CLM and NCSOCKS: a technical report

M. Cinque, F. Cornevilli, D. Cotroneo[*], A. Migliaccio, S. Russo and G. Virnicchi

The Mobilab Group - www.mobilab.unina.it
Dipartimento di Informatica e Sistemistica, Università di Napoli Federico II
Via Claudio 21, 80125 - Napoli, Italy

1 Overview

This report provides the technical details of the CLM (Connection and Location Management) components and NCSOCKS (Nomadic Computing Sockets) API developed by the Mobilab Group. The aim of this work is supporting the application developer with a middleware architecture providing an API specifically suited for Nomadic Computing environments. Nomadic Computing (NC) is referred in literature as a distributed computing model in which the communication takes place over strongly heterogeneous network infrastructures. Such infrastructures are composed of one or more wireless domains, glued together by a fixed infrastructure (the core network), and provide anytime, anywhere access to mobile devices [1]. In this kind of infrastructure, the mobile terminals communicate each other through the core network which is accessed by means of Access Points (APs). Each AP, with its covering zone, defines a cell in which a mobile device can communicate with the fixed infrastructure.
The proposed architecture deals with two major issues:

- Mobility Management, i.e. all the activity related to support users with mobile terminals to enjoy their services through wireless networks when they are moving into a new service area [2]. Mobility management involves handover management, that is all the activity related to manage the wireless network connection in spite of device movements from a cell to another. Since different APs can use different wireless communication technologies (e.g. Bluetooth [3], Wi-Fi [4], IrDA [5], etc.), also vertical handover procedures (i.e. handover between AP of different techs [6]) have to be addressed.

- Location Management, i.e. the management of the current mobile device location. By location, we mean the symbolic location, that encompasses abstract ideas of where something is [7]: in the kitchen, in the lab, next to a picture in a museum.

Through the NCSOCKS interface, application developers can support their NC applications with:

- Connection Awareness: applications can be aware of the current state of the connection. This kind of awareness is especially recognized in NC environments, where

[*]Contacts D. Cotroneo (**cotroneo@unina.it**) to obtain CLM and NCSOCKS source code.

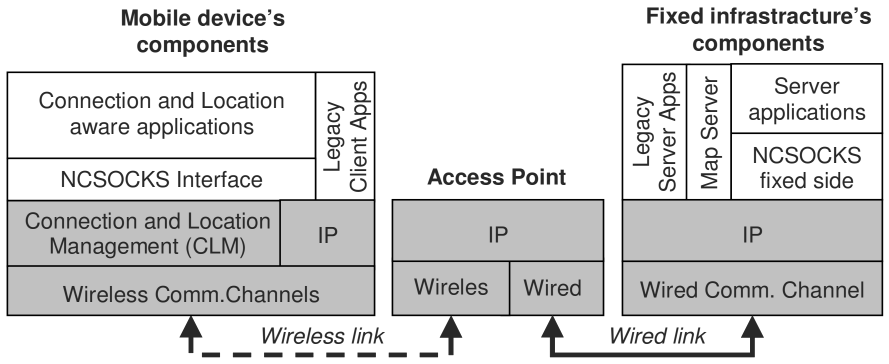

Figure 1: Abstract machine model of our solution

the disconnections are the rule, rather than the exception. Thus the application can adapt their behavior on the basis of the current state of the connection.

- Location Awareness: applications can be aware of the current device symbolic location, thus they can adapt their behavior accordingly to the current location.

In the following sections, the overall architecture, its peculiar aspects, and implementation details, are deeply described.

2 Overall Architecture

The architecture we propose is depicted in figure 1. Three are the major components that build the architecture: the Connection and Location Management (CLM), the Map Sever, and the Nomadic Computing Sockets (NCSOCKS) Interface.
The former run on mobile terminals and it is in charge of handling connections with the APs. In particular, it is responsible of performing handover operations between different cells. To this aim, CLM uses information about the current location of the mobile device by keeping a map of the neighboring APs. Such a map is managed by the Map Server, which run on the core network. The Map Server provides the current APs topology and the technology being used by each AP (e.g., Bluetooth, Wi-Fi, and Irda). The CLM we implemented is a software module placed between application and kernel space. It provides a set of communication services, by means of a user-level library (the NCSOCKS Interface), allowing applications to request an IP-based communication channel. In particular, the CLM layer provides applications with location and connection awareness support. Using NCSOCKS library enables applications to be notified about changes in the channel status, in terms of raised exceptions. By this way, applications can perform the correct actions, adapting their behavior according to the connection status (connection awareness) and the symbolic location (location awareness).

2.1 CLM Layer

As described earlier, the CLM component is in charge of implementing handover procedures and of providing the current location information. The handover procedure is typically composed of three phases [8]:

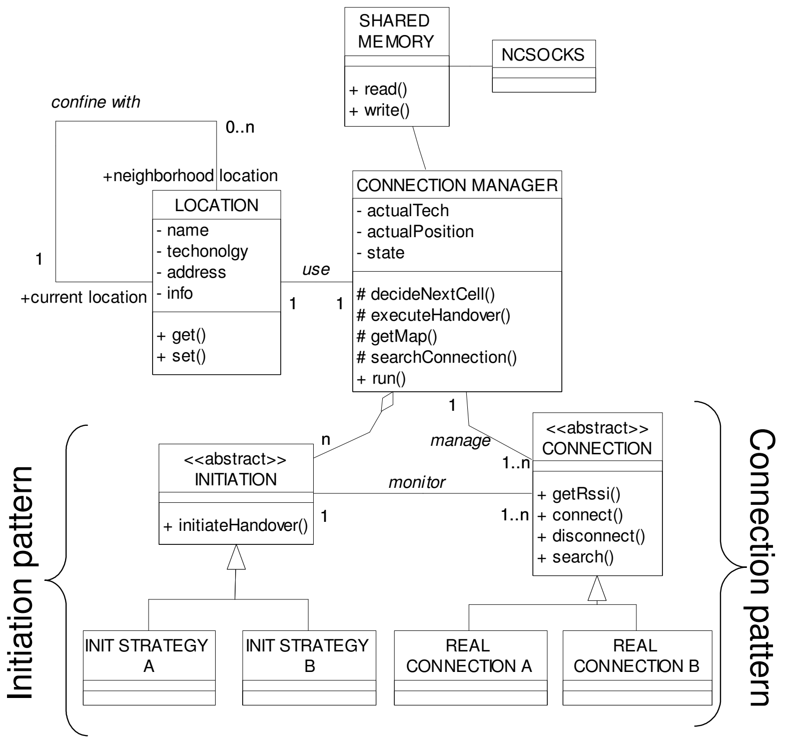

Figure 2: CLM Design Pattern

- Initiation: the objective of the initiation phase is to recognize the need for handover. To this aim, the network status is monitored in order to decide when the migration has to be started;
- Decision: once the need for handover is recognized, it is necessary to decide the new AP to connect;
- Execution: in this phase, a connection is established to the new AP, which have been discovered in the previous phase, and the information about the current location is updated.

The initiation phase is strongly dependent on technology being used. For instance, in the case of a Bluetooth cell, in which the mobile device acting as a slave can manage at most one connection, the initiation phase can use only the information about the signal strength of the actual connection, both in the case of horizontal and vertical handovers. In order to design a solution which is independent of the technology, we adopt a strategy pattern [9], using n different initiation strategies for n different kind of cells. In the decision phase, a new AP is chosen. The decision can be taken by monitoring some parameters of the wireless link between mobile device and APs (namely the Receiver Signal Strength Indicator - RSSI). In order to reduce the number of APs to be inquired, a predictive handover scheme is adopted. This scheme uses the environment topology map, where the next cell is chosen among the neighbored APs. The topology map is used also for locationing purposes: the `Connection Manager` knows the mobile device symbolic location by knowing the AP currently used. During the handover, a new AP is chosen (among the old AP neighborhood) and the location information is updated. The decision algorithm

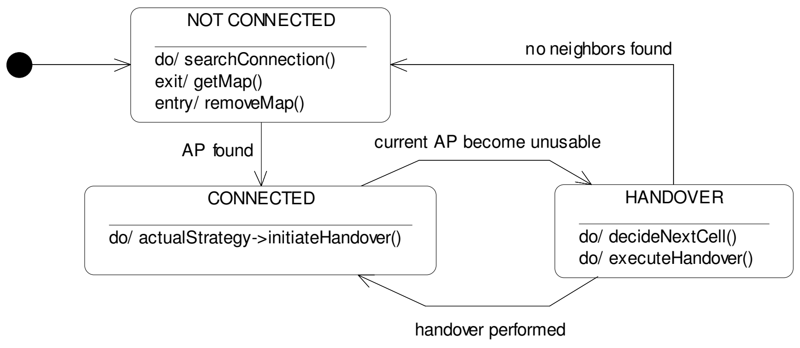

Figure 3: CLM state chart diagram

has to be implemented in order to choose an AP covering the zone in which the device is currently present.
We here present the design of the CLM layer as a design pattern. This let us to propose a general solution which can be adopted by designers facing with similar problems. The class diagram of CLM is depicted in figure 2. The `Connection Manager` holds information about the connection status, actual location, and technology being used. The connection status can assume three distinct values: i) *NOT CONNECTED*, i.e. the device is not connected to any AP; ii)*HANDOVER*, i.e. the mobile device is performing an handover procedure; and iii) *CONNECTED*, i.e. the device is connected to an AP. Two strategy patterns are adopted, namely *Initiation* and *Connection*. The former aims to define a family of initiation algorithms, encapsulate each one, and make them interchangeable in spite of technology being used. The latter provides a connection abstraction, in terms of connections, disconnections, AP discovering, and signal strength monitoring procedures. The topology map is implemented by means of `Location` objects and built by invoking the `getMap()` method, which has the effect of requesting the overall map to the Map Server. The `Location` objects hold also the information about the technology adopted by each cell. This information allows the `Connection Manager` to use the initiation strategy (through the `initiationStrategy()` method) and the connection primitives accordingly to the actual cell technology. In order to shade some light on the `Connection Manager` behavior, figure 3 depicts its UML state chart diagram.

2.2 NCSOCKS Layer

The NCSOCKS interface provide the applications with transport communication facilities. Since they have been designed in order to deal with nomadic settings, the NCSOCKS are responsible of the maintenance of transport channels in spite of lower layers changes. Indeed, through the NCSOCKS, applications can communicate with the fixed infrastructure also in spite of device disconnections, due to signal absence or handover being performed. These facilities are provided by the interaction between the NCSOCKS and the CLM. The NCSOCKS handle the packet delivering by knowing the current connection status and, in particular circumstances, also the applications can be notified about changes in the connection status. At the same time, applications can request the NCSOCKS in order to read, or to be notified, about the current device symbolic location.

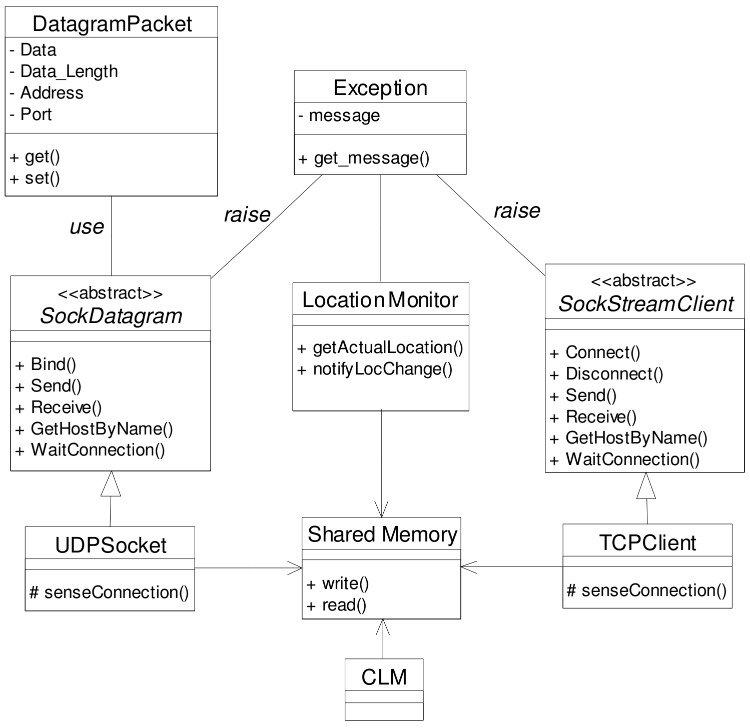

Figure 4: NCSOCKS class diagram

Figure 4 depicts the NCSOCKS class diagram. As figure shows, the API is similar to the standard Java sockets library, except for the following: firstly there is a class, the `Location Monitor`, that provides the applications with the awareness of the actual symbolic location; the applications can request the actual location, and can be notified about location changes; the current location is provided by the CLM layer trough a shared memory. Secondly, communication methods adopt a synchronization paradigm taking into account the connection awareness support. In order to explain such a synchronization paradigm, the most significant methods are described in the following:

- `senseConnection() : status` - As the name suggest, the `senseConnection()` method performs a connection status sensing. The status information are gathered through a shared memory, written by the `Connection Manager` and containing two fields: the *state* of the connection (CONNECTED, NOT CONECTED and HANDOVER) and a counter, *refconn*, indicating the number of connections that have been made since the CLM has been started-up. Each new connection establishment (after an handover or a reconnection due to a quite long channel unavailability), causes *refconn* to be incremented. A `senseConnection()` method call causes the following actions to be performed: i) the shared memory contents are read; ii) if the NCSOCKS local memorized connection number is less than *refconn*, then the transport level services are restarted (e.g. trough a sockets re-initialization); indeed, in this case, some disconnections have been occurred, causing sometime the change of the IP interface being used (e.g. after a vertical handover); iii) the local memorized connection number is adjusted to *refconn*; iv) the read connection *state* is returned.

- `Send() : int` - the `Send()` primitive provides a data transport service and return the number of delivered bytes. If the communication cannot succeed, exceptions are thrown indicating the connection status (e.g. *connection unavailable* or *connection in progress*, that is a too long handover operation). The connectionless implementation uses UDP; in this case, the `Send()` C++ signature is:

  ```
  int UDPSocket::Send (DatagramPacket & packet, int persist_duration =
          8, int persist_interval = 2) throw (Exception)
  ```

  The PDU exchanged is a `DatagramPacket` object, which encapsulate the destination host IP address and port, the payload and its length. As far as the connection oriented implementation is concerned, it uses TCP, and the `Send()` C++ signature is:

  ```
  int TCPClient::Send (const string & packet, const int length, int
  persist_duration = 8, int persist_interval = 2) throw (Exception)
  ```

  The PDU exchanged is a string containing the message payload. Obviously, there is no need to specify the destination address. Both the connectionless and connection oriented solutions, are designed in order to check the wireless connection availability before the data delivery. After the check, three situation might occur:

  1. The wireless connection is available: data are immediately sent;
  2. The wireless connection is unavailable: a *connection unavailable* exception is thrown to the application;
  3. An handover is being performed: the `send()` waits for `persist_duration` seconds and control the connection status every `persist_interval` seconds. If in this time the connection became available, data are sent, otherwise, if the `persist_duration` time is exceeded or if the connection becomes unavailable, a *connection in progress* or a *connection unavailable* exception is thrown.

  In the following, the send primitive core C++ simple code is shown:

  ```
  switch (senseConnection()) {
        case CONNECTED :
             data delivering //through TCP or UDP
             break;
        case HANDOVER :
             if (WaitConnection(persist_interval, persist_duration,
                STATE_ DISCONNECTED + STATE_CONNECTED) == 0)
                      throw (CommException("Connection in progress"));
             else return(Send(packet));
             break;
        case DISCONNECTED :
             throw (CommException("Connection unavaible"));
             break;
  }
  ```

Hence the `Send()` primitive uses the `WaitConnection()` primitive described below.

- `WaitConnection (const int interval, const int dur, const int trigger): int` - this primitive allows the applications to wait for an event, for a time `dur`, specified by the `trigger` parameter, completely enabling the connection awareness support. The event can be the passage to a CONNECTED status (`trigger` = STATE_ CONNECTED), NOT CONNECTED status (`trigger` = STATE_ DISCONNECTED), or from an HANDOVER status to a different one (`trigger` = STATE_ CONNECTED + STATE_ DISCONNECTED). The `interval` parameter specifies how often the status has to be requested during the `dur` time.

- `Receive() : int` - the `Receive()` primitive provides a non-blocking (with timeout) data reception service, and return the number of effectively received bytes. Compared with the `Send()` primitive, the handover situation is handled in a different manner: since it makes no sense to await the connection availability (the expected packet is probably already lost), the effective reception is performed only if the connection is available. The connectionless implementation provides the following interface:

  ```
  int UDPSocket::Receive(DatagramPacket & packet,int timeout) throw
                                (Exception)
  ```

  whereas, in the connection oriented implementation, the interface becomes:

  ```
  int TCPClient::Receive(string & packet, const int length, int
                      timeout) throw (Exception)
  ```

  Both the connectionless and connection oriented solutions, implement the non-blocking behavior by using the Unix `select` system call.

2.3 Map Server

The Map Server is a fixed-side component. It provides the mobile devices `Connection Manager` with the current environmental topology map. Since each environment has its topology, each environment has to have a Map Server, capable of accepting map requests from mobile terminals. Furthermore, the topology may change (an AP may fail, or the system administrator may decide to add or remove a covered zone), so the Map Server has to recognize topology changes and provide the mobile terminals with the last updated map version. To this aim, the Map Server is designed as a set of distributed objects: one object per AP (`AP monitors`), which are in charge of performing the AP monitoring, and one `Central Manager`. Through the `Central Manager` user interface, the topology can be managed by the system administrator, whereas the `AP monitors` notify the manager about current APs status, thus the topology changes not handled by human administrators are automatically handled by the Map Server.

3 Design and Implementation Details

The CLM and NCSOCKS components have been implemented by using the C++ language and the Linux OS (kernel 2.4.19 or above) primitives. The code have been compiled both for PCs and laptops (having installed a Linux mandrake 9.1 distribution) and cross-compiled for the Strong ARM target machine in order to be run on Compaq IPAQ 3970 (with a Linux Familiar 0.7.1 distribution). Current development efforts are involving the Map Server implementation and the java-porting of the NCSOCKS library. The Map Server is being implemented using java and JacORB (www.jacorb.org), a java implementation of the CORBA specification. As far as the implementation is concerned, our CLM has been conceived as a daemon process running on mobile devices. Communication between users applications and the CLM is made possible through NCSOCKS by using shared memory IPC system calls, which are encapsulated in the SharedMemory class. The following sections describe some key details of the CLM design and implementation, that is the integration of the Bluetooth and Wi-Fi technologies, the IP channel configuration and the description of our decision algorithm designed to address also locationing issues.

3.1 Integrating the Bluetooth Technology

3.1.1 Connection

The Bluetooth connection is achieved by a specific `PANConnection` class, which implements the `Connection` abstract class. Such a class is in charge of creating *IP over L2CAP* channels based on Bluetooth Personal Area Network profile (PAN) and Bluetooth Network Encapsulation Protocol (BNEP) [3]. The Bluetooth channel access and power monitoring is implemented by using BlueZ, the official Linux Bluetooth stack implementation. On the AP side, we implemented a specific daemon process, namely `NAPdaemon`, which is in charge of accepting incoming PAN connections and bridge them upon a unique virtual datalink interface, as suggested by BlueZ implementors (`http://bluez.sourceforge.net /contrib/HOWTO-PAN`). To this aim, the Linux kernel we use has to be compiled with Bridge Control (brctl 802.1d) option. Behind the AP we enable a Network Address Translantion Server (NAT), allowing bluetooth-enabled devices to use private IP network addresses. Further information about the `NAPdaemon`, its source code and installation guidelines can be found at `www.mobilab.unina.it/BlueNAPHOWTO.htm`.

3.1.2 Initiation Strategy

The Bluetooth technology do not support neither horizontal nor vertical handover procedures, where horizontal handovers are intended as handover procedures between APs using the same technology. We thus manage both handover types in the same manner, that is, when the mobile terminal leave a Bluetooth cell it will be able to reconnect either to another Bluetooth enabled AP or to an AP using some other technology. In our solution, Bluetooth APs are Bluetooth master device, whereas mobile terminals are slaves. This assumption is needed to overcome some limitations of the current Bluetooth hardware implementations, in which slave devices can manage at most one connection at the same time. We decide also to not monitor AP of technologically different cells at the same time in order to reduce the interferences near the mobile device (e.g. using Bluetooth and Wi-Fi at the same time can result in high interferences [10]). With this assumption, only the RSSI of the actual connection is monitored, reducing the battery consumption.

A problem that arise with this kind of assumption is that the initiation can be affected by transient RSSI degradation, due to temporary interferences and shadowing. For this reason, we propose a count and threshold scheme, namely alpha-count scheme [11], capable of discriminate transient and permanent RSSI degradations. We define the alpha-count function as follows:

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

During the L-th measurement, if RSSI falls below the threshold S_{RSSI}, the value of the $\alpha^{(L)}$ function is incremented, otherwise the value is decremented by a positive quantity *dec*. An handover is initiated when $\alpha^{(L)}$ becomes greater than a threshold α_T , which represents the tolerance of the algorithm. This means that, the value of such a parameter has to be carefully tuned in order to achieve a trade-off between the availability and accuracy of the location mechanism. Indeed, α_T along with values of *dec* and S_{RSSI} indicates which is the zone covered by an AP. So each AP has to have its characteristic parameters to be held in its `Location` object.
The proposed initiation strategy have been implemented through a specific `InitBlue` class that implements the abstract `Initiation` class.

3.2 Integrating the Wi-Fi Technology

3.2.1 Connection

The Wi-Fi connection is achieved by a specific `WiFiConneciton` class, implementing the `Connection` abstract class. Since the IP abstraction is already provided by Wi-Fi adapters, this implementation was straightforward. The RSSI monitoring have been performed by using the source code of the `iwconfig` command, a Wi-Fi configuration tool for Linux.

3.2.2 Initiation Strategy

Since the Wi-Fi technology automatically performs the horizontal handover, the initiation phase has only to recognize the need for a vertical handover. Hence, the RSSI is monitored with the α-count scheme and, when the α function overcome the threshold (hence, horizontal handovers are not capable of improving the RSSI level), a vertical handover is performed. However, during the Wi-Fi utilization, it is needed to keep track of the current location of the device. This task is actually implemented by keeping track of the current Wi-Fi AP being used. Further improvements can be obtained by implementing triangulation techniques among Wi-Fi APs. Indeed, whereas Bluetooth sensors cover little zones that are likely representative of the current device symbolic location, Wi-Fi APs cover bigger zones, resulting in a low locationing accuracy.

3.3 IP addresses configuration

Since the network layer is abstracted trough IP, allowing also the portability of legacy client applications, the IP addresses have to be assigned. The address assignment must be performed after a vertical handover (after which a different network interface will be used) and after an horizontal handover in the case of technology that does not automatically handle it. The configuration is performed using automatic techniques, such as DHCP or Zero Conf IP (`http://files.zeroconf.org`). This means that, after the handover operation, the IP address might change. However, if the mobile-side applications are clients of stateless servers residing on the fixed infrastructure, the IP address changing do not represent a problem. This is the normal situation for a wide class of legacy client-server application, such as UDP applications or some TCP applications (SMTP mail upload and POP or IMAP mail retrieval, web browsing) that are characterized by TCP connections being short enough to make the cost of having to re-attempt an operation (e.g. an HTTP transaction that was aborted due to handover or the download of a single email) relatively small. In any case, the implementation of mobile and/or state-full servers, requires the adoption of proxy-based solution (e.g. providing the servers mobility support and session management using the Session Initiation Protocol - SIP [12]). To this aim, the NCSOCKS can be adopted as the core communication infrastructure between the mobile clients/servers and the proxy, residing on the core network.

3.4 Decision Algorithm and Locationing Issues

The decision algorithm is encapsulated in the `decideNextCell()` method of the `Connection Manager`. In the actual implementation, we adopt a topology-based schema. The decision algorithm is in charge of electing a new AP among the neighbored APs. The decision is taken by using a score criteria: let $N = \{ng_1, ..., ng_n\}$ to be the set of neighbored APs. For each $ng_i \in N$ a score $s(ng_i)$ is evaluated on the basis on some parameters (RSSI, delay, delay variation, information loss). The decision algorithm selects the AP ng^* with the score $s^* = max_{ng_i \in N}\ s(ng_i)$. As for the locationing issues, we assume that a mobile device is in the room x when it is attached to the AP x. The initiation phase, with a fine tuning of its parameters, assures that when a mobile device leaves a room, an handover will be performed. The decision algorithm assures that when a mobile device enter in a room x, with an AP x, it is quite certain that the AP x will be chosen. In fact, the score parameters used by the algorithm are strongly influenced by the distance between the device and the AP and the presence of walls (the norm is that neighbors of the old AP are shadowed each other by means of walls), as studies, such as [10], confirm. For this reason, since the AP x is the closest and not covered by walls, its score will be better. However, even if pathological situations can lead to the decision of a wrong AP, poor values of the signal strength, which are measured on the selected AP, will result in the initiation of a new handover,thus correcting the error.

4 Conclusions and Future Works

This work presented a communication architecture for Nomadic Environments. We presented the design of the proposed architecture used a pattern-oriented design approach. Indeed, our effort has been progressed from the design to the implementation issues, trying to make design independent on technology details. The objective of future activity will be thorough evaluation of the effectiveness of the communication service, both in terms of robustness and performance. Further analysis are needed to assess the decision process by implementing the score criteria decribed in section 3.4. Finally, we plan to use our communication library as communication service core in our enhanced distributed object computing middleware for nomadic environments, namely ESPERANTO.

References

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

[2] J. Z. Sun and J. Sauvola. Mobility and mobility management: a conceptual framework. *Proc. 10th IEEE International Conference on Networks, Singapore*, pages 205 – 210, 2002.

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

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

[5] P. J. Mogowan, D. W. Suvak, and C. D. Knutson. *IrDA Infrared Communications: an Overview.* www.irda.org.

[6] C. Lebre, R. Titmuss, and P. Smyth. Handover between heterogeneous IP networks for mobile multimedia services. *Exploiting Mobility Conference*, pages 25 – 31, September 1998.

[7] J. Hightower and G. Borriello. Location systems for ubiquitous computing. *Computer*, 34(8):57–66, August 2001.

[8] P. Reynolds. Mobility management for the support of handover within a heterogeneous mobile environments. *3G Mobile Communication Technologies, 2000. First International Conference on (IEE Conf. Publ. No. 471)*, pages 341–346, 27-29 March 2000.

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

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

[11] A. Bondavalli, S. Chiaradonna, F. Di Giandomenico, and F. Grandoni. Threshold-based mechanisms to discriminate transient from intermittent faults. *IEEE Transactions on Computers*, 49(3):230 – 245, March 2000.

[12] E. Wedlund and H. Schulzrinne. Mobility support using SIP. *Proceedings of the 2nd ACM International Workshop on Wireless Mobile Multimedia (WoWMoM'99)*, 1999.