synchronization, and also space coupling (since an invoking object holds a remote reference to each of its invokees). Several attempts have been made to remove synchronization coupling in remote and avoid blocking the caller thread while waiting for the reply of a remote invocation. A first variant consists in providing a special flavor of asynchronous invocation for remote methods that have no return values. For instance, CORBA

provides a special one-way modifier that can be used to specify such methods. Obviously the multiplicity allowed by the remote procedure call is one-to-one.

- **Tuple Space**: The tuple space (also known as distributed shared memory) paradigm provides hosts in a distributed system with the view of a common shared space across disjoint address spaces, in which synchronization and communication between participants take place through operations on shared data. The notion of tuple space was originally integrated at the language level in Linda [23], and provides a simple and powerful abstraction for accessing shared memory. A tuple space is composed of a collection of ordered tuples, equally accessible to all hosts of a distributed system. Communication between hosts takes place through the insertion/removal of tuples into/from the tuple space. Three main operations can be performed: write() to export a tuple into a tuple space, take() to import (and remove) a tuple from the tuple space, and read() to read (without removing) a tuple from the tuple space. The interaction model provides time and space decoupling, in that tuple producers and consumers remain anonymous with respect to each other. The creator of a tuple needs no knowledge about the future use of that tuple or its destination. An in-based interaction implements one-to-one semantics (only one consumer reads a given tuple) whereas read-based interaction can be used to implement one-to-many message delivery (a given tuple can be read by all such consumers). The tuple space paradigm proposed by Gelernter et al. does not provides synchronous decoupling because consumers pull new tuples from the space in a synchronous style (read() and take() are blocking primitives). However available implementations (e.g. JavaSpace [8] , TSpaces [24]), do provide non-blocking primitives

so that providing complete synchronous decoupling.

- **Message Passing**: Message oriented is often used to refer to a family of models rather than to a specific interaction scheme. Message queuing and publish/subscribe are tightly intertwined: message queuing systems usually integrate some form of publish/subscribe like interaction. Such message-centric approaches are often referred to as message-oriented middleware (MOM). At the interaction level, message queues recall much of tuple spaces: queues can be seen as global spaces, which are fed with messages from producers. From a functional point of view, message queuing systems additionally provide transactional, timing, and ordering guarantees not necessarily considered by tuple spaces. In message queuing systems, messages are concurrently pulled by consumers with one-of-n semantics. These interaction model is often also referred to as point-to-point (PTP) queuing. Which element is retrieved by a consumer is not defined by the element's structure, but by the order in which the elements are stored in the queue (generally first-in first-out (FIFO) or priority-based order). Similarly to tuple spaces, producers and consumers are decoupled in both time and space. As consumers synchronously pull messages, message queues do not provide synchronization decoupling. Some message queuing systems offer limited support for asynchronous message delivery, but these asynchronous mechanisms do not scale well to large populations of consumers because of the additional interactions needed to maintain transactional, timing, and ordering guarantees.

Table 1.1 summarizes the decoupling, initiative, and multiplicity properties of the afore-mentioned communication paradigms. The tuple space model is the most

Table 1.1: Communication paradigms and their attributes according to proposed dimensions

	RPC	MOM	PS	TS
decoupling	all	none	none	none
initiative	pull-based	pull-based	push-based	both
multeplicity	1-1	1-1	both	both

flexible communication model.

1.5.3 A standard approach

The WSDL [15] is an XML vocabulary for describing network services as a set of endpoints exchanging messages about each others capabilities. The messages may contain document-oriented (i.e. messages or tuples) or procedure-oriented information (i.e. method signatures).

A WSDL document defines services as collections of network endpoints (or *ports*). In WSDL, the abstract definition of endpoints and messages is separated from their concrete network deployment or data format bindings. This allows the reuse of abstract definitions: messages, which are abstract descriptions of the data being exchanged, and port types, which are abstract collections of operations. The concrete protocol and data format specifications for a particular *portType* constitute a reusable binding. A port is defined by associating a network address with a reusable binding. A collection of ports defines a service.

A *portType* is a named set of abstract operations and the abstract messages involved. The *portType* name attribute provides a unique name among all *portTypes* defined within the enclosing WSDL document. An operation is specified by the involved messages. WSDL defines four primitive types of operations that an endpoint can support:

- **One-way**: The endpoint receives a message. The data exchange only occurs in one direction. The service only gets an input, without having to provide an output. Hence, the *portType* only consists of an input message.

- **Request-Response**: The endpoint receives a message, processes it, and sends a correlated message to the requestor. This is the most common scenario, as known from HTML pages.

- **Solicit-Response**: The endpoint sends a message and receives a correlated message. This is the complementary operation type to Request-Response.

- **Notification**: The endpoint sends a message, but it does not expect a response. This type of operation is complementary to the One-way operation.

SOAP [15] is an XML-based protocol for messaging and remote procedure calls. At its core, a SOAP message has a very simple structure: an XML element (the *Envelope*) with two child elements, one of which contains the optional *Header* and the other the *Body*. For Web Services, SOAP offers basic communication, while WSDL does inform about what messages must be exchanged to successfully interact with a service.

It is noteworthy that both WSDL and SOAP are highly expressive and allow the user to specify whichever communication primitives and communication paradigms he needs.

Chapter 2

Nomadic computing platforms

Several works addressed the issue of supporting mobile computing applications since the end of '90s. Focusing on distributed computing middleware, research effort has been mainly progressed along the following directions: i) extending traditional middleware implementations with primitive mobile-enabled capabilities, and ii) proposing middleware which adopt mobile-enabled computing models.

The former aimed at preserving the distributed computing model and to let legacy applications run transparently in mobile computing settings, whereas the latter aimed at proposing suitable computing models for mobile computing environments. Solutions belonging to the first approach include [25, 26, 27, 28, 29, 30, 31]. Solutions belonging to the second approach include [11, 9, 10, 32, 13, 33, 34].

In the following, detailed description of these middleware solutions are provided. Finally, according to what stated in sections 1.2.1, 1.3.1, 1.3.2, a comparison framework is provided.

2.1 Extending traditional middleware

2.1.1 Wireless CORBA

The Telecommunications Domain Task Force (DTF) of the OMG issued a Request for Proposals (RFP) on *Wireless Access and Terminal Mobility in CORBA* [30], which requested a standardized solution to wireless and mobile communication in the CORBA framework. The overall system architecture is divided into three separate domains, each of which contains a central component: i) *Home Domain*: it is the mobile terminal's administrative home. This is presumably connected to the organization administering the terminal. The *Home Location Agent* in the Home Domain is responsible for tracking the location of each terminal owned by its domain. The Home Domain may also contain CORBA services that may be treated specially by mobile terminals; ii) *Terminal Domain*: it consists of everything on the mobile terminal. As is proper with a CORBA specification, the internal structure of a domain is not specified at all, only its outside interfaces. The outside interface of the Terminal Domain is the *Terminal Bridge*. All CORBA invocations whose one endpoint is on the mobile terminal go through the Terminal Bridge, which communicates using a specified protocol with its counterpart on the network side; iii) *Visited Domain*: the counterpart of the Terminal Bridge on the fixed network is called the *Access Bridge*. In the connection between the two Bridges the Access Bridge is the passive side, which is contacted by the Terminal Bridge. The domain containing the Access Bridge is called the *Visited Domain*.

When a CORBA object is on a mobile terminal, invocations intended for it need to be routed somehow to the terminal's current Access Bridge. This is accomplished

with Mobile IORs. Mobile IOR contains the address and port number of the mobile terminal's Home Location Agent, instead of the address and port number of the actual object on the terminal. When a Home Location Agent receives an invocation intended for an object on a mobile terminal under its administration, it reads the Mobile Terminal Profile contained in the target IOR of the invocation and send a `LOCATION_FORWARD` response to redirect the invocation to the current Access Bridge of the terminal identified in the Mobile Terminal Profile. The Access Bridge can determine the target terminal in the same way as the Home Location Agent and tunnel the invocation to it. If the terminal has already left the Access Bridge, it can either respond with a `LOCATION_FORWARD` (if it knows the terminal's current location) or `OBJECT_NOT_EXIST` (which should cause the client to retry the invocation at the Home Location Agent, if such existed).

The Bridges communicate with each other using the *GIOP Tunneling Protocol* (i.e. GTP). The GIOP connection logically is directly between the object on the terminal and the object on the fixed network. The Access Bridge is responsible for translating between the IIOP used by the network object and the GTP used by the terminal. GTP messages over the wireless link are transferred by the adaptation layer that guarantees reliability and ordered delivery of messages. The actual transport layer can be any transport used over wireless links; if the transport layer does not provide sufficient reliability, it is the responsibility of the adaptation layer to provide this on top of it.

When a mobile terminal migrates from a domain to another, handoff procedure are needed to be performed. There are backward handoff, where an existing connection is switched to go through a new access point, and forward handoff, where a lost

connection is re-established after a sudden loss. When a terminal performs a handoff, its old Access Bridge and its Home Location Agent need to know about terminal location. For this purpose the Wireless CORBA specification defines Mobility Events that may be generated whenever the terminal moves. Both the Terminal Bridge and Access Bridge generate these events when a terminal performs a handoff or loses its current connection. These mobility events are specified to pass through a Notification Channel, though there is no specified procedure for outside applications to discover this channel. Moreover, the Wireless CORBA specification does not describe how such procedures are triggered.

2.1.2 Dolmen

Dolmen project [31] has the objective to study the impact of terminal mobility on client server interaction mechanisms over a wireless access and how this can be supported by a CORBA compliant environment. The mechanism used by a client to invoke an operation offered by a server comprises two steps: i) retrieval of a reference to an instance of the interface that gives access to the desired operation; ii) invocation of the operation across the interface, using the obtained reference, provided that a valid reference has been obtained. In that respect, terminal mobility implicitly means frequent changes of object references; in particular those of nomadic objects because the mobile terminal and the objects contained therein are continuously on the move. Dolmen proposes a solution in which the impact of terminal mobility on the basic client server interaction mechanism is tackled by using CORBA bridging techniques.

In the CORBA architecture, bridging is one of the cornerstones of building interoperability support between different communication environments. A common use

for an interoperability bridge is to act as a gateway between a CORBA domain and a non-CORBA environment, and translate between IIOP and a particular ESIOP designed for that environment. More precisely, a bridge resides between two domains and translates each message related to an object invocation across the domain border into a format understood by the destination domain.

The concept of bridging is exploited to interconnect mobile terminals to the fixed network. Implementing two half-bridges, one residing in a mobile terminal and the other in a well-known access point within each mobility domain in the fixed network, allows to introduce an efficient light-weight Inter-ORB protocol for use over the wireless access network. The wireless access domain and part of the core network domain is divided into mobility domains. The core network part of each mobility domain instantiates a set of mobile-specific support services, including one or more Fixed DPE Bridges (FDBRs) that serve as access points to the fixed network. The rest of the core network domain serves fixed terminals and acts as a backbone network. Each mobile terminal has its own ORB that provides object services to the applications running on the terminal.

Invocations of objects outside the local access domain are directed to the Mobile DPE Bridge (MDBR) on the mobile terminal. The MDBR forwards the invocation to the FDBR, which then invokes the desired object. The FDBR acts as the representative of the mobile terminal within the fixed network, invoking operations in other objects on behalf of the mobile terminal. The FDBR also accepts invocation requests for objects located on the mobile terminal from objects within the core network. The FDBR forwards an invocation request to the MDBR, which then invokes the actual object and returns the response through the FDBR

When a mobile terminal is in contact with the core network, a physical signaling connection (a dedicated signaling channel) exists between the two bridges. When the terminal moves to another mobility domain, this signaling connection must be released, a new FDBR within the new domain must be contacted, and a new signaling connection must be created. This procedure is referred to as a bridge handoff.

Object invocations that are in progress during a bridge handoff are reliably and correctly completed despite the momentary break in connectivity via buffering invocation related messages in the old FDBR until the mobile terminal has successfully connected to a new FDBR. The bridges must also perform recovery after an unexpected loss and subsequent re-establishment of the signaling connection.

The LWIOP protocol provides the means for such a recovery: each LWIOP message must be acknowledged by the receiver before it can be discarded by the sender. In the event of a communication error, any unacknowledged messages can be re-sent after the communication channel has been re-established. Recovery from a loss of signaling connection also often entails a bridge handover, since the new connection may be established through a different base station.

2.1.3 Alice

Alice [26] allows CORBA applications running on mobile devices to communicate transparently with standard CORBA applications using IIOP. The architecture allows server as well as client objects to reside on mobile hosts without relying on a centralised location register to keep track of their whereabouts. IIOP clients and servers residing on mobile hosts are able to interact with IIOP servers and clients on the wired network using standard IPv4 and without requiring the wired clients and

servers to know that they are interacting with clients and servers on a mobile host. In particular, no support for Mobile IP is required.

To address mobility issues, Alice uses a session layer type approach in conjunction with application support without relying on other approaches. Mobile hosts are connected to mobility gateways via wireless links. The mobility gateway has several roles, one of which is to act as a proxy for a mobile host, relaying incoming and outgoing communications over wired connections as shown with the solid lines. Another role is to perform address translation and redirection for the higher layers.

A mobile host can change mobility gateway as it moves, causing a handoff from the old to the new mobility gateway. This involves transferring state information from the old to the new mobility gateway and tunneling open connections for the remainder of their lifetime. The Alice's architecture consists of three layers: i) the Mobility Layer (ML) provides mobility support that is independent of both CORBA and IIOP and that can also be used to support other protocols such as HTTP; ii) the IIOP Layer implements the IIOP protocol independently of mobility; iii) the S/IIOP layer, used when both client and server objects are to be hosted on mobile devices. The Swizzling or S/IIOP Layer provides the IIOP support that is required specifically in mobile environments where server objects are to be hosted on mobile devices.

The ML plays several roles in the architecture. First, it hides broken TCP connections from the layer above it by performing transparent reconnection attempts. In an IIOP context, this assures at-most-once invocation semantics even in the presence of broken wireless connections. Second, the ML on the mobile host lets the layer above it allocate TCP/IP ports on the mobility gateway for incoming connection attempts. This is necessary to allow clients on the wired network to create TCP connections to

the mobile device. Such connection attempts are sent to the mobility gateway which creates corresponding logical connections to the mobile device. Third, it performs handoff between mobility gateways, in case the mobile host moves from one gateway to another. Finally, it can (optionally) notify higher layers about the current network connection point. In particular, this information is used by the S/IIOP layer to perform the object reference translation described below.

The S/IIOP layer is the mobility-aware component of the IIOP implementation and is used in tandem with the IIOP layer to support server objects on the mobile host. The S/IIOP layer is used by the IIOP layer to perform operations which are affected by mobility, especially publication and encoding of object references. In CORBA, each server object has its own object refer reference, called an Interoperable Object Reference (IOR), that uniquely identifies and locates the object. At least one (hostname, port number) pair is part of the IOR. When an IOR is created on a mobile host, the (hostname, port number) pair of the mobile host is replaced by that of the mobility gateway. Such an IOR is said to be swizzled. S/IIOP on the mobile host uses the underlying ML to obtain information about the current network connection point in order to perform this swizzling of IORs.

This allows a client on the fixed network to contact the mobility gateway instead of the mobile host. S/IIOP on the mobility gateway is in turn configured to forward incoming requests to the server object on the mobile host. S/IIOP exports a traditional sockets-like interface to the layer above as well as operations to create and destroy object references. The IIOP layer allows the layer above it to communicate with other CORBA applications, such as those supported by CORBA 2.0 compliant

ORBs or other IIOP implementations. The implementation expects a standard sockets interface from the layer below and can be supported directly above TCP/IP, the ML or S/IIOP layer as required.

2.1.4 The proxy platform Π^2

The proxy platform Π^2 [27] is based on special equipment on the borders between wireless and wired domains, where proxies act on behalf of the mobile users. Such proxies help to reduce the communication requirements for the wireless link and, therefore, integrate mobile users into distributed applications. They bridge the protocols used in the wired domain and in the wireless domain, hence dealing with address and format translation. Furthermore, these proxies may be enhanced by components allowing value-added services to support context- and location-aware applications.

Basically, the platform defines a sort of specially tailored Environment-Specific Inter-ORB Protocol ESIOP to cope with the difficulties on the wireless link. Furthermore, to handle legacy applications, special gateways are designed bridging this ESIOP and IIOP. Gateways also deal with addressing and filtering aspects.

To transfer the proxy concept onto CORBA, Π^2 uses a protocol proxy to overcome the shortcomings of TCP along with filtering and compressing data strategies to improve the transmission quality. Content-specific proxy may apply context- and location-specific information to achieve value-added services. The current location of the mobile user, the user's profile and information on the actual situation the user is in may be considered when relaying data to the mobile end system.

The proxy platform Π^2 acts as a mediator between a CORBA client and a CORBA server. It works similarly to a generic request level bridge, but allows a more common

solution for problems often occurring in mobile and heterogeneous environments and the integration of further functionality to provide additional services.

To provide the desired functionality the proxy platform Π^2 modifies the implementation of the ORB. The client ORB and the proxy ORB are changed in the same way. These modifications imply that both client and server applications and the server ORB do not have to be changed. On call processing, it is necessary to forward all requests from the client to the appropriate proxy. This is done by setting up a "'tunnel"' between the client and a proxy and between the proxy and the server. In a scenario with several proxies between the client and the server the proxies are also connected via "tunnels".

Transport connection set-up is done in the ORB with the information found in the object reference. Because the object reference of the server object is changed to the object reference of the proxy the ORB creates a transport connection to the proxy and sends the request to the proxy. To enable forwarding of CORBA requests on the proxy it is necessary to get type information of the current invocation. Such information is used to analyze the request for further processing like filtering, caching of parameter values, etc.

Π^2 can be integrated in an existing distributed system in several ways. The integration is determined by the number of installed proxies, the location of the proxies and the functionality of each proxy. The integration has implications on manageability, the round-trip time or latency and the functionality of the system. In the simplest integration scenario, one proxy is located on the access node of the wireless network. No other proxy is used in the system.

A more useful integration scenario supplies an additional proxy located on the

mobile node. This approach gains much more flexibility and transparency. The protocol that is used on the wireless link can be changed transparently to the client application or client ORB. Connection disruption and reconnection can be hidden to the client and the server and can be handled by the proxies on the endpoints of the wireless link. The proxy on the mobile node also allows transparent integration of additional services on the client.

2.2 Mobile-enabled middleware proposals

2.2.1 Rover

The Rover Toolkit [34] offers applications a distributed object system based on a client-server architecture. Clients are Rover applications that typically run on mobile hosts, but can also run on stationary hosts as well. Servers, which may be replicated, typically run on stationary hosts and hold the long-term state of the system. Communication between clients is limited to peer-to-peer interactions within a mobile host (using the local object cache for sharing) and mobile hosts server interactions; there is no support for peer-to-peer, mobile host to mobile host interactions.

The Rover toolkit provides mobile communication support based on two ideas: relocatable dynamic object (RDO) and queued remote procedure call (QRPC). A relocatable dynamic object is an object (code and data) with a well-defined interface that can be dynamically loaded into a client computer from a server computer, or vice versa, to reduce client-server communication requirements. Queued remote procedure call is a communication system that permits applications to continue to make non blocking remote procedure calls even when a host is disconnected- requests and

responses are exchanged upon network reconnection. Rover gives applications control over the location where the computation will be performed. In an intermittently connected environment, the network often separates an application from the data upon which it is dependent.

By moving RDOs across the network, applications can move data and/or computation from the client to the server and vice versa. Use of RDOs allows mobile-aware applications to migrate functionality dynamically to either side of a slow network connection to minimize the amount of data communicated across the network. Caching RDOs reduces latency and bandwidth consumption. Interface functionality can run at full speed on a mobile host while large data manipulations may be performed on the well-connected server. All application code and all application-touched data are written as RDOs. RDOs may execute at either the client or the server. Each RDO has a "home" server that maintains the primary, canonical copy. Clients import secondary copies of RDOs into their local caches and export tentatively updated RDOs back to their home servers.

RDOs may vary in complexity from simple calendar items with a small set of operations to modules that encapsulate a significant part of an application (e.g., the graphical user interface for an email browser). Complex RDOs may create a thread of control when they are imported. with the object cache. When a client side application issues an import or export operation, the Toolkit satisfies the request depending on whether the object is found in a local cache and on the consistency option specified for the object. Once an object has been imported into the client-side application's local address space, method invocations without side effects are serviced locally by the object. At the application's discretion, method invocations with side effects may

also be processed locally, inserting tentative data into the object cache.

Operations with side effects also insert a QRPC into a stable operation log located at the client. Each insert is a synchronous action. Support for intermittent network connectivity is accomplished by allowing the log to be incrementally flushed back to the server. Thus, as network connectivity comes and goes, the client will make progress towards reaching a consistent state. The network scheduler contributes to log transmission optimization by grouping operations destined to the same server for transmission and selecting the appropriate transport protocol and medium over which to send them. Rover is capable of using a variety of network transports. Rover supports both connection- based protocols (e.g., HTTP over TCP/IP networks) and connection-less protocols (e.g., SMTP over IP or non-IP networks). The network scheduler leverages the queuing of QRPCs performed.

2.2.2 Xmiddle

Xmiddle [10] allows mobile to communicate and sharing information with other hosts. Mobile peers may come and go, allowing complicated ad-hoc network configurations. In order to allow mobile devices to store their data in a structured and useful way, each device stores its data in a tree structure, a sort of expressive tuple representation. Trees allow sophisticated manipulations due to the different node levels, hierarchy among the nodes, and the relationships among the different elements which could be defined. Xmiddle defines a set of primitives for tree manipulation, which applications can use to access and modify the data, basically a set of primitives to access a tuple space.

Xmiddle provides an approach to sharing that allows on-line collaboration, off-line data manipulation, synchronization and application dependent reconciliation. On each device, a set of possible access points for the private tree are defined so that other devices can link to these points to gain access to this information; essentially, the access points address branches of trees that can be modified and read by peers. In order to share data, a host needs to explicitly link to another host's tree. Access points to a host's tree are a set that are called *ExportLink*. Xmiddle allows mobile hosts to share data when they are connected or replicate the data and perform operations on them off-line; reconciliation of data takes place once the hosts reconnect.

A host also records the branches that it links from other remote hosts in the set *LinkedFrom*, and the hosts linking to branches of the owned tree in the set *LinkedBy*. These sets contain lists of tuples (host; branch) that define the host that is linking to a branch, and from whom a branch is linked, respectively. When two hosts are connected they can share and modify the information on each other's linked data trees. Each host has full control over its own tree, however it is obliged to notify other connected hosts that link to the modified part (branch) of its tree about the changes introduced.

Hosts may explicitly disconnect from other hosts using the disconnect primitive. Xmiddle supports explicit disconnection to enable, for instance, a host to save battery power, to perform changes in isolation from other hosts and to not receive updates that other hosts broadcast. Disconnection may also occur due to movement of a host into an out of reach area, or to a fault. In both cases, the disconnected host retains replicas of the last version of the trees it was sharing with other hosts while connected and continues to be able to access and modify the data; a versioning system is in place

to allow consistent sharing and data reconciliation.

Xmiddle implements its services on top of standard network protocols, such as UDP or TCP, that are provided in mobile networks on top of, for instance, a Bluetooth data-link layer, and MAC and physical layer. The current prototype is however based on UDP upon Wireless Lan, which is an other possible option. The protocol stack consists of the following layers: i) the presentation layer implementation maps XML documents to DOM trees and provides the mobile application layer with the primitives to link, unlink and manipulate its own DOM tree, as well as replicas of remote trees; ii) the session layer implementation manages connection and disconnection.

These two layers consist of a Xmiddle Controller, which is a concurrent thread that communicates with the underlying network protocol and handles new connections and disconnections, triggers the reconciliation procedures and handles reconciliation conflicts according to application specific policies. The Xmiddle Primitives API provides mobile applications with operations implementing the XMIDDLE primitives, such as link, unlink, connect and disconnect. The ability to link to trees from other devices introduces a client/server dependency between mobile hosts.

2.2.3 Lime

Lime [13] is a middleware supporting the development of applications that exhibit physical mobility of hosts, logical mobility of agents, or both. LIME adopts a coordination perspective inspired by the Linda model. The context for computation, represented in Linda by a globally accessible, persistent tuple space, is represented in LIME by transient sharing of the tuple spaces carried by each individual mobile unit.

In Linda, processes communicate through a shared tuple space that acts as a repository of elementary data structures, or tuples. A tuple space is a multiset of tuples that can be accessed concurrently by several processes. Each tuple is a sequence of typed parameters, and contains the actual information being communicated.

Tuples are added to a tuple space by performing *out* operation, and can be removed by executing a *in* operation. Tuples are anonymous, thus their selection takes place through pattern matching on the tuple content. The argument is often called a template, and its fields contain either actuals or formals. Actuals are values, formals act like "wild cards", and are matched against actuals when selecting a tuple from the tuple space. If multiple tuples match a template, the one returned by in is selected nondeterministically. Tuples can also be read from the tuple space using the *rd* operation. Both *in* and *rd* are blocking, i.e., the process performing the operation blocks until a matching tuple is found in the tuple space. A typical extension to this synchronous model is the provision of a pair of asynchronous primitives *inp* and *rdp*, called probes, that allow non-blocking access to the tuple space.

Linda characteristics resonate with the mobile setting. In particular, communication in Linda is decoupled in time and space, i.e., senders and receivers do not need to be available at the same time, and mutual knowledge of their location is not necessary for data exchange. The global context for operations is defined by the transient community of mobile units that are currently present. Since these communities are dynamically changing according to connectivity and migration, the context changes as well.

In the model underlying LIME, the shift from a fixed context to a dynamically changing one is accomplished by breaking up the Linda tuple space into many tuple

spaces, each permanently associated to a mobile unit, and by introducing rules for transient sharing of the individual tuple spaces based on connectivity. From the perspective of a mobile unit, the only way to access the global context is through a so-called interface tuple space (ITS), which is permanently and exclusively attached to the unit itself. The ITS contains tuples the mobile unit is willing to make available to other units, and that are concretely co-located with the unit itself. This represents the only context accessible to the unit when it is alone. Access to the ITS takes place using the Linda primitives already mentioned, whose semantics is basically unaffected.

Nevertheless, this tuple space is also transiently shared with the ITSs belonging to the mobile units that are currently part of the community. Hence, the content perceived through the ITS changes dynamically in response to changes in the set of co-located mobile units. Upon arrival of a new mobile unit, tuples in the ITS of the new unit are merged with those, already shared, belonging to the other mobile units, and the result is made accessible through the ITS of each of the units. This sequence of operations, called engagement, is performed as a single atomic operation. Similar considerations hold for the departure of a mobile unit, resulting in the disengagement of the corresponding tuple space and the removal of data perceived by the remaining units through their ITSs.

Transient sharing of the ITS constitutes a very powerful abstraction, as it provides a mobile unit with the illusion of a local tuple space that contains all the tuples coming from all the units belonging to the community, without any need to know them explicitly. In an ad hoc network, LIME mobile hosts are connected when distance between them allows communication.

2.2.4 L^{2}imbo

L^{2}imbo [32] is based on the Linda model but includes a number of significant extensions which address the specific requirements necessary for operation in mobile environments. In particular, the system incorporates the following key extensions: i) multiple tuple spaces which may be specialized to meet application level requirements, e.g. for consistency, security or performance; ii) an explicit tuple type hierarchy with support for dynamic sub-typing; iii) tuples with QoS attributes including delivery deadlines; iv) a number of system agents that provide services for QoS monitoring, the creation of new tuple spaces and the propagation of tuples between tuple spaces.

In addition to general purpose tuple spaces L^{2}imbo allows the creation of tuple spaces with support of non-functional requirements, such as security (user authentication), persistence and tuple logging (for accountability in safety critical systems). Crucially, it is also possible to create a range of QoS-aware tuple spaces. In order to create a new tuple space clients communicate with the appropriate system agents via a universal tuple space (UTS). Clients specify the characteristics of the desired tuple space and place it into a common tuple space. The appropriate system agent accesses this tuple, creates a tuple space with the required characteristics and then places it into the common tuple space.

The fields in this tuple denote the actual characteristics of the new tuple space (which may be different to those requested in best-effort systems) and a handle through which clients can access the new space. Clients can make use of the new tuple space by means of a use primitive which provides access to a previously created tuple space. This primitive communicates with a membership agent through the universal tuple space and returns a handle if the tuple space exists and certain

other criteria are met. The precise criteria vary from tuple space to tuple space and can include checks on authentication and access control functions or relevant QoS management functions.

At a later time, handles can be discarded by an agent using a discard primitive. An appropriate tuple is then placed in the universal tuple space so that the membership agent can take appropriate steps. Tuple spaces are destroyed by placing a tuple of type terminate into the tuple space. These tuples are picked up by system agents within the tuple spaces themselves and invoke a system function to gracefully shut-down the tuple space.

This model can be applied recursively. It is possible to access a tuple space through the universal tuple space and then find that this tuple space has system agents supporting the creation and subsequent access to tuple spaces. This recursive structure provides a means of creating private worlds offering finer grain access control. Every site in L^2imbo has an associated local management tuple space together with a number of QoS monitoring agents. These monitoring agents monitor key aspects of the system and inject tuples representing the current state of that part of the system into the management tuple space. Some typical forms of QoS monitoring agent are: i) connectivity monitors, which watch over the characteristics of the underlying communications infrastructure and make available information such as the current throughput between hosts; ii) power monitors, which review the availability and consumption of power on a particular host. In particular, applications can obtain power information on host peripherals and may utilize hardware power saving functionality as appropriate; iii) cost monitors, which determine the cost associated with the current communications links between hosts.

2.3 Comparison framework

Figure 2.1 summarizes the comparison among related work in the area of nomadic computing middleware. As the Figure shows none of the proposed solutions adopts a pervasive approach in dealing with mobility-related issues. Mobility affects all the layers of the ISO/OSI protocols stack, thus requiring integrated mechanisms to be adopted at all layers, or in other words, a *cross-layer* approach. This means that mechanisms and solutions to be provided by a nomadic computing middleware need to be tightly designed in order to be an effective response to mobility challenges [35]. All the analyzed solutions rely on services offered by the underlying layers, namely, network and transport layers, such as *MobileIP*, TCP and/or UDP, thus adopting the classical *layered* approach.

To better clarifies motivations behind the work of this thesis, drawbacks, and weaknesses of the considered solutions, every dimension of the framework reported in Figure 2.1 is carefully considered:

- *disconnectedness*: every solution tries to deal with disconnectedness proposing sorts of decoupled interaction mechanisms. For instance, Wireless CORBA introduces Terminal Bridges, Access Bridges and Home Location Agents, and let them communicate with each other via a tunneling protocol. This helps to make parties *space-decoupled*, however, remote methods invocations they use to interact are still synchronously tight, since their implementation relies on GTP which is a connection-oriented protocol. Dolmen, Alice, Π^2, adopt similar mechanisms. Xmiddle, Lime and L^2imbo adopt a decoupled communication paradigm, however, Xmiddle, Lime do not rely on another entity to let the communication be space decoupled. The same issue affects Rover.

Dealing with disconnectedness means that middleware should adopt a fully decoupled communication paradigm, which allows both the middleware to handle device disconnections and reconnections and the application to go further in the computation even if the counterpart is not available.

- *variable connectivity*: Wireless CORBA, XMiddle and Lime aside, every solution provides mechanisms to deal with unpredictability of wireless connections. Dolmen, Alice and Π^2, L^2imbo provide smart proxies that embed strategies to cope with bandwidth variability. Rover introduces the concept of Relocatable Dynamic Objects (i.e. RDOs) to be downloaded from an entity to another in order to cope with bandwidth drops and disconnection. Although this may be an effective mean, is useless without any sort of network status prediction and adaptation.

  Dealing with variable connectivity means that middleware should adopt strategies that are aware or the connectivity status and let its behavior change during its operational phase to improve the efficiency in its usage as well as the availability perceived by the application. Such strategies should not introduce high overhead and deal with power constraints of mobile devices.

- *processing power*: almost none of the analyzed solutions deal with processing power constraints of mobile devices. Rover and L^2imbo provide respectively RDOs and Monitoring Agents. RDOs are used to shift a computation on a more powerful node while Monitoring Agents are used to know the status of the mobile device's battery. None of the considered solutions deploy middleware component according to power constraints, but let the application developer do

it.

Dealing with processing power means that middleware should be deployed according mobile device's power constraints. Heavy computational tasks such as handoff procedures, data storage, synchronization etc. should be left to middleware component that run of powerful hosts of the nomadic computing infrastructure.

- *heterogeneity*: heterogeneity is strictly related with the pervasiveness of the proposed approach. Most of the considered solutions adopt a *transparent* approach where the heterogeneity is hided to the middleware by means of underlying standard network protocols stack. Although, this is an effective approach to adopt, none of the analyzed solutions provide mechanisms to switch seamlessly from a network technology to another, and keeping the application developer from the low-level details.

  Dealing with heterogeneity means that middleware should implement mechanisms to exploit the flexibility offered by mobile devices, which mostly are equipped with more than one wireless interface. This may impact on the need of handoff procedures when a device has to switch either from a technology to another or from a wireless access point to another.

- *programming abstractions*: although all the considered solutions provide the developer with an object-oriented application programming interface, only Wireless CORBA, Dolmen, Alice, Π^2 and Rover adopt a Distributed Object Computing (i.e. DOC) model, while XMiddle, Lime and L^{2}imbo adopt a Tuple Space model. Remote method invocations are far more successful than tuple

space *write/read/take* primitives since they are easier to understand and more effective to apply. However, space-, time-, and synchrony- decoupling of tuple space make them suitable to use in nomadic computing environments.

Dealing with programming abstractions means that middleware should provide a powerful programming abstraction, directly related to a powerful communication model, which is also suitable to be applied in the context of nomadic computing environments.

- *advanced middleware services*: while Wireless Corba, Dolmen, Alice, Xmiddle and Lime do not provide any advanced middleware service, Π^2, XMiddle, and Lime can be distinguished since they provide some sorts of built-in mechanisms to respectively let the computation be location-aware, the data sharing be proximity-aware and the application be physical mobile. Rover and L^2imbo, instead, provide explicit services to adapt the behavior of the application to the surrounding context.

  Dealing with advanced middleware services means that middleware should provide building blocks and services to make nomadic computing scenario easy to be realized, without much effort from the designer.

- *tools for design and development*: any of the considered platforms provides advanced tools for supporting design and development of nomadic computing application, such as visual tools, code generators etc.

  Dealing with such issue means that middleware should provide such tools in order to let the designer focus on the real needs of the application and leave the middleware do the rest of the job.

- *interoperability*: basically, any of the considered platforms care about interoperability. Only CORBA-based solutions may benefit of some sort of interoperability due to OMG effort. However, middleware are supposed to be software systems that make distributed applications interoperable, this is one of their major goals. This is especially true in nomadic computing environments, where the diversity is the rules rather than the exception.

  Dealing with this issue means that middleware should provide mechanisms to be interoperable with other middleware. This could be achieved via the implementation of technologies and middleware bridges. However, the approach should be scalable with regard to the number of middleware solutions that will be proposed. It is not feasible to make middleware interoperable via a multitude of bridges.

Finally, Wireless Corba, Dolmen, Alice and Π^2 all share a crucial drawback: the adoption of the *rpc* mechanism: *rpc* is inadequate in mobile computing environments due to tightly coupling in space, time and synchronization [11]. Xmiddle, L^2imbo and Lime propose different computing models according to a tuple oriented approach.

Although such approaches provide mobile computing applications with time, space and synchronism decoupling, they are poorly structured and typed. DOC middleware have been successful in promoting high quality and reusable distributed software [14], and providing applications with such computing models is a step backwards.

Rover provides a distributed object model while adopting a sort of decoupled communication paradigm, but it has some weaknesses; firstly, authors are concerned about the burden of implementing application-specific adaptation strategies defining methods to update objects, to detect and to resolve conflicts. Secondly, Rover

does not provide any mobility management in terms of both handoff procedures and decoupled mechanisms.

Rover does not deal with device movements, which may lead to inconsistency of network-level connections, and it does not let client and server interact via an intermediate counterpart. This is the basic assumption to provide time and space decoupling.

PLATFORMS	pervasiveness	SOLUTIONS							
		mobility-related				*application-related*			
		disconnectedness	**variable connectivity**	**processing power**	**heterogeneity**	**programming abstractions**	**advanced middleware services**	**tools for design and development**	**interoperability**
WIRELESS CORBA	U	NPWT+RMI	U	U	PD-GTP	DOC	N	U	OMG
DOLMEN	U	NPWT+RMI	RASM	U	NAL	DOC	N	N	OMG
ALICE	U	NPWT+RMI	SP	U	U	DOC	N	N	OMG
P\	U	NPWT+RMI	FMA	U	PD-GTP	DOC	LAP	N	OMG
ROVER	U	QRPC	RDOs	RDOs	NAL	DOC	CCMN	N	N
XMIDDLE	U	DTS-SYN	N	U	U	TS	N	N	N
LIME	U	DTS-ASY	N	U	U	TS	N	N	N
L<IMBO	U	QOS-TS	FMA	FMA	NAL	TS	QMAA	N	N
		efficient mobility management				rich computing model and API			standard interface
		decoupled communication paradigm		technology transparence			easy to use in NC		
		REQUIREMENTS							

U	=	Unspecified, left either to the specific implementation, or to particular extensions, or yet to underlying layers
N	=	None
NPWT+RMI	=	Normal proxy approach with tunnelling, the interaction mechanism is still remote method invocation
QRPC	=	Queued Remote Procedure Call
DTS-SYN	=	Distributed Tuple Space, synchronous access primitives
DTS-ASY	=	Distributed Tuple Space, asynchronous access primitives
QOS-TS	=	Possibility to specify Tuple space with QoS attributes
SP	=	Adoption of smart proxies to handle variability of connection
RDOs	=	relocatable dynamic objects
FMA	=	Filter Manager or Agents to manipulate requests and responses or data structures in general
RASM	=	Resource Adapters in Service Machine approach
PD-GTP	=	Design of a new protocol, particularly GIOP Tunnelling Protocol, and rely on underlying network stack
NAL	=	Network Abstraction Layer
TS	=	Tuple space computing programming
DOC	=	Traditional Distributed Object Computing programming
LAS	=	Proxies may be aware of mobile user's locations
CPMN	=	Objects Caching, Prefetching, Migration, and Notification
QMAA	=	QoS Monitoring and Adaptation
OMG	=	Interoperability between ORBs and other middleware, if the specification is implemented

Figure 2.1: Comparison framework among nomadic computing solutions

Chapter 3

The Esperanto Broker

This chapter describes assumptions and terminology used in this thesis. It also provides an overview of the proposed platform and how the above mentioned issues have been addressed. The Figure 3.1 illustrates the architectural view of the Esperanto Broker.

3.1 Assumptions and Definitions

Wireless access points of Nomadic Computing infrastructures may be clustered based on several criteria (such as their geographic location or the ownership). In the following sections we refer to such clusters as *Domains*, and to the permanent infrastructure interconnecting these Domains as the *Core Network*. The Core Network can be characterized by a certain level of performance. Wired networks parameters (i.e. bandwidth, latency, and transmission reliability) are an order of magnitude higher than the parameters of wireless networks. Permanent hosts are more powerful than mobile devices. These considerations let the use of the Core Network to provide mobile devices with a support for mobility management and/or middleware services.

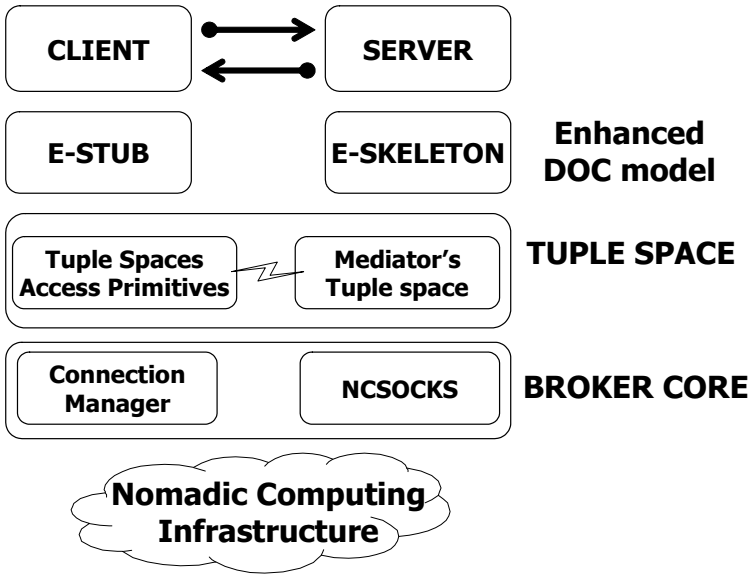

Figure 3.1: The Esperanto Broker: architectural overview

3.2 Dealing with nomadic computing challenges

In the following we present how the Esperanto Broker deals with challenges highlighted in section 1.2.1.

- **mobility-related issues**:
  - *disconnectedness*: changes in network access points due to user's movements as well as mobile device's power constraints may cause device's disconnections, which prevent application objects from communicating with their counterparts. To improve the availability of device connectivity the Esperanto Broker provides integrated mechanisms at any layer of the software stack to let devices handoff between two adjacent wireless access points and two wireless domains. To achieve *disconnected* interactions,

the Esperanto Broker provides applications with a decoupled communication paradigm (i.e. tuple space).

- *variable connectivity*: either voluntary or unpredictable changes may cause variations in bandwidth, latency, reliability, error rate and delay of the network link. The Esperanto Broker adopts internal communication primitives (i.e. NCSOCKS) whose implementation strategies address dinamicity of network link, providing applications and middleware components with awareness of network performances.
- *processing power*: mobility management procedures like handoffs, as well as mobility-aware strategies to deal with variable connectivity may be computational intensive. The Esperanto Broker takes into account resource limitations of mobile devices and adopts an approach where *high-computational* middleware components run on the fixed-side of the nomadic computing infrastructure, where permanent and powerful nodes are located.
- *heterogeneity*: some terminals are able to use different access technologies either simultaneously or one at a time. The Esperanto Broker allows both vertical and horizontal data-link layer handoff. The former are handoffs among access points of the same technology, while the latter are handoffs among access points of different technologies. This to achieve a higher availability of device connectivity.

- **application-related issues**:
  - *programming abstractions*: the Esperanto Broker provides an object-oriented

Application Program Interface (i.e. API) as well as a computation model which is coherent with the remote method invocations. It has also a plethora of features that makes it suitable for mobile computing settings. More precisely, the Esperanto Broker i) decouples method invocations interposing a tuple space between clients and servers; ii) introduces the way to specify both *client-initiated* and *server-initiated* remote method invocations; iii) provides one-to-many remote methods invocations.

- *mobile-enabled services*: several mobile-enabled services and mechanisms should be provided in a nomadic computing middleware. The Esperanto Broker provides two of the most obvious ones: i) a location aware service, which allow the application to be aware of the user's current location; and ii) a group communication mechanism, which allows the application components to easily make rendezvous.
- *tools for software design and development*: middleware should support fast service development and deployment. The Esperanto Broker provides a visual tool, namely *ESERV*, that allows the developer to literally *draw* object's interfaces. The development process is made much easier since most of the application code is automatically generated.
- *interoperability*: the Esperanto Broker does not have the ambition to be *the* nomadic computing middleware. It lacks in some other important issues (like those of security), hence it is reasonable that other middleware could be adopted in developing nomadic computing applications. However, any middleware should manifest some sort of interoperability facilities. The Esperanto Broker is interoperable with *Web Services*. Esperanto clients

may invokes web services transparently, and vice versa, web clients may invoke Esperanto servers transparently. The interoperability between the Esperanto Broker with any other middleware *A* is achieved providing a mapping from *A* toward the *Web Services* architecture.

3.3 The Esperanto distributed computing model

Distributed object systems built on the EB are systems in which all entities are modeled as Esperanto objects. We decided to adopt such a computing model since, as compared to the event-based and the tuple-based, it has many advantages: it is very popular, it is well understood and proficiently applied, it aids to reduce the design and development effort. Each Esperanto object implements interfaces defined in Esperanto's Interface Definition Language (E-IDL), which consists of simple extensions to the standard OMG IDL. E-IDL is introduced to further improve the effectiveness of the distributed computing model and to aid developers to easily implement next-generation application scenarios.

By means of E-IDL, developers can specify how Esperanto objects interact according to the communication paradigms proposed by the Web Services Description Language (WSDL) specification. Four are the standardized communication paradigms: i) *request/response*; ii) *one-way*; iii) *solicit/response*; and iv) *notify* [15]. The *solicit/response* and the *notify* paradigms may involve one or more service requesters. The one-to-many communication paradigm is provided by the EB as a built-in mechanism. The idea to extend the IDL instead of providing developers with direct mapping to the WSDL language has two main reasons: i) using a high-level language improves

the portability among different programming languages; and ii) IDL is a pretty common language, developers who are familiar with it would not regret to understand few extensions to the standard language. As further explained in the section 4.5.1, the E-IDL compilation generates intermediate files that can be used to achieve interoperability between Web Services and Esperanto applications.

Client objects access the methods in the E-IDL interfaces via RMIs. Esperanto RMIs are built upon a tuple space (i.e. using write, read and take, plus asynchronous delivery primitives) since objects need to operate in completely decoupled fashion [11, 21]. Such primitives make also the group communication simple to realize. Figure 3.2 shows how to build Esperanto RMIs. Client and server objects are space-decoupled since they interact only via well-known tuple space access points; they are time-decoupled, since the tuple space is carried out by permanent nodes which are always active; they can be synchrony-decoupled since the tuple space provides primitives which let them to asynchronously communicate to one another. This approach overcomes the intrinsic limitations of the *rpc* that forces objects to communicate in a tightly coupled fashion, in that i) it needs to locate the server object, ii) it needs the server to be active to connect to it, iii) it blocks the client, and iv) it can operate only via a pull model in one-to-one multiplicity.

As far as RMIs semantic is concerned, it is worth noting that the *request/response* paradigm's semantic (and of its dual paradigm, *solicit/response*) remains unchanged. Once a client issues a request (i.e. it writes a tuple in the space), it will remain blocked until the server pushes back the reply (i.e. waiting to take the response from the space). Whenever clients and servers need to communicate in asynchronous fashion, they must recur to the asynchronous *oneway* (and its dual *notify*). This

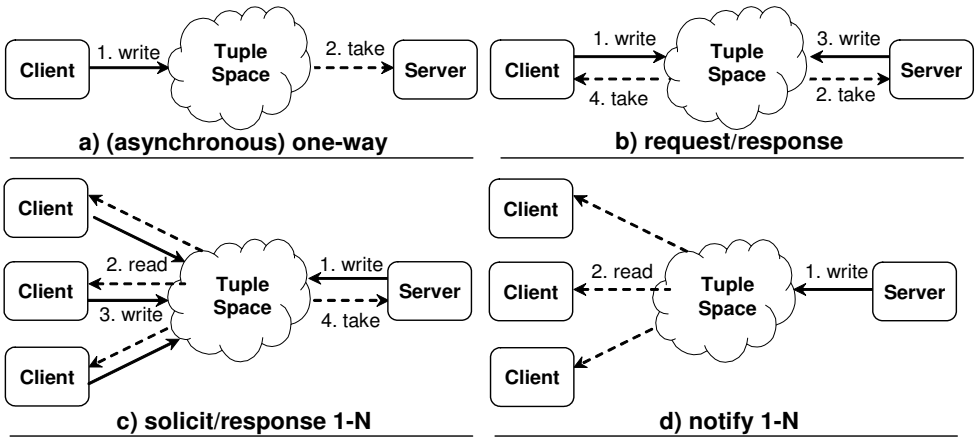

Figure 3.2: Simple implementation of the Esperanto RMIs via tuple space primitives

approach has the main advantage to provide developers with the flexibility to choose the most suitable communication paradigm to application's needs, preserving them from being confused by esoteric semantics.

3.3.1 Esperanto Peers and Esperanto programming model

The Esperanto Broker (i.e. EB) provides programming interfaces and models for distributed object-oriented computing applications. Like all technologies, EB has unique terminology associated with it. Although some of the concepts and terms are borrowed from similar technologies such as CORBA, others are new or different. Understanding these terms and the concepts behind them is key to having a firm grasp of EB itself. The most important terms in EB are explained in the following:

- **Esperanto Peer**: an Esperanto Peer is a "virtual" entity capable of being located by the Esperanto Broker and having requests invoked on it. It is virtual in the sense that it does not really exist unless it is made concrete by an implementation written in a programming language. The Esperanto Peer

reference is used by the Broker to direct requests to concrete objects. An Esperanto Peer may be located on both sides of the computation, i.e. client-side or server-side. For instance, if a E-IDL interface contains both client-initiated and server-initiated paradigms, it will require the implementation of two Esperanto Peers, one client-side and the other one on the server-side.

- **server-side and client-side paradigms**: the programming model of an object-oriented middleware, such as CORBA, provides client objects with the method invocation abstraction via the implementation of the stub/skeleton pattern. A client may invoke a method on a server object via the stub, which makes the remote invocation as it were local, and the server replies to the clients via skeleton, which dispatches the method coherently. This way to exchange messages is so-called *client-initiated*. The Esperanto broker programming model allows server objects to invoke methods on client objects as client and server roles were on the other way around. Therefore, server objects (i.e. objects that implement skeleton classes) may use stubs and vice versa, client objects (i.e. objects that instantiate stubs) may implement skeleton classes. This is done to implement *server-initiated* paradigms.

- **multiple method invocations**: classical remote method invocations are one-to-one paradigms, a client may communicate with one server at time and vice-versa. This is reasonable since the method invocations are basically *client-initiated* primitives. This is still true for the Esperanto *client-initiated* primitives (i.e. *oneway* and *reqres*). However, *server-initiated* primitives allows the server to contact more than one client at time, both to send a notification, and to require responses.

3.3.2 ESERV: The Esperanto Service Descriptor

The ESERV is a graphic tool that allows developer to draw Esperanto interfaces, and generate code automatically. To specify an Esperanto interface the developer has to submit the following information: i) interaction primitives (i.e. *reqres*, *oneway*, *solres*, *notify*); ii) methods signature; iii) parameter directions (i.e. *in*, *out*, *inout*). Once the developer has prepared the service he is interest in, he can decide to generate the application code. He can: i) generate the client-side code; ii) generate the server-side code; iii) generate both; iv) generate the WSDL description of the Esperanto interface; v) generate the bridge to make an Esperanto server interoperable with a web client; vi) generate the bridge to make a web service interoperable with an Esperanto client. Figure 3.3 shows a screen shot of the ESERV tool.

3.4 The Esperanto tuple space model

The Esperanto tuple space is distributed throughout special network nodes that are located on the Core Network. Conversely to other approaches such as Lime and Xmiddle [13, 10], the EB core running on mobile devices acts as proxy to the distributed space, i.e. it implements the primitives for writing/reading tuples in/from the remote shared memory. This to achieve small memory footprint and low computational overhead, and allow resource constrained devices to run Esperanto objects. Permanent nodes which carry out the shared memory are named *Mediators*. A single Mediator is dedicated on each Domain of the Nomadic Computing infrastructure. Objects running on mobile devices exchange remote method invocations' parameters (i.e. method signature along with remote object reference) via the Mediator of the

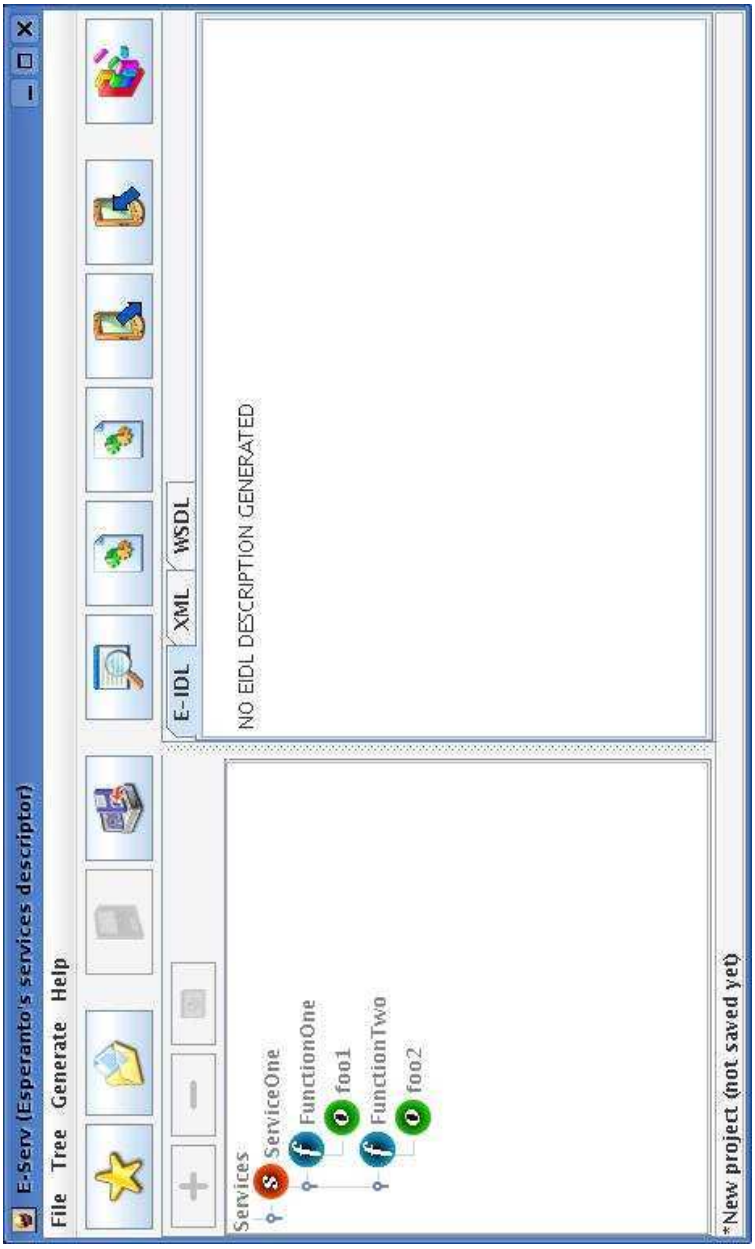

Figure 3.3: A screen shot of the Esperanto Service Descriptor

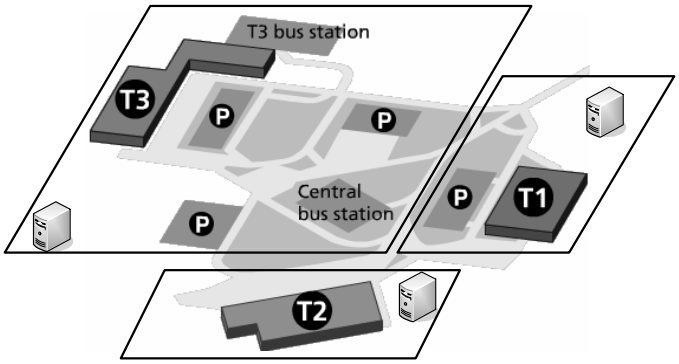

Figure 3.4: Nomadic Computing domains and Mediators allocation

Domain where mobile devices are currently located. Mediators cooperate to allow objects to interact when mobile devices are in distinct Domains.

The Figure 3.4 exemplifies how the network infrastructure of a hypothetic airport can be decomposed in several *Domains* and how Mediators may be assigned to them. Each Domain may have several wireless access points that provide mobile devices with connectivity to the Core network: a mobile device located in the terminal $T1$ may communicate with a mobile device located in the terminal $T2$ being unaware of the tuple space distribution. This has several advantages: i) it simplifies the EB design on the mobile device's side; ii) it may improve infrastructure scalability; iii) it avoids single point of failures. Therefore, the EB results in two different modules: i) the *Mobile-side*, which encompasses the components carried out by mobile devices; and ii) the *Mediator-side*, which encompasses the components carried out by nodes of the Core Network, where Mediators are running on.

Mediators's crucial tasks pose several design issues: Mediators need to be reliable and need to have reliable network connectivity. This can easily achieved if Mediators run on permanent nodes of the Core network, as already stated. However, the side-effect of the proposed approach lies in the fact that mobile devices cannot

communicate to one another if no Core network connectivity is provided. Ad-hoc scenarios are beyond the scope of the Esperanto Broker.

3.5 The Esperanto mobility management

3.5.1 The Esperanto holistic support

Mobility affects all the software layers of a computer system stack. To be successful in addressing the mobility challenges of nomadic computing systems, one must provide a holistic support in mobility management, tackling the challenge at all layers: data-link layer, network layer, transport layer, and middleware layer. This can be done either by means of a layered-approach or a cross-layer approach. Esperanto takes a cross-layer approach in providing a holistic support.

3.5.2 The data-link mobility management

The *Connection and Location Manager* (i.e. CLM) layer aims to provide mobile devices with the *Anytime, Anywhere Access* to the Core Network. CLM provides the built-in mechanisms (i.e. hand-off procedures) to guarantee the continuity of *data-link level* communications despite disconnections (i.e. no coverage area, drops in bandwidth, etc.). Such hand-off procedures are both horizontal and vertical, which means that the CLM is able to connect a mobile device to wireless access points of the same or different technologies. Addressing heterogeneity at the data-link level represents an important step for pursuing the realization of the Next-Generation Wireless Internet scenarios [36].

3.5.3 The network mobility management

There are several approaches to deal with mobility at the network layer. MobileIP [37] is one of the most common. The most important issue that has motivated the IETF to adopt the MobileIP approach was to prevent connection oriented communications (such as those atop of TCP) from being corrupted. However, it is widely recognized that TCP has poor performances in mobile settings [38]. For this reason, the Esperanto Broker refuses entirely the adoption of connection-oriented communication at transport layer. This allows to not have any mobility management at the network layer, except for a small service which reconfigures the network interface, every time a mobile device connect to a new access point. The network interface may change address, subnet and default gateway. The counterpart service on the core network will take care of any request and reply coming from and directed to it.

3.5.4 The transport mobility management

The transport layer of the Esperanto Broker consists of connectionless communication primitives like those of datagram sockets (they are called NCSOCKS). Issue related to corruption, duplication or out of order packet delivery is carried out by the upper tuple space layer. In some sort, the tuple space layer actually represents the transport layer for the Esperanto Broker, since stubs and skeletons are built atop of it. However, these primitives can still represent the transport layer for legacy applications, which want to use them directly. Such primitives take care of channel variability and provides mechanisms to notify applications about the status of the connectivity. Since the above mentioned network may reconfigure the network interface during a data-link handoff, mechanisms are implemented to guarantee that sockets identifiers are still

valid when applications try to use them.

3.5.5 The middleware mobility management

The EB provides mobility management at each layer of the infrastructure. Beyond the facilities provided by the CLM, the Esperanto Broker has its crucial mobility management activities at the middleware layer. These procedures are required for the following reasons: i) the Esperanto *Mobile-side* platform accesses only to the shared memory of the Mediator which the mobile device is currently connected to; ii) if the user migrates from a Domain to another, the *Mobile-side* platform will need a reference to the new Mediator.The GSM architectural model has inspired the middleware layer handoff procedure: i) each mobile device has a home agent (i.e. the Mediator) which stores accounting information and tracks its Domain migrations; and ii) a particular *Mobile-side* component (i.e. the Middleware Mobility Manager), is in charge of triggering the handoff procedure during Domain migrations.

3.6 The Esperanto Broker Core

3.6.1 The Esperanto cross-layer approach

A cross-layer design approach is a design technique where layers in a software stack are designed tightly, that is, data and status about a layer are passed to the higher layers and vice-versa without having firm boundaries that currently exist in modular software stacks. Adopting a cross-layer approach in designing mobile-enabled platforms has two significant advantages over traditional layered implementations that preserve the modularity of a sofware stack.

Firstly, it can result in better application performance and better resource utilization, since the implementation is closely coupled to underlying mechanisms and exploit them efficiently. Secondly, it allows the implementation of effective adaptation mechanisms at higher levels. By knowing low level information, and more generally, global system state information, the way to abstract and synthesize at higher levels is far more effective. Such an approach is adopted to design mostly of the Esperanto Broker and especially the Esperanto Broker Core.

3.6.2 The Connection and Location Manager

The cross-layer approach has been adopted in designing the CLM in order to let applications know about low-level network connection status information. By means of the information flow to/from the CLM, an application can set its QoS requirements in terms of bandwidth, delay, link cost, and location precision. Besides, it can request, or be notified, about connection status changes. The status consists of several information: i) the availability (coded in *connected*, *disconnected*, and *handoff*), ii) the wireless technology being used and its cost, iii) the bandwidth level, iv) the delay level, and v) the mobile device location (in terms of its coarse grained symbolic location, such as the room name).

3.6.3 The Nomadic Computing Sockets

The Esperanto Broker Core consists of CLM, and of the *Nomadic Computing SOCKetS* (NCSOCKS) layers. Upon the CLM services, NCSOCKS provide a transport level object-oriented API that enables developers to be aware of mobility and of wireless network conditions. Such information encompasses current connection status (e.g.,

whether the mobile device is connected, disconnected, or it is performing a handoff) and the connection characteristics (e.g. the available bandwidth, the current delay, the link cost, etc). The NCSOCKS transport is used to implement the proxy on the mobile-side EB. Implementation strategies of the primitives to access the tuple space are enhanced taking advantages of information about the connection status provided by the CLM via the NCSOCKS layer. For instance, strategies to implement retransmission and/or synchronization depends on the network status and have the objective to cope with temporary disconnections.

3.7 Summary of the Esperanto Broker features

Figure 3.5 summarizes the Esperanto Broker's features and compares them with those of the related nomadic computing platforms. The Esperanto Broker does not have lacks in dealing with any of the crucial challenge of nomadic computing environments. To better compare the Esperanto Broker to the other solutions, let us consider the following dimensions:

- **efficient mobility management**: mobility issues like disconnections, variations in network performances and mobile device constraints are needed to be dealt with mobility management procedures and strategies. Such procedures should aim at improving the availability of the device connectivity. Almost all platforms analyzed in Chapter 2 have most of the above mentioned issues not addressed via any effective solution.

- **decoupled communication paradigm**: device disconnections and degradations in network performances affect the ability of an application object to

be available for communicating with counterparts. To improve such an availability, objects should be provided with decoupled communication paradigms. Although any solution analyzed in Figure 3.5 provides a decoupled communication paradigm, either it looses expressiveness of the computing model, or it still adopts sort of synchronous interaction primitives.

- **technology transparency**: the ability to use different access technologies either simultaneously or one at a time should be exploited by any mobile-enabled middleware. The Esperanto Broker implements handoff strategies that allow the device to be connected to the core network seamlessly despite the wireless technology. None of the solutions detailed in Figure 3.5 addresses the heterogeneity in the same way that the Esperanto Broker does. The common approach (adopted by those that face this issue) is to provide an abstraction layer, which hides the underlying technologies and deal with them separately.

- **rich computing model and API**: to be widely adopted a nomadic computing middleware should provide a powerful computing model, and advanced services to aid the designer/developer to build applications. The Esperanto Broker joins *remote method invocations* and *tuple space* together to exploit advantages of both. It also provides mobile-enabled services such location-aware and group-aware services. None of the considered alternatives proposes such a computing model.

- **easy to use in Nomadic Computing**: none of the considered solution provides tools for design and development similar to ESERV. It simplifies the process of designing object interfaces and make the code generation faster.

- **standard interface**: as far as mechanisms to allow interoperability are concerned, CORBA-based solutions rely on the *General Inter ORB Protocol*, whereas other solutions are not concerned with interoperability at all. The Esperanto Broker is interoperable with the *Web Service* standard. By means of bridges, Esperanto clients may invoke web services and vice versa, web client may invoke Esperanto servers. Since *Web Services* are becoming the standard *de facto* in developing and deploying distributed services, our decision to allow interoperability with the Esperanto Broker and any other middleware solution seemed a good way to achieve it. Eventually any middleware solution shall be interoperable with *Web Services*.

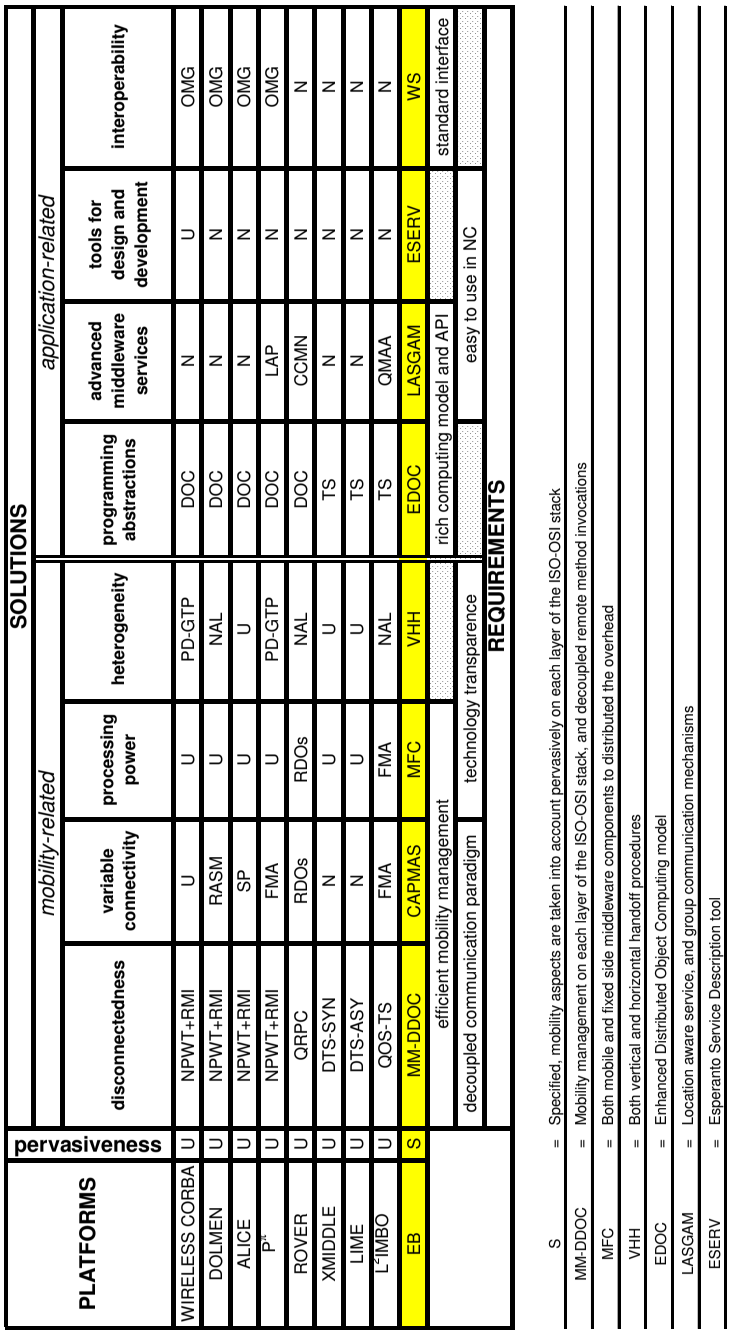

Figure 3.5: Features of the Esperanto Broker compared to features of related nomadic computing platforms

Chapter 4

Design and implementation strategies

4.1 The Architecture of the Esperanto Broker

The EB *layered* architecture is depicted in Figure 4.1 [1]. According to the Esperanto DOC model, the EB *Mobile-side* module allows Esperanto objects to interact via RMIs, despite device movements and/or disconnections. Interactions among Esperanto objects take place via stubs and skeletons objects, which are in charge of performing remote method invocations via Tuple-oriented Primitives. These provide decoupled access to the shared memory located on Mediators. Moreover, capabilities for dealing with device mobility are provided by the NCSOCKS, the CLM, and the Middleware Mobility Manager.

The Tuple Space infrastructure is distributed among Mediators: each Mediator provides its own connected mobile devices with access primitives to the shared space. Mediators cooperate with EB Mobile side to carry out middleware layer handoff procedures. The *Mobile-side* EB deployment consists of two daemon processes and of a run-time library. The *Mediator-side* EB deployment consists of several CORBA

[1] More details and source code are available at *http://www.mobilab.unina.it/Prototypes.htm*

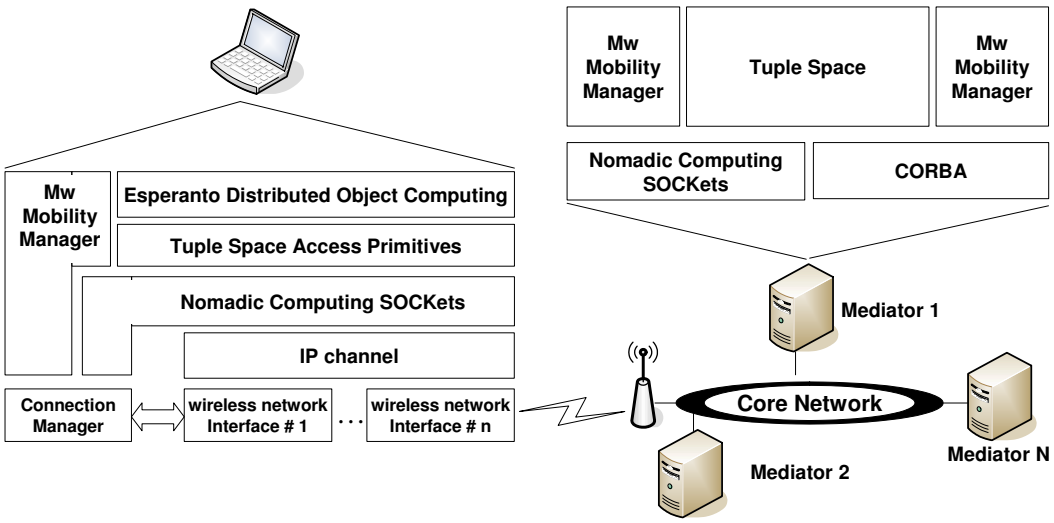

Figure 4.1: The Esperanto Broker Architecture

servers. In the following, we describe the crucial design and implementation details of all the layers the Esperanto Broker consists of.

4.2 The Mobile-side mobility management

As already stated, Domains of the NC infrastructure may be clusters of both heterogeneous wireless access points and access points of the same technology. During user's movements, a mobile device needs to connect with an access point which can be either in the same Domain the user is already in, or in a different one. It is thus clear that procedures to manage handoff has to be provided at both data-link and network layers (i.e. when device migrates between two access points within the same Domain) and at middleware layer (i.e. when device migrates between two access points which belong to different Domains).

As for the Esperanto mobile-side is concerned, how we dealt with this issue is

briefly described in the following: i) data-link level: the *Connection and Location Manager* provides procedures when handoff takes place between two access points whether they belong to the same Domain or not; ii) network level: no procedure has been actually provided, since the broker is built upon connectionless communication channels to overcome connection-oriented channels drawbacks [29, 38]; iii) middleware level: the *Middleware Mobility Manager* provides procedures only if data-link handoffs take place between two access points belonging to different Domains.

4.2.1 Middleware Mobility Manager

The Middleware Mobility Manager is a daemon running on each mobile device, which is in charge of detecting the device migration, and of triggering the domain handoff (i.e. middleware handoff) on both sides of the EB. To this aim, the daemon uses the following map:

```
<domains>
  <domain>
   <domainId> the Esperanto domain identifier </domainId>
    <mediator>
     <id> the Mediator identifier </id>
     <address> Mediator IP address </address>
     <port> Mediator UDP port </port>
    </mediator>
    <WirelessAccessPoints>
      <address> MAC address </address>
    </WirelessAccessPoints>
  </domain>
</domains>
```

The map describes how wireless access points are organized in Domains and which mediators are assigned to each Domain. Whenever the connected access point is not in the list of the current Domain, a domain handoff is triggered. More precisely, the

daemon's activities are depicted in the following loop:

```
this->init();
while(true) {
  this->waitEvents();              // CLM notifies WAPs handoff
  this->lookup();                  // look the map up
  switch(this->event) {
    case AP_HANDOFF:
      this->notifyTDL();           // notifies the tuple space proxy
      break;
    case DOMAIN_HANDOFF:
      this->updateMediatorRef();   // updates the Mediator's reference
      this->notifyTDL();
      this->sendGreetings();       // triggers the Mediator-side handoff
  } // switch
} // while
```

The daemon performs an initialization phase first: it identifies the current Domain location (i.e. it identifies the Mediator to communicate with), and sends the first *Greetings* message (to advertise the device's presence to it). During the loop phase the daemon passively waits for handoffs, at both data-link layer (i.e. transitions between two Wireless Access Points, WAPs) and middleware layer (i.e. transitions between two Mediators). On the data-link handoff, the daemon sends the event to the tuple space proxy, otherwise it triggers the domain handoff via the *Greetings* message to the new Mediator. To accomplish the handoff, the new Mediator needs to know the mobile device identity, the Domain where the device is coming from, and the Mediator that stores the device's accounting information. How the Mediator-side handoff works and which information are transferred between them are illustrated in section 4.8.

4.2.2 Connection and Location Manager

The Connection and Location Manager[2] handles handoffs between wireless access points of different technologies (i.e. *vertical handoff*) and between access points of the same technology (i.e. *horizontal handoff*). The handoff procedure consists of the following phases: i) *Initiation* (the network status is monitored to decide when to start a migration); ii) *Decision* (once the need for handoff is triggered, a new access point has to be selected); and iii) *Execution* (the connection to the selected access point is established).

The CLM layer, which is implemented as a daemon running on mobile devices, is in charge of: i) making the handoff transparent to technologies being used; and ii) pursuing the objective of high connection availability. Since mobile devices might move around different areas with no coverage or high interference, the CLM has to avoid a sudden drop in network bandwidth or a loss of connection entirely trying to perform handoff toward a more reliable or a less overloaded access point. To this aim, it keeps a map of the neighboring access points. During the *Decision* phase, the daemon decides to migrate toward the closest available access point. Other decision criteria may be implemented as well (e.g. the least overloaded access point, the fastest access point, etc).

As for the CLM implementation, we dealt with Bluetooth and Wi-Fi wireless short-range access networks. Thanks to the object-oriented design benefits, the adopted approach can be applied to other wireless technologies. In fact, as Figure 4.2 shows, the CLM is designed according the strategy pattern [39]. Such a pattern basically consists of decoupling an algorithm from its host, and encapsulating the

[2]further details about the CLM can be found in our previous work [36]

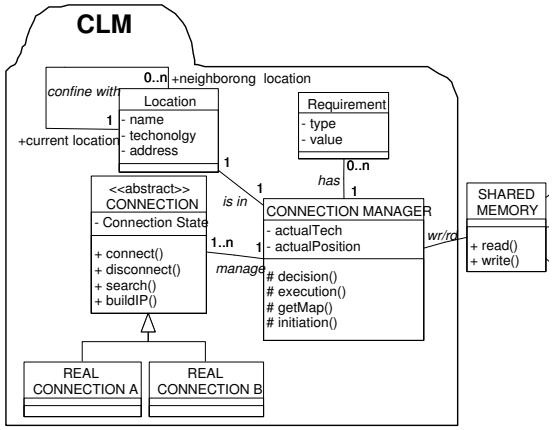

Figure 4.2: Connection and Location Manager class diagram

algorithm into a separate class. In other words, an object and its behavior are separated and put into two different classes. Changes in the algorithms won't affect the class interface.

As for implementation details, we have started using Linux. As for Wi-Fi, the implementation was straightforward, inasmuch the IP abstraction is already provided by Wi-Fi adapters. This is not the case of Bluetooth technology, where a more deep study of BlueZ (*http://bluez.sourceforge.net*), the official Linux Bluetooth stack, has been needed. In particular, since Bluetooth does not support IP natively, the Personal Area Network (PAN) profile and the Bluetooth Network Encapsulation Protocol (BNEP) have been used. On the access point side, we have experienced the implementation of Bluetooth wireless access points by using simple Bluetooth USB dongles attached to PCs. Behind the access point we enable a Network Address Translation Server (NAT), allowing Bluetooth-enabled devices to use private IP network addresses.

4.2.3 Achieving availability: the proposed handoff scheme

The focus of the proposed handoff scheme is to minimize connection unavailability. Starting from the assumption that the device is in a zone covered by access points, otherwise, no connections can be established, connection unavailability can be caused by two kind of events: i) handoff occurrence, and ii) cell overload during an handoff, that can occur whenever a device tries to connect to an AP which cannot manage more connections.

We explicitly note that the connection unavailability for the event i) is negligible as compared to the time spent for the event ii). Furthermore, event i) does not occur if a soft handoff scheme is adopted. For this reason, we assume a soft handoff strategy. This means that we are only concerned with the event ii). Under the above mentioned assumptions, the availability, i.e. the probability that the connection is available during system operations, can be expressed as:

$$aval = 1 - unav = 1 - P_r(O \cdot H) = 1 - P_r(O) \cdot P_r(H) \tag{4.2.1}$$

where $P_r(H)$ is the probability that a handoff occurs and $P_r(O)$ is the APs overload probability. Thus, our goal is to minimize both $P_r(H)$ and $P_r(O)$.

Although we assumed to use a soft handoff strategy, it is should be noted that, in order to minimize the $P_r(O)$ term, soft handoff schemes are not the best choice. In fact, as previous studies stated [40], the overload probability often increases with respect to hard handoff as the number of channels used by mobile terminals grows. However, we are concerned with soft handoff schemes as they help in minimizing the unavailability period due to the handoff per se. Thus, a trade-off between soft and hard handoff should be adopted.

Our proposal consists of using a Last Second Soft Handoff (LSSH) scheme, in which the initiation phase takes place using only the information about the AP currently in use, as in hard handoff, and only in the decision phase multiple connections are established, as in soft handoff. Hence, the LSSH scheme presents the characteristic of using a wireless interface at time during the initiation phase. This also results in i) better energy efficiency due to low power consumption, and ii) interference reduction, indeed using for instance Bluetooth and Wi-Fi simultaneously may produce significant interference [41].

As far as the probability of a handoff $P_r(H)$ is concerned, it should be noted that the initiation phase can be performed using diverse sets of information and techniques, such as broken link recognition and AP monitoring through RSSI. The solution implemented in CLM is RSSI based, for several reasons: i) it allows the handoff to be proactive, ii) the RSSI parameter is already provided by the wireless interface, without performing intrusive measures needed to obtain other parameters, such as throughput or delay; this also reduces the power consumption, and iii) RSSI is an indication of the device position with respect to APs; this helps to achieve load balancing on APs depending on device distribution in the environment.

According to the LSSH scheme, the probability $P_r(H)$ is minimized if the initiation phase is performed only when RSSI permanently goes below a certain threshold. Indeed, transient signal degradations can trigger unnecessary handoff procedures, increasing the probability $P_r(H)$. The mechanism adopted to keep the handoff probability low is the α-count. The α-count function $\alpha^{(L)}$ is a count and threshold mechanism. It takes the L-th measured RSSI as an input, then $\alpha^{(L)}$ is incremented by 1 as the current RSSI falls below the threshold S_{RSSI}. Similarly, $\alpha^{(L)}$ is decremented

by a positive quantity *dec* if the L-th measured RSSI is greater than the S_{RSSI}. A handoff is triggered as soon as $\alpha^{(L)}$ becomes greater than a certain threshold α_T. The function $\alpha^{(L)}$ is thus defined as follows:

$$\alpha^{(L)} = \begin{cases} \alpha^{(L-1)} + 1 & \textit{if} \quad RSSI^{(L)} < S_{RSSI} \\ \alpha^{(L-1)} - dec & \textit{if} \quad RSSI^{(L)} \geq S_{RSSI} \\ & \quad \textit{and } \alpha^{(L-1)} - dec > 0 \\ 0 & \textit{if} \quad RSSI^{(L)} \geq S_{RSSI} \\ & \quad \textit{and } \alpha^{(L-1)} - dec \leq 0 \end{cases}$$

The α-count mechanism avoids to trigger handoffs procedures due to transient RSSI degradations. Indeed, a handoff is triggered if the degradation becomes permanent, i.e. $\alpha^{(L)}$ reaches α_T. Obviously, the values of α_T, *dec* and S_{RSSI} parameters have to be accurately tuned in order to achieve a trade-off between early and late handoffs. Further details on how to tune such parameters can be found in [36].

4.3 Nomadic Computing Sockets

4.3.1 The classes framework

The NCSOCKS provides a C++ API to access an IP-based communication channel. The API provides the UDP communication abstraction, since the TCP is rather inadequate for mobile computing systems [29, 38]. *DatagramPacket*, and *UDPSocket* are the classes provided to send and to receive UDP datagrams: the former represents a packet used for the payload delivery, whereas, the latter is the socket used for sending and receiving datagram packets over the network. These classes implement mobility-aware strategies to cope with issues related to the device mobility (rapid

disconnections/reconnections or handoff periods) during data transmission. Such strategies are driven by the network status provided by the CLM.

4.3.2 Implementation strategies

In order to illustrate the above mentioned strategies, let us consider *send* and *receive* primitives:

```
int UDPSocket::Send(DatagramPacket &packet, int t, int m)
int UDPSocket::Receive(DatagramPacket &packet, int timeout)
```

The *send* primitive provides a data transport service and return the number of delivered bytes. It works as follows: if the channel is established, it will send packets to the IP destination; if the connection establishment is in progress, it will wait for a time t; then it will try to send packets (up to a certain value m) only if the channel reaches the *connected* status within the timeout. In the other cases, an exception is raised. The *receive* primitive provides a blocking data acceptance service (with timeout), and return the number of received bytes. It will receive packets only if the connection is established, otherwise an exception is raised. However, applications can specify a timeout in order to wait for the reconnection. In order to implement mobility-aware transmission, the NCSOCKS layer implements special primitives which interact with the CLM layer:

```
Event UDPSocket::WaitConnection(const int timeout,
                                const int slots, const Event& tr)
Status UDPSocket::senseConnection()
```

4.3.3 Mobile-aware facilities

The *WaitConnection* primitive allows upper layer to wait for a particular event *tr*, until the timeout expires. The *tr* represents the device status transition, for instance, from the status DISCONNECTED to the status CONNECTED. As the name suggests, the *senseConnection* returns information about the device and network status, such as the device state (i.e. connected, disconnected, or handoff), the location (i.e. which access point the device is connected to), the quality of the connection (i.e. the receiver signal strength indicator), the network bandwidth etc.

Applications may also require to be notified in connection status changes. As Figure 4.3 shows, applications are provided by the NCSOCKS with a class, the Connection Monitor, which is used by them to set their requirements and to read or register their interest in some connection status information. The monitor provides a set of setRequirement()-like methods to set application level requirements.

Since different mobile-enabled applications on the same device may ask for contrasting requirements, exception are raised in order to let the application (or the user) to relax such requirements, if any, or abort the execution. Unspecified values for some connection attributes are automatically set to non-conflicting default values. In this way, all applications running on the same mobile device are forced to agree with the same non-conflicting requirement set.

On the other hand, the monitor allows applications to read connection status information, and to register a callback with it to be executed when specified changes occur in some connection status information. In this way, applications can adapt their behavior accordingly. Finally, applications can also force a handoff triggering, if the expected requirements are not satisfied.

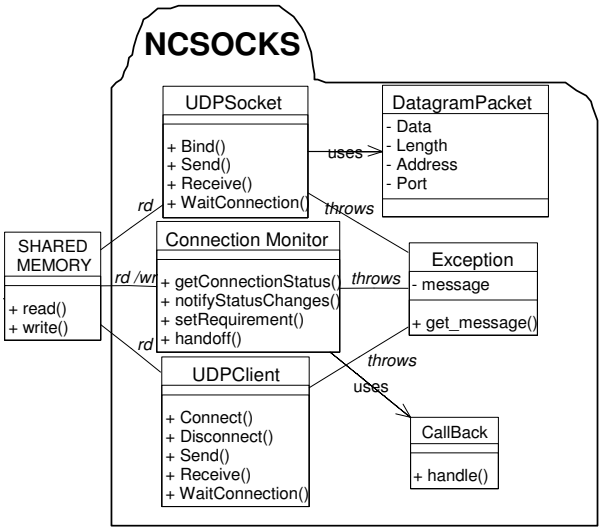

Figure 4.3: NCSOCKS class diagram

4.4 The Esperanto Broker tuple space

The primitives provided by this layer are the following:

```
void write(const Tuple& tuple)
Tuple read(const Tuple& template, Time timeout)
Tuple take(const Tuple& template, Time timeout)
int subscribe(const PeerId& receiver, const ParamList& pl,
                                    const CallBcakRef callback)
void unsubscribe(const int subscriptionId)
Event detect(const unsigned int timeout)
```

The *write* accepts a tuple, containing application-level information (e.g. either the signature of a method to invoke or the method's return value). The *read* (*take*) accepts a tuple template (i.e. a tuple with wildcards), containing parameters needed to retrieve (to remove) a tuple from the space, and returns the matched tuple, if any. The requester will wait to get the tuple until the timeout expires. The *subscribe/unsubscribe* are used for the asynchronous tuple notification. The application has to provide the template of the tuple it is interested in, and the callback reference

to invoke when the tuple is available. The *detect* is used to know the current status about the device connectivity.

4.4.1 The tuple data structure

The tuple structure is illustrated below:

```
<tuple>
  <sender> who writes the tuple </sender>
  <receiver> who needs the tuple </receiver>
  <parameters>
     <parameter>
        <name> the parameter semantic </name>
        <type> the type </type>
        <value> the value </value>
     <parameter>
  </parameters>
  <lease> time to the removal from the space </lease>
</tuple>
```

The schema has been designed to achieve efficiency of the template matching algorithm. Since this layer underlies stubs and skeletons, the matching algorithm is especially computed when a server object needs to retrieve the pendent invocation requests to its methods. It is thus important to have low latency in such an operation. To this aim, *sender* and *receiver* are kept separated from the parameters list and they are used as indexes for accessing the shared space.

4.4.2 Tuple space access primitives

Write, *read/take* and *subscribe/unsubscribe* are the building blocks for the implementation of Esperanto RMIs. We implemented RMI strategies taking into account the efficiency as primary requirement. To this aim, server objects subscribe themselves

to the reception of any tuple they are interest in. Whenever a client object writes a tuple in the space, this is immediately notified to the server, if connected. Otherwise, the tuple remains in the space and can be obtained via a pro-active tuple retrieval. The interaction may take place with success even though objects are not active at the same time (i.e. time decoupling), in fact the space itself is in charge to store requests and replies. They do not need to know the counterpart location (i.e. space decoupling), since the space itself is in charge to keep their references. They can interact asynchronously, since *oneway* and *notify* directly map onto *write* and *subscribe* primitives.

4.4.3 Implementation strategies

Tuple space access primitives are implemented through NCSOCKS. Similarly to the *send/receive* primitive, tuple transmission has been implemented using mobility-aware strategies. For instance, the write primitive does not try to send the tuple if the device is performing an handoff or is disconnected. The algorithm passively waits for the device re-connection (via the *waitConnection*) and then delivers the tuple. If the disconnection is permanent it will raise an exception. Such a strategy is crucial for saving computational cycles and battery's energy. In fact, whether the device is disconnected or not, underlying sockets are still valid and the algorithm would try to deliver packets since no feedbacks on the nature of the failure are provided by the operating system.

4.4.4 Cross-layer approach

The *detect* primitive is used to collect information about the device connectivity. Such information is presented in the form of an event: an event may be the device's migration between two Domains or the device's migration between two WAPs. The *detect* primitive can collect only local events, which means that it cannot know if any other device has migrated toward a different Domain. If no event occurs, it will block the caller until timeout expires, otherwise it notifies the particular occurred event. Such a primitive is crucial to let Esperanto stub and skeleton classes be able to implement mobility-aware strategies during the phases of tuple retrieval and dispatching. In fact, when a handoff occurs, objects get disconnected, and thus unable to interact. This may affect method invocations (at client-side), or dispatching operations (at server-side). As shown in the following paragraph, stubs and skeletons implement proper strategies in order to deal with these crucial issues.

4.5 The Esperanto Interface Definition Language

The Esperanto IDL extends the OMG IDL in order to provide the following communication paradigms: i) *request/response*; ii) *one-way*; iii) *notify*; and iv) *solicit/response*. The *notify* is the *one-way* paradigm that allows a server object to send messages to one or more clients, while the *solicit/response* 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 illustrate the use of the IDL extensions, consider the following IDL, which defines an interface named *MyService*:

```
interface MyService {
   oneway void fooA(in int op);
```

```
    reqres bool fooB(in long op1, out string op2, inout long op3);
    solres void fooC(in string op1, out string op2, out double op3);
    notify void fooD(in float op);
};
```

The Esperanto IDL adopts a *service-centric* approach. This means that E-IDL *describes interactions* between objects rather than describing just methods to be invoked on the remote server. The *oneway* and *reqres* qualifiers describe that client and server objects can communicate according to the pull model, while the *solres* and *notify* qualifiers describe that a client and server objects can communicate according to the push model. As for the former interactions, client objects must invoke *fooA* and *fooB* methods on server-side, whereas for the latter interactions, server objects must invoke *fooC* and *fooD* methods on client-side. Due to these considerations, for *oneway* and *reqres* methods, **in** parameters are passed from the client to the invoked server object, whereas **out** parameters are passed back from the invoked server object to the client object (only for *reqres* methods). Conversely, for *solres* and *notify* methods, **in** parameters are passed from the server to the invoked client objects, while **out** parameters are passed back from invoked client objects to the server object. Parameters labeled as **inout** are allowed only in *reqres* methods and their values may be passed in both directions. It should be noted that *oneway* and *reqres* methods must be implemented on server-side, while *solres* and *notify* methods must be implemented on client-side.

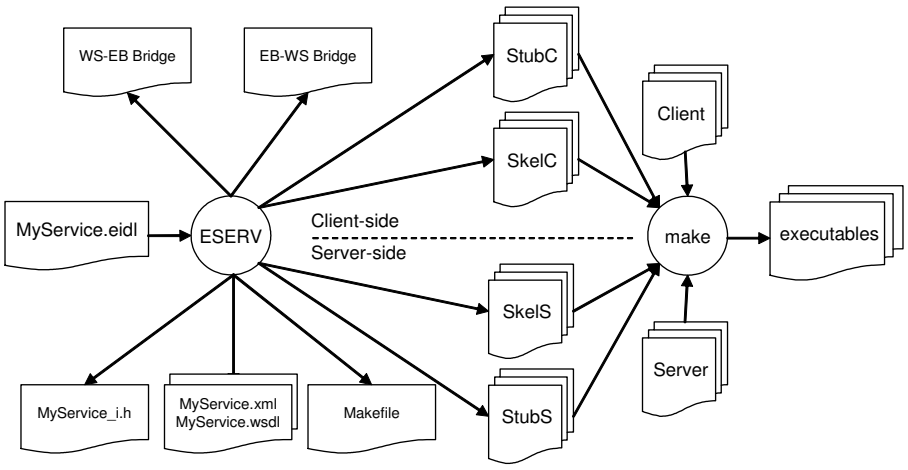

Figure 4.4: The life-cycle of Esperanto applications

4.5.1 The life-cycle of an Esperanto application

Developing an Esperanto application is a CORBA-like process, composed of two main steps: i) the *E-IDL* files compilation; ii) the application and stub/skeleton files compilation. The Figure 4.4 depicts such a process for the interface *MyService*. The E-IDL compilation process is supported by a GUI editor[3], ESERV (i.e. Esperanto SERVice descriptors), which allows the developer to design Esperanto interfaces and to compile them. As Figure 4.4 shows, ESERV produces two sets of files: i) helper files (i.e. to compile stub/skeleton classes, and to provide the WSDL mapping); ii) stub/skeleton implementation files. More precisely, the E-IDL compilation produces four *namespaces*: *stubC*/*skelC* used for building the client-side application, and *stubS*/*skelS* used for building the server-side application. In the further discussion we present the E-IDL mapping to stub/skeleton classes and the underlying tuple space infrastructure considering the example interface *MyService*.

[3]More details and source code are available at *http://www.mobilab.unina.it/Esperantodwnd.htm*

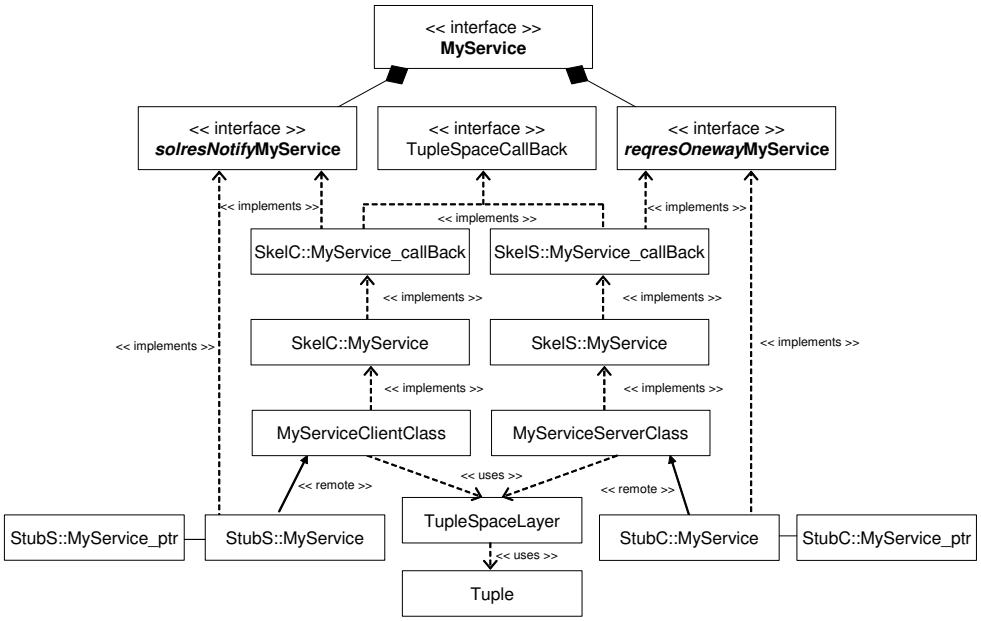

Figure 4.5: The class hierarchy produced by the E-IDL compilation process

4.5.2 Mapping the Esperanto Interfaces

The Figure 4.5 shows the hierarchy that ESERV produces when it translates E-IDL *MyService* into C++ classes. There are two couples of stub and skeleton classes. *StubC::Myservice* and *SkelS::MyService* implement the pattern for all *reqres* and *oneway* methods via implementing the abstract class:

```
class ReqresOnewayMyService {
  public:
    virtual void fooA(const int op) = 0;
    virtual bool fooB(const long op1, string &op2, long &op3) = 0;
    ...
}
```

StubS::Myservice and *SkelC::MyService* implement the pattern for all *solres* and *notify* methods via implementing the abstract class:

```
class SolresNotifyMyService {
  protected:
```

```
    virtual void _c_fooC(const string &op1, string &op2, double &op3) = 0;
    virtual void _s_fooC(const string &op1, fooCGroup &fCg) = 0;
  public:
    virtual void fooD(const float op1) = 0;
  ...
}
```

Stubs and skeletons provides also helper methods to aid the developer writing the code.

4.5.3 Mapping WSDL to Esperanto

ESERV produces the WSDL description of the Esperanto interfaces. Building such a mapping is very straightforward. ESERV generates an intermediary file where it describes the Esperanto interface according to the *xml* syntax. The following fragment shows how the E-IDL representation of the *fooA* method has been translated into the *xml* representation:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--File automatically generated by ESERV 0.1-->
<services>
  <service name="MyService">
    <function TTEreq="25" TTEres="" name="fooA"
                          return_type="bool" type="reqres">
      <operand flow="IN" name="op1" type="long"/>
      <operand flow="OUT" name="op2" type="string"/>
      <operand flow="INOUT" name="op3" type="long"/>
    </function>
  </service>
</services>
```

Once such a translation is performed, the mapping to *PortTypes* (i.e. the operations performed by the web service), *messages* (i.e. the messages exchanged), *types* (i.e. the data types used) and *bindings* (i.e. the communication protocols adopted) is achieved by parsing the *xml* tags, i.e. *service*, *function* and *operand*.

4.6 The Esperanto DOC abstraction

4.6.1 The Esperanto Peers

Stubs need a reference to skeletons. An Esperanto reference, named a *PeerId*, is a triple of attributes: i) a *Peer Reference* to identify the object running on the mobile device; ii) a *Device Reference* to identify the device among the others being connected to the Nomadic Computing infrastructure; and iii) a *Domain Reference* which is the identifier of the Mediator whose Domain hosts the device. This reference does represent the remote object and it is used by the EB in order to deliver tuples to the object.

4.6.2 Client-side mapping

As stated, the client-side mapping entails two namespaces:

- *StubC*: it contains the stub-side of *reqres* and *oneway* methods. Stubs make RMIs transparent to invokers by behaving like a local object. The Figure 4.6 (a) shows how the client-side *reqres* RMIs map to the underlying tuple space (*oneway* RMI mapping is pretty similar). When a client object invokes the *foo()* method, the stub marshals the request into a tuple and writes it to the remote shared memory. Afterward, it takes the response from the remote shared memory, and unmarshals the tuple returning the results to the invoker. *Reqres* RMIs behave like regular synchronous remote invocations, but client and server objects are time and space decoupled by means of the Mediator. If the stub gets disconnected after requesting the invocation, the Mediator stores the reply that can be retrieved later, without raising any communication exceptions.

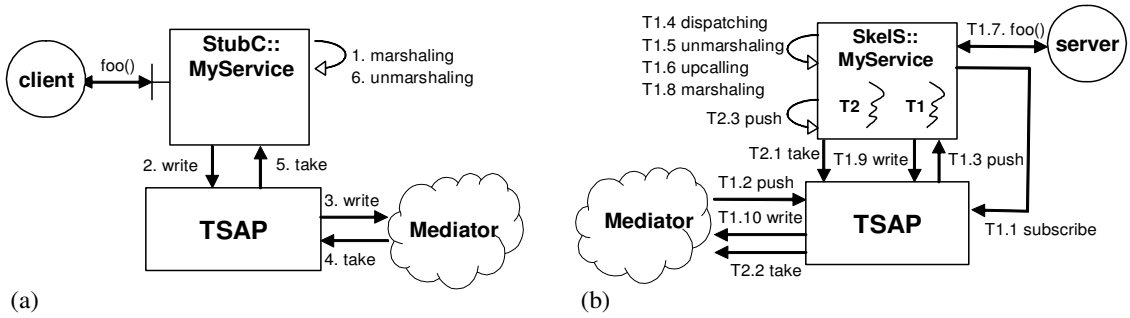

Figure 4.6: The implementation of *reqres* RMIs: (a) client-side mapping; (b) server-side mapping

- *SkelC*: it contains the skeleton-side of *solres* and *notify* methods. *SkelC* skeleton behaves similarly to *SkelS* skeleton, however they differ in functionalities they provide to the developer. Main difference lies in the possibility to disable server's solicits and/or notifications. Such a feature may be useful to save cpu cycles and/or device's battery. *Solicits* and *notifications* are enabled via a helper method named *_bind*. The lease may be set to a specific value. When the lease expires, the client object has to reinvoke the *_bind* method if it is still interested in server messages. A lease equal to 0 corresponds to an *unbind* operation.

4.6.3 Server-side mapping

As stated, the server-side mapping entails two namespaces:

- *SkelS*: *skelS* defines classes used for implementing the skeleton-side of *reqres* and *oneway* methods. The skeleton is in charge of retrieving requests from the tuple space, dispatching them and up-calling the relative functions. If a *reqres* is involved in the process, the skeleton must also push back the response. The Figure 4.6 (b) shows how the server-side *reqres* RMIs map to the underlying

tuple space. The requests retrieval is implemented by means of two strategies: i) via an asynchronous delivery, which takes place in two phases (first, the skeleton subscribes a callback to be invoked whenever tuples match the template it is interested in, and then, waits passively for them); and ii) via a pro-active search, where the skeleton periodically takes all the tuples from the space that for some reasons could not have been notified to it. The skeleton implements both the *TupleSpaceCallBack* and *ReqresOnewayMyService*. The former is the callback invoked by the the tuple space for the asynchronous delivery, whereas the latter provides the pure virtual methods that the server class must implement, so to let skeleton up-call them.

- *StubS*: it contains classes used for implementing the stub-side of *solres* and *notify* methods. *StubS* stub behaves similarly to *StubC* stub, however they differ in the implementation: the former allows the server to solicit and/or notify one or more clients at the same time. To handle this situation, server invokes methods passing a special-purpose data structure, *fooCGroup*, which stores client object *PeerIds* and (for each of them) parameters being exchanged. If the server wants to send solicits/notifications to every listener, the data structure will empty. If it does not, the data structure will contain only entries for clients that are interest in the solicits/notifications. When the control has been passed back to the invoker, the *fooCGroup* contains responses of client objects which are bound and achievable. In the following, we illustrate a partial implementation of the *solres fooC* method:

```
  ...                                                  // MARSHALLING
this->_tuple = _fooC_request_tuple_;
this->_tuple.setSender(this->_this);
```

```
this->_tuple[_TUPLE_REQ_ID_INDEX_].value =
                                    ulong2string(this->_request_id++);
this->_tuple[_fooC_REQ_TUPLE_PARAM_OP1_INDEX_].value = op1;
if (fCg.size() == 0) { // to every client object
      ...
} else {   // only to the selected client objects
 fooCGroup::iterator pos;
 for (pos = fCg.begin(); pos != fCg.end(); pos++)
  try {                                              // WRITE
   this->_tuple.setReceiver(pos->first);
   this->_tsl.write(this->_tuple);
   this->_tuple = _fooC_reply_tuple_;                // TAKE
   this->_tuple.setReceiver(this->_this);
   this->_tuple = this->_tsl.take(this->_tuple, TIMEOUT);
   if (!this->_tuple.isEmpty()) {                    // UNMARSHALLING
    pos->second->op2 = _tuple[_fooC_REP_TUPLE_PARAM_OP2_IDX_].value;
    pos->second->op3 =
         string2double(_tuple[_fooC_REP_TUPLE_PARAM_OP3_IDX_].value);
   } else {
     delete pos->second;
     pos->second = NULL; // Peer unavailable: delete the PeerID
   }
 } catch (const Exception& e) { ... } // catch
} // else
```

Before writing the request tuple in the remote shared memory, the stub checks which kind of solicit the invoker has been requested: in the case of a solicit to every client, *fooCGroup* is empty, and the stub writes the request tuple setting for it the shared group identifier. The tuple retrieval consists of several take operations until it returns an empty tuple. In the case of a selective solicit, the *fooCGroup* contains the list of *PeerId*s which the invoker wants to solicit. The stub writes the request tuple setting the specific *PeerId* and waiting for the response tuple via a take operation.

4.6.4 Implementation strategies

In previous sections we stated that skeletons retrieve tuples periodically. This is to cope with device disconnections which inhibit the asynchronous delivery of tuples. In fact, if an object is temporarily disconnected (due to either a Domain handoff or an access point handoff), tuples remain in the shared memory and no notifications can be delivered to the interested object. Skeletons may be aware of such an event via the *detect* method, thus retrieving tuples issuing a *take* operation. This mobility-aware tuple retrieval strategy is illustrated in the following code fragment:

```
  ...
Tuple retrieved, tmplt;
tmplt.setReceiver(this->_service);
Time idleTime = MAX_TIME;
while (true) {
  trigger = this->_tsl.detect(idleTime);
  switch (trigger.id) {
    case DOMAIN_HANDOFF: idleTime /= 4; break;
    case WAP_HANDOFF: idleTime /= 2; break;
    case DETECT_TIMEOUT_EXPIRED: idleTime += idleTime/2;
  } // switch
  if (trigger != DETECT_TIMEOUT_EXPIRED)
    do {
      retrieved = this->_tsl.take(tmplt, TIMEOUT);
      if (!retrieved.isEmpty()) this->push(PUSHING_DONE, retrieved);
      else break;
    } while (true);
} // while
```

The time between each retrieval is set according to mobile device status. The possible events and the actions taken are: i) a timeout expiration means that the device is steadily located in a Domain. Then the asynchronous tuple notification works fine. The skeleton may slow the pro-active tuple retrieval down, saving computational cycles and battery; ii) a WAP handoff means that the most of the time the

device is disconnected or performing the transition. Most likely the skeleton has not been notified of some requests (which are stored in the tuple space, though). There is a need to speed up the pro-active tuples retrieval from the space; iii) a Domain handoff means that the device might have been disconnected recently. Most likely, some undelivered tuples might still be on the space. There is a need to shorten the time to the next pro-active tuples retrieval.

4.7 The Esperanto Mediator

Each Mediator performs two tasks: i) it implements the tuple space, providing client and server objects with the distribution transparency; and ii) it is charge of device mobility management. To accomplish these tasks a Mediator collaborate with other Mediators. This collaboration is achieved by means of CORBA middleware, namely, to send/receive tuples to/from remote shared spaces, and to update device location references. The Mediator itself is a distributed component, implemented as a set of distributed CORBA objects. We used TAO [42] as the CORBA platform for the implementation of our prototype.

4.7.1 Implementation strategies

The Figure 4.7 shows a detailed UML CORBA component diagram of the Tuple Space Layer. As figure shows, this layer consists of four CORBA servers: the *Bridge*, which carries out the mapping between the corresponding *Mobile-side* layer and the *Mediator-side* tuple space layer, the *Tuple Dispatcher*, which hides the distribution of the shared memory, the *Tuple Manager*, which implements the access primitives

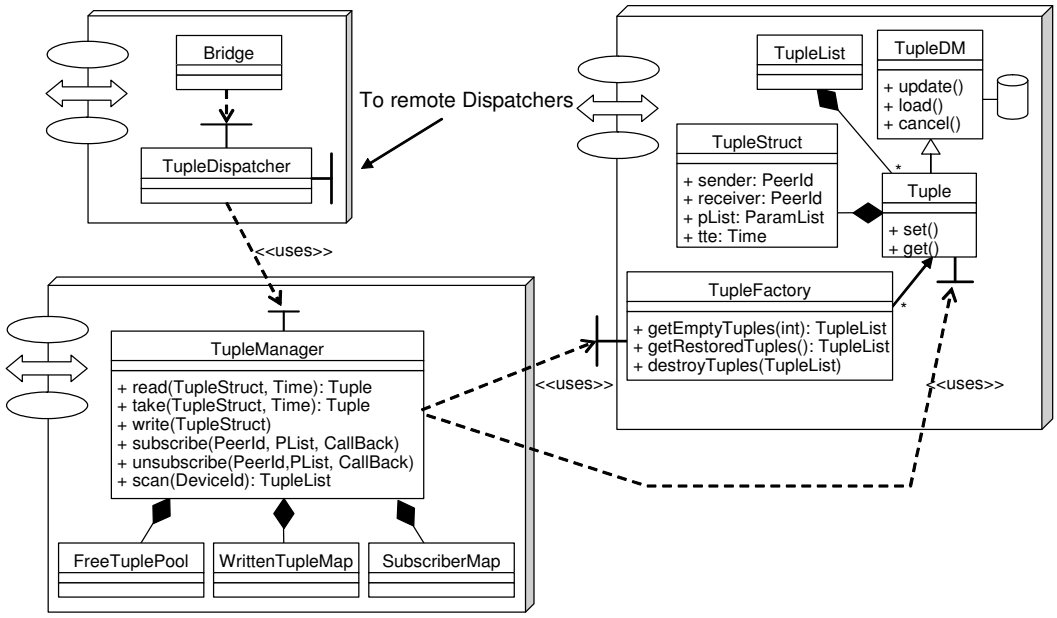

Figure 4.7: UML CORBA component diagram of the Tuple Space layer

to the local tuple space, and the *Tuple Factory*, which acts as a factory of tuples. The idea behind the Esperanto tuple space implementation is to achieve tuples persistence preserving the transparency of a specific database technology. To this aim, we encapsulate the persistence strategy (e.g. XML-native DB, RDBMS, file system) in CORBA servants, i.e. *Tuple* servants. More precisely, the persistence strategy is encapsulated in the *TupleDM* skeleton class, which currently implements a serialization of the tuple attributes into a XML file. The IDL representation of the tuple structure is the following:

```
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 lease;
};
interface Tuple {
  attribute TupleStruct ts;
};
```

Thus, the Mediator's tuple space consists of a set of Tuple servants which are ready to serve *write*, *read* or *take* requests. To keep the incoming request's latency low, Tuple servants are created at initialization time via the *Tuple Factory* and are accommodated in a pool of available servants.

4.7.2 Tuple space access primitives

The *Tuple Manager* is the component in charge of implementing the tuple space *primitives*:

1. *write*: Whenever a tuple is requested to be written, the *Tuple Manager* first checks if any callback is subscribed to its reception. In this case, it pushes the tuple to the skeleton via the registered callback. If the no callbacks are available or they are temporarily disconnected or unable to receive tuples, it stores the tuple by setting the *Tuple* servant and marking it as *busy*.

2. *read/take*: Whenever a *read/take* request is issued, the *Tuple Manager* performs a matching algorithm between the template provided in input and the tuples stored by busy *Tuple* servants. A template matches a tuple if the following

conditions hold: i) tuple and template XML schemes are the same (the empty parameters list works like a wildcard); ii) tuple and template receivers are the same; iii) tuple and template senders are the same (if specified); and iv) tuple and template parameters list are the same, i.e. each parameter has the same name, type, and value (if any).

3. *subscribe/unsubscribe*: by issuing *subscribe* requests, objects ask for tuples to be delivered as soon as they are available in the local space. In this way, they are not charge of pro-active retrieval saving cpu cycles and battery energy. Listener are implemented via the following callback interface:

```
interface CallBack {
  void push(in TupleStruct ts) raises (EsperantoException);
};
```

4. *scan*: a *scan* request is issued by other Mediators. The method provides a mechanism to retrieve tuples whose receiver objects are running on a specific mobile device. A *scan* request is issued when a mobile device migrate from a domain to another. During such a migration, it is most likely that some tuples cannot be asynchronously delivered and will remain in the old Mediator tuple space. As soon as the device becomes connected again and the handoff succeeds, tuples need to migrate to the new Mediator tuple space, so that object running on the mobile device can retrieve them pro-actively.

4.7.3 The shared space distribution

The *Tuple Dispatcher* hides the distribution of the tuple space by wrapping the local *Tuple Manager* and cooperating with remote *Tuple Dispatchers*. Interfaces implemented by the *Dispatcher* are the following:

```
interface ToBridge { // implements delivery protocols among Mediators
  oneway void write(in TupleStruct ts);
  Tuple read(in TupleStruct tTemplate, in Time timeOut);
  Tuple take(in TupleStruct tTemplate, in Time timeOut);
  oneway void subscribe(in PeerId pId, in ParamList pl, in CallBack pT);
  oneway void unsubscribe(in PeerId pId, in ParamList pl, in CallBack pT);
};
interface ToRemoteDispatcher {
  void write(in TupleStruct ts) raises (EsperantoException);
  ToRemoteDispatcher whereis(in DeviceId dev) raises (EsperantoException);
};
```

More precisely, as far as write operation is concerned, the *Tuple Dispatcher* works as follows:

1. if the tuple receiver is running on a mobile device located in the Domain where the request is coming from, the *Dispatcher* writes the tuple locally (i.e. by invoking the *write* method on the *Tuple Manager*).

2. if the tuple receiver is running on a mobile device located in a different Domain, it forwards the tuple to the Mediator where the device is currently located (i.e. by invoking a remote *write* method on the remote *Dispatcher* reference). Due to device mobility, this reference may be obsolete, therefore the *Dispatcher* first inquires the Mediator which tracks device migrations (by invoking the *whereis* method on the relevant *Dispatcher*), and afterward forwards the tuple by invoking the remote *write* method on the up to date *Dispatcher*'s reference.

Finally, the remote *Dispatcher* invokes a *write* operation on the local *Tuple Manager*.

3. if the tuple receiver refers to a group of Esperanto objects, the *Dispatcher* forwards the tuple to each remote Dispatcher available.

As for the read/take operation, thanks to the above mentioned *write* strategy, read requests are always processed as tuple retrieval on the local *Tuple Manager*. Also *subscribe* and *unsubscribe* primitives work on the local *Tuple Manager*.

4.8 The Mediator-side mobility management

Since the tuple space access primitives on the EB *Mobile-side* are implemented by using the NCSOCKS communication layer, and in particular by using the UDP protocol, the Mediator provides a component, i.e. the *Bridge*, which carries out requested operations as CORBA RMIs and gives the results back to the *Mobile-side* EB. The *Bridge*, which is a multithreading UDP server, carries out the following operations: i) it accepts and interprets requests coming from the *Mobile-side* when Tuple Space Access Primitive are invoked; ii) it translates parameters contained in the NCSOCKS datagrams into CORBA-compliant parameters; iii) it invokes the correspondent method on the *Tuple Dispatcher* CORBA servant; and iv) it makes the inverse translation, and gives back the result (if any).

The Figure 4.8 shows the *Mediator-side* components which are involved in the middleware layer handoff procedure. Each Esperanto mobile device has a Mediator (called *Home* Mediator) which stores information about it and its migrations among the Nomadic Computing Domains. The *Device Manager* is the component which has

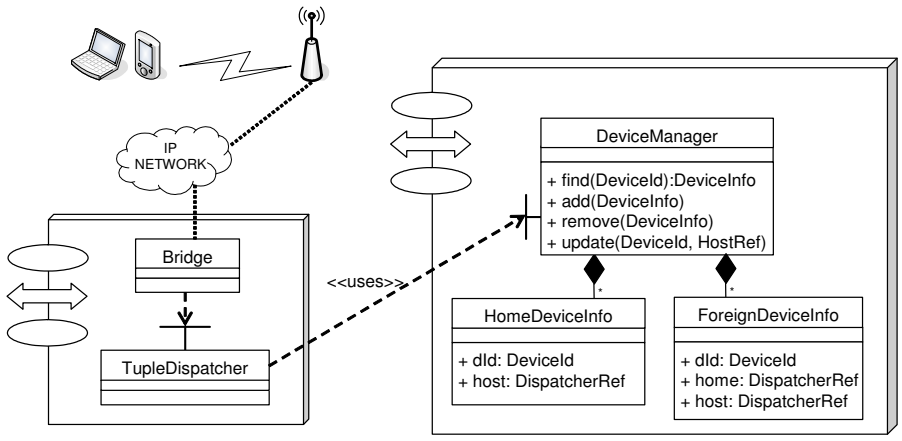

Figure 4.8: Mediator components involved in the middleware mobility management

such a responsibility. As stated in the Section 4.2, the handoff procedure has been triggered by the *Mobility Manager* daemon, when it sends the *Greetings* message to the *Bridge*. Once the handoff has been triggered, these are the steps to complete the procedure:

1. the new Mediator's *Dispatcher* notifies the *Home* Mediator that the *Host* Mediator ha been changed (i.e. by invoking the *notify* method on the *Home* Mediator's *Dispatcher*)

2. it transfers all the tuples concerning objects running on the mobile device, from the old shared memory to the local shared space (i.e. by invoking the *moveTuples* method on the old *Host* Mediator's *Dispatcher*).

The accounting information needed to do the above mentioned steps are: i) the identifier of the mobile device (i.e. *DeviceId*), ii) the reference to the current Domain, which is the current Mediator the device is being connected to (called *Host*). The latter information is used by the *Dispatcher* in order to correctly forward tuples through the distributed shared space. Beyond such information, the *Device Manager*

caches *Host* Mediator's references of *foreign* Esperanto devices, so that avoiding the *Host*'s reference resolution each time a tuple needs to be forwarded.

4.9 Bridges for interoperability with *Web Services*

There are two basic requirements a bridge generator has to meet to achieve interoperability with Web Services. When linking Esperanto applications with Web Services the Esperanto interface definitions need to be translated into corresponding Web Service descriptions. Afterward, one program for each mapping (i.e. Web Services to Esperanto Broker and vice versa), has to be generated that transposes the Web Service invocations into equivalent Esperanto invocations and vice versa.

Using an Esperanto-based application, the remote interfaces are defined in the E-IDL. Web Service based applications, on the other hand, commonly specify their interfaces using the WSDL. Therefore, an IDL file containing the Esperanto interface definitions first needs to be translated into a corresponding WSDL file. To carry out this task, the ESERV parses the IDL file and transforms the parse tree into a WSDL document, as shown in section 4.5.3. Afterward, the tool generate two executable components:

- *Bridge toward Esperanto Services*: such a component is in charge of mapping SOAP messages into Esperanto RMIs. Basically, web clients are unaware of sending SOAP messages toward an Esperanto server, since the bridge appears as a web service, which publish itself into an UDDI registry and marshals SOAP messages into Esperanto RMIs, sending them to the server that implements the E-IDL specification.

- *Bridge toward Web Services*: such a component is in charge of mapping Esperanto RMIs into SOAP messages. Basically, Esperanto clients are unaware of making RMIs toward a web service, since the bridge appears as an Esperanto server, which is compliant to the E-IDL definition and marshals Esperanto RMIs into SOAP messages specified into the WSDL document generated by ESERV, and vice versa.

4.9.1 Mapping SOAP messages to Esperanto RMIs

Having translated the IDL interface definitions into the WSDL format, the next step is to generate the bridge itself. Its main task is to interpret the received Web Service messages, map them to the corresponding Esperanto method invocations and finally return their results again as a Web Service message. Obviously, it must be ensured that all of the bridge's messages conform to the WSDL document.

The main component of the bridge is a multi-threaded web server, which receives and decodes incoming HTTP/SOAP messages using the *libcurl* and *libxml* C++ libraries. Having received a message, in accordance to the "façade" design pattern [39], its task is to determine, which proxy is responsible for handling the message. The bridge then invokes the corresponding substitute's operation in the appropriate proxy class.

Once the bridge is running within the target system, it can be reached via the URI specified in the deployment descriptor. The first time the bridge is used, it resolves the symbolic names of the remote Esperanto peer. The obtained references are buffered for later access to reduce unnecessary overhead. If the bridge receives an incoming SOAP message it will extract its content and determine the appropriate

proxy class and the correct method to invoke on the Esperanto server.

The message content is passed as parameter to the operation, which redirects the invocation to its dedicated Esperanto Peer. The result of the invocation is then, again, packaged by the proxy into a SOAP fragment and passed back to the web server. Here, the SOAP message is terminated and sent back to the calling web client. Should an exception be raised during this process, an error message will be generated. If a user-defined exception is raised, the proxy takes care of constructing the error message; internal errors or Esperanto exceptions are handled by the bridge itself.

4.9.2 Mapping Esperanto RMIs to SOAP messages

The main task of the bridge from Esperanto clients toward web services is to interpret the Esperanto RMIs, map them to the corresponding SOAP message invocations and finally return their results again. To this aim, the method's parameters list is wrapped in a tuple to be easily parsed via *libxml* into a SOAP envelope.

The main component of the bridge is an Esperanto skeleton, which dispatches and up-call incoming RMIs, while the real server parses them into SOAP messages. Since the bridge acts as a web client, it has to be able to find the potential web service, which should be registered itself at a UDDI registry. To overcome possible limitations, once the bridge is running within the target system, it can reach the actual web service via the URI specified in a configuration file, which has to be hand-edited.

Then an Esperanto server can invoke methods on the web service. Using information from the WSDL document, the Esperanto server is able to send a request to the actual service and has to wait for the appropriate response. As in the previous case,

the details of the bridge functionalities are completely transparent to the Esperanto clients. The bridges receives the request, and, according to the received data, communicates with them. The response of the web service is translated into a tuple via the skeleton services and returned to the Esperanto client.

Chapter 5

Developing Esperanto applications

To test the effectiveness of the Esperanto approach, we have employed the EB in educational projects of the basic distributed programming courses at the University of Naples. Several mobile applications have been successfully developed. In this section we illustrate *SmartMall*, an advertisement manager application where the scenario is an outlet mall. The manager has several aims: to suggest walking paths, to appeal customers with promotions about goods, to ask them for feedbacks, etc. While walking around shopping areas, customers may want to buy products, to search for items, and make reservations at a food court's restaurant.

5.1 Requirements issues

5.1.1 Mobility-related issues

While walking among different shopping areas, customers may experience periods of disconnections. Moreover, the advertisement manager service may suggest walking paths which depend on the current shopping areas where customers are located, or send promotions which are related only to shops in the customers's nearby and based

on costumer's profile. On the customer side, he/she may want to reserve a table at the nearest food court area, or restrict the search for a product only to them sold by shops in the nearby.

Therefore the following issues need to be addressed:

- *track costumers walking paths*: the service infrastructure needs to know current and past customer's locations. To achieve this, the mobile-side service infrastructure has to notify current device's location, whereas the fixed-side infrastructure has to keep tracking device movements.

- *profile costumers attitudes and needs*: the service infrastructure needs to know about costumers attitudes and needs as they get into the *SmartMall*, i.e. they get the mobile device. To address this, the mobile-side service infrastructure has to query costumers on basic matters, such as shoppoing's aim, and forward it to the fixed-side service infrastructure.

- *make costumer service provisioning aware of his/her location*: the service delivery has to be dependent on costumer's location. To this aim, the fixed and mobile infrastructure has to provide services whose computations depend on the possible shopping areas of the *SmartMall*.

- *let costumer use services despite mobile terminal disconnections*: service provision has to be feasible even if costumer is disconnected. For instance, the costumer should be able to roll the list of products despite he/she is connected or not. To this aim, the mobile-side service infrastructure has to implement strategies to cache data from the fixed-side service infrastructure.

5.1.2 Application-related issues

The advertisement manager needs to send promotions to whoever may be interested, or only to a set of customers that have expressed the intention to receive promotions about products sold in a specific shopping area. Feedbacks may be requested either on per-customer basis or to any customer. It is thus clear that promotions can be sent to one or more customers, while feedbacks can be sent back by one or more of them.

Therefore the following issues need to be addressed:

- *pushed-based service delivery*: the service infrastructure has to be able to send both time-triggered (i.e. based on a certain period of time) and event-triggered (i.e. based on the fact that costumers have come in a certain area) information. Costumers do not request for such information, they are just notified by the infrastructure asynchronously and pro-actively. To this aim the mobile-side service infrastructure has to receive such information and show them to the costumer.

- *one-to-many service delivery*: the fixed-side service infrastructure needs to interact with more than one costumer at the same time. To address this, the fixed-side service infrastructure has to rely on a *broadcast* mechanism to send information out to all costumers.

- *selective one-to-many service delivery*: the fixed-side service infrastructure needs to interact with more than one costumer at the same time. However, costumers may be selected based upon their profiles and/or their location. To address this, the fixed-side service infrastructure has to rely on a selective *broadcast*

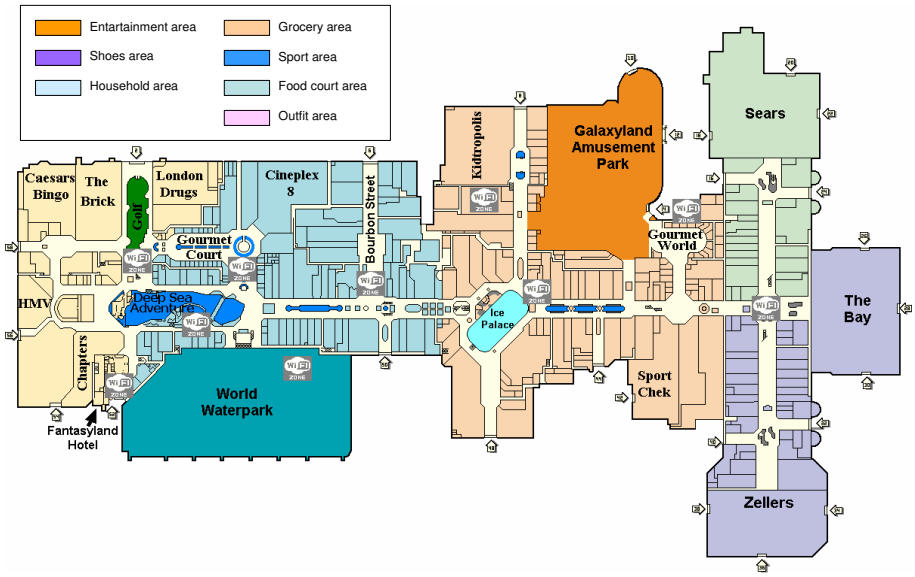

Figure 5.1: Partitioning of a hypothetical shopping mall

mechanism to send information out to the interested costumers.

5.1.3 The Esperanto approach

The aforementioned issues are easily addressed with the adoption of the EB. The proposed platform copes with mobile devices disconnections, provides mechanisms to cluster shops in several domains, provides both manager-initiated and customer-initiated communication paradigms, and allows the manager to directly contact one or more customers. By means of conventional middleware it would be very hard to implement such a mobile computing application.

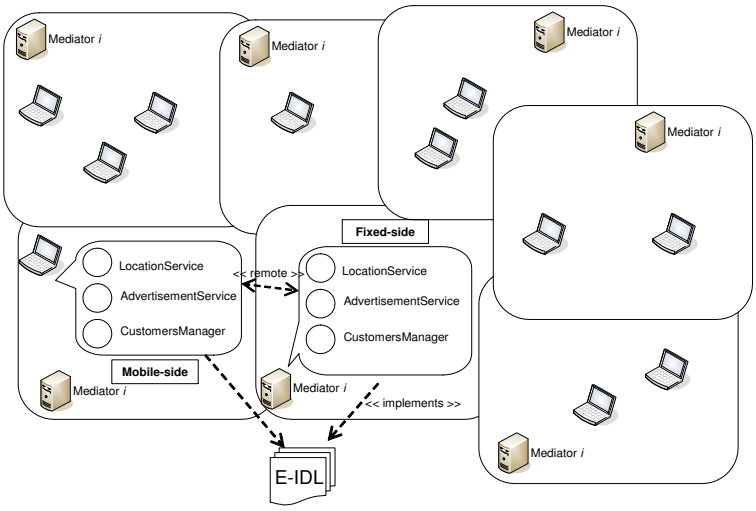

Figure 5.2: Conceptual diagram for the *SmartMall* scenario

5.2 Design issues

5.2.1 Architecture design

As far as the application architecture is concerned, we have assumed that there is a Domain for each shopping area (e.g. food-court area, clothes and shoes area, grocery's area, etc.) and that each Domain requires the deployment of the following entities: i) the location service; ii) the advertisement service; iii) the customers manager. Figure 5.1 depicts how such a partitioning of a hypothetical shopping mall may happen.

Basically, a Mediator will run in each domain, as well as the the above mentioned entities. Therefore, the conceptual diagram that describes such a system decomposition at a very high level of abstraction is the one shown in Figure 5.2.

5.2.2 Interfaces design

According to issues detailed in section 5.1, the crucial interfaces needed to be implemented are the following:

```
interface LocationService {
  oneway void sendLocation(in DeviceId customer, in DomainId from);
  notify void notifyLocation(in DeviceId customer, in DomainId location);
};

interface AdvertisementService {
  solres void askForFeedbacks(in string question, out string feedback);
  solres void suggestWalkPath(in string direction);
  solres void sendPromotion(in string promotion);
};

interface CustomersManager {
  reqres int makeReservation(in int howMany, inout DomainId foodCourt);
  reqres bool buyProduct(in int productId, in DeviceId custumer);
  reqres bool searchFor(in int productId);
};
```

We assume that a customer gets an Esperanto-enabled device at the mall's entrance, so that he/she is able to contact a *CustomersManager*. The *LocationService* allows both customers to send their Domain location, and *advertisementServices* to be notified of their location changes. An *AdvertisementService* suggests walk paths, sends promotions and ask for feedbacks to customers, as soon as they get inside the shopping area it is in charge of.

Client-side application gets the current device location by means of a local service implemented via the *detect()* primitive. The compilation process of the above mentioned interfaces generates the necessary stub and skeleton classes to invoke methods on both sides of the service infrastructure.

The E-IDL and the DOC model help to easily describe and extend interactions and entities: for instance, it may be possible to let customers communicate to one another.

5.3 Development issues

The development process of any Esperanto application follows the basic steps reported in the following:

5.3.1 Drawing interfaces

The task of drawing Esperanto interfaces consists of specifying, for all interfaces that are needed to be implemented by Esperanto Peer, the list of methods, along with the specification of the interaction paradigm and the list of passing parameters. Using ESERV, such a task is not error-prone, inasmuch ESERV check for errors (for instance, *oneway* methods cannot have *out* parameters).

Figure 5.3 shows how the specification of the above mentioned interfaces appears in ESERV. Once the developer has done with the interfaces visual description, he/she can generate the E-IDL file along with the stub and skeleton code.

5.3.2 Code Generation

The code generation provides stubs and skeletons to implement the application logic. In the following, part of the code generated for the *AdvertisementService* interface is shown.

```
class AdvertisementService: protected AdvertisementService_CallBack {
  public:
    AdvertisementService(const PeerId& service): AdvertisementService_CallBack(service) {}
    virtual ~AdvertisementService() {}
    void _bind(const Time lease) throw (EsperantoException) {
      try {
        AdvertisementService_CallBack::_bind(lease);
      } catch(const EsperantoException& ee) { throw; }
    }
    bool _isBound() {
      return AdvertisementService_CallBack::_isBound();
    }
// Methods
```

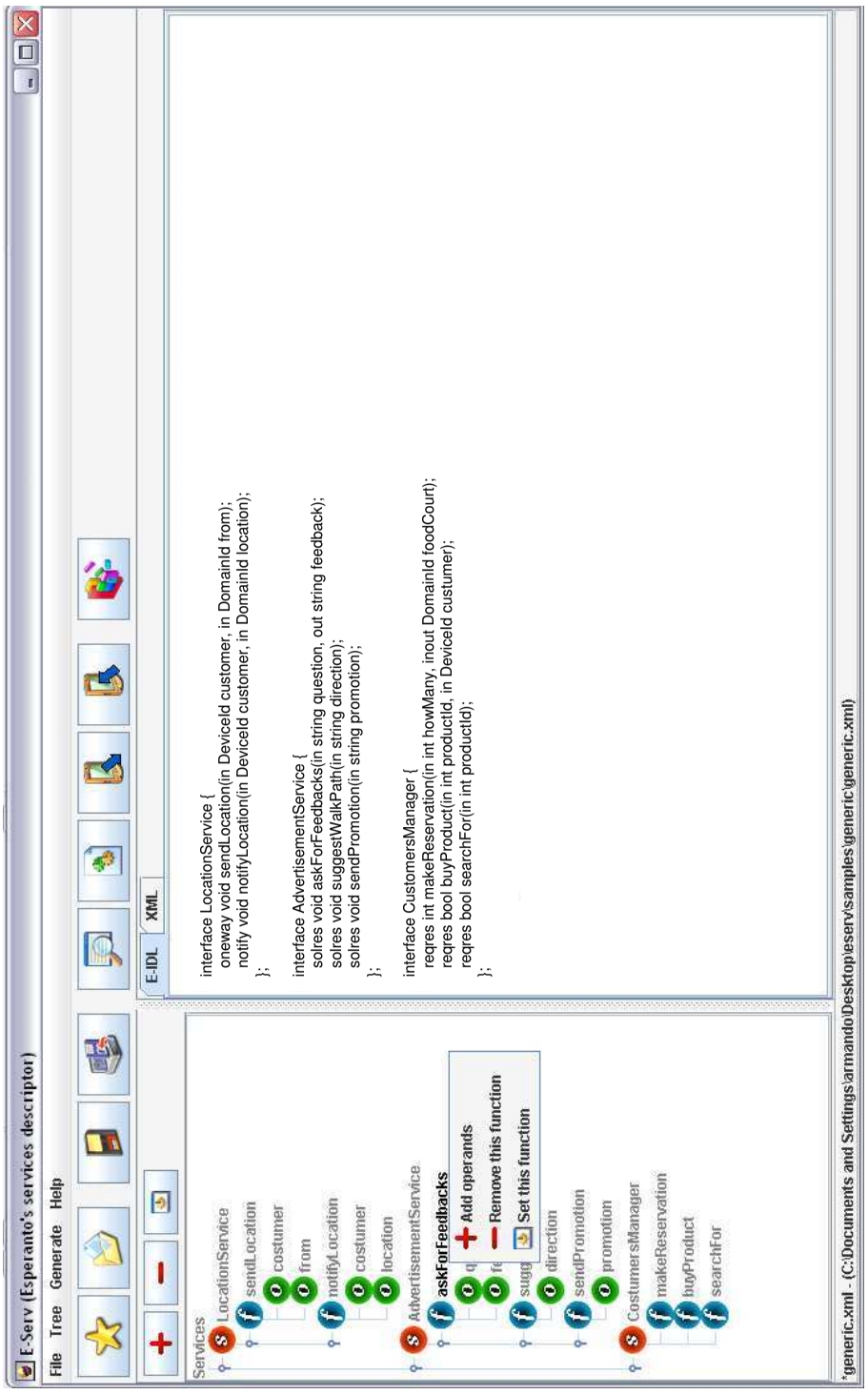

Figure 5.3: Screen shot of the ESERV tool to design Esperanto interfaces

```
    virtual void askForFeedbacks(const string& question, string& feedback) = 0;
    virtual void suggestWalkPath(const string& direction) = 0;
    virtual void sendPromotion(const string& promotion) = 0;
};
```

As mapping rules state, the interface compilation will generate two classes, more precisely a stub server-side and a skeleton client-side. This is due to the fact that no *oneway* and *reqres* methods are available in the interface definition. The C++ code above shows the skeleton client-side that the developer needs to implement, while the C++ below shows the server-side stub that the developer must use.

```
class AdvertisementService {
  public:
    void askForFeedbacks(const string& question, askForFeedbacksGroup& askForFeedbacks_group)
     throw (EsperantoException);
    void suggestWalkPath(const string& direction, suggestWalkPathGroup& suggestWalkPath_group)
     throw (EsperantoException);
    void sendPromotion(const string& promotion, sendPromotionGroup& sendPromotion_group)
     throw (EsperantoException);
    GroupIDs _activeGroup() throw (EsperantoException);
  private:
    AdvertisementService(const PeerId& service);
    ~AdvertisementService() { pthread_mutex_destroy(&_mutex); }
    PeerId _this;
    PeerId _group;
    GroupIDs _groupIDs;
    Tuple _tuple;
    TupleLayer _tdl;
    pthread_mutex_t _mutex;
    timestamp _request_id;
    int _ref_count;
    friend class AdvertisementService_ptr;
};
```

5.3.3 Building Esperanto Peers

Once all the automatic code is generated, the developer may focus solely on the application logic. The application logic will be embedded only into the virtual pure methods specified by skeleton classes declaration. For instance, as far as the *AdvertisementService* is concerned, the simplest way to implement the *askForFeedbacks* method will be the following:

```
...
virtual void askForFeedbacks(const string& question, string& feedback) {
  cout << "\nPlease provide a feedback to the question: " << question << endl;
  cin >> feedback;
  return;
}
...
```

Each costumer is asked to reply to the question with a particular feedback. On the stub side, the developer needs to ask for feedback either on event-triggered base or on time-triggered base. Let's suppose the service sends out feedback requests when required by the shopping mall manager. For instance, the simplest way to implement such a logic is shown in the following piece of code:

```
...
string question;
while (true) {
  askForFeedbacks_group feedbacksGroup;
  cout << "Insert the question you wanna ask: " << flush;
  cin >> question;
  cout << "Please, wait for reply..." <<  flush << endl;
  costumers->askForFeedbacks(message, feedbacksGroup);
  askForFeedbacks_group::iterator pos;
  for (pos = feedbacksGroup.begin(); pos != feedbacksGroup.end(); pos++) {
    cout << "Feedback from costumer: " << pos->first.PeerId2str()
                                    << " -> : " << pos->second->feedback << flush << endl;
    delete pos->second;
  } // for
} // while
...
```

The Manager is prompted to issue the feedback to each costumer. In the code above, feedbacks are issued to every costumer since the *feedbacksGroup* structure is empty. Whenever he/she needs to ask feedbacks to only some costumers the application logic would be the one shown below:

```
//
// I - asks for available costumers
//
  int i = 0;
  GroupIDs costumersIDs = costumers->_activeGroup();
  cout << "Active costumers:" << flush;
  GroupIDs::iterator pos;
  vector<PeerId> peerIdVect;
  for(i = 0, pos = costumersIDs.begin(); pos != costumersIDs.end(); pos++, i++) {
    cout << "\n" << i << " - " << (*pos).PeerId2str() << flush;
```

```
    peerIdVect.push_back(*pos);
  } // for
bool end = false;
askForFeedbacks_group feedbacksGroup;
//
// II - selects the costumers to solicit
//
  do {
    cout << "\nSelects Peers to solicit (-1 to end): " << flush;
    cin >> i;
    end = (i == -1);
    if (!end)
    feedbacksGroup.insert(pair<tdl::PeerId,
                        askForFeedbacks_groupRef>(peerIdVect[i], new askForFeedbacks_group));
  } while (!end);
//
// III - isses the questions
//
  if (feedbacksGroup.size() > 0) {
    cout << "Insert the question you wanna ask: " << flush;
    cin >> question;
    cout << "Please, wait for reply..." <<  flush << endl;
    costumers->askForFeedbacks(message, feedbacksGroup);
  } // if
//
// IV - checks for feedbacks from available costumers
//
askForFeedbacks_group::iterator posFeedbacksReply;
  for (posFeedbacksReply = feedbacksGroup.begin();
                        posFeedbacksReply != feedbacksGroup.end(); posFeedbacksReply++) {
    if (posFeedbacksReply->second != NULL) {
      cout << "Feedback from costumer: " << pos->first.PeerId2str() <<
                        " -> : " << pos->second->feedback << flush << endl;
      delete pos->second;
    } else {
      cout << "- " << posFeedbacksReply->first.PeerId2str() <<
                              " -> is no longer available" << flush << endl;
    }
  } // for
...
```

As the piece of code shows, activities being done are the following:

1. the Manager asks for available costumers;

2. he/she fills the *feedbacksGroup* structure up with the costumers he/she wants to solicit;

3. he/she issues the question and sends it to interested costumers;

4. he/she checks for feedbacks provided by the costumers who have replied.

5.4 Deployment issues

As far as the deployment is concerned, the client-side application prototype has been cross-compiled for the *StrongArm* platform (i.e. to run on PDAs), while the server-side application has been compiled for the traditional *i386* platform. Tests ran in our laboratories, where two shopping areas were simulated and three customers were moving around hallway and rooms. Deployment usually requires a preliminary phase where the following activities need to be done:

- *Description of the Esperanto domains*: this phase requires that domains, which the the Nomadic Computing infrastructure is decomposed in, and Mediators needed to be allocated are described via *xml* syntax.
- *Description of the Nomadic Computing infrastructure*: this phase requires that wireless access points, which are connected to the Nomadic Computing infrastructure, are described in terms of their characteristics and relate to each other.
- *Tuning configuration parameters*: this phase requires that parameters to control the reaction time of the handoff strategies are setup.

5.4.1 Description of Esperanto domains

The description of the Esperanto domains is required to let the Mobile-side Esperanto Broker be aware of the partitioning of the Nomadic Computing infrastructure. Such a description is stored in a *xml* file, which contains for each domain (i.e. the *outfit_area*, the *sport_area*, etc.), how many mediators are running, how to contact them, and which are the wireless access points that cover the domain area.

In the following a fragment of such a file is reported:

```
<domains>
  <domain>
    <domainID>OUTFIT_AREA</domainID>
    <mediator>
       <bridgeAddress>10.0.0.35</bridgeAddress>
       <bridgePort>10000</bridgePort>
       <accessPoints>
          <address>00:0E:6A:FD:DA:BD</address>
          <address>00:0E:6A:FD:DA:AD</address>
          <address>00:0E:6A:FD:CB:EF</address>
       </accessPoints>
   </mediator>
</domain>
<domain>
    <domainID>SPORT_AREA</domainID>
    <mediator>
       <bridgeAddress>10.0.0.35</bridgeAddress>
       <bridgePort>20000</bridgePort>
       <accessPoints>
         <address>00:0B:AC:E8:59:32</address>
         <address>00:0E:6A:E9:44:23</address>
       </accessPoints>
   </mediator>
</domain>
...
</domains>
```

As the fragment shows, even if the number of the running mediators are more than one, there is only one physical host that is involved. This can have a good impact on deployment costs, since the number of domains and mediators do not affect the number of physical nodes needed to let the middleware run.

5.4.2 Description of the Nomadic Computing infrastructure

The description of the Nomadic Computing infrastructure consists of the list of the available wireless access points along with configuration parameters that are needed by the Mobile-side Esperanto Broker, when a handoff triggers. In the following, an *xml* fragment shows the description of some access points and their relationships.

```
<infrastructure>
  <locations>
     <loc>
        <name>galileo</name>
        <tech>blue</tech>
        <address>00:0E:6A:FD:DA:BD</address>
        <initstr>blue</initstr>
```

```
        <srssi>3</srssi>
        <alphathr>6</alphathr>
    </loc>
    <loc>
      <name>mercurio</name>
      <tech>blue</tech>
      <address>00:0B:AC:E8:59:32</address>
      <initstr>blue</initstr>
      <srssi>3</srssi>
      <alphathr>6</alphathr>
    </loc>
    ...
  </locations>
  <neighbors>
    <loc="galileo">
      <neigh>mercurio</neigh>
    </loc>
      <loc="mercurio">
      <neigh>galileo</neigh>
    </loc>
    ...
  </neighbors>
</infrastructure>
```

Basically, such a list is used to determine which wireless access point to contact when device is about to be disconnected by old access points (for this, a *neighbors* section describes such relationships). For each access point, parameters used by the handoff algorithm to tune α-count behavior are also specified.

5.4.3 Tuning configuration parameters

The tuning phase is especially needed to setup the data-link handoff algorithm. Such a tuning consists of choosing the proper values for parameters that control the α-count scheme behavior. Basically, the proposed α-count scheme affects the *reaction time* of the initiation strategy. If we define the time within which the handoff is triggered as the reaction time T_r, once the mobile device reaches a wireless cell boundary, the handoff toward a new access point should be triggered at most after T_r seconds. The aim of the tuning process is to keep this time as low as possible.

Obviously, T_r is a function of α-count parameters, that is $T_r = f(S_{RSSI}, \alpha_T, dec)$;

hence, once the expected dimension of a cell is fixed, it is necessary to tune the α-count parameters in order to achieve a certain T_r when the boundaries are reached. We define an experimental mean to tune the α-count parameters, given the cell dimensions and the expected T_r. For each cell, the tuning process encompasses three steps:

- *Experimental evaluation of Received Signal Strength Indicator (i.e. RSSI)*: the frequency distribution at the cell boundary of the RSSI is determined. To this aim, the mobile device is located at the boundary of the cell and the RSSI samples are recorded in order to obtain statistic information.

- *Simulation of the α-count algorithm*: once the RSSI distribution is determined, it is used to run a simulation set of the α-count algorithm to estimate the T_r as function of the triple $(S_{RSSI}, \alpha_T, dec)$. More precisely, triples of parameters $(S_{RSSI}, \alpha_T, dec)$ are chosen arbitrarily and the expected T_r samples are collected.

- *Choosing the correct parameter values which guarantee the expected T_r*: once the previous steps are done, choosing the desired T_r and the respective triples of parameters $(S_{RSSI}, \alpha_T, dec)$ is quite straightforward.

To exemplify such a process, let consider the following example: according to the first step, we capture RSSI values in several parts of the cell boundary region. As for the second step, α-count simulation results are depicted in figure 5.4, in order to evaluate T_r as function of α_T, dec, and S_{RSSI} parameters.

Once the desired T_r is determined (as an example $T_r = 4$ in the figure, which is emphasized by a dashed line), it is possible to determine different $t = (S_{RSSI}, \alpha_T, dec)$ triples that produce the expected T_r (for example both $t_1 = (S_{RSSI} = 3, \alpha_T = 4.3, dec = 0.2)$ and $t_2 = (S_{RSSI} = 3, \alpha_T = 6, dec = 0.8)$ triples can be used).

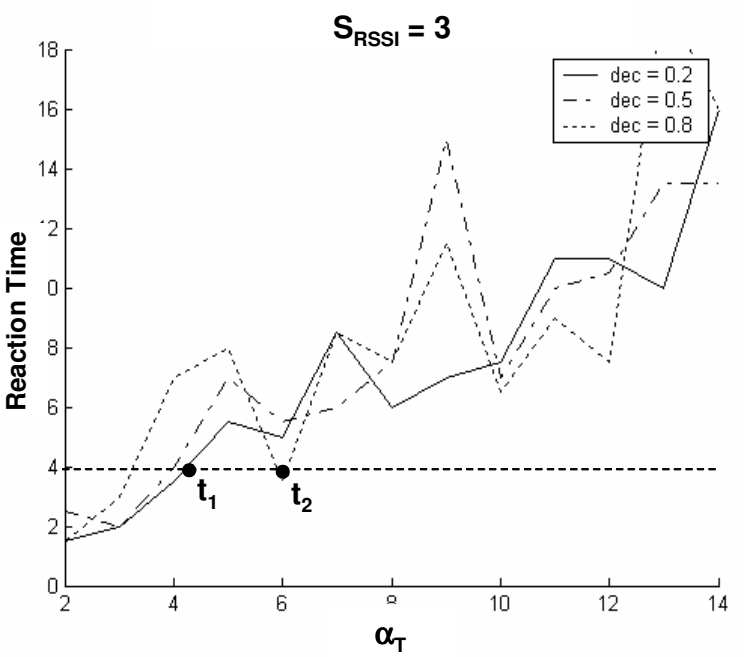

Figure 5.4: α-count parameters tuning

By evaluating the RSSI frequency distribution in several cell boundaries, it is possible to achieve different parameters values for different cell sizes. Then the correct set of values can be chosen by the CLM with respect to application requirements (e.g. for location sensitive applications, it is better to use small cells, whereas for high bandwidth applications is better to use big cells in order to rarely change the current good AP).

5.5 Conclusions

Tuning parameters for handoff algorithms and for mobility-aware strategy timeouts are usually the most delicate parts of the entire development process. However, extending and modifying the Esperanto network infrastructure is easy as modifying

an xml document. Adding/removing domains, mediators and wireless access points requires only to know basic information like IP and MAC addresses. A further improvement would be the implementation of a GUI application which actually allows the developer to draw such an infrastructure and which generate the xml document in automatic way.

The EB has all the building bocks to develop and deploy next-generation mobile computing applications. The developer is shielded from low-level details, group communication and mobility mechanisms, and can focus only on the application business logic using a powerful computing model. As for the design phase, the developer is provided with a high-level tool, i.e. ESERV, which allows to draw interfaces and the involved communication paradigms without any effort in knowing the language's syntax.

As for the development phase, he/she is required to write only the business logic code, since all the skeleton code, i.e. makefiles, and stub/skeleton classes are generated automatically. As for the deployment phase, he/she is required to configure the particular Esperanto network infrastructure, plug in the needed access points, and run commands to start daemons, on the mobile-side, and Mediators on the fixed-side.

Several under-graduated projects have proved that students dealt with Esperanto as good as they did with traditional middleware like CORBA. Esperanto has a GUI tool to generate and compile E-IDL interfaces, and other GUI tools (i.e. utilities to configure both mediators and the esperanto network infrastructure) may be developed to further improve its usability factor.

Chapter 6

Experimental results

An experimental campaign has been conducted to evaluate the performances of the implemented prototype. The Esperanto Broker has been tested under different scenarios and load conditions, collecting several measures of latency and throughput. Measurements to estimate the impact of mobility on remote interactions have also been estimated. In the following, we present the experiment design and setup, and analyze the obtained experimental results.

6.1 Experiment design

6.1.1 Experiments aims

Experiments were aimed at estimating:

- *the performance penalty due to the client-server decoupling via Mediators*: Esperanto objects communicate to one another having Mediators as their intermediates; this may result in a cost on the overall interaction latency.
- *the EB's performance behavior as compared to MIWCO*: the Esperanto Broker

prototype may not be optimized. Comparing it to an similar middleware may give us clues about how to further improve it.

- *the overhead of stub/skeleton layer built upon the tuple space*: building a distributed object computing model over a tuple space model may have a cost in marshaling and unmarshaling operations.

- *the scalability of the infrastructure*: Mediators are crucial for handling middleware activities such as handoff procedures and tuple dispatching operations; they may represent a bottleneck of the entire middleware infrastructure.

- *the impact of mobility on the overall performances*: a mobile device may experience disconnections during a handoff. Estimating how mobility affects the overall performances is crucial for determining mobility-aware strategies to reduce such effects.

6.1.2 Comparing Esperanto Broker and MIWCO

MIWCO [12] has been chosen as the middleware to compare the Esperanto Broker with. MIWCO is a MICO's extension to add support for Wireless Access and Terminal Mobility as specified by OMG's Telecom Domain Task Force [30]. In such an infrastructure mobile terminals are connected to the fixed network through General Inter-ORB Protocol (i.e. GIOP) tunnels. The endpoints of this tunnel are called *Terminal Bridge* (on the mobile side), i.e. TB, and *Access Bridge* (on the core side), i.e. AB.

The bridges communicate using the GIOP Tunneling Protocol. Each terminal has a *Home Location Agent*, i.e. HLA, in their home network. The HLA is responsible for

tracking its terminals as they move from one AB to another. In addition, the HLA uses location forwarding to redirect invocations intended for objects on terminals to their proper addresses. On terminals, all clients' invocations are rerouted to the TB.

The servers on the terminals need to create Mobile IORs for their objects. Such an IOR contains the addresses either of the HLA or of the terminal's current AB, instead of the server's IOR. The AB acts as a proxy object, using the GIOP Tunneling Protocol to handle the invocation.

MIWCO has lots of similarities to the Esperanto Broker and this is the reason why we chose it as a middleware solution to compare the Esperanto Broker with. In fact, the TB has responsibilities similar to the mobile Middleware Management daemon, the HLA has similar responsibilities to the *Home Mediator*, whereas the AB has similar responsibilities to the *current Mediator*. However, the wireless CORBA specification does not offer built-in primitives for one-to-many communication. Besides, network monitoring algorithms needed to trigger handoff procedures, are not specified, and are assigned to external components.

6.1.3 Testbed and experimental scenarios

The experimental campaign is conducted in the context of the two following scenarios:

- *Scenario I*: application objects run on mobile devices which are located in the same Domain. This means that Esperanto objects interact with the same Mediator, whereas CORBA objects running on MIWCO ORB (i.e. TBs) interact with the same AB.

- *Scenario II*: application objects run on mobile devices which are located in different Domains. This means that Esperanto objects interact with distinct

Mediators, whereas CORBA objects running on MIWCO ORB (i.e. TBs) interact with different ABs.

All experiments were performed using boxes equipped with 1GB of RAM and PIV 1.8Ghz CPUs. Linux 2.6.9 has been the platform operating system. As for mobile devices, we used Compaq IPAQ 3970 running Linux familiar v0.7.1, equipped with Wi-Fi 802.11b and Bluetooth cards. As for the core network, we have used a switched 100 MBits Ethernet. Mediators communicate to one another via TAO 1.4. During tests, the external load was the normal background load of active service daemons.

6.1.4 Performance metrics

We adopted the following performance metrics: i) method invocation's latency; ii) throughput of tuples handled by Mediators; iii) the invocation's latency during device handoffs, iv) data-link and domain handoff latencies. To compare latency measures between EB and MIWCO, we let clients and servers interact via the *request/response* paradigm. To this aim, we used the following IDL and E-IDL interfaces:

```
// OMG IDL                          // E-IDL
interface Measure {                 interface Measure {
   long foo(in string op);             reqres long foo(in string op);
 };                                 };
```

This does not represent a limitation since the *request/response* latency can be used as a fine-grained estimation for the performance evaluation of the other interaction paradigms. As for network interconnection, we used GIOP Tunneling Protocol, GTP, (for MIWCO) and NCSOCKS (for the EB) over Bluetooth.

6.2 Experiment setup

Hardware required to set up the testbed consist of two mobile devices, two fixed hosts and two access points. As for the software is concerned, both the application and the middleware code have been instrumented with probe points to get timestamps. Such timestamps have been used to estimate the following entities: i) invocation's latency (i.e. measured on the client-side as the round-trip time of a method invocation); ii) throughput (i.e. measured on the Mediator-side as the number of requests per second processed by the Bridge server). They have been evaluated as function of payload size and of number of server objects connected to Mediators.

To this aim the experimental code has been parametrized with respect to these entities; iii) invocation's latency during handoffs (i.e. measured as the average latency of consecutive remote invocations when one or more handoff does occur); iv) handoff's latency (i.e. measured as the time the Mobility Manager from when it reveals the need for a handoff and reconnect to the Mediator). The experiment collected all the timestamps at the same time.

6.3 Empirical results

6.3.1 Latencies comparison

Table 6.1 shows the latency of *request/response* invocations as function of payload size in both scenario I and scenario II. We measured latencies at both stub/skeleton (i.e. E-DOC) and Tuple Space Access Primitives (i.e. TSAP) layers. Such latencies are compared to the MIWCO's latencies.

Table 6.1 illustrates that the EB has a performance cost respect to MIWCO.

Table 6.1: Latency at different layers of EB compared to MIWCO latency

		invocation latency (msec)				
		1B	10 B	100 B	1KB	10 KB
MIWCO	scenario I	73.6957	74.0677	74.8794	122.126	411.373
	scenario II	74.0427	74.3985	74.9931	123.614	482.856
E-DOC	scenario I	169.501	172.543	174.726	193.797	588.869
	scenario II	179.161	184.147	187.065	215.9	692.236
TSAP	scenario I	167.231	171.686	173.18	192.104	587.591
	scenario II	171.118	176.848	178.973	195.446	596.521

Table 6.2: Throughput respect as function of connected objects

	throughput (requests/sec)				
	1 object	25 objects	50 objects	100 objects	150 objects
scenario I	7.41	11.30	17.60	18.04	19.65
scenario II	3.30	7.32	14.50	16.25	18.08

However, as payload increases such a gap will decrease, demonstrating that our prototype does not introduce unacceptable overhead. Moreover, the low ratio between the latency using the DOC abstraction and the latency using the tuple space layer demonstrates that building a *middleware over another middleware* does not get performance worse.

6.3.2 Throughput

Table 6.2 shows the throughput achieved by Mediators as function of the number of connected objects. Table 6.2 illustrates that the EB has a predictable behavior even in presence of high load situations and that Mediators do not represent a bottleneck of the EB infrastructure. Throughput increases as connected objects increase. However, the small number of requests per second is basically due to Bluetooth's bandwidth limitations.

Table 6.3: Latency respect as function of connected objects

	invocation's latency (msec)				
	1 object	25 objects	50 objects	100 objects	150 objects
scenario I	147.66	154.48	157.29	154.24	163.34
scenario II	169.69	160.44	162.84	167.10	176.74

Table 6.4: Latency of data-link and domain handoffs

	handoff latency (sec)			
	min	average	max	st.dev.
data-link	2.147	4.552	8.566	1.764
domain	10.017	10.388	13.041	0.826

6.3.3 Latency

Table 6.3 shows the latency of *request/response* invocations as function of the number of connected objects. The invocation's latency remains basically stable despite the number of objects connected to Mediators. This is thanks to a design implementation choice. Server objects receives their tuples via notifications to callbacks which are stored in a Standard Template Library (i.e. STL) map. Such a map offers O(1) access.

6.3.4 Mobility impact

Table 6.4 shows the minimum, maximum and average latencies for both data-link and domain handoffs.

Table 6.5 compares the minimum, maximum and average values of the invocation's latency during a data-link handoff with the respective values of invocation's latency recorded in regular situations.

Table 6.6 compares the minimum, maximum and average values of the invocation's latency during a domain handoff with the respective values of invocation's latency

Table 6.5: Invocation's latency with and without data-link handoffs

	invocation latency (sec) - scenario I			
	min	average	max	st.dev.
w/o handoff	0.156	0.229	0.763	0.087
with handoff	1.321	2.015	3.345	0.594

Table 6.6: Invocation's latency with and without domain handoffs

	invocation latency (sec) - scenario II			
	min	average	max	st.dev.
w/o handoff	0.151	0.188	0.602	0.471
with handoff	1.426	2.977	4.780	0.831

recorded in regular situations.

Tables 6.4, 6.5, and 6.6 provides useful understandings: i) invocation's latency increases during handoffs due to loss of network connectivity: the burden is greater during domain handoffs than during data-link handoffs since in the latter case no tuples need to be moved between Mediators.

However, latencies are almost stable as the analysis of min, max and average values suggests; ii) domain handoff's latency is greater than the data-link handoff, since more activities are required by the Mobility Manager to perform the device migration. However, latencies are almost stable as the analysis of min, max and average values suggests.

6.4 Analysis of results

Empirical experiments prove the attractiveness of the proposed platform. Although our first prototype has a cost in terms of performance, it shows a predictable behavior in presence of device mobility and high load situations. Invocation's latency remains

basically stable despite how many objects are connected to the platform, whereas handoff procedures introduce predictable overhead.

Chapter 7

Conclusions

7.1 Conclusions

There is an increasing demand of middleware for nomadic computing applications. Due to the inherent characteristics of such environments, these platforms have to address two fundamental issues: i) device disconnections and limitation of wireless networks may force users to experience short periods of service unavailability, and ii) the complexity to design and develop next-generation mobile computing applications.

This thesis proposed the Esperanto Broker, a communication platform for Nomadic Computing systems. The platform's design faced major mobility issues: it coped with disconnections due to device mobility, and it let applications be decoupled according to the disconnected *lifestyle* of mobile computing. Developers are provided with advanced services to support their effort in designing next-generation mobile applications.

Mobility issues like disconnections, variations in network performances and mobile device constraints are needed to be dealt with mobility management procedures and strategies. Almost all current platforms do not address the above mentioned issues

via any effective solution as the Esperanto Broker does. It addresses mobility issues via an integrated approach, i.e. both at data-link, network and middleware levels.

Device disconnections and degradations in network performances affect the ability of an application object to be available for communicating with counterparts. To improve such an availability, objects should be provided with decoupled communication paradigms. Although any solution analyzed provides a decoupled communication paradigm, either it looses expressiveness of the computing model, or it still adopts sort of synchronous interaction primitives. According to the Esperanto Broker approach, decoupling interactions are achieved via a tuple-space underlying infrastructure. To support developers with advanced services, the Esperanto Broker enhances the *distributed objects computing* model providing the abstraction for the communication paradigms standardized by the W3C.

In other words, the Esperanto Broker joins *remote method invocations* and *tuple space* together to exploit their advantages respectively. It also provides mobile-enabled services such location-aware and group-aware services. Esperanto applications are modeled as sets of objects that are distributed over mobile devices which communicate via remote method invocations (RMIs). RMIs natively implement *pull* and *push* models, in both one-to-one and one-to-many multiplicity. None of the considered alternatives proposes such a computing model.

A powerful computing model, and especially advanced tools, aid the designers and developers to build applications, as ESERV does. It simplifies the process of designing object interfaces and make the code generation faster. None of the considered solution provides tools for design and development similar to it.

As far as heterogeneity of wireless technologies is concerned, the common traditional approach to deal with this is to provide an abstraction layer, which hides the underlying technologies and deal with them separately. The Esperanto Broker implements handoff strategies that allow the device to be connected to the core network seamlessly despite the wireless technology.

Finally, as far as mechanisms to allow interoperability are concerned, Esperanto clients may invoke web services and, vice versa, web client may invoke Esperanto servers. Since *Web Services* are becoming the standard *de facto* in developing and deploying distributed services, the decision to allow interoperability with the Esperanto Broker and any other middleware solution seemed a good way to achieve it. Eventually any middleware solution shall be interoperable with *Web Services*.

This thesis has discussed the design and the implementation of the Esperanto Broker. The thesis also focused on evaluating performances of the implemented prototype and the effectiveness of its usability factor.

The EB has all the building bocks to develop and deploy next-generation mobile computing applications. The developer is shielded from low-level details, group communication and mobility mechanisms, and can focus only on the application business logic using a powerful computing model. As for the design phase, the developer is provided with a high-level tool, i.e. ESERV, which allows to draw interfaces and the involved communication paradigms without any effort in knowing the language's syntax.

As for the development phase, he/she is required to write only the business logic code, since all the skeleton code, i.e. makefiles, and stub/skeleton classes are generated automatically. As for the deployment phase, he/she is required to configure the

particular Esperanto network infrastructure, plug in the needed access points, and run commands to start daemons, on the mobile-side, and Mediators on the fixed-side.

Several under-graduated projects have proved that students dealt with Esperanto as good as they did with traditional middleware like CORBA. Esperanto has a GUI tool to generate and compile E-IDL interfaces, and other GUI tools (i.e. utilities to configure both mediators and the esperanto network infrastructure) may be developed to further improve its usability factor.

From a methodological point of view, we believe that the experience brings two important benefits to developers community: i) it has shown that the Esperanto Broker may provide an effective mean for supporting applications running over a Nomadic Computing environment, and ii) it has provided most of the implementation techniques we used, which can help middleware developers to understand how to integrate mobility mechanisms in current middleware implementations. From an experimental point of view, the performance evaluation provides an evidence of the platform attractiveness.

Bibliography

[1] R. Bagrodia, W.W. Chu, L Kleinrock, and C. Popek. Vision, issues, and architecture for nomadic computing. *IEEE Personal Communications*, 2(6):14–27, 1995.

[2] L. Kleinrock. Nomadic computing and smart spaces. *IEEE Internet Computing*, 4(1), 2000.

[3] L. Kleinrock. On some principles of nomadic computing and multi-access communications. *IEEE Communications Magazine*, 38(7), 2000.

[4] A. Gaddah and T. Kunz. Does modern middleware address mobile computing requirements? *In Proc. of the 8th World Multi-Conference on Systemics, Cybernetics and Informatics*, 5:493–499, 2004.

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

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

[7] Microsoft. DCOM tecnology, 2004. www.microsoft.com/com/tech/DCOM.asp.

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

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

[10] 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.

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

[12] Jaakko Kangasharju. *Implementing the Wireless CORBA Specification*. PhD thesis, Computer Science Department - University of Helsinki, 2002. http://kotisivu.mtv3.fi/ashar/software/miwco/laudatur-jjk.pdf.

[13] A. L. Murphy, G. P. Picco, and G. C. Roman. LIME: A Middleware for Physical and Logical Mobility. *In Proc. of 21st IEEE Int. Conf. on Distributed Computing Systems*, pages 524 – 533, 2001.

[14] S.S. Yau and F. Karim. A Lightweight Middleware Protocol for Ad Hoc Distributed Object Computing in Ubiquitous Computing Environments. *In Proc. of 6th IEEE Int. Symp. on Object-Oriented Real-Time Distributed Computing*, pages 14–16, 2003.

[15] W3C. Web Services Activity, 2006. http://www.w3.org/2002/ws/.

[16] L. Kleinrock. Nomadicity: anytime, anywhere in a disconnected world. *Source Mobile Networks and Applications archive*, 1(4), 1996.

[17] Microsoft. Tablet PC on the Go: Scenarios for Powerful Mobile Computing, August 2004.

[18] M. Burgess and S. Fagernes. Pervasive Computer Management: A Smart Mall Scenario Using Promise Theory. *Lecture Notes on Computer Science*, 3775, 2005.

[19] Cristiano di Flora. Service Discovery and Delivery in interoperable Nomadic Computing Systems. Master's thesis, Università degli Studi di Napoli Federico II, 2005.

[20] Graziano Almerindo. Achieving Secure Service Provision in Nomadic Computing Systems. Master's thesis, Università degli Studi di Napoli Federico II, 2005.

[21] P.T. Eugster, P. Felber, R. Guerraoui, and A.M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computer Survey*, 2003.

[22] M. Franklin and S. Zdonik. A Framework for Scalable Dissemination-based Systems. In ACM Press, editor, *SIGPLAN: ACM Special Interest Group on Programming Languages*, pages 94 – 105, 1997.

[23] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, Volume 7 Issue 1:pages 80–112, 1985.

[24] IBM. Tspace home page, 2004. www.almaden.ibm.com/cs/TSpaces/index.html.

[25] S. Adwankar. Mobile CORBA. *In Proc. of 3rd IEEE Int. Symp. on Distributed Objects and Applications (DOA '01)*, pages 17–20, 2001.

[26] M. Haahr, R. Cunningham, and V. Cahill. Supporting CORBA Applications in a Mobile environment. *In Proc. of the 5th annual ACM/IEEE Int. Conf. on Mobile Computing and Networking*, 1999.

[27] ¡. Π^2 A Generic Proxy Platform for Wirelss Access and Mobility in CORBA. *In Proc. of the 19th ACM Symp. on Principles of distributed computing*, pages 191–198, 2000.

[28] S. Campadello, O. Koskimies, K. Raatikainen, and H. Helin. Wireless Java RMI. *in Proc. of 4th Int. Enterprise Distributed Object Computing Conference*, 2000.

[29] 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.

[30] OMG. Wireless Access and Terminal Mobility in CORBA Specification, 2001.

[31] M. Liljeberg, K. Raatikainen, M. Evans, S. Furnell, N. Maumon, E. Veldkamp, B. Wind, and S. Trigila. Using CORBA to Support Terminal Mobility. *in ACM Proc. of the Global Convergence of Telecommunications and Distributed Object Computing*, 1997.

[32] N. Davies, A. Friday, S. P. Wade, and G. S. Blair. L2imbo: a distributed systems platform for mobile computing. *Mobile Networks and Applications*, 3(2):pages 143 – 156, 1998.

[33] P. R. Pietzuch and J. M. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. *In Proc. of 22nd IEEE Int. Conf. on Distributed Computing Systems Workshops (ICDCSW '02)*, 2002.

[34] A. D. Joseph, J. A. Tauber, and f. Kaashoek. Mobile Computing with the Rover Toolkit. *IEEE Transactions on Computers*, 48(3):337 – 352, march 1997.

[35] T.S. Rappaport S. Shakkottai and P.C. Karlsson. Cross-layer design for wireless networks. *IEEE Communications Magazine*, 41(10), 2003.

[36] M. Cinque, D. Cotroneo, and S. Russo. Achieving All the Time, Everywhere Access in Next-Generation Mobile Networks. *ACM SIGMOBILE Mobile Computing and Communication Review (MC2R) Journal*, 9(2):29–39, 2005.

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

[38] 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.

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

[40] Y. B. Lin and A. C. Pang. Comparing soft and hard handoffs. *IEEE Trans. on Vehicular Technology*, 49(3):792–798, May 2000.

[41] J. Lansford, A. Stephens, and R. Nevo. Wi-Fi (802.11b) and Bluetooth: Enabling coexistence. *IEEE Network*, pages 20 – 27, September/October 2001.

[42] 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.