Technische Universität Carolo-Wilhelmina zu Braunschweig

Institut für Nachrichtentechnik

Schleinitzstraβe 22, 38106 Braunschweig

Studienarbeit:

Automatic connection configuration and service discovery for small devices using Java ME CLDC

Ivan Bella López

Oktober 2008

Betreuer: Dipl.-Inform. Jan Sonnenberg

Declaration of authorship

I hereby declare that I have written this thesis on my own and without any help from others. I certify that I have mentioned all information sources and other aids used and that I have indicated them respectively.

Braunschweig, 14.10.2008 Ivan Bella López

Abstract

Nowadays the newest mobile devices are enabled to run Multimedia and to communicate via Wifi technologies. Though the connection with the powerful mobile devices is implemented is not in Connected Limited Devices Configuration (CLDC) devices which cover an important market share.

Although the configuration complexity for the connection establishment tends to be transparent to the user most of Wifi communications like Bluetooth are built following a service-specific profile (depending on the technology or brands) and the consequence is the lack of communication between them.

In this project a software application for CLDC devices has been developed providing automatic configuration, service discovery process and media files exchange using an open generic framework to guarantee a service-general profile.

Thanks to my family and friends for their support, to Jan Sonnenberg and Professor Ulrich Reimers to let me develop my thesis with them.

“Caminante no hay camino, se hace camino al andar.”

“Wanderer there is no way, you make the way as you go.”

Antonio Machado (1875-1939)

Table of contents

1. Introduction

The increase of device connectivity capacity for the latest mobile devices is converging into an interconnected world. The lack of the independent software implementation and the capabilities of limited devices (CLDC devices) for connectivity create a gap which is needed to solve. To offer independent software built on a common API and totally transparent to the user (automatic connection and service discovery) jointly whit data exchange is the main aim of this research, the task to determine a suitable technology and implement an application for connectivity between CLDC devices and other devices.

To carry out the software, technologies which provide automatic connection and service discovery were studied, compared and chosen taking into account the capabilities of CLDC devices. The field of use of these technologies covers from a car infotainment scenario, industrial applications, domotics… Finally, a software application has been developed in order to demonstrate the applicability of the chosen technology.

2. Theoretical part

2.1. Introduction

In order to choose a technology is mandatory to do a research to determine which technology is most suitable to implement. The studied technologies (Jini, JXTA, UPnP and Web Services) are useful for the project purpose due to provide automatic connection and service discovery and provide some kind of data exchange. In addition, these technologies are in development so can be improved to offer better resources. Moreover, all of them are well known open protocols.

For each technology, and overview and basic features are explained. Nevertheless, the technology explanation is focused on the service discovery process and the connection between the devices. This section is divided in four steps:

- Boot: where clients and services attempt to initiate the discovery process.
- Advertising: where a service provider publishes information about its services
- Request: where a client looks for a desired service
- Response: where a client receives the means to get the requested service.

2.2. JINI

2.2.1. Historical approach

Jini has its origins with the David Gelernter's and Nick Carriero's work that created the Linda[i] concept. A decade ago they used Tuple Spaces[ii], the base to develop Java Spaces[iii] and Jini. The Jini Community was initially established in January 1999 with the release of the first version of the Jini Technology Starter Kit from Sun Microsystems. The last version of the Jini Technology Starter Kit has been released in February 2004.

Jini technology is being employed in a wide range of applications that require the ability to respond automatically to changes in their operating environments. Some scenarios exemplify Jini technology uses cases like compute grids.

The many problems to which Jini technology-based compute grids are currently being applied include modelling, simulations, data processing, rendering, and pattern matching.

2.2.2. Overview

Jini comes up as a Java extension, which provides the construction of secure and distributed systems consisting of federations. It is used to build adaptive network systems which are scalable and flexible. Jini enables services to be added or deleted from a federation at any time according to demand. Moreover, Jini brings object-orientation to the network. Jini use Java RMI[iv] (Java Remote Method Invocation), which provides code mobility and event notification mechanism that allows objects to be triggered. The details are transparent to the users and they only need to know the interface of the service.

The Lookup service (called “Reggie”) is one of the main parts of the Jini system. The Lookup service, which is a Jini service, can be viewed as a directory service where services are found and resolved. The registered services in the Lookup service are the base of the Jini structure. The Lookup service keeps available the service information in the Jini federation.

Jini consists on a trio of protocols: discovery, join, and lookup. A Jini service uses the discovery protocol (see 2.2.3) to announce its presence on a network. When a service finds a Lookup service (or is founded by a Lookup service) the service register itself in the Lookup service sending the specific software code that provides an implementation of the service interface (the service proxy) and other information about the service. This process is the join protocol (see 2.2.3.2). Finally, if a client wants to find a service (see 2.2.3.3), a request is made to the Lookup service. Once the

required service has been discovered, the interface implementation is downloaded to the client that can then access the service using Java RMI.

The use of a service is controlled via the Jini leasing mechanism. A service is requested for a time period and, then, granted for negotiated period between the service user and provider. This lease must be renewed before its expiration. Otherwise, the resources associated with the services are released. Leasing enables the Jini systems to be robust and maintenance-free. For example when a device leaves the group or fails abruptly without having a chance to deregister properly.

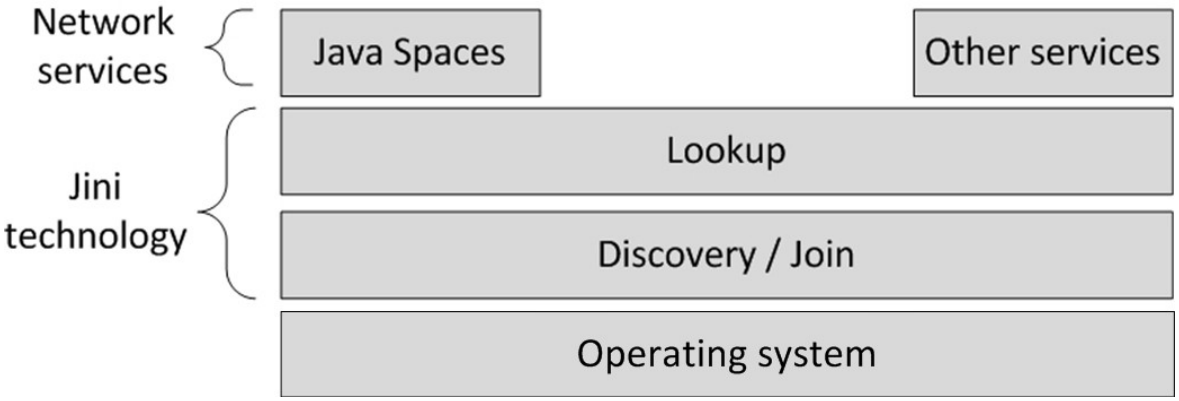


Figure 1. Jini architecture

Based on: http://www.gigaspaces.com/wiki/download/attachments/9994725/IMG958.gif

2.2.3. Protocols

2.2.3.1. Discovery Protocol

Through this protocol, Jini clients find Jini services and services find Lookup services. The Jini platform includes multiple mechanisms for Service Discovery:

- Multicast Request Protocol: is used when an application or service first becomes active, and needs to find Lookup services in the vicinity.
- Multicast Announcement Protocol: is used by Lookup services to announce their presence to the services that may have interest in the community.
- Unicast Discovery Protocol: is used to establish communications with a specific Lookup service known a priori.

2.2.3.2. Join Protocol

Once a Lookup service is located the service provider loads a service object into it. This service object contains the Java programming language interface including the methods that users and applications will invoke to execute the service, along with other descriptive attributes.

2.2.3.3. Lookup Protocol

Jini services locate each other through specific lookup mechanisms. The Lookup process revolves around Lookup services such as Reggie, which keep the information of the services available in a group (djinn[v]).

Once a Lookup service is discovered by a client (see 2.2.3), the client retrieves a proxy from it and uses it to perform lookups for the services. A client performs a lookup specifying a Java interface that the desired service must implement. The client can also include attributes that provide more details about the service. It is up to each service, when it registers with a Lookup service, to provide values for entries that clients may use.

In response, the Lookup service returns one or more proxies for services that match what the client asked for. At this point, the client has successfully looked up services and the service object is loaded into the client. The final stage is to invoke the service.

2.2.4. Service discovery

2.2.4.1. Boot

When clients and service providers start try to discover the Lookup service which is associated to the djinn they are. This can be done via multicast or unicast depending on the situation (see 2.2.3).

2.2.4.2. Advertising

Once a Lookup service is found, services register with it. A registered record is an instance of the `serviceItem` Java class shown below:

```
public class ServiceItem implements Serializable {
```

```
        Public ServiceItem (
                    ServiceID serviceID,
                    Object service,
                    Entry [] attributeSets)
        {...}

        public ServiceID serviceID;
        public Object service;
        public Entry[] attributeSets;
}
```

A service registration contains the service assigned UUID (`serviceID`), a proxy-object (`service`) and a set of attribute value pairs describing the service (`attributeSets`).

2.2.4.3. Request

When a Lookup service is found by a client, client's queries are sent to that Lookup service using RMI. A query object is an instance of the following `ServiceTemplate` Java class:

```
public class ServiceTemplate implements Serializable {
      public ServiceTemplate(
                  ServiceID serviceID,
                  Class[] serviceTypes,
                  Entry[] attributeSetTemplates)
      {...}

      public ServiceID serviceID;
      public Class[] serviceTypes;
      public Entry[] attributeSetTemplates;
}
```

All the fields describe the requested service. The field `serviceID` refers to the UUID of the requested service. The `serviceTypes` include the list of Java classes the service may be an instance of, and `attributeSetTemplates` is a set of attribute-value pairs describing the requested service. The Lookup Service matches the query against its records.

If no Lookup service is found, a client can act in its place using the Peer Lookup mechanism. Basically consists on emulate the packet is used by a Lookup service to request services to register. It is only used to discover services. It is never used like a Lookup service and this mechanism never responses to other clients which are looking for a Lookup service.

2.2.4.4. Response

The Lookup service sends every instance which matches with the client request. The one which has a proxy-object is used to invoke the service via RMI.

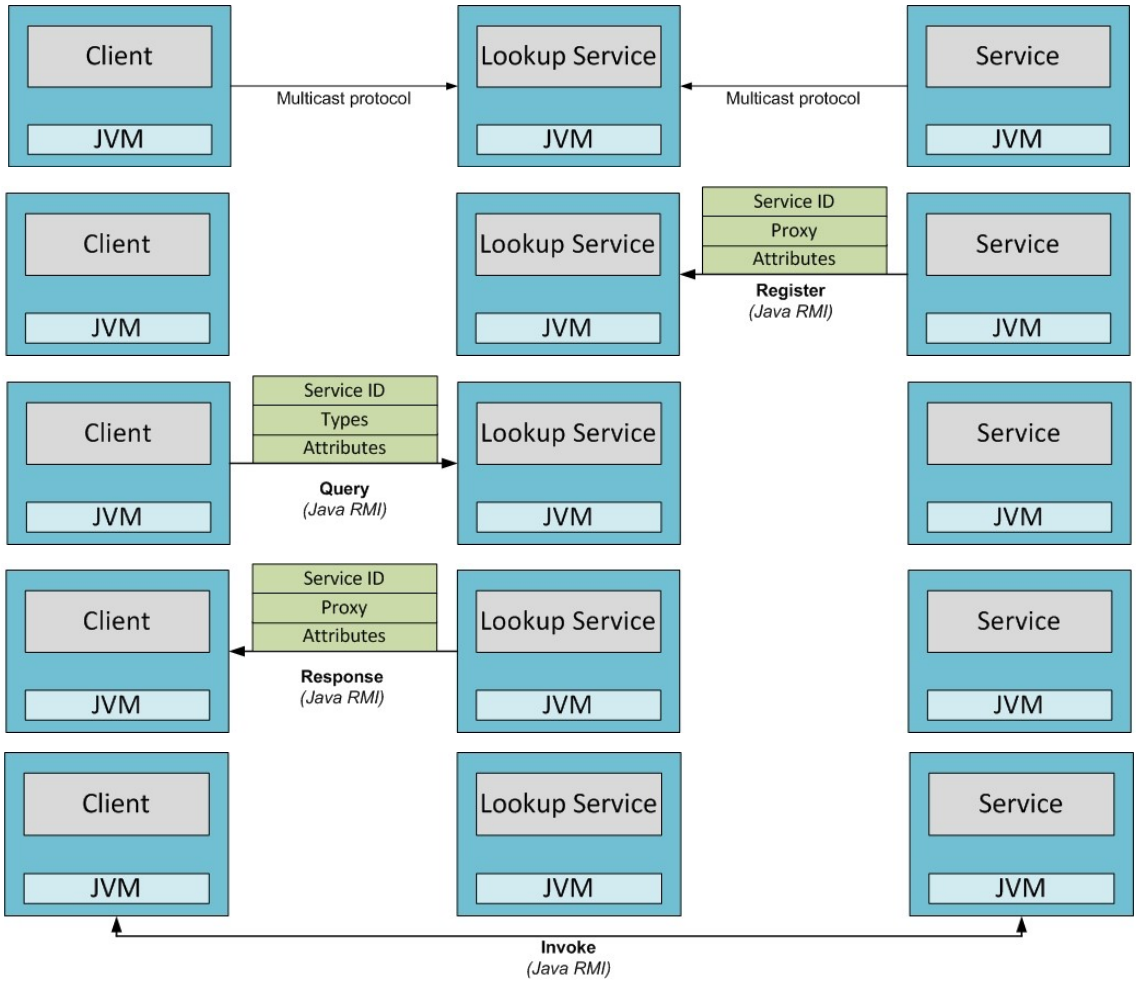


Figure 2. Jini service discovery steps

Based on: http://www.sun.com/software/jini/whitepapers/architecture.html

2.2.5. Summary

Jini has a full dependence on Java to enable its targets. It assumes that devices support JVM (Java Virtual Machine) even though a Jini-proxy can be used for a cluster of limited devices (see Jini Surrogate Protocol).

Jini's service proxy concept is one of strongest features not found at Universal Plug and Play (UPnP), JXTA or Web Service. The no-need-for driver scenario presumes the Jini

devices to support standard interfaces are already available in the network. It means all manufactures of a certain device type must consent to the standard interface.

2.3. JXTA

2.3.1. Historical approach

JXTA is an open source peer-to-peer (P2P) protocol specification begun by Sun Microsystems. It was announced by Sun in April 2001 being the first to include standard protocols and multi-language implementations. Sun remains actively involved in the development of JXTA. The last version 2.5 was published in November, 2007.

There are over 17,000 members of Project JXTA's community, many of which are using and continually improving the technology. Nokia, Verizon, Brevient, CodeFarm and Zudha are using JXTA technology.

Nokia is using JXTA to create a peer-to-peer server network for their Automated Network Services and network appliance management. Zudha, based in India, created a P2P instant messaging application. The application called ZIM-Pro.

2.3.2. Overview

JXTA provides a peer distributed model to discover each other and interact, and an infrastructure to establish communications between peers. The targets of JXTA are to build an interoperable, platform independent and ubiquitous network. JXTA is based on a three-layer model (see 2.3.4) and has six protocols (see 2.3.3) for P2P communication.

Through these protocols, peers using different transport protocols and hardware platforms, and built by different languages, are allowed to discover each other, communicate, organise into the necessary structures and monitor activity.

JXTA has provisions for security. Security mechanisms are based on TLS[vi] (Transport Layer Security), digital certificates and certificate authorities.

JXTA peers create a virtual network where any peer can interact with other peers and resources directly, even when some of the peers and resources are behind firewalls, NATs[vii] (Network Address Translation) or on different network transports. Peers send messages through pipes, which are an abstraction over the actual network layer. Pipes are uni-directional, so there are in and out pipes, and end-points can be bound to multiple peers.

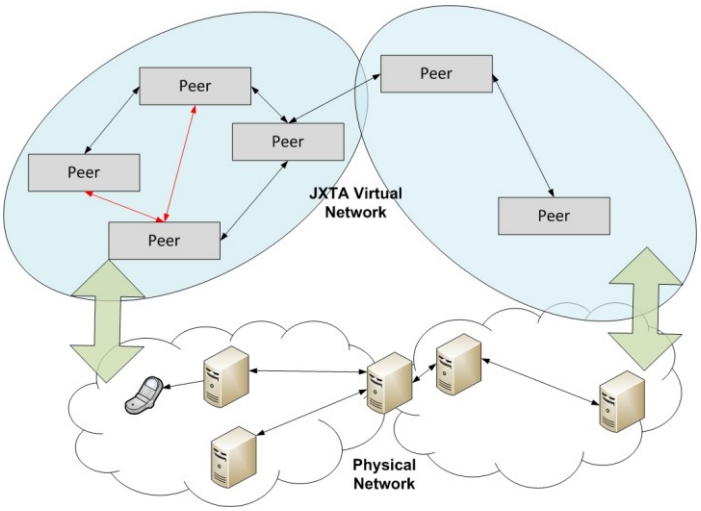


Figure 3. JXTA virtual network

Any peer wishes to make a service available on a JXTA network needs to create an advertisement for the service. An advertisement is coded in XML and it announces the existence and some properties of a peer, a peer group, or a pipe. The advertisement allows other peers in the same peer group to find it, using a standardized search mechanism, until the expiration time of the advertisement has passed. When a peer finds an advertisement, it usually puts it in its local cache. Other peers can retrieve it from there as well as retrieving the advertisement from the actual service provider. This mechanism provides additional redundancy and scalability of JXTA networks.

There are two main types of peers:

- Edge Peer
    - Minimal Edge Peer: does not cache advertisements or route messages, e.g. PDAs, cell phones.
    - Edge Peer: caches advertisements, replies to discovery requests but does not forward them.

- Super Peers
    - Rendezvous Super Peer: caches advertisements and forwards discovery requests.
    - Relay Super Peer: maintains routing information about peers that cannot directly connect to other peers.

2.3.3. JXTA protocols

2.3.3.1. Peer discovery protocol

The Peer discovery protocol defines how peers advertise their own capabilities and discover others. This protocol is used for resource queries.

2.3.3.2. Peer resolver protocol

The Peer resolver protocol defines a request/response protocol that allows a peer to send queries to a specific peer, multiple peers or to the Peer Group and receive responses to its query.

2.3.3.3. Peer information protocol

The Peer information protocol allows a peer to obtain information about other peers.

2.3.3.4. Peer binding protocol

The Peer binding protocol allows a peer to establish a virtual communication channel (i.e. pipe) with one or more peers by binding endpoints.

2.3.3.5. Endpoint routing protocol

The Endpoint routing protocol is used to determine the route between two endpoints. If no direct link exists, routing is made through other peers.

2.3.3.6. Rendezvous protocol

The Rendezvous protocol allows messages to be propagated within a Peer Group. In a Peer Group, peers can be either rendezvous peers or peers that are listening to rendezvous peers. Rendezvous peers propagate messages to the associated peers.

2.3.4. JXTA architecture

The line between the layers is not fixed. Depending on the situation, one peer might see a functionality as a service and another peer might see it as a complete application.

2.3.4.1. Application Layer

The application layer hosts code that pulls individual peers together for a common piece of functionality. For the JXTA specification and related bindings to be successful, developers need to fill out the application layer.

2.3.4.2. Service Layer

This layer implements services that are integrated to JXTA. Services include searching and indexing, protocol translation, authentication and others. JXTA services implement session, presentation and application layers in the OSI model.

2.3.4.3. Platform Layer (JXTA Core)

The core implements a minimal set of primitives. Primitives include discovery, transport, creation of peers and peer groups and others. The JXTA core implements transport, network and data link layers in the OSI model. The core services are:

- Membership service. Used by the current peer group members to reject or accept a new group membership application.
- Discovery service. Provides peers, groups and other advertisements.

- Monitoring service. Allows a peer to get information about another peer's status.
- Resolver service. Used to send queries throughout the P2P network. It is used by other services like Discovery service, to distribute a discovery query to peers involved in the same group.
- Endpoint service. Addressing mechanism used by peers to communicate with others. Endpoint service allows a communication to be established between peers using different communication protocols.
- Pipe service. Provide a virtual connection between peers involved in a communication.
- Rendezvous service. Act as a proxy for queries over the network. Indexes over Peer Group advertisements can be distributed among the Rendezvous peers.

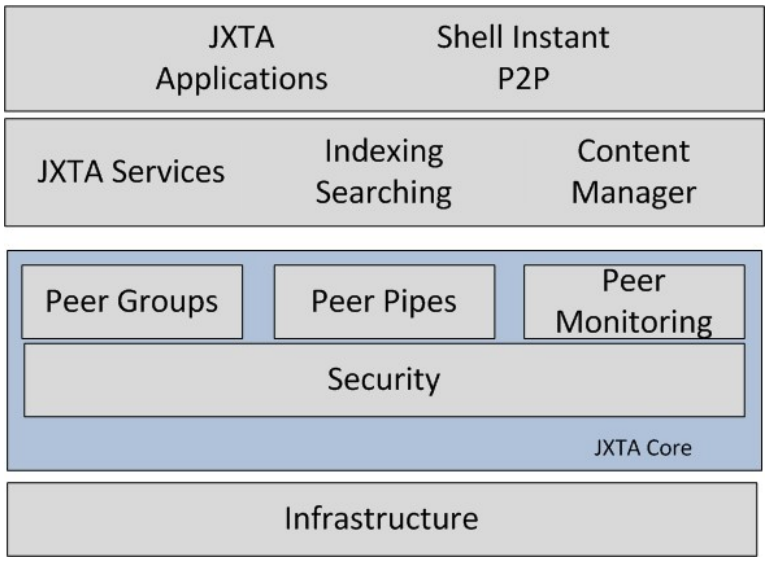


Figure 4. JXTA architecture

Based on: http://www.2dconcept.com/articles/2/architecture.gif

2.3.5. Service discovery

2.3.5.1. Boot

On start-up, JXTA try to join into a Peer Group. Firstly, a peer belongs to a generic Peer Group (`WorldPeerGroup` or `NetPeerGroup`) which is a generic implementation of JXTA standard. If the group accepts the peer subscription, the peer obtains an authenticator

that allows it to join the group. Finally, the peer obtains the Membership service as implemented by the group and a credential that the peer will use to authenticate its messages.

2.3.5.2. Advertising

The services are defined by modules. JXTA associates three kinds of advertisement to each module: Module Class Advertisement (description of the service behaviour), Module Specification Advertisement (means to invoke the service) and Module Implementation Advertisement (reference to an implementation of the service). JXTA-enabled services are typically published using a Module Specification Advertisement. The advertisement has to specify a communication channel through a pipe advertisement that must be used by a peer to invoke the service.

2.3.5.3. Request

Firstly the peer tries to resolve its queries locally. If the query cannot be resolved locally it is then encapsulated in a Peer Resolver query to be sent over the Peer Group. When a Peer Group implements the Rendezvous Protocol, queries are sent to Rendezvous Peers. These peers are most likely to cache an index over the requested service advertisement. Rendezvous peers propagate the query to associated peers, using the Pipe Service. This service abstracts the underlying communication model. If the Rendezvous peer cannot resolve the query, it propagates it among other known Rendezvous peers. Queries are mapped to the cached advertisements in a way that is not specified by the JXTA specification. Peers respond to the query with a maximum number of responses as specified in the query.

2.3.5.4. Response

Peers receive a certain number of service advertisements encoded in XML. Depending on the advertisement type the peer will handle the service in a certain way.

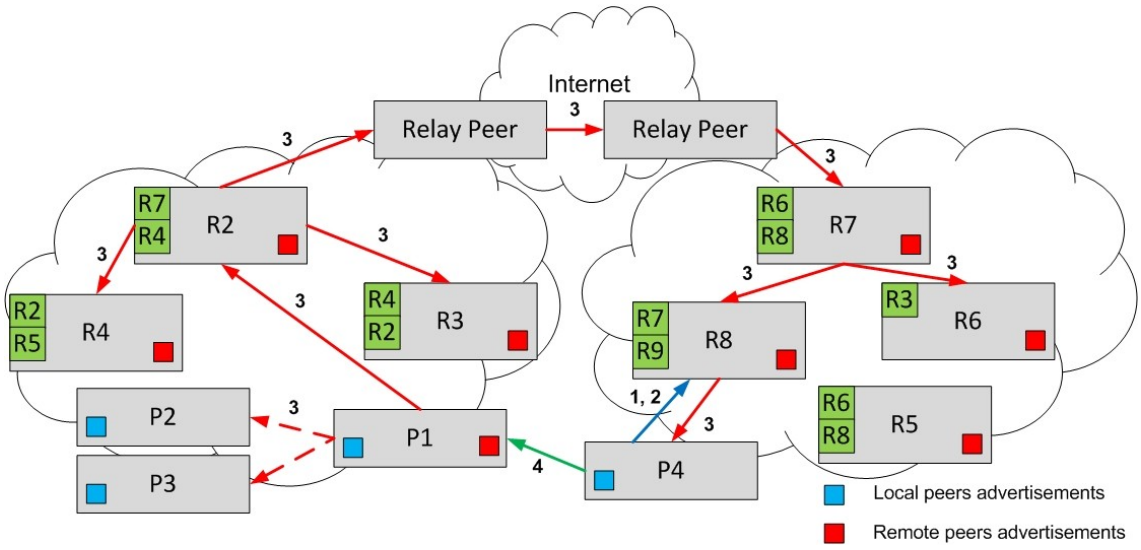


Figure 5. JXTA discovery process

Based on: http://www.buet.ac.bd/cse/users/faculty/reazahmed/cse6809/discovery-ieee.pdf

2.3.6. Summary

JXTA is the only discussed P2P technology. JXTA technology is designed to enable peers providing various P2P services to locate each other and communicate with each other.

A virtual network overlay is used to allow peers to interact directly and organize independently of their location.

Also, is designed to be independent of programming languages and deployment platforms. JXTA is accessible not only for PCs; CLDC/MIDP/CDC devices are enabling to join JXTA technology using JXTA-JXME[viii].

2.4. UPnP

2.4.1. Historical approach

UPnP was published primarily by Microsoft and Intel and is maintained by the UPnP Forum which was founded in 1999. In August 2008, the UPnP Forum involve more than 863 leading companies in computing, printing and networking, consumer electronics, home appliances, automation, control and security and mobile products. The first

version of the UPnP device architecture was released in 2000. On September 2008, UPnP Device Architecture Version 1.0 was approved as ISO standard. Also, UPnP is supported by Windows XP and Windows Vista. In Windows Vista, UPnP is integrated with PnP-X and Function Discovery.

Nowadays is being common to find UPnP-enabled devices. Today UPnP can already be used at home to share music, video and photos with compatible devices like Pinnacle, D-Link, Philips, Microsoft… This interconnectivity is done by UPnP server applications as Microsoft's Windows Media Connect or the Philips Media Manager software.

2.4.2. Overview

UPnP technology provides a decentralized, open networking architecture that uses TCP/IP and Web technologies. UPnP devices announce themselves and their supported services on the network. Control Points, which act on the consumer's behalf, catch the announcements and initiate queries based on consumer's needs. In UPnP, a device can dynamically join a network, obtain an IP address and learn about the presence and capabilities of other devices. A device can leave the network automatically without leaving any unwanted state behind. UPnP features can be divided in six steps:

2.4.2.1. Addressing

In addressing, a device receives an IP address either through DHCP[ix] or Auto-IP (see 2.4.3.1).

2.4.2.2. Discovery

When a device is added to the network, the device advertises its services to Control Points. Similarly, when a Control Point is added to the network, UPnP offers methods for the Control Point to search for devices (see 2.4.3.2).

2.4.2.3. Description

After a Control Point has discovered a device the Control Point retrieves the device's description from the URL provided by the device in the discovery message. The UPnP

description is expressed in XML and includes a list of any embedded devices or services, as well as URLs for control, eventing and presentation.

2.4.2.4. Control

Once a Control Point has the device description, the Control Point can send actions to a service device. To do this, a Control Point sends a suitable control message to the control URL for the service. Control messages are also expressed in XML using the Simple Object Access Protocol (SOAP). As function calls, in response to the control message, the service returns action-specific values.

2.4.2.5. Eventing

The service can publish its updates (by sending event messages). A Control Point may subscribe to this updates. Event messages contain the names of one or more state variables and the current value of those variables. These messages are also expressed in XML and formatted using the General Event Notification Architecture[x] (GENA) which can be unicast over normal HTTP or encapsulated in SSDP (Simple Service Discovery Protocol) HTTP over multicast for group notification.

2.4.2.6. Presentation

If a device has a URL for presentation, the Control Point can retrieve a page from this URL, load the page into a browser and depending on the capabilities of the page, it allows a user to control the device and/or view the device's status. It is an optional step.

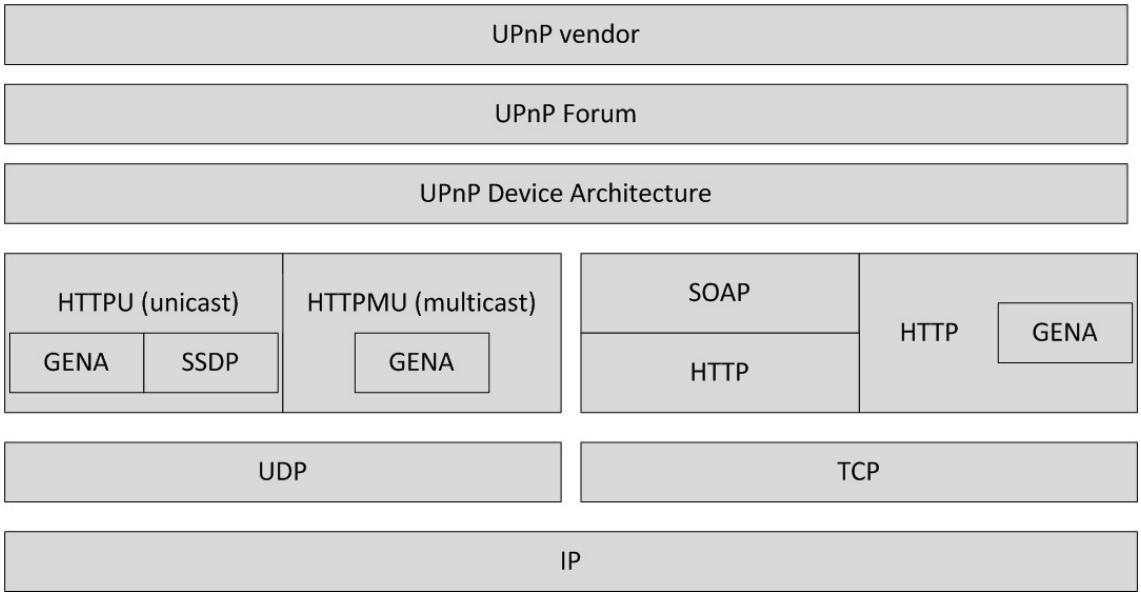


Figure 6. UPnP Stack

Based on: http://technet.microsoft.com/en-us/library/Bb457049.upnpxp04_big(en-us,TechNet.10).gif

2.4.3. Service discovery

2.4.3.1. Boot

In contrast with Jini and JXTA, UPnP acts on a lower level. UPnP assign an IP address in start-up. A subset of Web Service, Device Profile for Web Service (See 2.5.3), acts on this level too. When an UPnP device is plugged in, it will attempt to obtain an IP address. The specification recommends the use of DHCP or Auto-IP. Firstly, it tries to get an IP address from a DHCP server. If no DHCP server is reached, Auto-IP is used. Auto-IP enables a device to join the network without any explicit administration. Auto-IP chooses a random IP address in the range “169.254.x.x/16” and verifies that it is not already assigned (by broadcasting Address Resolution Protocol (ARP) requests).

2.4.3.2. Advertising

UPnP uses SSDP to discover and to announce services. SSDP uses HTTP over multicast (HTTPMU) and unicast UDP (HTTPU). When a device is added to the network, the device advertises itself to the Control Points. No central directory or lookup service is used. A discovery message contains few and basic information about the device or its services. These advertisements contain typically the type of the advertised service or device, and a URL where a detailed description can be found.

2.4.3.3. Request

Requests may be multicasted by Control Points over the network. They are typically based on the type of the requested device or service. Standard device and service types are assigned by a naming authority. When a device receives a query, it matches it against itself and its embedded services and devices. The responses to queries are unicasted.

2.4.3.4. Response

Similarly to Jini, UPnP provides an event notification mechanism to trigger service updates. The lack is security is not addressed by the UPnP design.

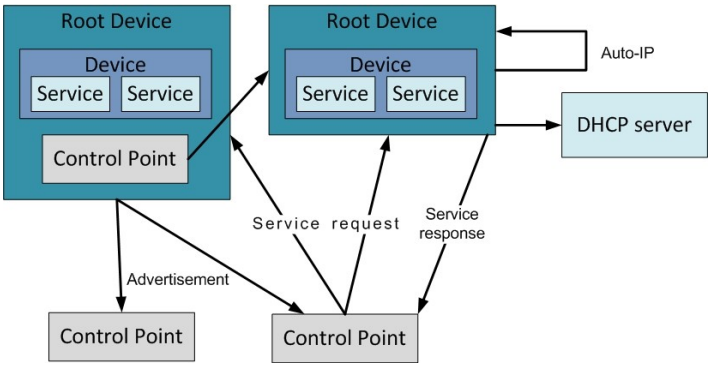


Figure 7. UPnP discovery process

2.4.4. Summary

The independence of programming language and platform, the variety of UPnP-enabled devices and the support of big companies makes UPnP be a promising technology to be consider.

UPnP is the unique studied technology which supports address assignment using Auto-IP or DHCP which let UPnP acts in a lower level and covers from address assignment (see 2.4.3.1) to service presentation (see 2.4.2.6).

2.5. Web Service

2.5.1. Historical approach

Web Service came up for a standardization communication needed between different platforms and programming languages.

Previously some attempts of standards like DCOM[xi] and CORBA[xii] were created. Even though, it did not become a success due to vendor-dependent implementations.

Another problem was the RPC[xiii] (Remote Procedure Call) use. It created several security problems and it has the disadvantage that its implementation in Internet was almost impossible (due to firewalls). Finally, Web Service arose to enable interoperability between different platforms.

Several parties began to consider a new version of the standard, the one which nowadays uses XML, SOAP, WSDL (Web Service Definition Language) and UDDI (Universal Description Discovery and Integration). Web Service was standardized by World Wide Web Consortium (W3C). It defines Web Service as "a software system designed to support interoperable machine-to-machine interaction over a network".
Nowadays Web Services are widely used between machine-machine communications and recently a subset came up, Device Profile for Web Service (DPWS) which is supported by Microsoft and Windows Rally Technologies.

2.5.2. Overview

Web Services provide a standard communication way between different software/framework/platforms. Web Service is based on XML, SOAP, and WSDL. Universal Description, Discovery and Integration (UDDI) is used for Web Service discovery. A Web Service is accessible through SOAP.

Operations can perform a specific task or a set of tasks. It is described using a standard, formal XML notation, called its service description that provides all of the details needed to interact with the service, including message formats, transport protocols, and location. Web Service descriptions are expressed in WSDL.

The architecture consists on three roles (service provider, service requester, service registry) and three steps (publish, find, and bind). A service provider creates a Web Service and its service definition and then publishes the service with a service registry based on a standard called the Universal Description, Discovery, and Integration (UDDI) specification.

Once a Web Service is published, a service requester may find the service via the UDDI interface. The UDDI registry provides the service requester with a WSDL service description and a URL. The service requester may then use this information to directly bind to the service and invoke it.

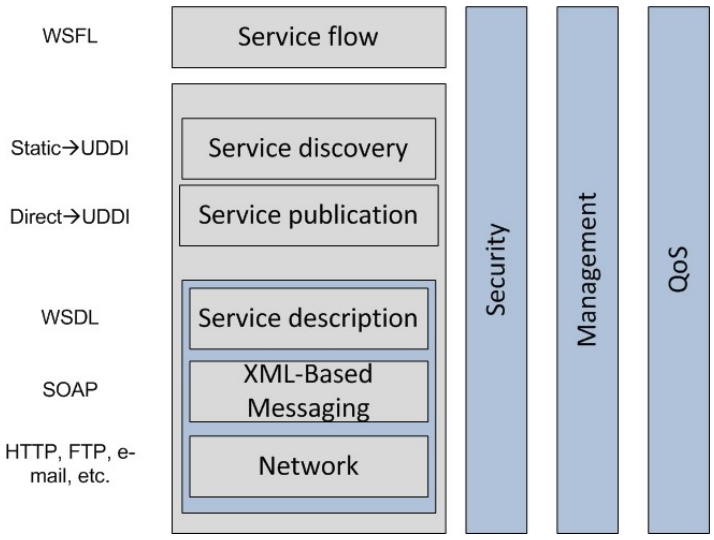


Figure 8. Web Service stack

http://www.idealliance.org/papers/xml2001/papers/html/images/03-02-03/Web%20Services%20Stack.jpg

2.5.3. Device Profile for Web Service (DPWS)

DPWS brings Web Service functionalities (Web Service messaging, discovery, description, and eventing) to resource-constrained devices. It enables the use of existing tools provided by Web Service platform providers for developing DPWS based devices and easier integration with existing Web Service deployments, Windows technologies and UPnP. It defines an extensible metadata model for describing the characteristics of devices. It provides guidance on security.

There are a number of possible usage scenarios for the DPWS like automation, domotics… The DPWS specification was initially published in May 2004 and has been submitted for standardization in July 2008.

DPWS uses SOAP, WS-Addressing, and MTOM/XOP for messaging. It supports SOAP-over-HTTP and SOAP-over-UDP. It uses WS-Discovery for discovering a device (hosting services), WS-Eventing for optionally managing subscriptions to. It uses Web Service Description Language (WSDL) to describe the device and WS-Transfer to retrieve service description and metadata information about the device.

WS-Discovery provides a Web Service based lightweight dynamic discovery protocol to locate Web Service. It does not require specialized network intermediaries to aid discovery. It is transport independent; it may be used over HTTP, UDP, or other transports.

2.5.4. Service discovery

2.5.4.1. Boot

The bootstrapping step is not addressed; the location of the directory is assumed to be known by the consumers and service providers.

2.5.4.2. Advertising

Service providers build the WSDL before registering it with the directory. In UDDI, the registered WSDL document contains information about the organization that provides the service and the service itself (e.g. name, access points, and operations). This is done according to the UDDI data model. There are several basic components inside the data model like `BusineesEntity` or `BusinessService` and `tModel`. UUID keys can be assigned to each business entity, service, binding template, or `tModel`.

2.5.4.3. Request

Queries follow the XML format. Only key lookups with basic qualifiers are supported for querying in the UDDI registry. Responses to queries are ensured to be accurate and complete since the registry has the full control over the information stored inside the registry.

2.5.4.4. Response

The result of querying in Web Service is often a list of all service descriptions (in XML format). Services then can be invoked through the service handles or access points defined in the UDDI entry. These access points usually take the form of URLs.

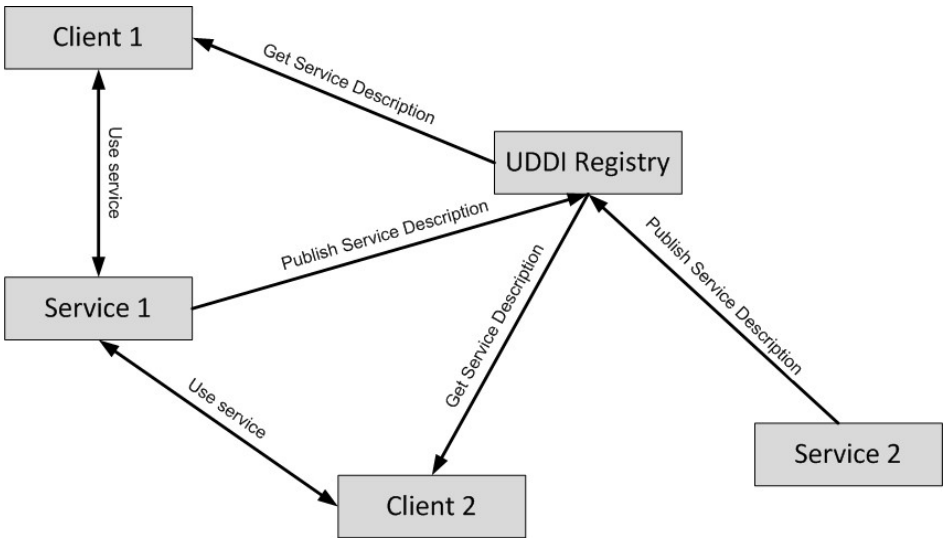


Figure 9. UDDI discover mechanism

2.5.5. Summary

Web Service is a high level technology very used in machine-to-machine communication and the only studied technology which its directory architecture is centralized. The acceptance and standardization of Web Service make it ubiquitous. Unlike Web Service are commonly used in Internet its field of use can cover more scenarios; it can be useful in other situations as smart homes or industrial automation. Moreover with DPWS which oriented for devices its market share can grow more.

2.6. Technology comparison

The table below sum up some not mentioned features between the described technologies before.

If we compare from the point of view of programming language and platform independence JXTA, UPnP and Web Service are the most independent technologies and Jini is fully-build in Java.

The two Sun technologies, Jini and JXTA let code mobility but without the support of the Java platform of Jini is limited; JXTA, on the other hand, is completely independent of Java, JXTA uses pipes to invoke. UPnP and Web Service use SOAP (UPnP GENA too).

The way they communicate, the architecture used, the scalability, the directory architecture… sometimes can be successful in some scenarios or not appropriate in others. So, in this point we can not talk about a winner or a loser or we can not determine which the best technology is. Depending on the scenario, purposes, limitations, goals to achieve, requirements… some technologies take advantage from others.

	Jini	JXTA
Programming language	Java	Independent
Code mobility	Yes	Yes
Platform dependence	Java platform	Independent
Network dependence	IP	IP
Security	Yes	Yes
Standardization	Yes (de facto)	Yes (Standard)
Centralised/Distributed Architecture	Partial	Distributed
Address assignment	Not supported	Not supported
Name resolution	Not supported	Not supported
Service discovery	Multicast request protocol (MRP)	Advertisements
Service description	Java interfaces	Modules
Service invocation	Java RMI	Pipes
Service presentation	Java Classes	Not supported
Scalability	LAN scale	Internet scale
Directory architecture	Distributed (DA)	Distributed (Local cache)

	UPnP	WS
Programming language	Independent	Independent
Code mobility	No	No
Platform dependence	Independent	Independent
Network dependence	IP	IP
Security	No	Yes
Standardization	Yes (standard)	Yes (de facto)
Centralised/Distributed Architecture	Partial	Varies
Address assignment	DHCP / Auto IP	Not supported
Name resolution	DNS	Not supported
Service discovery	SSDP	UDDI
Service description	XML	XML / WSDL
Service invocation	SOAP / GENA	SOAP
Service presentation	HTML (optional)	HTML (optional)
Scalability	LAN scale	Internet scale
Directory architecture	Distributed (Local cache)	Centralized

3. Practical part

3.1. Introduction

Before to start thinking about which technology we should implement for devices communication is convenient to specify some targets. Though the primary objective is to provide automatic connection configuration, service discovery and file exchange with Java ME, it is important to define other objectives and bear in mind the CLDC limitations in order to try to rule out technologies. Thus, it is easier to find the best solution since the aim of the project can be achieved by most of the technologies; they differ in the way they work and its features.

The practical part consists on develop an application which provides connectivity between CLDC devices and other devices. The target of the practical part is to choose a technology that is especially applicable to connect CLDC devices and for which an appropriate API is available. Afterwards, to develop an application to demonstrate the capability of the chosen technology is mandatory. The application should achieve at least the following targets:

- To provide transparent connectivity between devices. The user does not need to configure anything before the application starts. It is done automatically by the application.

- To discover the available devices/services. The application will look for the services of other devices which implements the same API to. Also, the application has to be able to retrieve the information not only of one device at once, it has to be able to scan the network and retrieve the information about the services available in all devices connected.

- Automatic connection. The connection with the required service or device has to be automatic. It means, the user selects a desired service and the software will connect automatically without any other user intervention.
- File exchange capability. The application has to provide some kind of multimedia file exchange.

3.2. CLDC and MIDP overview

J2ME currently defines two configurations: CLDC and CDC. We will focus on its capabilities and restrictions.

3.2.1. The Connected Limited Device Configuration

Nowadays, there are two versions:

- CLDC 1.0 (Java Specification Request (JSR) 30)
- CLDC 1.1 (Java Specification Request (JSR) 139)

The following table explain what is defined and not defined in CLDC specification:

Defined	**Not defined**
The capabilities of the Java Virtual Machine (JVM)	Any APIs related to user interfaces
A very small subset of the J2SE 1.3 classes	How applications are loaded, activated or deactivated
A new set of APIs for input/output called the Generic Connection Framework.	

3.2.1.1. The Java CLDC Virtual Machine

The JVM used in the CLDC is restricted in certain important ways. These restrictions allow the JVM to fit the memory.

The primary restrictions on the JVM are:

- No object finalization or weak references.

- No JNI or reflection (hence no object serialization).
- No thread groups or daemon threads
- No application-defined class loaders.
- Remote method invocation (RMI)

CLDC 1.1 needs at least 192 kB of total memory available. CLDC 1.1 relaxes some of these restrictions, in particular reenabling support for floating point types. The CLDC also requires class verification to be done differently. Class files are processed by an off-device class verifier, a process called preverification.

3.2.2. The MIDP Specification

Profiles define the application programming interfaces that are required to write useful applications for a particular group or family of J2ME devices. The Mobile Information Device Profile (MIDP) defines a Java runtime environment for mobile phones

As of mid 2008 there are two versions and an upcoming third of MIDP: MIDP 1.0 (JSR 37), MIDP 2.0 (JSR 118) and MIDP 3.0 (JSR 271).

In MIDP 1.0 there are no standard facilities for data synchronization, even if the device supports the feature. Device manufacturers can and do provide their own device-specific APIs. Some of the differences between the two currents versions are:

MIDP 1.0 (JSR 37)	MIDP 2.0 (JSR 118)
No standard security functions	WAP Certificate Profile (WAPCERT) support based on the Internet X.509 Public Key Infrastructure (PKI) Certificate and the Certificate Revocation List (CRL) Profile.
Only HTTP connection	HTTP, HTTPS and SSL/TLS

MIDP 3.0 is still in development but some of the new features are:

- Multitasking (MVM) support:
    - Multiple MIDlet suites running simultaneously.
    - Background MIDlets, event-launched MIDlets
- Shared libraries for MIDlets (LIBlets)
- UI enhancements
    - Better support for devices with larger displays
    - Support for multiple/secondary displays
    - More scalable graphics model, new widget types
- Secure/Removable/Remote RMS stores
    - Networking enhancements: Multiple network interfaces - IPv6, VPN, IPSEC

3.2.3. The Connected Device Configuration

The CDC, known as JSR 218, was released on 2005. Devices that support CDC typically include a 32-bit microprocessor/controller and make about 2 MB of RAM and 2.5 MB of ROM available to the Java application environment. Contrary to CLDC, CDC has a full-featured JVM.

CDC devices can use optional packages, which is a set of technology-specific APIs that extends the functionality of a Java application environment. Some optional packages can be the RMI Optional Package (JSR 66) or the JDBC [xiv]Optional Package (Java Database Connectivity, JSR 169).

3.3. The API decision

To decide an API to develop the CLDC software was taken into account some prerequisites for the selection. The selection was focused on the limitations of CLDC devices and the practical research targets. Due to all the commented technologies can achieve the targets the main problem is the CLDC limitations.

Searching for available APIs which fit the desired requirements, Web Service for Devices (WS4D[xv]) library has been used to develop the software. WS4D brings Service-Oriented Architecture (SOA) and Web Service technology. It is based on a subset of Web Service technology; Device Profile for Web Service (DPWS). WS4D has Java ME stack available which is based on CLDC capabilities.

Other APIs were studied too. JXME and Jini can fit the targets of the project. As is told before, CLDC don't support RMI and by nature they can not send or receive multicast. Due to these limitations, Jini technology can not be used directly in CLDC devices, it is required an additional element. Jini solves this problem using the Jini Surrogate Protocol which is based on a proxy to act on behalf CLDC devices and the other elements of the federation. That is why using Jini a CLDC device can not be a full-enabled client of the Jini network.

JXME recently release the JXME 2.5 which can be a full-enabled client of JXTA but the choice of the API is not only a technological choice; it is also a political one. Web Services are commonly used in Internet for machine-to-machine communication and DPWS is supported in UPnP, Windows Vista and in Windows Rally Technologies. UPnP is a promising technology but unfortunately there is no open CLDC API available yet. If we compare UPnP and DPWS objectives, their objectives are similar but also DPWS is fully aligned with Web Service.

3.4. Implementation

3.4.1. Architecture

The architecture of the implementation follows the DPWS schema. It is composed by clients and devices. Clients are an entity, which can search for services and call operations of services. A device hosts one or more services and provides common functions like messaging or discovery. It is classified by its port types. According to the DPWS specification a device is a target service of the WS-Discovery specification.

Finally a service provides an implementation of one or more port types to DPWS clients. The messages a service receives and sends are specified by its port types.

The stack implements the following WS-* specifications: Web Service Addressing (WS-Addressing), Web Service Discovery (WS-Discovery), Web Service Eventing (WS-Eventing), Web Service Transfer (WS-Transfer), Web Service Metadata Exchange (WS-MetadataExchange).

The WS4D core package contains all code that is necessary to create a DPWS device with one or more services. This device is discoverable via WS-Discovery mechanisms and can be used as an event-source. The WS4D client package adds functionality to find remote devices and services in the network and to invoke the operations defined in the WSDL.

The implementation carried out within this thesis consists of three blocks: a client and two devices. Although more devices and clients can join the party this is a basic scenario. Clients are enabled to discover all the devices in the network and the services (hosted in devices) are prepared to handle more than one request at once (by threads).

The implemented services let download from the device to the mobile, upload from the mobile to the device or delete the files hosted in the device. Two kinds of services are implemented: picture and audio. The client is capable to use the services and store exchanged files. Also, the client can delete the files hosted in the remote device.

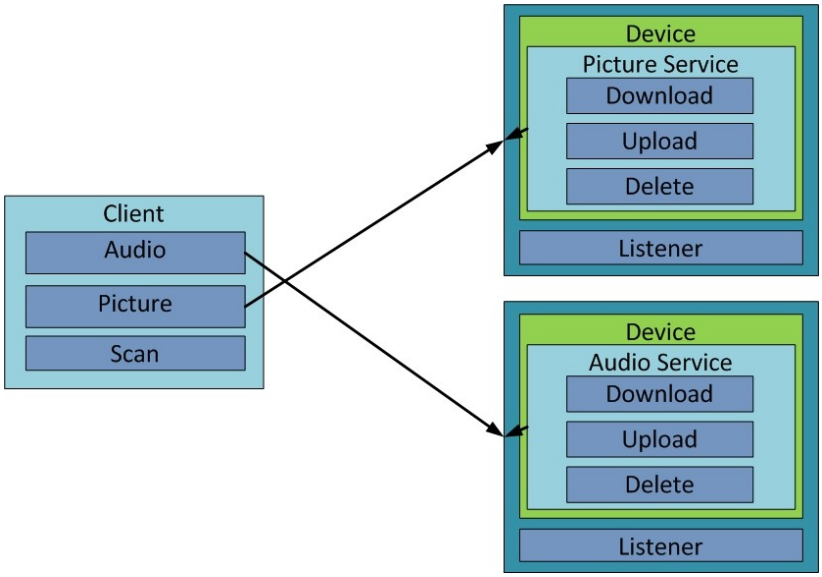


Figure 10. Architecture implementation

3.4.2. Client

The client is composed by scan, audio and picture module. Scan module lets the client look for the available service. The audio and picture modules are used to call the audio or picture service, respectively.

Taking a look in the client flowchart (figure 11) we can see the process of the application. Once the application is started, it checks for an IP. If no IP is set, Auto-IP is used. After that, scan modules send multicast messages following the WS-discovery protocol with the purpose of finding services and retrieve its information. So, up this point we have the automatic connection and the service discovery goals. The next step is to print this information in order to let the user the possibility of choice between the available services. Once a service is selected, the specific module to call the service (audio or picture) comes into play. And finally, here we provide the target of file exchange (if download or upload is chosen).

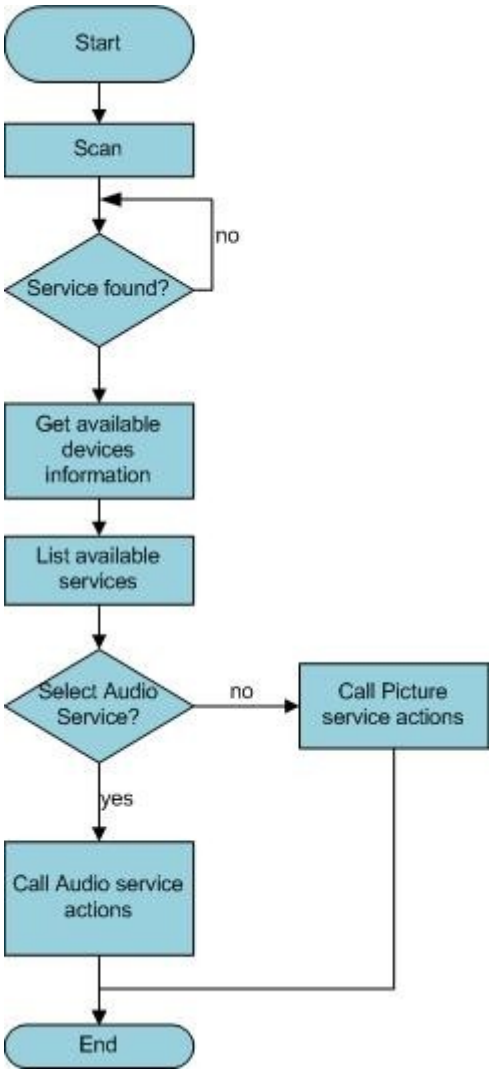


Figure 11. Client application flowchart

In the following UML diagram (figure 12) we can observe the application main class, the Launcher. It has all the methods and functions to call the `PictureActions` or `AudioActions` classes which are on charge of calling the services, i.e. Launcher acts on behalf the user and the calling service classes. Also, Launcher has other important tasks like storing audios and pictures downloaded from the device, browsing the saved files and handling the embedded camera and microphone to upload files to the device.

The Reminder class refreshes the information of the scan module about the founded services every 2 seconds. The `Video` and `VideoCanvas` classes enable the application the possibility of take pictures from the embedded mobile camera.

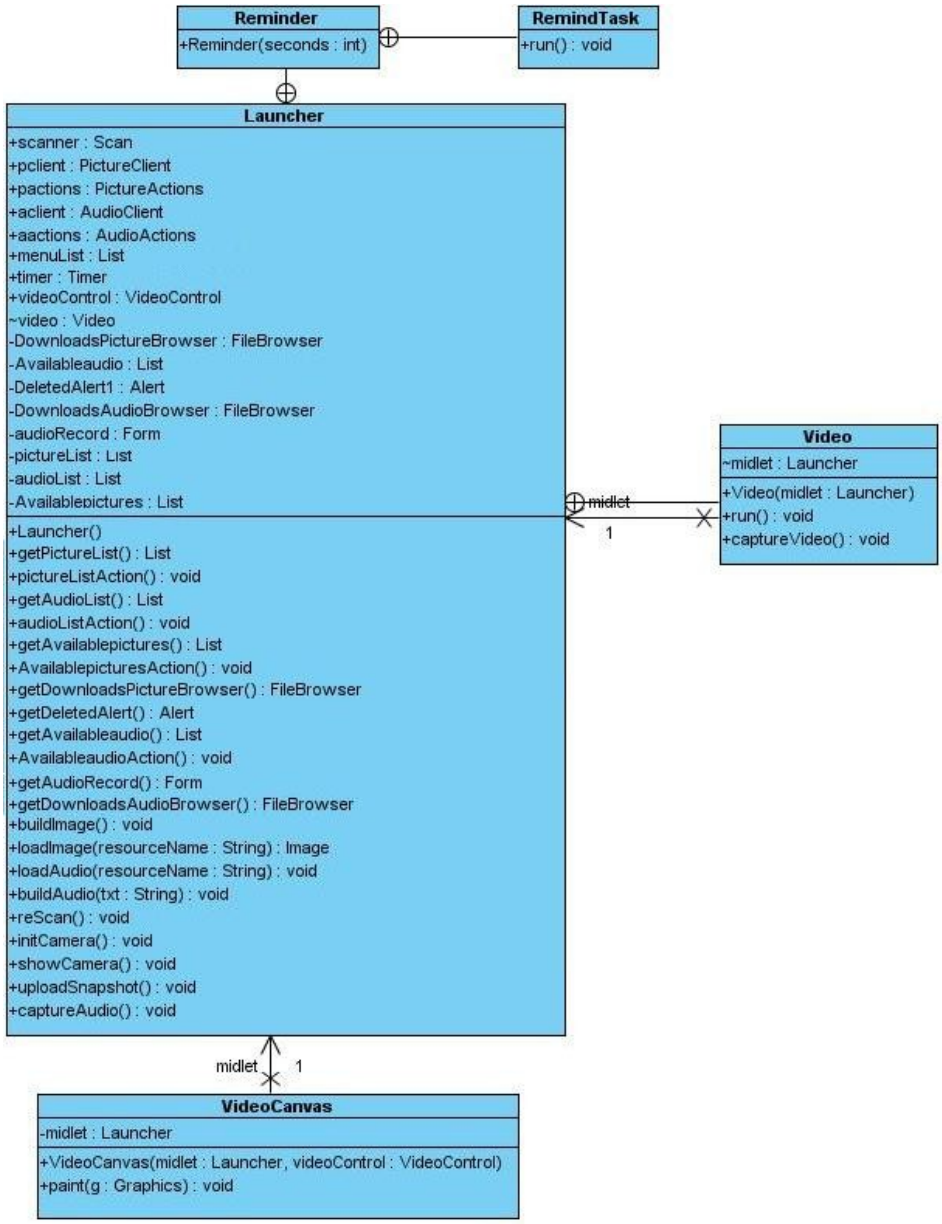


Figure 12. Main Client class, the Launcher UML

3.4.2.1. Scan module

As it said, the scan module looks for services. Once the application has an IP the Launcher class starts the scan module. For every founded device, the name and the model are saved in a Vector and a posteriori listed in the mobile.

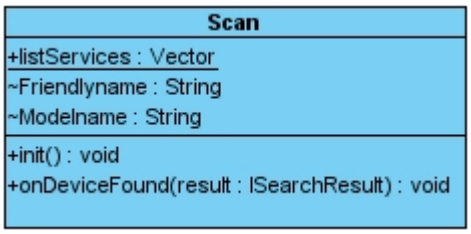


Figure 13. Scan module UML

3.4.2.2. Audio and Picture module

In order to call the services, we need a class entrusted of it. For every service has been implemented an action class (`PictureActions` and `AudioActions`). It is called by Launcher class requesting the service the user wants to call. As is shown in the UML below, the `Action` class has the four equivalent methods to action services.

Firstly, the action class tries to find the requested service filtering the searching by its name (specified by a scope) in order to not overload the mobile with WSDL documents (the mobile has to parse it). After that, the desired action can be invoked by the Launcher class specifying the name of the action as is shown in the UML (`ACTION_*_NAME`).

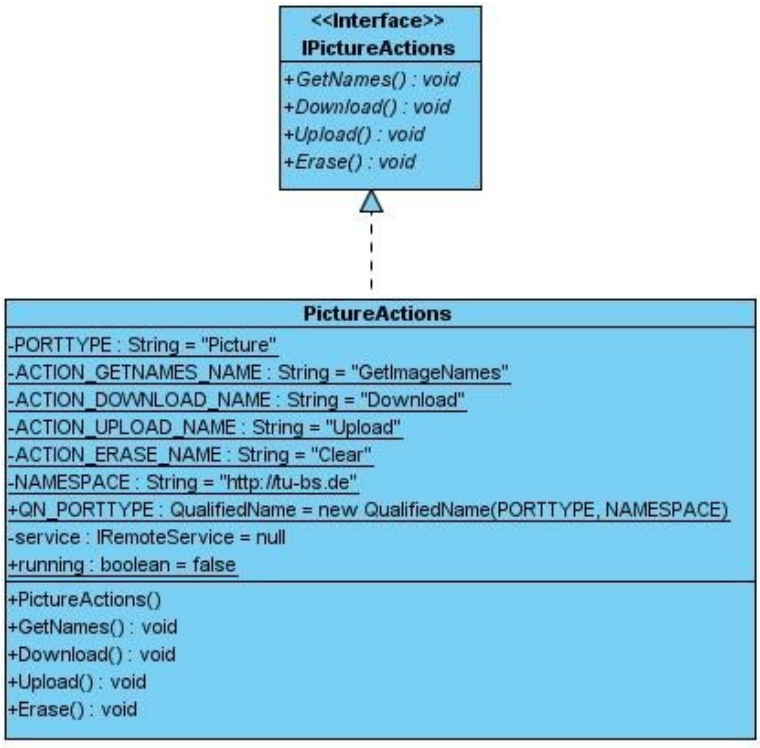


Figure 14. PictureActions UML

3.4.3. Device

The device is composed by the listener module and the device itself. The device hosts the services. In the implementation there are two devices: audio and picture. When the listener module detects a client joined the network the device is started and advices itself to the network to offer its services.

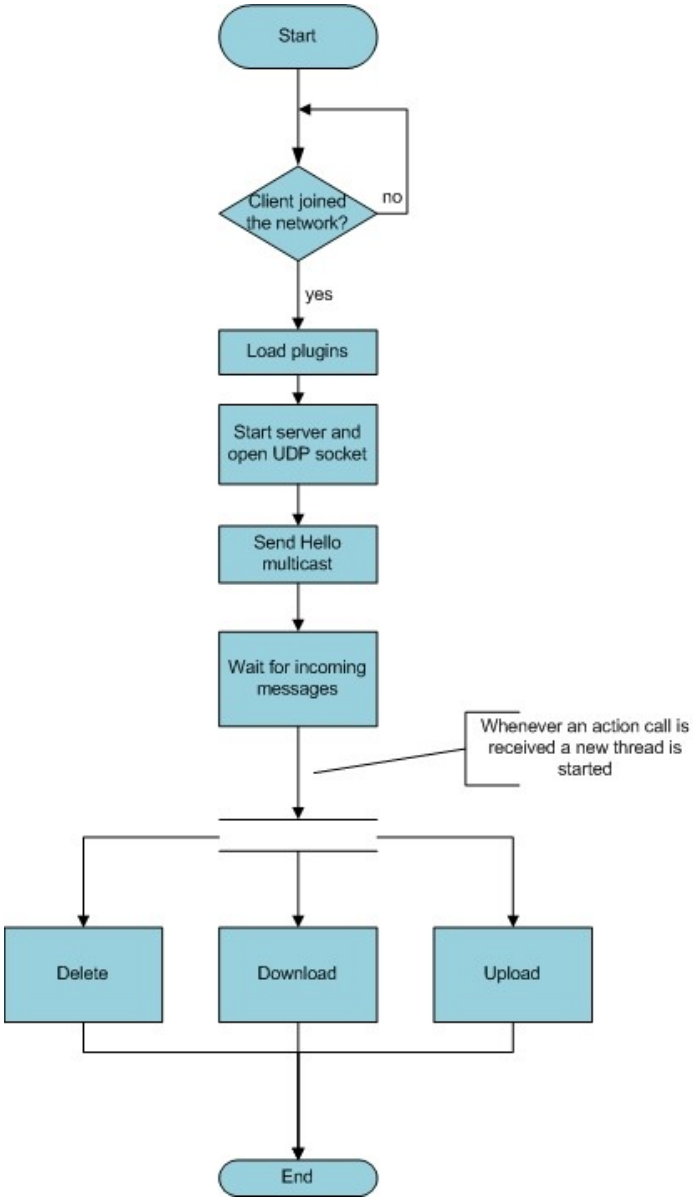


Figure 15. Device application flowchart

In figure 15 is shown how the device application runs. Firstly, the listener module is launched waiting for the connection of a client. When the listener detects a client in the network the device starts. As it happen in the client when the application is started, an IP is checked for. If no IP is set, Auto-IP is used. After that, the plugins of the stack are loaded. There are three plugins available: Device Administration Service, Attachment and Presentation. In this implementation only the attachment plug-in is loaded. Attachment provides the functionality to send and receive MIME attachments over SOAP and HTTP (in order to provide file exchange capability). When the device is started, it sends Hello messages via multicast following the WS-Addressing protocol and wait for responses. Note that in this scenario Hello messages are useless in order

to CLDC devices cannot receive multicast messages. Finally, when the client finds the device and the communication is established the client can use the services hosted in the device.

The device UML (figure 16) is quite simple. In the device information about itself is added like model, company, firmware version… and the start method boots the device.

Figure 16. Device UML

3.4.3.1. Listener module

As it said before, the listener module waits for a client join the network. The `Launch` class starts the application which calls the `Reminder` class. The `Reminder` class is listening the network every second looking for if any client is joining the network. Once a client is detected, the service is started. The service firstly boots the device and after that the service is started.

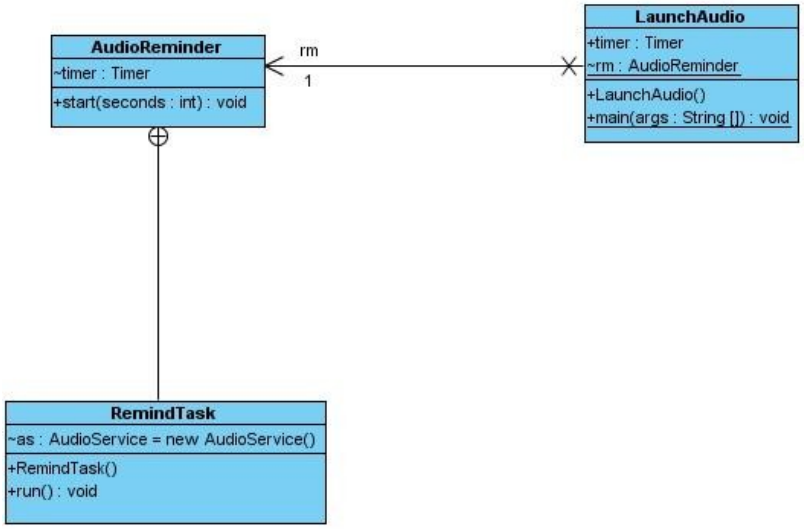


Figure 17. Listener module UML

3.4.3.2. Audio and video services

The two implemented services (audio and video) have the same structure. In the UML (figure 18) we can see the service consists on three classes. The `IAudio` or `IPicture` is

the interface where the methods are defined. The defined methods are the available actions (download, upload, delete). The `getAudioNames()` method is used when we want to download an audio file. Firstly, `getAudioNames()` is called to list in the mobile the available files in the device. Afterwards, the one selected from the list is downloaded calling the `download()`.

The next class is the `AudioService` or `PictureService`. In this class, the action names are defined. These names are used to match the request from the client. So, when the client wants to call an action, it is called by the name defined in `ACTION_NAME_*`. In this class is where the actions are defined. For every action, the in and out parameters are defined. In the `initialize()` method is where some properties are added. The properties are specified by the WS4D stack. With the properties it is possible to set an IP for device, to specify buffer sizes, to choose the multicast mode (J2SE or J2ME), to show debug information or to set timeouts.

The `fireEvent()` method is used to notify some changes in the device. This method is used when we delete the device gallery or when a file is uploaded to the device (which involves a change in the device).

Finally to carry out the actions `ByteBufferBasedAudio` class is used. In this class is where the action code resides. For every action a new thread is started. The stack handle timeouts for threads (also it is possible to specify a time by Properties).

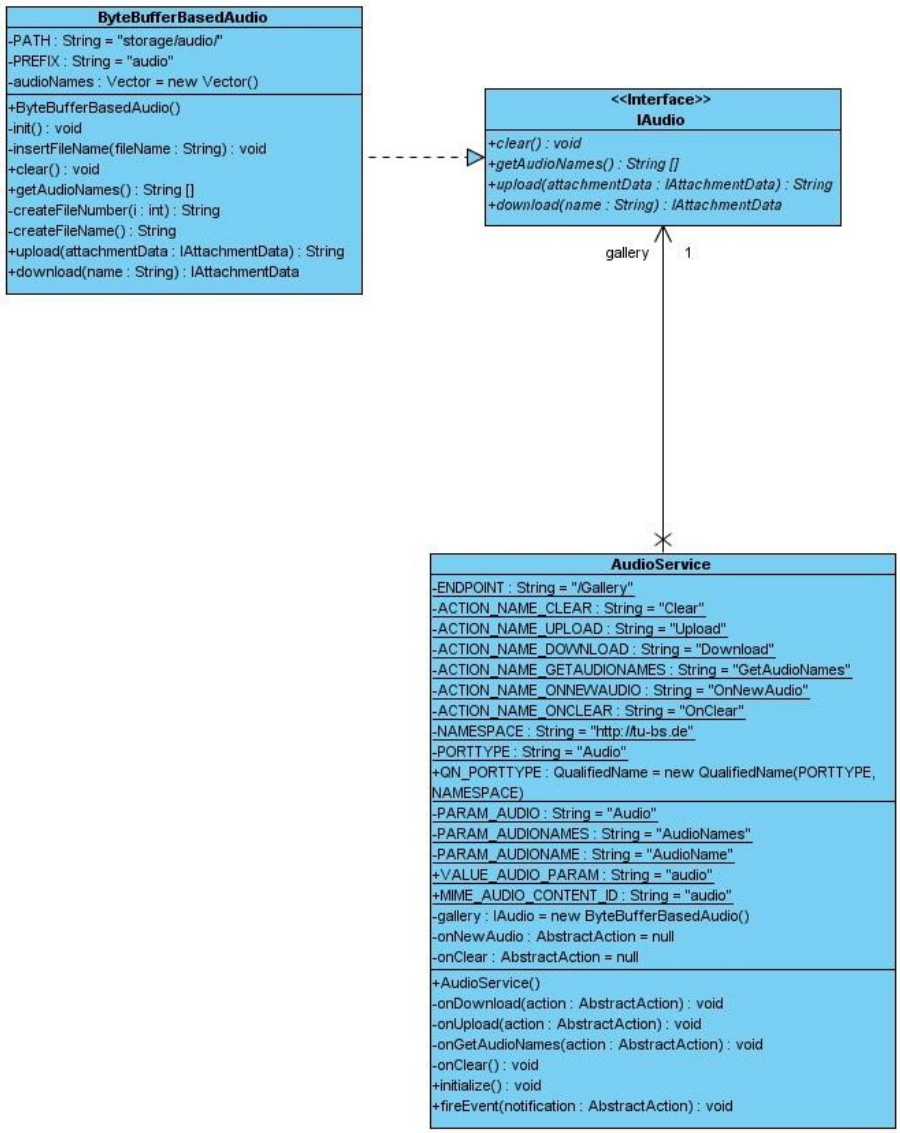


Figure 18. Audio/Video service UML

3.5. Quality assurance

The quality of the application is an important aspect to take care of. To assure the quality different tools can be used. Two parts must be checked: the device implementation and the CLDC device implementation. In addition, the communication between these two parts has to be checked too. Specifically in this project, DPWS Explorer has been used to check the device implementation; a Nokia N95 simulator (S60 3rd edition for Symbian OS) was used to check the CLDC device and a packet sniffer (Wireshark) to check the communication between them. After all, a final testing in real: with a real N95 communicating with the computer via Wifi. This final step was

very important due to some features only could be checked with the real phone. The simulator was not able to take photos or record audio and not able to send multicast messages. Also, the communication between the simulator and the computer was not via Wifi, it was internally.

3.5.1. Device testing with the DPWS Explorer

DPWS Explorer lets to search for devices and invoke its actions. Once the device is running we search for devices and list what is found. Clicking the devices we implemented (Audio and Picture) we can see all the services hosted in the device with all its information. This is the way to check all the information (name, manufacture, location, firmware version…) is correct. Also all the actions available from the service are listed. Moreover WSDL documents can be read.

To check all the actions are properly programmed, DPWS Explorer is able to invoke the actions. In the figure 19, a picture is loaded from the DPWS Explorer to the device. Previously, we select a picture (Input parameter) and then Invoke Action button is pressed. Once is uploaded the name of the picture saved in the car appears as the Output parameter. For every action in every service the invocations where executed and all of them work perfectly.

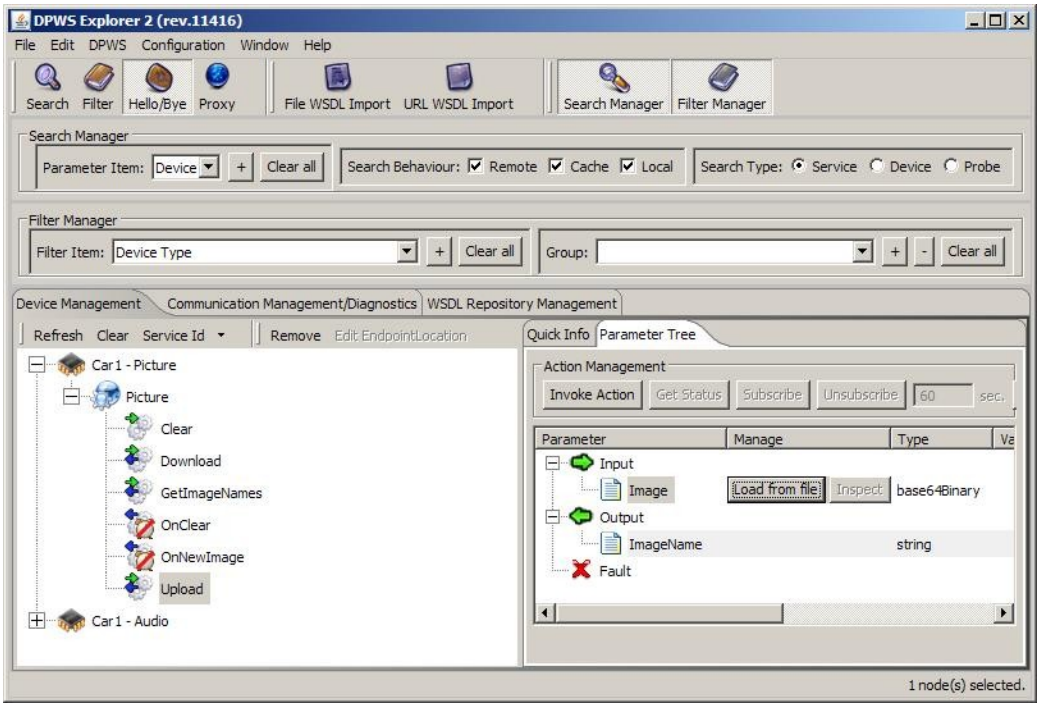


Figure 19. DPWS Explorer uploading a picture

Moreover, in DPWS Explorer we can check for its communication. As figure 20 shows we can see all the messages when the explorer is searching for devices and information about them: IP, direction, type, event. Also, it is possible to filter the messages if it is desired. Checking this feature no problems where detected.

On the other hand, the WS4D stack can show logs while the application is running. There are different options depending what type of information it is needed to read but an error log option is available to look for errors.

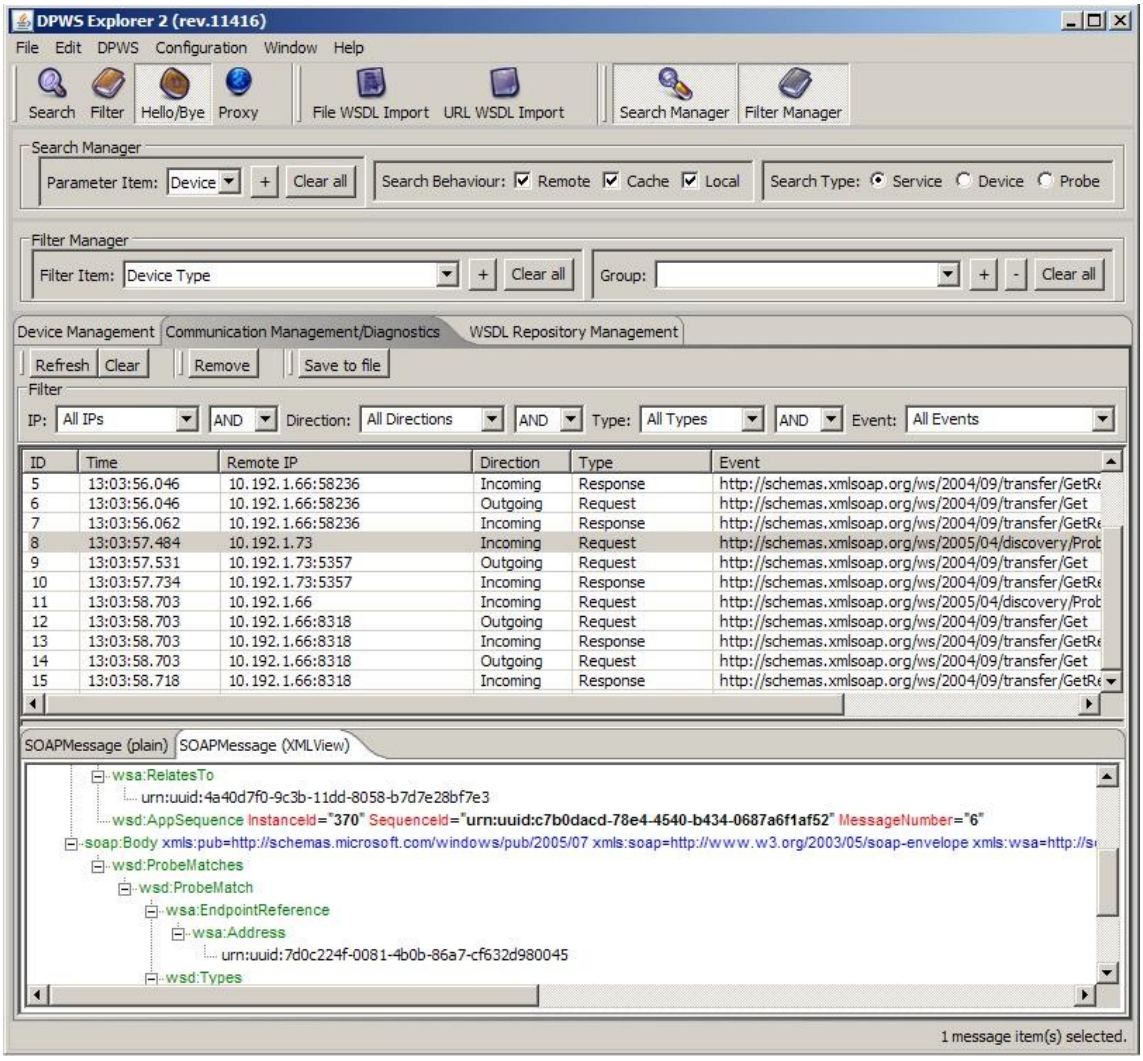


Figure 20. DPWS Explorer received and sent messages

3.5.2. Simulation of the mobile application on the PC

To check the mobile application a N95 emulator was used. It was not a full-enabled simulator but all the actions available from the device could be checked and invocated. Another task of the simulator was to try to obtain a good GUI application. That is the reason of using this simulator because in real this phone is used. Firstly, other Sun Java Wireless Toolkit 2.5.2 for CLDC and Series 40 5th Edition SDK emulators was used before developing the GUI interface with good results. Even it was not a target is important to build an application easy and fast to understand/use.

3.5.3. Communication. Packet sniffer

To assure the correct communication between the devices Wireshark was used. The communication was always good from the first time. There was not too much difference the simulation and the real situation. The only difference resides that in the real situation some packets were lost which meant a slower communication and sometimes timeout errors between the involved devices.

3.6. Example of use

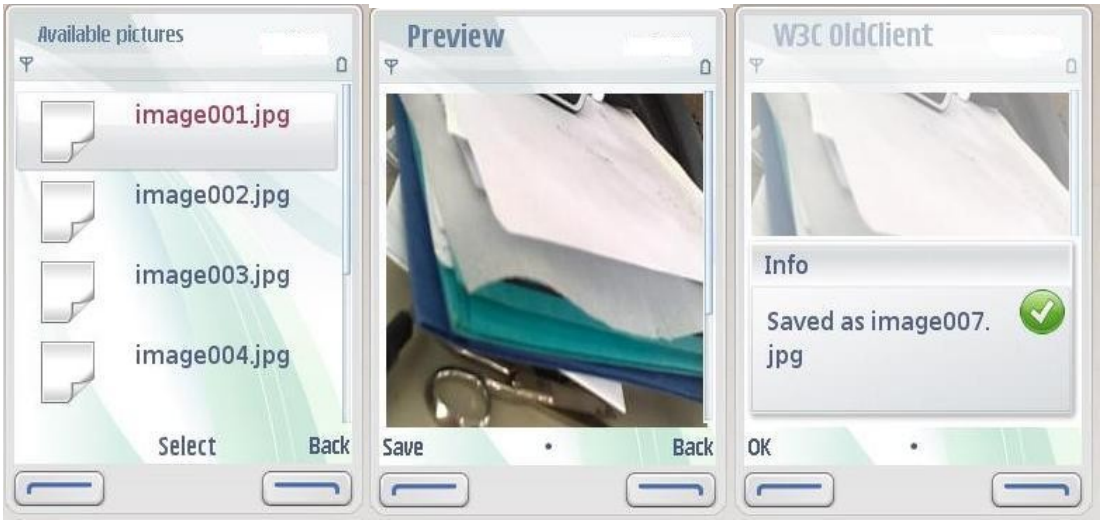
To explain the use, some screenshots was taken to make understand easily how the application works.

When the application starts, scan module starts to find which devices are connected to the network. This module retrieves the name and the location of all services in every device and lists them. A refresh option is implemented to search again. In this example we have two services in the same location.

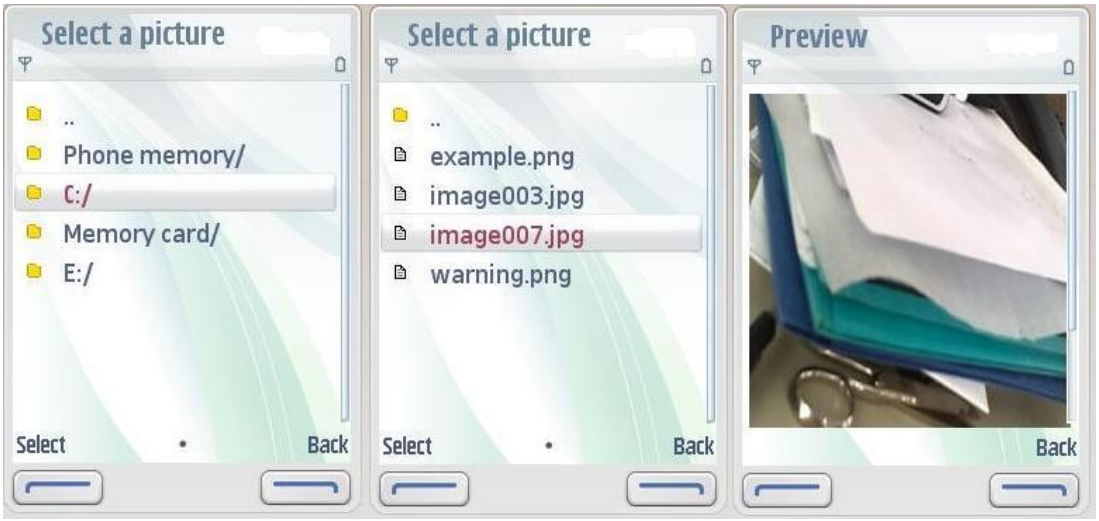
Once we have listed the services, the user can select one to start to use the service and after the selection, the application request for the service action names and we are going to a menu which and again a list with all the action is displayed. The following screenshot shows the actions of the picture service once we select it.

To upload a picture, a picture is taken from the embedded camera in the mobile phone. Once it is taken, it is stored in the remote device.

When we select the option Download pictures, we get the images stored in the remote device (in this case Car1). Also, we can download and save the desired picture. In the second screenshot we can see the downloaded image is the one that we uploaded before.



Finally, we can browse all the images saved in the mobile phone.



Also, we can delete remotely all the images of Car1.

The operation of the audio service is the same as the picture service. The only difference lies in audio service, record audio from 15 seconds when we select Upload audio and then like in the picture service the file is uploaded. All the other operations are the same.

3.7. Goals achieved

The implementation of the project has been fully successful. The main goals: automatic connection configuration, service discovery and file exchange has been achieved. Nevertheless, other kind of media like video could have been possible to exchange in order to cover more media types. Talking about the communication process there is no difference due to all kind of files are attached to the SOAP message; we only have to specify which type it is.

On the other hand, some kind of data synchronization would be a forgotten goal to achieve. With the project finished is not complicated to do a module which is on charge to do a data synchronization. Also, too much code is not needed since the synchronization is a combination of download/uploads files from one device to another and that is already implemented.

4. Conclusions

The limitations of CLDC devices have been the most remarkable point to choose the technology to develop the software. If CLDC devices would not have these restrictions all the studied technologies could be useful due to all of them achieve the purposes of the project and others features has not been taken into account to make the choice.

In some technologies there are solutions for the CLDC devices but it means other elements come into play. Moreover, these elements act on behalf of CLDC devices so CLDC devices would not be full-enabled device. For example, Jini has the Jini Surrogate Protocol.

Even though, the WS4D-J2ME stack is a good option. It meets all the requirements; it is based on the Connected Limited Device Configuration and can thus be used on all Java ME platforms and even on Java 2 Standard editions. This stack complies with the DPWS, which is supported by UPnP too.

The connectivity mobile-mobile is ruled out due to they don't support multicasting. This lack is one of the most important restrictions to provide connectivity between devices using WS4D stack. The stack makes possible to send multicast messages but not to receive.

To sum up, the main conclusion can be drawn is CLDC devices limitations make difficult to use some technologies to provide connectivity between them even though some limitations like multicasting can be solved by programming and others like serialization cannot.

4.1. Outlook

The implementation can be updated using the latest stack release. The current software was built according to the first one. The new version which is available since

august 2008 provides security mechanisms. The new stack supports HTTP Basic Authentication and the Transport Layer Security (TLS). To secure its services, the stack uses a https endpoint location and communicates only over SSL with other services. In addition, some other improvements are added like using multicast messages via broadcasting for some CLDC-Platforms which don't support Multicast.

On the other hand a possible implementation within OSGi (Open Services Gateway initiative) framework will be useful. OSGi Alliance is an open standards organization. OSGi have specified a Java-based service platform that can be remotely managed. The aim is to define open software specifications to let design compatible platforms to provide services.

5. Bibliography

Jini technology
http://www.jini.org/wiki/Main_Page
Visited: 01.10.2008

Jini architectural overview
http://www.sun.com/software/jini/whitepapers/architecture.pdf
Visited: 01.10.2008

JXTA protocols specification
https://jxta-spec.dev.java.net/JXTAProtocols.pdf
Visited: 01.10.2008

The JXTA solution to P2P
http://www.javaworld.com/javaworld/jw-10-2001/jw-1019-jxta.html
Visited: 01.10.2008

A review of Jini and JXTA
http://users.ecs.soton.ac.uk/ra/papers/Ashri-Agent-Middleware-Technologies.pdf
Visited: 01.10.2008

Understanding UPnP
http://www.eg3.com/goto.cgi?pageid=consumer/upnp&slot=record&url=http://www.upnp.org/download/UPNP_UnderstandingUPNP.doc
Visited: 01.10.2008

UPnP Device Architecture 1.0
http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.0-20080424.pdf
Visited: 01.10.2008

Web Service Architecture
http://www.w3.org/TR/ws-arch/
Visited: 01.10.2008

Service discovery protocols. A comparative study
http://bcr2.uwaterloo.ca/~rboutaba/Papers/Conferences/IM05-discovery.pdf
Visited: 01.10.2008

Automatic Configuration and Service Discovery for Networked Smart Devices
http://www.appliedinformatics.at/download/AutomaticConfiguration.pdf
Visited: 01.10.2008

Resource and Service Discovery in Large-Scale Multi-Domain Networks
http://www.buet.ac.bd/cse/users/faculty/reazahmed/cse6809/discovery-ieee.pdf
Visited: 01.10.2008

A Technical Introduction to the Devices Profile for Web Service
http://msdn.microsoft.com/en-us/library/ms996400.aspx
Visited: 01.10.2008

Device Profile for Web Service specification
http://specs.xmlsoap.org/ws/2006/02/devprof/devicesprofile.pdf
Visited: 01.10.2008

WS4D Java Multi Edition DPWS Stack tutorial
http://kent.dl.sourceforge.net/sourceforge/ws4d-javame/ws4d-javame-doc.pdf
Visited: 01.10.2008

Understanding the Connected Limited Device Configuration (CLDC)
http://www.ericgiguere.com/articles/understanding-the-cldc.html
Visited: 01.10.2008

Mobile Information Device Profile
http://java.sun.com/products/midp/midp-ds.pdf
Visited: 01.10.2008

Understanding Devices Profile for Web Service, WS-Discovery, and SOAP-over-UDP
http://msdn.microsoft.com/en-us/library/cc980021.aspx#_Toc208829803
Visited: 14.10.2008

WS4D
http://www.ws4d.org

6. Glossary

[i] Model of coordination and communication among several parallel processes operating upon objects stored in and retrieved from shared, virtual, associative memory. This model is implemented as a "coordination language" in which several primitives operating on ordered sequence of typed data objects, tuples, are added to a sequential language and a logically global associative memory, called a tuplespace, in which processes store and retrieve tuples.
http://wcat05.unex.es/Documents/Wells.pdf

[ii] Tuple Space is an implementation of the associative memory paradigm for parallel/distributed computing. It provides a repository of tuples that can be accessed concurrently. Tuple spaces were the theoretical underpinning of the Linda language.
http://c2.com/cgi/wiki?TupleSpace

[iii] Service specification which provides a distributed object exchange and coordination mechanism (which may or may not be persistent) for Java objects. It is used to store the distributed system state and implement distributed algorithms. In a JavaSpace all communication partners (peers) communicate and coordinate by sharing state. JavaSpaces is used to achieve scalability through parallel processing and provides for reliable storage of objects while reducing the complexity of traditional distributed systems.
http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/index.html

[iv] The Java Remote Method Invocation (RMI) system allows an object running in one Java virtual machine to invoke methods on an object running in another Java virtual machine. RMI provides for remote communication between programs written in the Java programming language.
http://java.sun.com/docs/books/tutorial/rmi/index.html

[v] Djinn is just another way to say "a Jini system", or if you will "an (actual software) system built using Jini".
http://www.jini.org/wiki/Djinn

[vi] TLS is a protocol for establishing a secure connection between a client and a server. TLS is capable of authenticating both the client and the server and creating an encrypted connection between the two.
http://www.tech-faq.com/tls-transport-layer-security.shtml

[vii] NAT (Network Address Translation or Network Address Translator) is the translation of an Internet Protocol address (IP address) used within one network to a different IP address known within another network.
http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci214107,00.html

[viii] The JXTA Java Micro Edition provides a JXTA compatible platform on resource constrained devices using the Connected Limited Device Configuration (CLDC) or the Mobile Information Device Profile 2.0 (MIDP), or Connected Device Configuration (CDC). The ranges of devices include the smart phones to PDAs. Using JXTA Java Micro Edition platform, any CLDC/MIDP/CDC device can participate in the JXTA network with any other JXTA device. (JXTA-J2SE, JXTA-C and JXTA-J2ME).
https://jxta-jxme.dev.java.net/

[ix] DHCP is a protocol for assigning dynamic IP addresses to devices on a network. With dynamic addressing, a device can have a different IP address every time it connects to the network. In some systems, the device's IP address can or even change while it is still connected.
http://www.faqs.org/rfcs/rfc2131.html

[x] Protocol provides for the sending of HTTP Notifications using administratively scoped unreliable Multicast UDP. Multicast UDP allows a single notification to be delivered to a potentially large group of recipients using only a single request.
http://msdn.microsoft.com/en-us/library/aa505982.aspx

[xi] Distributed Component Object Model (DCOM) is a proprietary Microsoft technology for communication among software components distributed across networked computers. DCOM, which originally was called "Network OLE", extends Microsoft's COM, and provides the communication substrate under Microsoft's COM+ application server infrastructure. It has been deprecated in favour of Microsoft .NET.
http://en.wikipedia.org/wiki/Distributed_Component_Object_Model

[xii] The Common Object Requesting Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG) that enables software components written in multiple computer languages and running on multiple computers to work together.
http://en.wikipedia.org/wiki/Corba

[xiii] Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located in another computer in a network without having to understand network details. RPC uses the client/server model.
http://searchsoa.techtarget.com/sDefinition/0,,sid26_gci214272,00.html

[xv] **http://www.ws4d.org**