1. Multimedia Applications

Multimedia applications are becoming always more common and requested. The GSM short text messages have evolved in multimedia messages, that may send text, audio, images and videos. Many of the services that will soon be available on the UMTS system are already successful services on the Internet. This is due to the rapid convergence between the Internet world and the Mobile Communications world. The trend is to provide on mobile terminals all multimedia services and applications the users can access on the Internet.

The Mobile world is now equipped to provide multimedia services to mobile users. Some service platforms have been defined and standardized, such as the MMS (Multimedia Messaging System) platform that enables the exchange of multimedia messages and the IMS (IP Multimedia Subsystem) domain that will supply IP-based services.

In this chapter some of these services, that are already deployed on the UMTS or soon will be, are presented.

1.1 Presence

1.1.1 Introduction

First Presence Services appeared on the Internet some years ago and at the beginning nobody expected that they would demonstrate to be so successful to become one of the most promising enabling services of the mobile world.

The basic need to know if your friend is on line at a specific time is common to both Internet and Mobile world. It represents the first expression of the concept of presence. This concept has been extended to include a number of other useful information about people the user wants to interact with.

The initial success of the presence service is intrinsically bound to the success of Instant Messaging (IM) applications. However, now it is clear that presence service is a more general enabling technology which is expected to play a relevant role for the enrichment of numerous services.

The opportunity to manage presence information and to reproduce on the mobile the success of Internet presence and instant messaging applications justify the interest of the mobile world to this service.

This paper discussed the status of the art of presence services and provides an overview of the challenges and the opportunities presented to mobile operators by presence services.

1.1.2 Presence services and the standardization

Even if initially Instant Messaging and Presence (IM&P) solutions were born from spontaneous or local Internet initiatives, quite early the need to have solutions for interoperability has brought the discussion about presence services into standardization.

First standardization efforts have been done by Internet Engineering Task Force (IETF). The Working Group (WG) that has leaded this work is called Instant Messaging and Presence Protocol (IMPP).

The RFCs issued by IMPP, RFC 2778 and 2779, represent the reference for the definition of main presence concepts. Within IETF, the most relevant results form a mobile perspective are those obtained by both IMPP and the SIP for Instant Messaging and Presence Leveraging Extensions (SIMPLE) WG because they have provide the basis for the 3GPP specifications of the Presence Service (PS).

It must be noticed that SIMPLE is compliant with IMPP guidelines for a common and interoperable presence protocol.

An relevant initiative on mobile presence is that of Wireless Village (WV) that is now migrated into the Open Mobile Alliance (OMA) and it is called Instant messaging and Presence Service (IMPS).

WV solution for presence and instant messaging is the result of the effort of three major mobile players as Nokia, Ericsson and Motorola who joint their forces in order to foster an early PIM solution suitable for existing GSM/GPRS mobile networks.

At the present time, there is no unique solution for presence services but rather a number of possible alternatives each suited to a specific environment. In this scenario, OMA is trying to find consensus for interoperability.

In the current market situation, the products of reference seem to be those based on SIMPLE/SIP since they are somehow aligned with both 3GPP and IETF specifications. It must be noted that OMA is gathering the heredity of both WV and Wireless Application Protocol Forum (WAPF). And, since WAPF IM&P solution was based on SIP/SIMPLE, now OMA is very concentrated on the WV and SIMPLE interoperability.

1.2 Instant Messaging

1.2.1 Overview

In the GSM world, SMS has been a huge success allowing subscribers to exchange messages in a convenient and cost-effective way that fits into their lifestyles. Similarly, in the IT world, Instant Messaging has recently proved to be an extremely attractive application for both business and private users. IMS offers the opportunity to introduce feature-rich SIP-based Instant Messaging that provides all the benefits of the mobile and IT world in a single application. As a flexible and adptable application. Instant Messaging offers full interworking with legacy SMS, MMS and email systems and will eventually replace them. Because of the popularity of these services, Instant Messaging is an extremely important application for subscribers and operators and one that is best supported in a SIP framework.

IMS Instant Messaging differs from traditional Instant Messaging in the following ways :

- Enables real time chat;
- IMS Instant Messaging delivers a rich set of subscriber presence states which can be updated by a combination of automatic and user driven requests;
- IMS presence model is consistent with the one defined in SIMPLE, and embraces an open event registration model;
- Messages of all types are handled in a uniform manner, providing the subscriber with a more consistent and satisfying user experience;
- IMS Instant Messaging supports multiple device types;
- IMS based Instant Messaging supports multiple address and login;
- Supports multiple messaging lists more easily;
- Any application on the network or handset can request and receive presence information via SIMPLE messages.

IMS based Instant Messaging offers all of the functionality expected of a robust carrier messaging system such as SMS combined with that of a reliable IP-based messaging system. Features include:

- Full send and receive functionality;

- Store and Forward support;
- Message History support;
- Selective blocking and reception priorities;
- One-to-Many and Many-to-Many messaging;
- Group Lists;
- Online and offline operation;
- Preset messages;
- Inter-carrier and Inter-platform support through gateways.

IMS Instant Messaging is more than a simple messaging application. It is a robust push solution that can unify all message delivery models within a service provider's network. The solution implements WAP push messages, SMS messages, and SIMPLE compatible CPIM messages. CPIM is a MIME-based model, so inherent support is also provided for MMS- and EMS-based functionality. WAP push, SMS, and MMS/EMS are implemented via gateways bridging into the rich MIME-based CPIM format. Messages of all types are handled in a uniform manner, providing the subscriber with a more consistent and satisfying user experience.
IMS Instant Messaging is fully SIP/SIMPLE compliant. It provides the wireless carrier with the full event subscription model supported by SIMPLE. Any application on the network or handset can request and receive presence information via SIMPLE messages. Any application or handset can send and receive messages as well. This is a very powerful feature and allows many exciting and innovative applications to be build on top of the Instant Messaging framework.

1.2.2 Instant Messaging and Presence

IMS Instant Messaging delivers a rich set of subscriber presence states. This presence model is consistent with the one defined in SIMPLE, and embraces an open event registration model. This is an important distinction of the IMS solution.
There are two types of subscriber presence states: *predefined user states* and *user-specified status*. Predefined user states are those that are pre-set in the application.

- Available;
- Ready to Chat;
- Away;
- Invisible;
- Busy;
- User Defined Status.

These states are updated through a combination of automatic (client and network driven) and user driven requests. As an example, the subscriber client will automatically make the user "Available" on device power up and "Away" when the device is powered off. Users may wish to augment those automatic capabilities by making themselves "Busy" when they go into a meeting or "Invisible" when they do not wish to be disturbed.
User-specified status allows users to append their own message to their current presence status. Typical messages might be "Maidenhead Office" or "Holiday". This combination of predefined states and user-specified status provides a powerful capability for business and private subscribers. Subscribers could be charged extra for joining the buddy service that alerts them to the status of their friends and colleagues. Furthermore, it is likely that subscribers with would feel a strong sense of belonging to their buddly list communities and thus would make more voice calls and send more instant messages to other members.
IMS Instant Messaging has also the ability to implement a presence "aging" function for the mobile device. Unlike wireline IM services that use key strokes or mouse movement to determine

if users are available, in a wireless environment it is difficult to determine if mobile subscribers are available to communicate even if their phones are on. The time from the last message transmitted/received by the mobile device is used by the aging function to determine the user's ability to communicate. A continuum of presence that is supported by this model.

- Signed On;
- Connected;
- Active;
- Dormant;
- Disconnected;
- Signed Off.

Buddy Lists are only one of many services that may be offered The full implementation of a SIMPLE event subscription model allows any presence subscription to be treated identically.

1.2.3 Instant Messaging Client Features

The experience of mobile subscribers differs enormously from that of a PC users. Their attention is typically split across several tasks, display and input capabilities are very limited with traditional handsets and they are almost always in a hurry. The IMS Instant Messaging solution has all the features of desktop IM, but has the flexibility to create a user experience specifically designed to enhance the mobile lifestyle.

The Buddy List is the "home-base" of the IM application. From it, mobile subscribers can see at a glance the Presence/Availability of other subscribers on their Buddy List. In addition, subscribers can always view their own Availability setting and easily change it with the toggle of the "Status" button. This is critical because the desire of users to accept message changes quickly and continually. Similarly, the most recent message is immediately viewable, as it is bannered across the top of the screen. Since mobile users are able to glance only sporadically at their device, unread, incoming messages are queued in threads that are indicated by **bold** buddy names. The Buddy List can be sorted either by name or by whether a new message is waiting to be read. This all puts users in control of what messages to view and when.

Another important feature of the IMS Instant Messaging solution is the support of messaging to SMS users who may not be listed on a Buddy List or have an Instant Message-enabled handset. While Buddy Lists have been proven to be a useful way to send messages, the support of messaging to SMS users enables sending messages to anyone with an SMS-enabled phone. The IMS Instant Messaging solution handles this by appending incoming messages from non-buddies to the Buddy List, without Presence information, until they are read and either replied to or deleted.

The Siemens/Dynamicsoft IMS Instant Messaging solution features an innovative tool for composing messages. The "notebuilder" message composition interface combines the flexibility of text input with the swiftness of "preset phrases" and "emoticons". For example, the user can enter a few words manually, such as "Really sorry", and then navigate down to select the preset phrase "I'll be late", which will be appended to the sent message. Finally, the user can append an "emoticon" in the same way and then send off a message that conveys just the right feel, but with minimal text entry.

Even while composing a message, the user can monitor and perform critical tasks. For example, the most recent unread message is still "bannered" across the top of the screen since it might contain information relevant to the message being composed. Users can also still view and modify their own availability status while composing.

1.3 Push

1.3.1 Introduction

Push technology is going to play a relevant role in the mobile scenario since it allows for new appealing services which foresee the distribution of information useful to mobile users like weather forecasts, traffic news, sport news or simply advertising. Typically push based services take advantage of user location information in order to provide the user not only with the information he may need but also with personalized data. Other services may require to push messages to all users who are in a given geographical area or in a given context at a specific time.

These services are provided on subscription basis since users have to explicitly give their availability for receiving the push contents, to specify the privacy rules and to agree on the charging model. The user should be able to stop the delivery of push services at any time.

Different business models can benefit from push technology. Either the mobile user or the content provider or both can be charged for this type of service as required by the business model.

It must be noticed that, with the respect to push services which are currently available on the Internet that are used for implementing distribution channels, the mobile push services are based on real push, i.e. the client does not poll the server for information updates. Clearly, the use of real push is justified by the need to save resources.

1.3.2 Push Solutions and Architectures

Since push services require essentially the asynchronous delivery of messages, simple push solutions can be based on existing messaging systems like SMS-C, MMS-C, and IM Server by providing a suitable interface to applications and servers. Both these messaging systems are able to guarantee the message delivery thanks to their store and forward capability.

However, this kind of solutions are not able to offer a suitable and complete infrastructure for push services because they lack of privacy support…

The major effort for the definition of an integrated push architecture for mobile services had been done by WAP Forum. It has identified an end to end push architecture and introduced the concept of Push Proxy, i.e. the functional entity that is in charge of receiving push requests from applications and to forward messages towards mobile users through the appropriate transport service (SMS, WAP).

However, the solution proposed by WAP Forum is focused on the GPRS scenario. Therefore, a push architecture suitable for 3G systems needs to be identified. Currently, 3GPP is working on push services for UMTS environments.

1.4 Games

1.4.1 Business Drivers for Games

Over the next few years, mobile games are seen as a source of significant increase revenue for operators. Mobile games appeal to market sectors (children, teenagers, young adults) that are already heavy users of mobile services (for instance, voice and SMS) and likely to be heavy users of Instant Messaging and Presence services, but are also price-sensitive. Offering attractive new services to this sector at the correct price point has the potential to significantly increase ARPU.

The strategy is to develop a flexible infrastructure for supporting mobile games with GPRS, UMTS and the evolution to IMS. Siemens is a founder member of the Mobile Games

Interoperability Forum and runs an active community programme for developers. and work with operators and third party developers.

Previous generations of mobile handsets have been very limited in terms of games available because of the restricted ability to download new games to the phone. The latest generation of phones such as the Siemens SL45i begins to overcome this limitation by introducing a Java runtime environment on to the handset. New applications including games can be easily downloaded to the handset. The figure shows the architecture proposed by Siemens to support games on Java-enabled handsets through the use of an download server. Users can be charged in a number of ways, for instance, premium rate SMS.

IMS presents even greater opportunities for the introduction of exciting and innovative concepts in mobile gaming, which is the key to increasing ARPU. One example is the wide game, where players receive location-dependent game information on their terminals as they move from place to place. Due to the engaing nature of the user experience the revenue opportunity may be significantly enhanced.

The SIP model is much more powerful and flexible one than the WAP session-based push model, and there is a significantly increased opportunity for multi-player games. These can be built on top of the standard facilities that IMS provides: presence, buddy lists, instant messaging, video streaming and support for m-commerce and location-based services.

Another novel feature that can easily be incorporated into IMS terminals is a motion detector. This translates the movement of the handset into game commands and can be used to convert the terminal into, for instance, a virtual wand in a fantasy wide game. All these features increase the richness and attractiveness of the mobile games increasing the amount of time that users will play the game and consequently the revenue generated for the operator.

2. UMTS and IMS

2.1 Why UMTS?

The third generation mobile network UMTS was born thanks to the great success achieved during the past decade by the following technologies: the GSM (Global System for Mobile Communication) network and the Internet.

The GSM, born initially for Western Europe (the first telephone call with this system was done in 1991 in a Helsinki Park), was exported in many others parts of the world (America, Asia, etc), growing up to one billion users. The "Short Message Service" represents an important source of profits for the mobile operators (a starting unexpected result when it was created). The most important limit of GSM is the available band shortage. Due to this limitation it does not support multimedia applications and the use of Internet services. The cellular telephony systems GSM grants a data transmission speed of a bare 10 Kb/s per user.

On the other hand, the Internet spreading has been so remarkable as to outdo the fixed switching packet traffic over the switching circuit during the year 2002. Its most important limit is poor mobility.

These two phenomena have upset the users way to communicate and they have brought institutions and operators to seek for new solutions. Expectations are for an enormous demand increase of multimedia services (already in the fixed network), together with mobility.

There is demand for a new network with the ability "to connect anything, anywhere, anytime". The second-generation technology limits should be overcome so to have an efficient use in mobile environment of tools like:

- Email;
- Web browsing;
- Corporate network services;
- Videoconference;
- E-commerce;
- Multimedia applications.

The UMTS network is not a support to Internet, the operators want the network to be, not a pipe to different domains, but a center of added value services. It must allow the user to be connected to the right thing, the right time, the right place. Only by these means the UMTS will return the operators the revenues they expect.

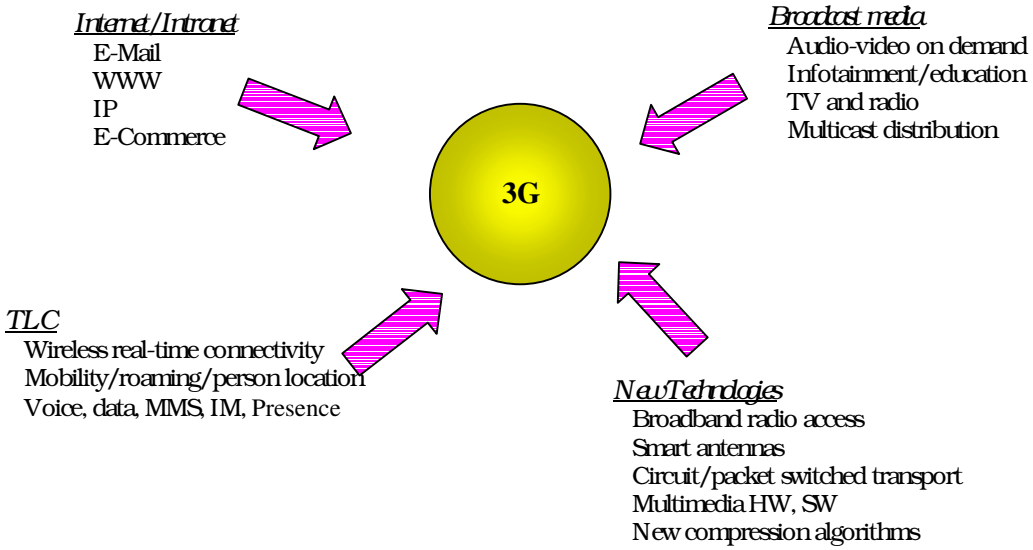

Fig. 2.1 Third generation convergence and integration

2.2 UMTS Specification Process

The UMTS was originated from the family of standard networks 3G named IMT-2000 (International Mobile Telecommunications 2000). Some of the requirements that must satisfy the IMT-2000 network are:

- To provide a worldwide covering to their users;
- To allow roaming between different networks;
- To support packet transmission (PS) and circuit transmission (CS).

The 3G standard are developed by Regional Standards Development Organizations. These organizations co-operate to the development of 3GPP (part of the family systems IMT-2000). Their goal is to supply a set of worldwide applicable specific standards. The proposals for 17 different standards INT-200 was exhibited at SDO in 1998: 11 for the Terrestrial system and 6 for the mobile satellite system (MSSs). The complete evaluation of such proposals was made at the end of 1998, and the necessary negotiations to create agreement between the different points of view ended in mid 1999. The ITU accepted all the 17 proposals as IMT-2000 standard.

The most important standard introduced are: UMTS (W-CDMA) from GSM and the CDMA2000 from IS-95 and CDMA (TD-SCDMA).

UMTS offers mobile users multimedia applications thanks to a greater data transmission speed (up to 2 Mbps) and it settles a standard global roaming.

The UMTS was developed by the 3GPP (Third Generation Partnership Project), a join venture between various SDO: ETSI (Europe), ARB/TTC (Japan), ANSI (USA), TTA (South Korea) and CWTS (China). To obtain total consent the 3GPP introduces UMTS in yearly releases::

- Rel. '99 (Major RAN Release, December 1999). The following elements have been introduced:
  - The new radio interface, WCDMA;
  - A new architecture for the RAN (Radio Access Network);
  - A new interface between the Core Network and the Access Network;
  - The "Open Service Architecture" for services;
  - Interworking rules among GSM and UMTS.
- Rel. 4 (Minor Release, March 2001). Are here specificated:
  - IP transport of Core Network protocols;
  - IP protocol header compression scheme (IETF);
  - Circuit switched domain evolution. MSC and MGW servers are now based on the Internet Protocol;
  - UTRAN access with an improved QoS;
  - An improvement on localization services;
  - The introduction of MMS and WAP protocols.
- Rel. 5 (Major CN Release, March 2002).
  - The IP Multimedia Subsystem (IMS) is introduced as a new Core Network element;
  - The WCDMA access scheme, MMS and Location services are improved.
- Rel. 6 (IMS part 2, expected at the ending of 2003). The focus of this Release is on the IMS. Services as Presence, Instant Messaging, Group Management and Conferencing are here included.

Open issues for future realeases are: 3GPP/3GPP2 IMS harmonization; WLAN-UMTS interworking and Multicast/Broadcast services.

We follow now giving a brief description of the UMTS network. An emphasis will be given to the IMS, given its importance for the work developed.

2.3 Network Architecture

The foundation idea of 3G architecture is to prepare a universal infrastructure able to support present and future services. The network is planned so as the technical changes and developments could be integrated without any upsetting of the network or the pre-existent services. The division between the technologies of access, transport, connection control and user applications, allows satisfying this requirement.

2.3.1 Network Models

The whole network architecture can be divided in subsystems according to the traffic nature that crosses it, the supporting protocols structure and the physical elements on which are located.

From the traffic point of view, the UMTS network has two domains, Packet Switching Domain, and Circuit Switching Domain. Based to the 3GPP TR 21.905, a domain refers to the highest physical entities level and to the reference points that divide such entities.

From responsibilities and protocols point of view, the network can be divided in access stratum and non-access stratum. The access stratum collects the protocols related to the User Equipment and the network access. The non-access stratum holds the communication protocols between the UE and the Core Network (domains PS and CS).

From the structural point of view, the network is sub divisible in User Equipment (UE), UMTS Terrestrial Radio Access Network (UTRAN) and Core Network (CN).

The User Equipment, that is the terminal used by the consumer, has some mandatory functions for the interaction between the network and the terminal, such as network registration functions, position adjournment, terminal profile identification, necessary algorithms execution for the authentication and codification of messages. The terminals, to support future developments and services, should:

- Offer APIs (Application Programming Interface);
- Offer means to download service information (parameters, script, and software), new protocols or up to date API;
- Support Virtual Home Environment using the same interface towards the user even in roaming.

The subsystem with access radio control is called UTRAN. His chief task is to create and to maintain the Radio Access Bearers for communications between UE and Core Network. Thanks to the RAB the Core Network equipment have the illusion of seeing a fixed connection with UE. They do not have the responsibility of the radio aspects. The UTRAN is sub divided in Radio Network Subsystems (RNS). A RNS consists of radio elements and control correspondents. Radio elements are Nodes B and control elements are the Radio Network Controller (RNC). Each Node B superintends a group of cells and participates to the Radio Resource Management (RRC) while the RNC manages and controls its domain radio resources (Node B linked). Node B and RNC manage also the Handover and the Macro difference (ability to maintain connection between terminal and network with more than one base stations).

The Core Network has all necessary network elements for switching and managing user's information. Besides, it is the basal platform of all the services supplied the UMTS users. It can depart, from a functional point of view, in the CS, PS, and IMS domains.

The CS domain has two fundamental network elements physically combined, they are the serving MSC/VLR and the GMSC. The serving MSC/VLR (serving Mobile Switching Center/Visitor Location Register), is responsible for the activities management about connection and circuit switching, for mobility management (position and registration up to date) and connection security. The GMSC (Gateway Mobile Switching Center), attends to the on and off connections for and from other networks. From the point of view of connection management, the GMSC routes toward serving MSC/VLR where the user is found. From the point of view of mobility management, the GMSC begins looking for position information, in order to find the correct serving MSC/VLR to set the call.

The PS domain has two network elements: the serving GPRS Support Node (SGSN) and the Gateway GPRS Support Node (GGSN). The SGSN nodes support packet communication towards the access network. They are responsible for the mobility management, the routing area adjournment, the position registration and the packet communication security. The GGSN nodes maintain the connection with other packet switching networks like Internet. The transport network that keeps the nodes GSN connected is the IP backbone and can be thought a true private Intranet. For this reason the IP backbone is separated from other networks through a firewall. The domain PS must also contain a DNS (Domain Name Server) for routing.

There are elements in the network whose function is not traffic transportation. These are: HLR (Home Location Register), AuC (Authentication Center) and EIR (Equipment Identity Register). These apparatuses contain information about addressing and identification related to the CS and PS domains and they are necessary for mobility management procedures. The VLR (Visitor Location Register) brings up to date position, security, etc.

2.4 IMS

The IP Multimedia Subsystem is based on the IP protocol. This new Core Network element is proposes to offer both traditional telephonys as well as packet switched services over a single converged packet based network The major shift in the UMTS architecture has been done with the introduction of Release 5. The wireless industry has well realized the advantages due to a converged network. Some of the reasons and benefits of a converged wireless network are:

- **Lower maintenance cost:** A single converged network based on IP results in reduced maintenance and operations costs. Again the management of the IP networking components is easier compared to telephony components due to open standard management platforms. The operators can manage the converged network with a smaller operational staff. Moreover, the operators need not invest in developing expertise in multiple technologies since the converged network will be based on one signaling and bearer network.

- **Enhanced Services**: The integration of voice and data networks offers opportunities for deploying enhanced multimedia services. Almost every service other than telephony services is available on the Internet today. The combination of Internet and telephony services opens a world of new revenue opportunities for service providers.

- **Rapid Service Deployment**: Development of a converged network based on a single standard allows for rapid deployment of new services. The configuration and co-ordination required to introduce new services is reduced due to the integrated management of wireless networks.

2.4.1 IMS Architecture

The components that will be presented are the Call Session Control Function (CSCF) components and the Media Gateway (MGW)/Media Gateway Control Function (MGCF) network components. UMTS Release 5 allows mobiles operating in packet mode to establish voice calls using SIP as the signaling protocol.

- **Call Session Control Functions:** The first key elements are the Call Session Control Functions (CSCF). The CSCF has taken the majority of the MSC functionality in the IMS architecture. The CSCF is analogous to the SIP server in the IETF architecture 3. Its function is to process signaling messages for controlling the user's multimedia session, to translate addresses and to manage the user's profile. The existing packet switched core network is used to support the bearer path for the multimedia session and the CSCFs are used to establish the sessions and perform features. The service control protocols are compliant with the Internet Engineering Task Force (IETF) based protocols. The protocol that is used for the majority of the signaling is SIP.The CSCF can play mainly three roles:

  - **Proxy-CSCF:** the Proxy Call Session Control Function (P-CSCF) is the mobiles first point of contact in the visited IMS network. The others only exist in the home network. The P-CSCF has two main functions. Its primary function is to be the Quality of Service Policy Enforcement Point within the visited IMS network. Its second responsibility is to provide the local control for emergency services.The P-CSCF forwards the SIP registration messages and session establishment messages to the home network. The Proxy-CSCF is analogous to the Proxy Server in the SIP architecture.

  - **Interrogating-CSCF:** The Interrogating Call Session Control Function (I-CSCF) is the first point of contract within the home network from a visited network. Its main job is to query the HSS and find the location of the Serving CSCF. This is an optional node in the IMS architecture. It could be configured so that the P-CSCF could contact the S-CSCF directly. The I-CSCF has a number of functions. It performs load balancing between the S-CSCFs with the support of the HSS. The I-CSCF hides the specific configuration of the home network form other network operators by providing the single point of entry into the network. The I-CSCF can also perform some forms of billing. If the I-CSCF is the gateway into the home network, it must support the firewall function.

  - **Serving-CSCF:** The Serving Call Session Control Function (S-CSCF) is the node that performs the session management for the IMS network. There can be several S-CSCFs in the network. They can be added as needed based on the capabilities of the nodes or the capacity requirements of the network. The S-CSCF in the home network is responsible for all session control, but it could forward the specific request to a P-CSCF in the visited network based on the requirements of the request. For

example, the visited network will be in a better position to support the local dialing plan or some other local service that the user may be interested in. The S-CSCF may be chosen differently based on the services requested or the capabilities of the mobile. One key advantage of this architecture is that the home network provides the service features. This means that the mobile is not restricted to the capabilities of the visited network as is seen in the current wireless network (i.e. if an MSC does not support a feature that you have subscribed to, you will not be able to use that feature.) This ability to allow the user to always be able to get access to their subscribed features is referred to as Virtual Home Environment (VHE.)

- **Home Subscriber Server:** As in the legacy mobile network, there is still a need for a centralized subscriber database. The Home Location Register (HLR) has evolved into the Home Subscriber Server (HSS.) The HSS interfaces with the I-CSCF and the S-CSCF to provide information about the location of the subscriber and the subscriber's subscription information. The HSS uses the only protocol that is not IETF based, the Cx interface 4 . The HSS and the CSCF communicate via the new Cx interface. The protocol on the Cx interface is not an IETF protocol, but it is IP based.

- **Media Gateway and Media Gateway Control Function:** In an environment where all of the sessions are between IP capable end user devices, there would be no need for anything other than the CSCF's and the HSS. In reality, there will be a very long transition period to completely eliminate the legacy PSTN and mobile networks. The IMS supports several nodes for interworking with legacy networks. These are the Media Gateway (MGW), the Media Gateway Control Function (MGCF), and the Transport Signaling Gateway (T-SGW.)

  - **Media Gateway Control Function:** The MGCF controls one or more MGW's, which allows for more scalability in the network. The MGCF manages the connection between the PSTN bearer (the trunk) and the IP stream. For simplicity the MGCF could be collocated with the MGW. The MGCF converts SIP messages into either Megaco or ISUP messages.

  - **Media Gateway:** If the MGCF is the brains of the operation then the Media Gateway (MGW) is the brawn. It is the workhorse that does the processing of the media bits between end users. Its primary function is to convert media from one format to another. In UMTS this will predominantly be between Pulse Code Modulation (PCM) in the PSTN and an IP based vocoder format. The MGW is likely to be a real-time hardware based platform. It is critical that it processes the bits as quickly as possible so that delay is not added to the transmission of the information.

  - **Transport Signaling Gateway:** The PSTN currently only understands SS7 and there is no incentive for it to provide support for anything other than SS7. SS7 has limitations and is not as flexible as IP. To prevent the need for the MGCF to support SS7 the Transport Signaling Gateway (T-SGW) was created. Its job is to convert SS7 to IP. The T-SGW converts the lower layers of SS7 into IP.

2.4.2 Distribution of CS functionality

In this paragraph we look at the at the essential functions of processing a call in the circuit switched world and see where these functions have moved to in the IMS network.

- **Call Control and Feature Processing:** In the circuit switched network the MSC did the call control to process a call. This function has been moved into the CSCF.

- **Billing:** At the end of the call the MSC must perform the billing function by generating a billing record. This function has been moved to the S-CSCF and the P-CSCF. The reason it is in both is so that the home network can bill the subscriber and the visited network can bill the home network for the subscriber's use of their resources.

- **Subscriber Profile Management:** The MSC was responsible for keeping a local copy of thesubscriber's profile that would be used to assist in processing a call. This function is now in the S-CSCF.

- **Mobility Management and Authentication:** The MSC performed mobility management to know the location of the mobile as it moves around the network. Since the mobile is communicating over an air interface, which cannot be protected, the MSC must also authenticate the identity of the user to ensure that it is not fraudulent. In previous releases of UMTS these functions were performed in the circuit switched network and in the packet switched network (i.e. this was performed in the MSC and SGSN separately.) It is redundant for both of these functions to be in both networks. In the IMS it will only be performed in the packet switched network (i.e. the SGSN.)

2.4.3 Services Architecture

The IMS services architecture allows deployment of new services by operators and 3 rd party service providers. This provides subscribes a wide choice of services. The S-CSCF is the anchor point for delivering new services since it manages the SIP sessions. However, services can be developed and deployed in a distributed architecture. Multiple service platforms may be used to deploy wide variety of services. The IMS defines three different was of delivering services:

- **Native SIP Services**: In the last few years, a wide variety of technologies have been developed by various organizations for developing SIP services. They include SIP servlets, Call Processing Language (CPL) script, SIP Common Gateway Interface (CGI) and Java APIs for Integrated Networks (JAIN). One or more SIP application servers may be used to deploy services using these technologies.

- **Legacy IN services:** While new and innovative services are required, the legacy telephony services cannot be ignored. The release '99 networks use CAMEL (Customized Applications for Mobile Enhanced Logic) Service Environment for deploying intelligent networking services such as pre-paid service and toll-free service.

- **3rd party services:** UMTS has defined Open Services Access (OSA) to allow 3rd party service providers to offer services through UMTS network. The OSA offers a secure API for 3 rd party service providers to access UMTS networks. Therefore, subscribes are not restricted to the services offered by the operators.

The S-CSCF uses the Cx interface to retrieve subscriber profile from the HSS. The S-CSCF interacts with different service platforms through IMS Services Control (ISC) interface that is based on SIP and its extensions. However, the OSA and CAMEL environments do not support ISC interface. The OSA Service Capability Server (SCS) performs mediation between the ISC and the OSA API. The IM-SSF performs mediation between the ISC and CAMEL Application Protocol (CAP).

2.4.4 Identification of Users

There are various identities that may be associated with a user of IP multimedia services. These may be:

- **Private User Identity:** Every IM CN subsystem user shall have a private user identity. The private identity is assigned by the home network operator, and used, for example, for Registration, Authorisation, Administration, and Accounting purposes. This identity shall take the form of a Network Access Identifier (NAI) as defined in RFC 2486. Its properties are:

  - The Private User Identity is not used for routing of SIP messages.

  - The Private User Identity is contained in all Registration requests, (including Re-registration and De-registration requests) passed from the UE to the home network.

  - The Private User Identity is a unique global identity defined by the Home Network Operator, which may be used within the home network to uniquely identify the user from a network perspective.

  - The Private User Identity is permanently allocated to a user (it is not a dynamic identity), and is valid for the duration of the user's subscription with the home network.
  - The Private User Identity is used to identify the user's information (for example authentication information) stored within the HSS (for use for example during Registration).

  - The Private User Identity identifies the subscription (e.g. IM service capability) not the user.

  - The S-CSCF needs to obtain and store the Private User Identity upon registration and unregistered termination.

2.4.5 Call Flows

The flux of call-setup signaling moves through the UTRAN from the UE towards the SGSN/GGSN, and is routed then to the CSCF and finally to the destination network (it may be another IMS network, or MGCF/MGW or another IP network). It is important to distinguish the elements that route messages from those that process them. When a UE sends a request to access a service, this is sent to a S-CSCF (through the P-CSCF and I-CSCF nodes) to ask the service. The SGSNs and the GGSNs have only routing tasks, they don't look the content of a message but check only the IP address so to be able to route it to destination.

The flux of media and data moves from the UE through SGSNs and GGSNs towards the destination network. This flux doesn't cross the CSCF network then. The IMS philosophy is to keep signaling separated from user's information.

2.4.6 How it works

The UE, so to be able to communicate with the IMS, must set up a SIP session. Lets now take a look to those that are the key steps that take to the setup of a session:

- **System Acquisition:** the first step is to power on the mobile and lock on to the UMTS system. Once the appropriate cell is selected, the UMTS mobile is ready to communicate signaling messages required to establish a data session.

- **Data Connection Setup:** once the system has been acquired, the next step is to establish the data connection or "pipe" to the SIP and other services. The UE does not know the IP address of the Proxy CSCF at this point to perform a SIP registration. The data connection is completed in a two step process using GPRS Attach and Packet Data Protocol (PDP) Context Activation message sequences.

  - **GPRS Attach:** the GPRS Attach process, as the corresponding procedure performed in the circuit-switched world (IMSI Attach), informs the network about the presence of the mobile terminal. Once registered, the network has knowledge of the location of the UE at a Routing Area level. The UMTS UE sends the Attach message to the Serving GPRS Support Node (SGSN), which includes the UE's International Mobile Subscriber Identifier (IMSI). The SGSN uses the IMSI to send a request to the UE's Home Location Register (HLR) for the uthentication parameters to help authenticate the subscriber. The HLR provides authentication information to the SGSN, enabling the SGSN to verify the veracity of the subscriber's IMSI. The successful completion of authentication procedure triggers the SGSN to send a location update (which provides the UMTS UE's IMSI) to the HLR and this triggers the subscriber's profile to be downloaded to the SGSN. This includes information such as the subscribed services, the QoS profile, any static IP addresses allocated and so on. The SGSN completes the Attach procedure by sending an Attach Complete message to the UE. With this step the UMTS network knows the location of the user.

  - **PDP Context Activation:** Once a UE is attached to an SGSN, it must activate a PDP address (in this case, an IP address) when it wishes to begin a packet data communication, including SIP services. Activating a PDP address sets up an association between the UE's current SGSN and the Gateway GPRS Support Node (GGSN) that anchors the PDP address. A record is kept regarding the associations made between the GGSN and SGSN. This record is known as a PDP context. This second and final step is required to establish the data connection or "pipe" to the Internet.. The activation of a PDP context activates an IP address for the UE; the UE may now exchange traffic using that IP address. By these means the path necessary to transport the SIP signaling to the P-CSCF through the GGSN is set. At this point the the UE comes to know the identity of the P-CSCF.

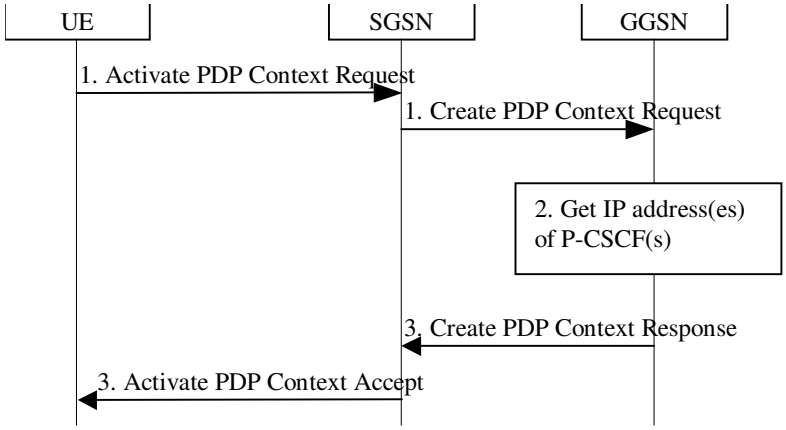

Fig. 2.2 P-CSCF discovery using PDP Context Activation signalling

- **Service Registration:** Before establishing an IP Multimedia session, the UE needs to perform the Service registration operation to let the IMS network know the location of the UE.

  - The reasons to perform registration are:

    - During service registration, the HSS acquires information on the current position of the user and follows updating its profile accordingly.

    - An authorization to be registered is necessary. The HSS checks user's authorization basing on its profile and on operator's limitations.

    - The UE needs a Serving-CSCF in its home network in order to obtain IMS services. During service registration, the home network selects a suitable Serving-CSCF for the UE and the subscriber profile is sent to the S-CSCF.

  - The flux of messages that starts with the sending of a REGISTER from the UE is as follows

    - To start the Service registration process, the UE sends the SIP Register message to the Proxy-CSCF. This message includes the subscriber identity and home networks domain name.

    - Upon receipt of the Register message, the P-CSCF examines the "home domain name" (e.g. ims_sprint.com) to discover the entry point to the home network (i.e. the I-CSCF) with help from DNS. The proxy sends the Register message to the I-CSCF with the P-CSCF's name, subscriber identity, and visited network contact name. The main job of I-CSCF is to query the HSS and find the location of the Serving CSCF. When the I-CSCF receives the Register message from the proxy, it examines the subscriber identity and the home domain name, and uses DNS to determine address of the HSS.

    - The I-CSCF sends a UMTS proprietary message, Cx-Query 7 , to the HSS with the subscriber identity, the home domain name, and the visited

domain name. The HSS checks whether the user is registered already. The HSS indicates whether the user is allowed to register in that visited network according to the user subscription and operator limitations/restrictions (if any). The Cx-Query Response is sent from the HSS to the I-CSCF.

- The I-CSCF sends the subscriber identity via the UMTS Cx-Select-Pull message to the HSS to request the information related to the required S-CSCF capabilities. This information is needed for selecting a S-CSCF. The HSS sends the required S-CSCF capabilities to the I-CSCF via the Cx-Select-Pull Response message..

- The I-CSCF, using the name of the S-CSCF, determines the address of the S-CSCF through a name-address resolution mechanism. The I-CSCF also determines the name of a suitable home network contact point, possibly based on information received from the HSS. The home network contact point may either be the S-CSCF itself, or a suitable I-CSCF in case network configuration hiding is desired. If an I-CSCF is chosen as the home network contact point, it may be distinct from the I-CSCF that appears in this service registration flow. I-CSCF then sends the Register message to the selected S-CSCF. The flow includes the P-CSCF's name, the subscriber's identity, the visited network contact name, and the home network contact point (if needed). The home network contact point will be used by the P-CSCF to forward session initiation signaling to the home network.

- The S-CSCF sends a Cx-Put message with the subscriber's identity and the S-CSCF name to the HSS. The HSS stores the S-CSCF name for that subscriber. The HSS sends the Cx-Put Response to the S-CSCF to acknowledge the sending of the Cx-Put.

- On receipt of the Cx-Put Response message, the S-CSCF sends the Cx-Pull message with the subscriber identity to the HSS in order to be able to download the relevant information from the subscriber profile to the S-CSCF. The S-CSCF stores the P-CSCF's name for use in session termination..

- In the 200 OK message, the S-CSCF sends the serving network contact information to the I-CSCF, who forwards it to the P-CSCF. The P-CSCF stores the information, and sends the 200 OK message to the UE. The I-CSCF releases all registration information after sending 200 OK.

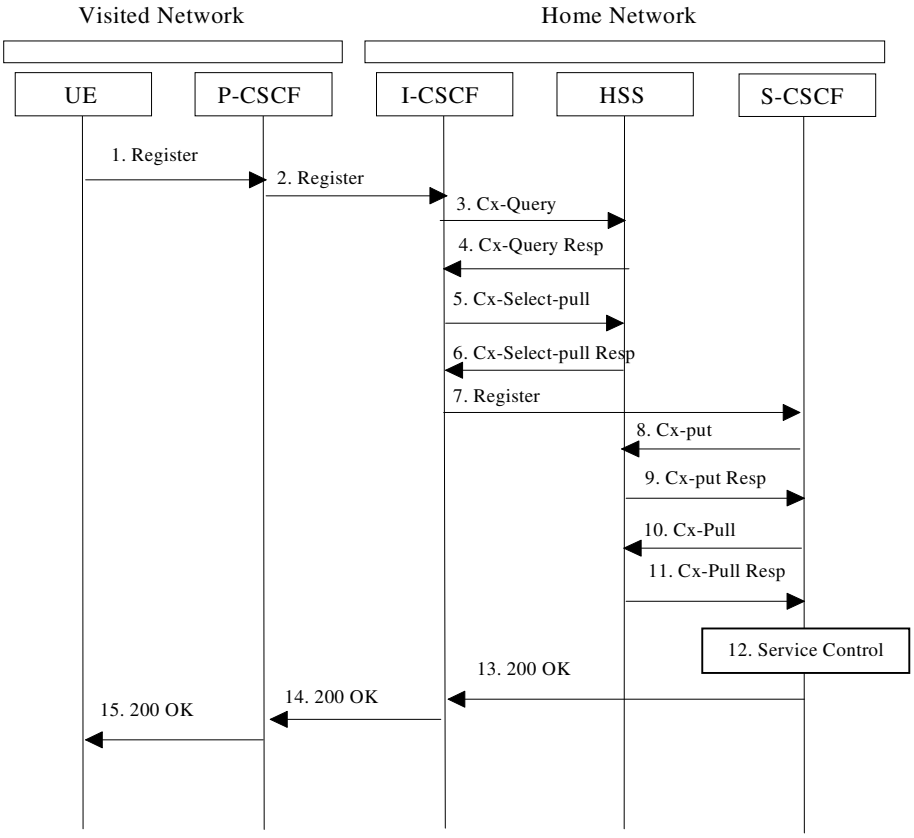

Fig. 2.3 Exchange of messages in the registration process

- **Session Setup Procedures:** In the IP Multimedia Subsystem specifications, an end-to-end session flow consists of three types of procedures: mobile origination, S-CSCF-to-S-CSCF, and mobile termination.

  - **Mobile origination:** in the example we assume that the UE is located in a visited-network when initiating the session:

    - The subscriber of the IMS mobile has either dialed digits or used a GUI on the mobile to determine the person that he wants to call. The mobile sends the SIP INVITE request to the P-CSCF. The INVITE message contains three key pieces of information. The first is the called party in the To header. The second key piece of information is the proposed SDP. This SDP may represent one or more media types for a multi-media session. The last piece of information is the From header that contains the calling party.

    - The P-CSCF remembers (from the registration procedure) the next hop CSCF for this mobile. This next hop is either the S-CSCF in the home network that is serving the visiting mobile, or an I-CSCF within the home network that is performing the configuration hiding function for the home network operator. This scenario assumes that the home network operator wants to keep its network configuration hidden, so the name/address of an I-CSCF in the home network was provided during

service registration, and the INVITE request is forwarded through this I-CSCF to the S-CSCF. The P-CSCF will look at the SDP portion of the SIP message and examine the proposed media types that were proposed by the calling party in establishing a session. The P-CSCF has the option at this point to remove some of the proposed media types based on the types of sessions the visited network wants to support.

- The S-CSCF validates the service profile, and performs any origination service control required for this subscriber. This includes authorization of the requested SDP based on the user's subscription for multimedia services..

- The S-CSCF also determines the location of the called party based on the information in the To header. As stated before, this could be to another IMS system, to a SIP capable device (a User Agent Client or a SIP proxy server) or to a Media Gateway Controller network to go to the PSTN. Which network is determined by using DNS to translate the address in the To header to an IP address. S-CSCF forwards the INVITE request to the destination. It is not clear from the standards if the S-CSCF is a stateless or stateful proxy. It seems logical to assume that it is a stateful proxy since the S-CSCF will support the billing function for the call session. By that same point, the P-CSCF is probably a stateful proxy as well.

- The called party responds with a provisional response that will include a SDP in the message body. This is the called party's suggestion on the media type.

- The S-CSCF forwards the SDP information to P-CSCF through I-CSCF.

- The P-CSCF again looks at the SDP field and removes media types that it does not want to support. The P-CSCF determines the type of resources that are required based on the media type that is requested.

- The P-CSCF then forwards the SDP information to the originating endpoint (i.e. the mobile).

- The mobile decides the final set of media streams for this session, and sends the Final SDP to P-CSCF. The P-CSCF will then perform the Policy Control Function and determine the resource requirements. There are a number of options that can happen at this point. The P-CSCF can send the policy information using a protocol like Common Open Policy System (COPS)8 to the GGSN. The P-CSCF could also just store this information in a database. The GGSN will send a request to the P-CSCF for "permission" to setup the requested resources.

- The P-CSCF sends the Final SDP message to the S-CSCF (via the I-CSCF if necessary.)

- The S-CSCF sends the Final SDP message to the called party.

- After determining the final media streams, the mobile initiates the reservation procedures for the resources needed for this session. The resource reservation takes the form of a secondary PDP context activation or a PDP context modification. At this point the mobile will request to take the current bearer path (that was used to send the signaling messaging) to a bearer that will support the media stream and the signaling. The GGSN will receive this request and either have the permission to establish (that was sent by the P-CSCF earlier) or the GGSN will send a request (via COPS) to the P-CSCF for permission to change the connection.

- When the resource reservation is completed, the mobile sends the COMET message (to show that the resource reservation has been successful) to the terminating endpoint, via the signaling path established by the INVITE message.

- The COMET message is sent to the l S-CSCF through the P-CSCF.

- The S-CSCF forwards the message to the UE.

- The destination party may optionally perform alerting. If so, it signals this to the originating party by a provisional response indicating ringing. This message goes through S-CSCF, I-CSCF, and P-CSCF to arrive at the originating mobile.

- When the destination party answers, the terminating endpoint sends a SIP 200-OK final response S-CSCF.

- The S-CSCF performs whatever service control is appropriate for the completed session setup.

- The S-CSCF sends a SIP 200 OK response to the P-CSCF through the I-CSCF.

- The P-CSCF informs that the reserved resources for this session should now be available.

- The P-CSCF sends SIP 200 OK response to the session originator.

- The originating mobile starts the media flow(s) for this session.

- The mobile responds to the 200 OK with a SIP ACK message, which goes through P-CSCF and S-CSCF. S-CSCF forwards the final ACK message to the terminating endpoint.

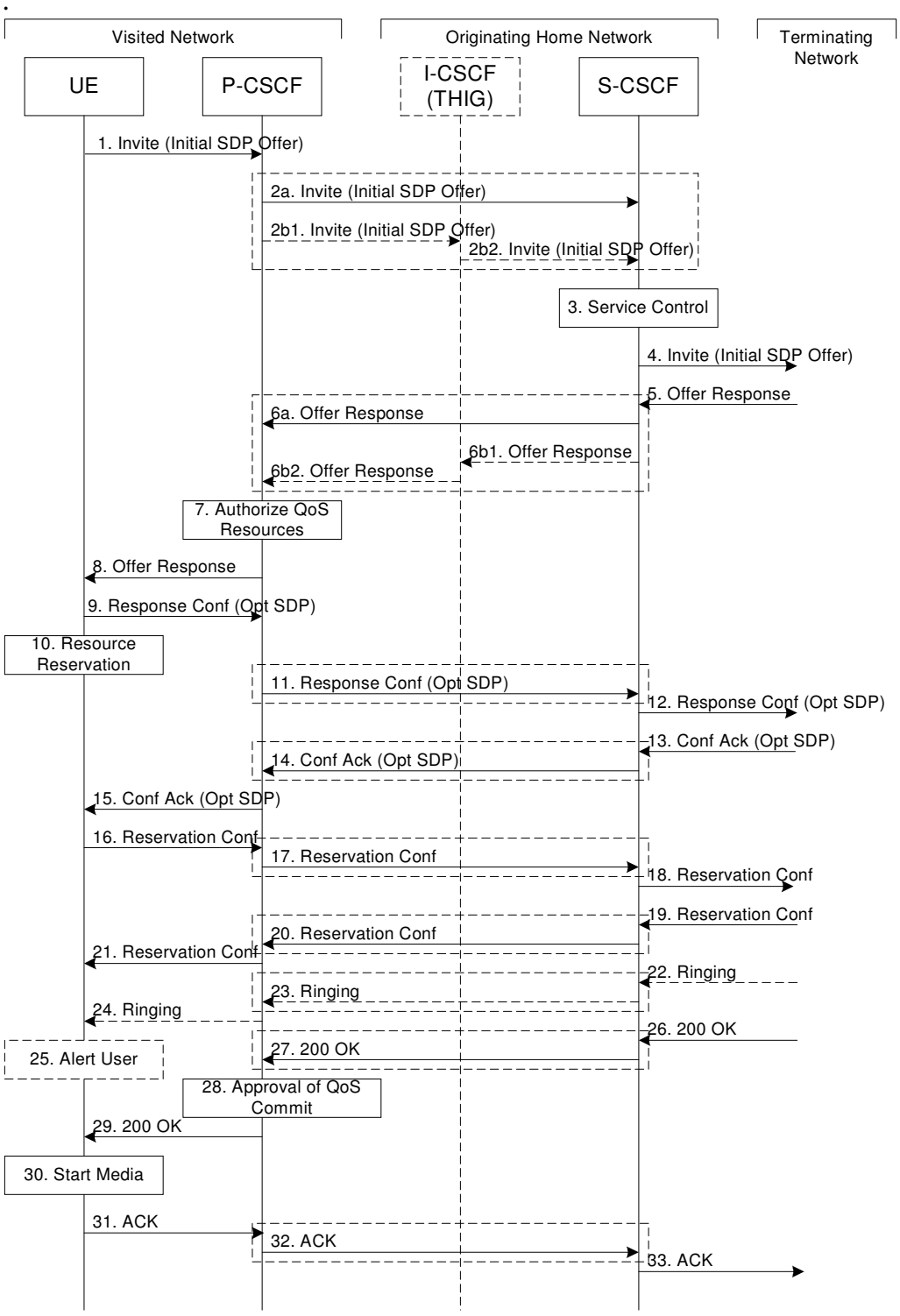

Fig. 2.4 Session setup started from a mobile located in a visited network

¬ Mobile termination is quite similar to the mobile origination except that the signaling is in the reverse direction. This scenario assumes that the mobile is located in a visited network. The use of the I-CSCF is, again, optional. If the originating network operator wants to keep the network configuration private, then the S-

CSCF will choose an I-CSCF, who will perform firewall function and pass messages to the P-Destinations CSCF.

- **INVITE:** The originating party sends a SIP INVITE message through the network to the destination mobile.
- **SDP negotiation:** The two end parties negotiate the media characteristics (e.g. number of media flows, codecs, etc.) for this session and make a decision on the media streams they will support for this session.
- **Resource Reservation:** The network reserves the necessary resources for supporting this session, after the media characteristics for this session have been agreed on.
- **Session setup confirmation:** Once resource reservation is completed successfully, the terminating mobile sends a SIP 200 OK final response and the originating point replies with a SIP ACK message to confirm the session setup.
- **Session in progress:** Once the P-CSCF approves that the reserved resources can be used, the mobile starts the media flow. After the session setup is confirmed, the session is in progress.

- PSTN/Legacy Networks Interaction: The IMS networks need to interact with PSTN so thatIMS users can establish services to PSTN users. The interworking between IMS networks and PSTN/legacy networks occur at two levels: One is the user plane level and the other is the signaling plane level. In the user plane, interworking elements are required to convert IP based media streams on the IMS side to PCM based media streams on the PSTN side. The Media Gateway (MGW) element is responsible for this function. The Media Gateway Control Function (MGCF) through the Megaco protocol controls the MGW elements. On the signaling plane level, the SIP signaling needs to be converted to legacy signaling such as ISDN Signaling User Part (ISUP). The MGCF is responsible for converting SIP signaling to legacy signaling such as ISUP. The MGCF is responsible for transporting ISUP signaling messages to a Trunking Signaling Gateway (T-SGW) over IP transport bearer. The T-SGW transports these ISUP messages over the SS7 bearer to either the PSTN or the legacy wireless networks. Please note that MGCF and T-SGW are logical functions. These functions may be implemented in one physical box.
    - The UE initiates the session by sending a SIP INVITE request, which includes the initial SDP. This request is forwarded all the way to the MGCF.
    - The MGCF initiates a Megaco interaction to pick an outgoing channel and determine the media capabilities (e.g. encoding format) of the MGW.
    - The UE and the MGCF negotiate the media characteristics (e.g. number of media flows, codecs, etc.) for this session.

- After determining the final set of media streams for this session, the UE initiates the reservation procedures for the resources needed for this session. Once the resources have been successfully reserved, the UE informs the MGCF.

- The MGCF communicates with the PSTN through the T-SGW to set up trunks for this session. The MGCP uses the legacy protocol such as ISUP to setup trunks.

- MGCF alerts the UE that the destination party has been contacted.

- The PSTN informs the MGCF that the destination party has answered.

- MGCF initiates a Megaco interaction to make the connection in the MGW bi-directional.

- MGCF sends a SIP 200 OK final response and the originating UE replies with a SIP ACK message to confirm the session setup.

- The UE starts the media flow for this session after it receives the SIP 200 OK response.

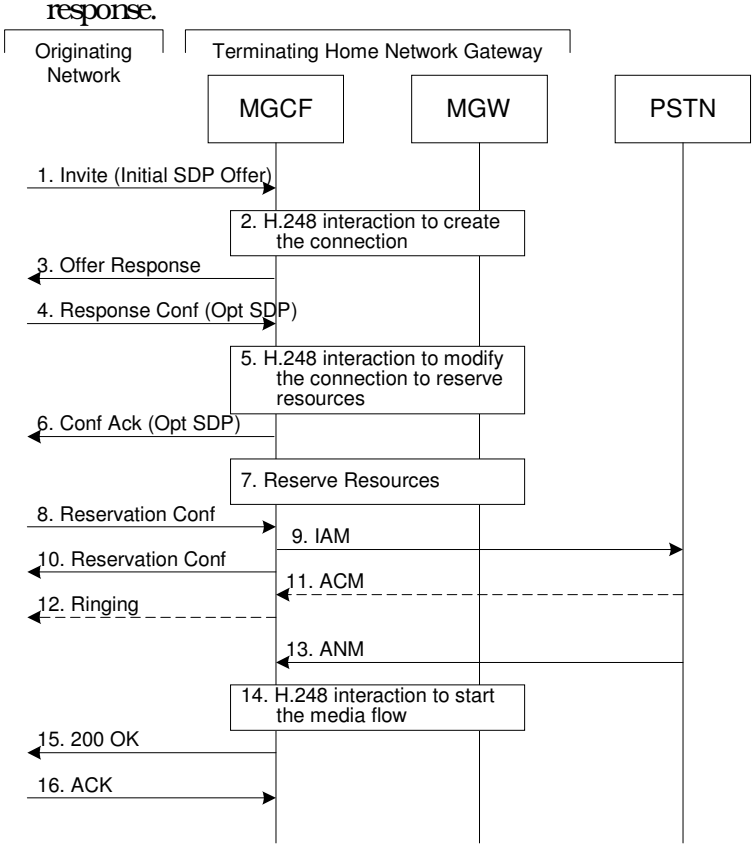

Fig. 2.5 Signaling call flow towards a PSTN user

3. Reference Scenario

3.1 Exploit Project

Mobile Network Operators are concerned with the risks that may come with the UMST. The need of multiple and heterogeneous technologies (PS and CS domains) increase costs and introduces significant interworking. The introduction of the PS domain can lead to a partial loss of control over applications and revenue streams. This may result in a loss of revenue and profit potential and in being relegated to an "access provider" role.

The Exploit project wants to demonstrate the advantages of the adoption of an IMS platform in this scenario. The advantages are seen in terms of:

- New services rapid development and deployment;
- Seamless to integration of existing services;
- Usage of a 3GPP standard platform.

The effort of the project is the definition, design and development of new services that may be tested on the IMS platform.

3.2 IMSes Platform

The Siemens IMS@vantage is part of the Siemens IP-multimedia service architecture. It provides MNOs (Mobile Network Operators) with the facility to offer voice-enabled multimedia services with packet-oriented networks. IMS@vantage is an access independent solution that works together with a packet-oriented access network, whether it is a UMTS Release 4 packet-switched domain, a GPRS network, a GSM network using enhanced data rates for GSM evolution (EDGE) or WLAN. This allows operators to effectively use the IMS@vantage core infrastructure as their control platform not only for UMTS radio access, but also for EDGE, GPRS, and license-free hot spot radio technologies such as WLAN access.

It is designed in such a way that it can also be used complementary to TDM networks such as global system for mobile communication (GSM) and circuit-switched UMTS. IMS@vantage can be easily introduced in parallel to such existing voice networks, and be used specially for multimedia applications or higher-bandwidth applications.

3.2.1 Platform for a Variety of Network Services

IMS@vantage provides five different types of services for multimedia service provisioning:

- **Basic network services:** registration (login as subscribed end-user), authentication and service authori-zation, mobile call control, and application triggering;
- **Interworking to legacy networks:** interworking to GSM and public telephony networks;
- **Enhanced network services (service enablers):** conferencing, presence information, payment, location information and click-to-call possibilities;
- **Enhanced user services and applications:** applications, which bundle enhanced network services to create new applications for mobile and fixed networks;
- **Operator services:** network element management and subscriber administration.

3.2.2 IMS@vantage Experimental System (ES) Architecture

IMS@vantage is the infrastructure needed to run new data and multimedia services. It consists of the IMS@vantage control network, the WLAN access network and software on the end-user devices. It provides the possibility to integrate new services and appli-cations developed by the mobile network operator or those provided for the MNO by ex-ternal application service providers (ASP). The following figure illustrates the architecture of the IMS@vantage ES, which is a laboratory test-system.

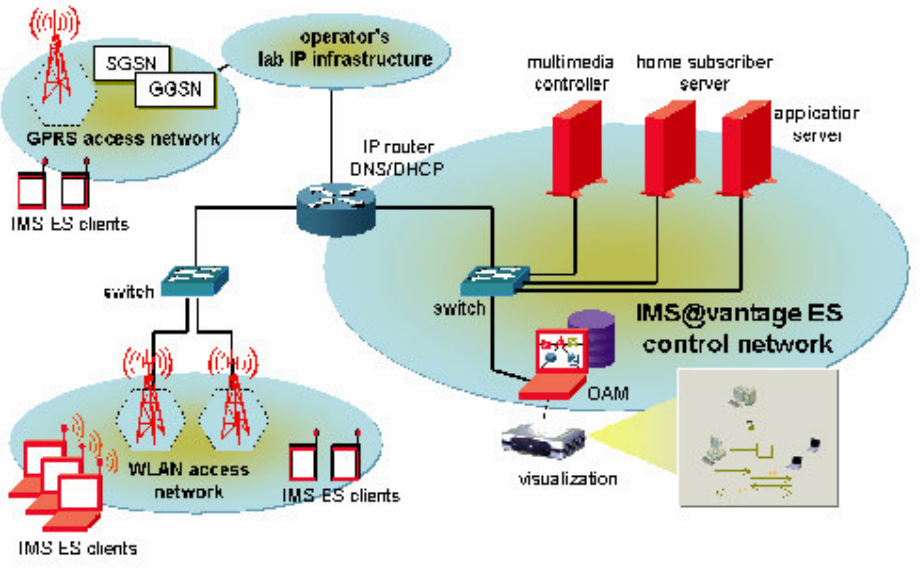

Fig. 3.1 The architecture of the IMS@vantage Experimental System (ES)

- **Access Networks:** the IMS@vantage ES allows access via:
  - GPRS;
  - WLAN.

GPRS access can be enabled via the MNO's existing second generation (2G) network. The infrastructure, usually consists of the following elements:

- GSM base stations;
- Serving GPRS support node (SGSN);
- Gateway GPRS support node (GGSN).

The elements contained in the WLAN access network are:

- The base stations (BS), also called WLAN access points, which enable the end-user devices with their WLAN access cards to connect to the WLAN access network;
- The switch that connects the WLAN access network with the router and the IMS control network.
- Notebooks with WLAN access cards;
- Handhelds (mobile phones and personal digital assistents, PDA) with WLAN or GPRS access facilities.

Fig. 3.2 WLAN Access Point

- **Control Network:** the IMS@vantage ES control network is the core of the IMS@vantage ES. It consists of the following elements:
  - The multimedia controller, which contains the following functions in a modular way:
    - the proxy call state control function (P-CSCF);
    - the interrogating call state control function (I-CSCF);
    - the serving call state control function (S-CSCF).
  - The home subscriber service (HSS) server;
  - The application server;
  - The operation, administration and maintenance (OAM) server with the rack console;
  - The router, which apart from routing contains the following functions:
    - domain name system (DNS);
    - network time protocol (NTP);
    - dynamic host configuration protocol (DHCP).

3.2.3 Element Description

The description of the IMS@vantage Experimental System functional elements follows:

- **Multimedia Controller:** one of the key elements of the IMS@vantage ES control network is the multimedia controller. The multimedia controller is responsible for call signaling. All three roles of a call state control function (CSCF), defined by the 3rd Generation Partnership Project (3GPP), are realized in the multimedia controller in a modular way.
  - **Proxy CSCF (P-CSCF):** it is the first point of contact for an end-user device when the IMS is contacted from inside the same administrative domain as the IMS. The P-CSCF:
    - Forwards SIP messages;
    - Translates IDs other than SIP unified resource identifiers (URI) into SIP URIs.

    In IMS@vantage ES the address of the P-CSCF is pre-configured on the end-user device.
  - **Interrogating CSCF (I-CSCF):** it is the first contact point, when the IMS is contacted by an IMS of another administrative domain. The I-CSCF:
    - Forwards SIP messages;
    - Assigns an S-CSCF (e.g., during registration);
    - Can be configured to conceal the internal network configuration, capacity and topol-ogy.

- **Serving CSCF (S-CSCF):** it performs session control and service triggering, it:
  - Acts as registrar (a server that accepts register requests);
  - Forwards SIP messages;
  - Interacts with the application server;
  - Authenticates according to HSS data.

- **Home Subscriber Service (HSS) Server:** the home subscriber service (HSS) is a database that contains subscriber-related information. This database includes data for:
  - Identification;
  - Authorized services;
  - Subscribed services.

  On the delivery of IMS@vantage ES the data for 50 subscribers is configured in the HSS (standard configuration).

- **Operation, Administration and Maintenance (OAM) Server:** OAM functions are HTTP-based and can therefore be performed by the administrator from the rack console or remotely from any workstation. The graphical user interface (GUI) greatly simplifies the OAM activities associated with the different network elements.
- **Application Server:** the IMS@vantage ES control network includes a multimedia application server over which applications are made available. Enhanced network services, such as call for-warding and network-initiated calls are realized with the help of this server. The application server is a SIP application server connected via SIP to the multimedia controller and offering various application programming interfaces (API) for service creation. Three APIs are available:
  - Java programming language (JPL) API;
  - Call processing language (CPL) API;
  - Hypertext transport protocol (HTTP) API.

- **Dynamic Host Configuration Protocol (DHCP):** domain name system (DNS) for name resolution, network time protocol (NTP) for time synchronization, and dynamic host configuration protocol (DHCP) for dynamic IP-address assignment are common elements/protocols of IP networks and for this reason are not described here in more detail.

3.3 Client Description

For development and testing purposes, two PDAs have been used: Compaq's iPAQ 3630 Fujitsu-Siemens' Pocket Loox (the target PDA for the IMS client).

The iPaq is powered by a 206 MHz Intel StrongARM 32-bit processor, 32 MB of RAM and 16 of ROM. The expansion-pack system allows to add functionality to suit particular needs. This expansion system has been used to achieve WLAN connectivity being able to attach a PCMCIA card slot. The problem of offering an integrated connectivity has been solved with later models. The color 240x320 TFT screen produces 4,096 colors (12 bit resolution per pixel), it is possible to view the screen from many angles. The Compaq iPAQ includes a microphone and a speaker as well as an audio-in jack. It features an infrared port for wireless data transfer. It connects to either USB or serial ports and allows to input data in your own handwriting, by soft keyboard, by voice recorder, or through inking.

The Pocket Loox features most of above properties and has some further ones. It is powered by a faster 400 MHz processor based on Intel's Xscale Microarchitecture, 64 MB of

RAM and 32 of ROM. It offers integrated Bluetooth connectivity and features of an integrated Compact Flash that was used to connect to the WLAN network. The touchscreen display is 240 by 320 pixels wide with 65536 colors.

Both of these PDAs permit pc connectivity via USB and serial interfaces, using Microsoft's ActiveSync software and both of them are powered with the Pocket Pc 2002 OS.

Fig. 3.3 PDA user

3.3.1 Pocket PC 2002

Pocket PC 2002, Microsoft' s PDA operating system, is more stable than the previous version and includes MSN Messenger and a remote access client. Pocket PC 2002 offers a Windows XP-like user interface, which includes 3D icons, pop-up alerts that appear atop running applications, and a customizable Today screen. The X in the upper-right corner of all program screens has been introduced, which closes the current application screen without shutting down the application itself, thereby freeing up RAM. Among the included software Pocket Windows Media Player, which supports streaming media, and a handheld version of MSN Messenger. The OS also now has Transcriber, handwriting-recognition software that permits to write whole words or even sentences rather than single characters, along with another new character recognizer. Having a mobile Internet access, Pocket PC' s terminal client grants access to Windows NT and other servers and Windows 2000-level password security for confidential data.

By summer of the present year a new edition of the Pocket Pc OS is expected; Pocket Pc 2003. The plus of this new operating system is its better performance on PDAs powered with the new Xscale processor. In fact, even if more powerful, applications thought for the StrongARM processor perform very poorly on this new platform with Pocket Pc 2002.

IMS ES client software is Java based and runs on top of the JEODE Java Virtual Machine installed on the PDAs.

3.3.2 Java Virtual Machine

The java language has been introduced by Sun Microsystems in 1995 with the purpose of deploying dynamic content into web pages. The main idea was that of creating web pages that not only contained static text and images, but also dynamic multimedia (video and animations) supported by interactivity. Java has then extended its application to web server programming and to software targeted for consume electronics (cell phones, PDAs, etc.). Java is a C/C++ based language. This, together with some of its features as no platform dependence and the extensive offer of packages explains why it is so widely used.

Since PDAs are definitely different from PCs and workstations in terms of hardware and software capabilities, a range of java virtual machines specifically targeted for these devices has been developed.

The first jvm targeted for limited devices has been PersonalJava. This jvm is based on the java 1.1 specification. Supported libraries are limited with respect to those available on a PC version and are not updated to the Java 2 specification. Included packages are:

- java.awt;
- java.util;
- java.io;
- java.rmi.

More recent, Java 2 Micro Edition (J2ME) is Sun' s version of Java aimed at machines with limited hardware resources such as PDAs, cell phones, and other consumer electronic and embedded devices. J2ME is aimed at machines with as little as 128KB of RAM and with processors a lot less powerful than those used on typical desktop and server machines. J2ME actually consists of a set of profiles. Each profile is defined for a particular type of device (cell phones, PDAs, microwave ovens, etc.) and consists of a minimum set of class libraries required for the particular type of device and a specification of a Java virtual machine required to support the device. The virtual machine specified in any profile is not necessarily the same as the virtual machine used in Java 2 Standard Edition (J2SE) and Java 2 Enterprise Edition (J2EE). A profile in itself does not do anything; it just defines the specification. Profiles are implemented with a configuration. A configuration may be thought as an implementation of a J2ME profile for a particular type of device such as a PDA. Some of the configurations currently available are the CDC (Connected Device Configuration) that targets high end PDAs and the CLDC (Connected Limited Device Configuration) that targets less powerful devices. The Connected Device Configuration will be the basis for the successor to Sun's previous attempt at a Java environment for consumer device, PersonalJava. PersonalJava has had limited success in consumer market segments.

The jvm that has been used within the project is an improved and optimized version of PersonalJava for Windows CE. This jvm, called JEM-CE is provided by Insignia. The class libraries included in the JEM-CE (the jvm is based on Insignia's Jeode) runtime component are:

- java.applet;
- java.awt and sub-packages;
- java.beans;
- java.io;
- java.lang and sub-packages;
- java.math;
- java.net;
- java.rmi and sub-packages;
- java.security;
- java.sql;
- java.text;
- java.util and sub-packages.

The main difference in programming on this platform, with respect to PCs and workstations, is the GUI. PersonalJava does not provide the Swing package and so it is necessary to get back to the AWT (Abstract Windowing Toolkit) package, deprecated in Java 2, in order to use graphics.

To be able to deploy an application on the PDA, the necessary steps are:

- Compile the program including the necessary packages in the classpath using jdk1.2.2 (with the JEM-CE version of jeode it is expressivily asked to use this version of the jvm, instead of the 1.1.8 used with the normal version);

- Copy the generated .class files on the PDA and run the application calling the jvm from a .lnk file (a sort of batch file on the PDA, more or less must be invoked the same command that would be executed on the DOS shell in the Windows environment).

Sun offers an emulator, that may be downloaded from its site, that directly on the PC performs a check whether the written and compiled code on the PC is compliant with the PersonalJava specification. Once confident with the environment, however, it is easy to deploy directly the bytecode on the terminal.

3.3.3 Jeti Application

The IMS client software provided by Siemens is Jeti. Jeti is a java application that runs on JEM-CE for PocketPc 2002. The IMS client permits authorized users to register on the IMSes platform and to enjoy Instant Messaging, two-party Chat and Presence services. The version installed on the PC permits to perform real-time audio/video calls too. Here follows the GUI of the client as it looks on the PDA.

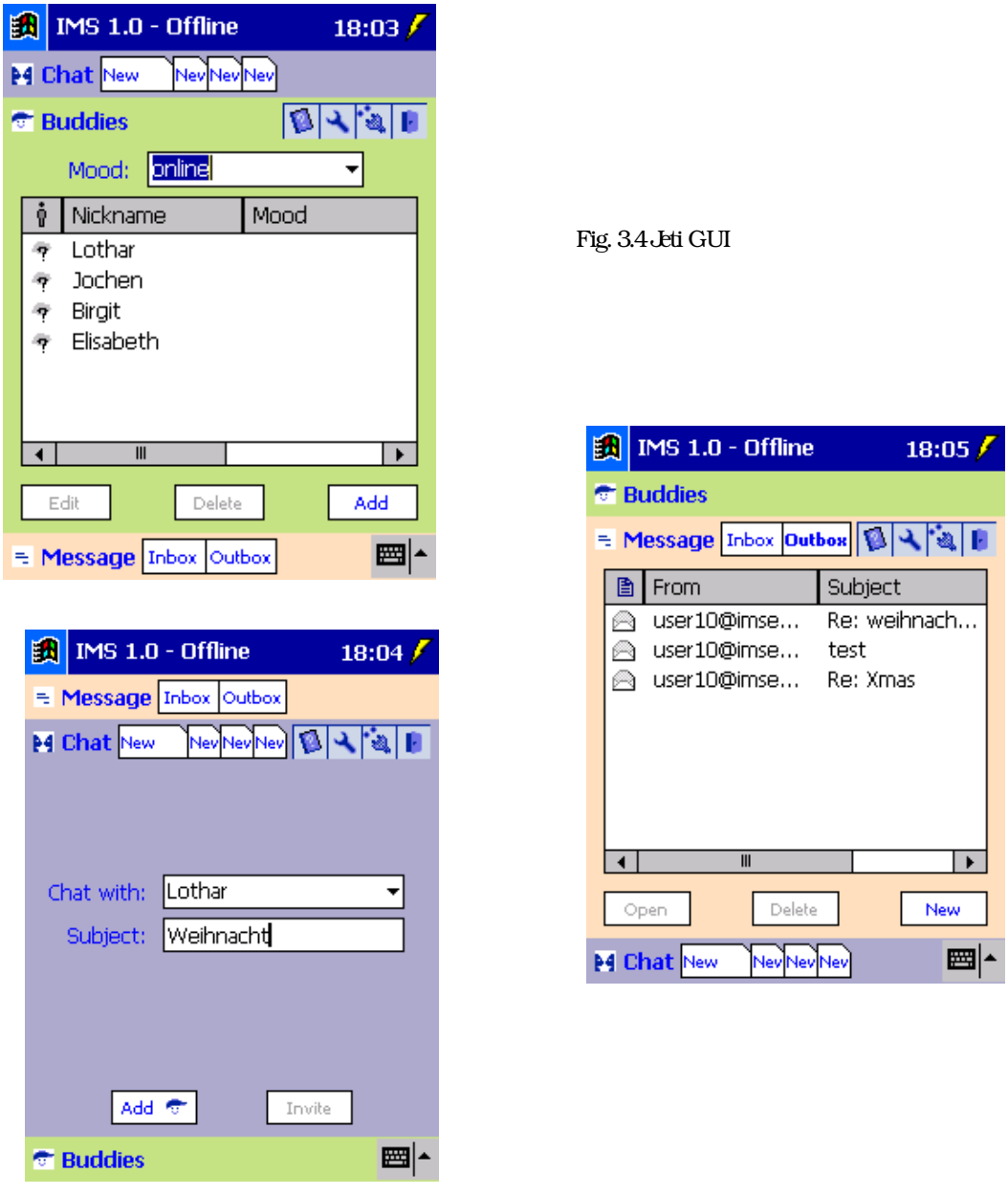

Fig. 3.4 Jeti GUI

The GUI is based on a rotating spaces environment. This makes the client user-friendlier and limits the waste of resources (memory and CPU). The four buttons on the up-right angle are linked to the following tasks (from the leftmost to the right): online help, configuration data, plug in management and exit. The configuration data permits the user to insert the P-CSCF with which the terminal talks, the registration settings (registration expire time, SIP URI), the availability of the user to publish its presence information or not.

The client has been built on java and embedded visual c++ (the use of c++ is tied to platform dependent issues and to the request of major performances for the GUI).

3.4 Selected Services

As already stated at the beginning, one of the main tasks of the project has been the selection of SIP based services to be demonstrated by Exploit. In general, the consumer services may be divided in the following segments: "Information" services like infotraffic and infoweather, "Communication" services like audio and video communication, "Entertainment" services like rolegames and lotto, "Transactions" based services as e-banking and e-commerce. All of the listed services are very promising, but those that combine Information and Entertainment service features have been considered as the most challenging from the IMS point of view. Infotainment services allow to demonstrate SIP value added in the support of person-to-person and person-to-machine communications with respect to other protocols. Other services that have been analyzed by the project are those of the Communication area, in particular the following services have been identified:

- Push messages to Close User Groups (push services for a community of users);
- Virtual Bulletin Board (a bulletin board keeps users informed about an event of interest);
- Polite Telephony (thought for applying privacy filtering criteria to calls);

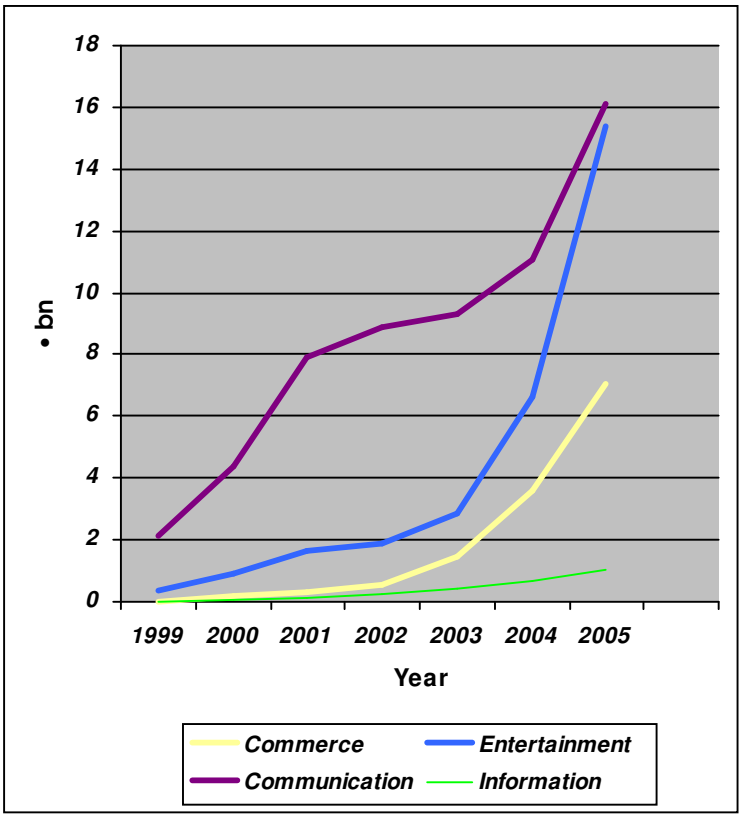

Fig. 3.5 Expected revenues (Durlacher UMTS Report)

- Click To Dial (web initiated calls, very useful for call centers for example);
- Email Breakout (delivery of IM through SMTP protocol, in case the user isn't online);
- SMS Breakout (delivery of IM through SMS);
- Text To Speech (conversion of IM content to speech);
- Click To Play (it gives the user the opportunity of enriching the session setup of a call with audio, images and text);
- Wake Up Call (opportunity of scheduling calls from a buddy list);
- Push To Talk (half-duplex calls at lower rates, in a walkie-talkie fashion);

Finally, the project decided to focus on infotainment services and among the candidates the following applications have been selected:

- Fantacalcio;
- Rolegames;
- ECity.

Fantacalcio is very popular game in Italy, linked to the soccer championship. The main reason that has lead to this choice has been its popularity. Each user build a fanta-team starting with an initial budget and takes part to a championship. Among the tasks that are performed by a user in taking part in a game there are the trade of players and the exchange of opinions on players and teams between the participants. The systems has the responsibility of keeping the user updated with the latest news on players and scores and to coordinate the game.

Rolegame, chosen for its many players around the world and in the Internet, is a game in which fantasy worlds and characters take place. Each player choses its character (a Dragon, a Witch or an Elf, for example) and starts its journey in a fantasy world. He will find battles and

enemies to beat on its way. Players interact among them in the exchange of cards (cards may represent a potion, or vital strength or a weapon) and their trading. The system has the responsibility of keeping the user updated with the latest news on players and scores and to coordinate the game.

ECity groups a set of public utility services designed for communities of people living on or visiting a city. Two applications of Ecity have been implemented for the first phase of the project: Carsharing and BuddyFinder. The former was thought as a response to the increasing disappointement among people due to traffic jams. The latter can help a user to create new relations. The user inserts the main characteristics of the type of person he is serching for (the filter criteria may be based on age, sex and hobbies for example) and the system returns a list of buddies that responsd to that criteria. The other applications of Ecity are: Ads&Advisors, Infotraffic and InfoATAC.

All these service need the capabilities that are provided by the IMS ES. Instant Messaging has a part in the playing of games, in the trading tied to it and in the exchange of information among users. Chat sessions permit the exchange of IM, it is the means by which, for example, Rolegames battles are performed. Presence information has a role in the detection of other players and in setting availability to play.

4. Jeti Client Architecture

The Exploit client has been designed based on the existing IMS client, i.e. Jeti. Jeti has been designed with the intent of providing mobile network operators with the ability to add their own services without being forced to design and implement a complete new client for the IMS ES. Registration with the IMS and basic services such as Chat, IM and Presence are available in Jeti.
In the following chapters a description of the Jeti architecture will be given. This represents the basis for the Exploit client design and it permits an understanding of the reasons for the changes and add-ons that have been done for the project.

4.1 Reference Architecture (JETI)

The architecture of the all-java version of Jeti is as follows.

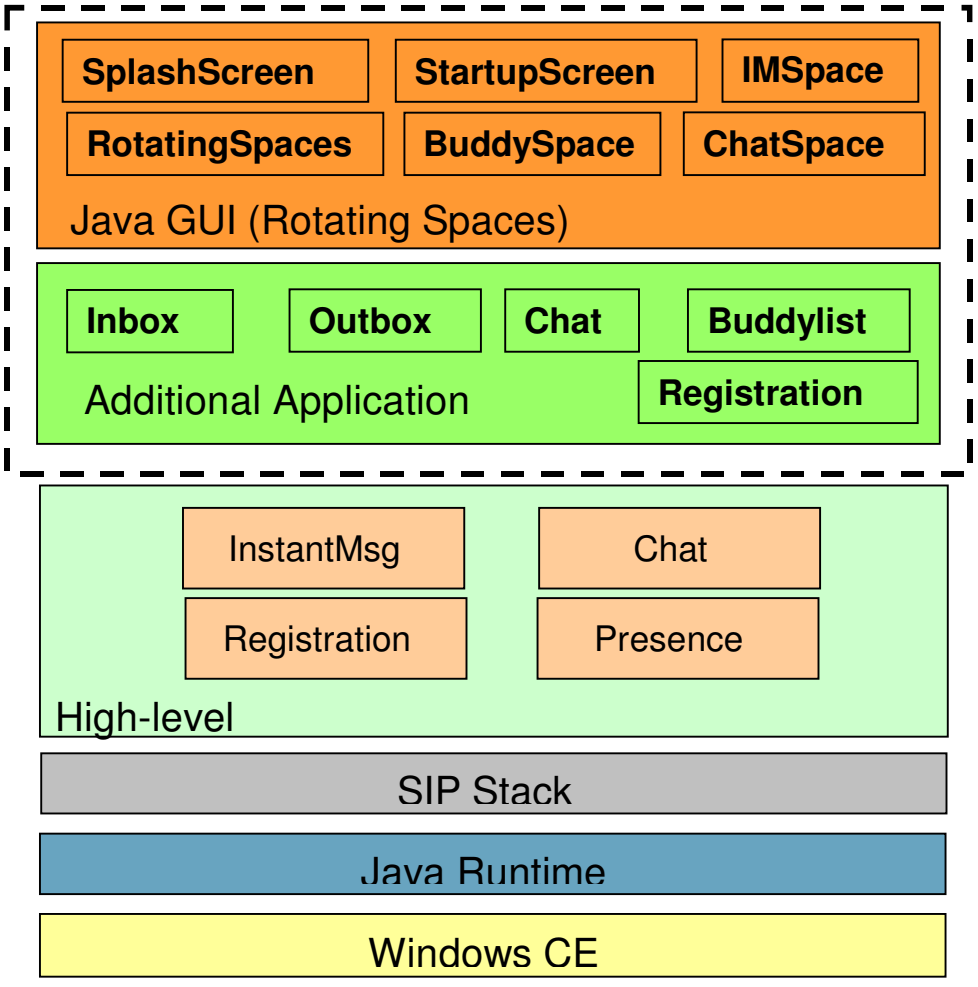

Fig. 4.1 All-java Jeti architecture

The Jeti application is divided in several strata. It provides High Level API that sits on top of a Jain-like SIP Stack. This API provides features such as Registration, Presence, Chat and IM. On top of this API, other functionalities have been added in support of the above services: Inbox (for IM), Outbox (for IM), Chat management, Buddy list and Registration management. The GUI is the topmost stratum in the figure; it is based on the "rotating spaces" model (i.e. three

spaces, IM, Chat and Buddy list that exchange control on the screen; this is because the screen space is limited and it is not possible to have the three services active all together).

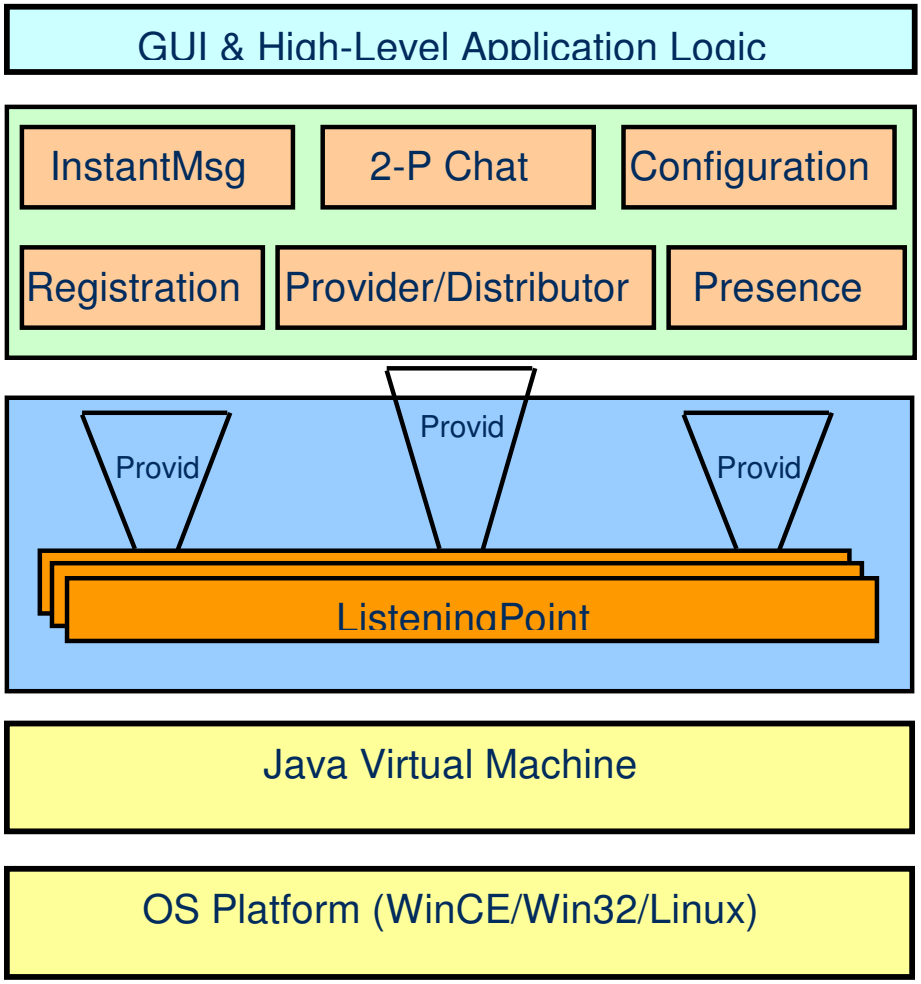

Fig. 4.2 Closer look on the High Level API and on the SIP Stack

The architecture of the SIP Stack is close to JSR32, but it is not a true SIP Jain implementation. The main differences from the JAIN SIP APIs are:

- No support of the JAIN interface;
- No support of the SipFactory interface;
- Limited support of SIP headers concerning the SIP parser (but full read/write access to any SIP header);
- Support to only UDP as transport protocol.

The reasons that at the time of development led to this choice are:

- The focus of JSR32 which is Java 2, and not PersonalJava or J2ME;
- The necessity to keep the stack footprint as small as possible, because of the limited environment it runs on (PDA).

The SIP stack follows the RFC 2543 specification. The SIP Stack is configurable using a Java property file. Configurable properties are: the P-CSCF address, P-CSCF port, client port, log file name, log level.

The developer may have access to the SIP Stack through the High Level API; this permits the MNO to use the existing services available (IM, Presence, Chat) and to build on these new applications and services.
The SIP Stack may be accessed directly too. This choice provides a better flexibility in the change of existing services.

- **SIP Stack access via High Level API:** the connecting point between the High Level API and the SIP Stack is the ProviderDistributor class. Each SIP request must be sent through an instance of this class, implemented as a Singleton pattern. The ProviderDistributor class provides a Plug-In interface for additional customer specific services, which need a connection to the CSCF via SIP protocol. This is the only available approach if the user wants to re-use existing services already built in the High Level API.
- **Direct access to the SIP Stack:** the class SipStack is implemented as a singleton pattern and available via getter method. The Provider/Listener model is used for sending/receiving of the SIP requests/responses (as in Jain SIP). The SipProvider class provides methods for sending SIP requests and responses. The class SipListener defines an interface, which is necessary for the stack to inform a user of incoming SIP requests/responses/timeouts. Class ListeningPoint provides SIP messages that come from a specified UDP port. Examples:
    - ¬ Creating and sending a SIP request:

```
//creating the SIP request
Request request = new Request(Request.MESSAGE);
request.setRequestURI(remoteUri);
//adding SIP headers
request.addHeader(new HeaderObject(HeaderObject.TO,
remoteUri.toString()));
request.addHeader(new HeaderObject(HeaderObject.FROM,
myUri.toString()));
...
//setting the body of the SIP method
request.setBody("Merry X-mas and Happy New Year"));
//sending the request using SipProvider object
//(direct access to the SipStack)
sipProvider.sendRequest(request);
//or sending the request using ProviderDistributor
//(access via High-Level API)
providerDistributor.sendRequest(request);
```

    - ¬ Receiving and analyzing a SIP message (response):

```
public void processResponse(SipEvent event) {
            //getting a Response object from the event
            Response response =(Response)event.getMessage();
            //getting SIP response code
            int status = response.getStatusCode();
            //getting transaction ID of the SIP transaction
            long tid = event.getTransactionId();
            //processing of a response to the MESSAGE
            //(Instant Message)
            if( isMessageId(tid) ) {
                  if(response.getStatusCode() >= 300) {
                        //inform the user about failed
                        //delivery
                  }
                  else if(response.getStatusCode() >= 200) {
                        //inform the user about successful
                        //delivery
                  }
            }
      else if (...) { ... }
```

```
}
```

As already explained, it is possible to re-use the available services provided by the High Level API. Here is a more in depth view of this package:

- **Registration:** this feature provides a way to Register/Deregister the client application from the IMS. All services available in the High Level API require registration. The authentication in the IMS is performed through a challenge/response mechanism. This service provides an automatic re-registration feature. There is a single point of access to registration features:

```
static Registration getRegistration()
      throws ConfigurationException,
             SipException
```

  The methods used for registration are blocking, waiting for a response or a timeout. The properties used from the configuration file to perform this task are the user's SIP address, authentication information and the registration expire time (by default two hours). Reregistration is based on an asynchronous mechanism, the observer pattern. An interface, RegistrationListener, must be implemented and one of its methods registrationSucceed and registrationFailed may be invoked depending on the receipt of a success or failure response. An example here follows of how the registration mechanism may be implemented with High Level APIs:

```
public class Foo implements RegistrationListener {
 public Foo() throws ConfigurationException, SipException {
   Registration.getRegistration().addRegListener(this);
 }
 public void doRegistration() {
   try {
      Registration.getInstance().register();
   } catch(RegistrationException re) {
      // Update GUI/Set application to offline mode
   }
 }
 public void registrationSucceed(RegistrationEvent event) {
   System.out.println("Reregistration successful.");
 }
 public void registrationFailed(RegistrationEvent event) {
   // Update GUI/Set application to offline mode
 }
}
```

- **Instant Messaging:** by means of this service, content can be sent in the body of a SIP MESSAGE. Valid contents are both text and attachments. Automatic fragmentation and reassembly is done for those messages that exceed the MaxBodySize parameter configured in the property file. As for Registration, there is a single point of access to the IM service:

```
static InstantMsg getInstantMsg()
      throws ConfigurationException, SipException
```

  Three methods for sending IM are available, one for sending only text, another for an attachment only and the last for both text and an attachment:

```
void sendIM(String uri, String subject, String msg)
    throws SendMsgException
```

```
void sendIM(String uri, String subject, Attachment att)
  throws SendMsgException
```

```
void sendIM(String uri, String subject, String msg,
            Attachment att) throws SendMsgException
```

In order to be notified of the events linked to this service (success or failure indications), the MsgListener interface must be implemented (which provides methods sendMsgSucceed() and sendMsgFailed()). Here follows an example on how this class may be used:

```
public class Foo implements MsgListener {
   private InstantMsg im;

   public Foo() throws ConfigurationException, SipException {
         im = InstantMsg.getInstantMsg().addMsgListener(this);
   }

   public void sendMessage() throws SendMsgException {
         im.sendIM("user01@imses.local", "test","Helloworld.");
   }

   public void sendMsgSucceed(MsgEvent evt) { ... }

   public void sendMsgFailed(MsgEvent evt) { ... }
   public void messageReceived(MsgEvent evt) {
         System.out.println("Sender: " + evt.getPeer());
         System.out.println("Subject: " + evt.getSubject());
         System.out.println("Text: " + evt.getMessage());
   }
}
```

- **Chat:** the Chat APIs provide methods that permit a user to initiate, control, close a chat session and handle chat invitations by other parties. Only two-party chats are supported. As for other services, a listener interface must be implemented in order to handle incoming events. Here an example of how a chat may be implemented; the main difference with respect to other services is that a chat constructor takes a Boolean argument, if true the instantiated object acts as a server and listens to incoming requests, otherwise (false) it acts as a client.

```
public class Foo implements ChatControlListener, MsgListener {
    private Chat server;

    public Foo() throws ConfigurationException, SipException {
      server = new Chat(true);
      server.addChatControlListener(this);
      server.addMsgListener(this);
    }

    public void initiateSession() throws ConfigurationException,
                                SessionException, SipException {
      Chat chat = new Chat(false); // client chat object
      chat.addChatControlListener(this);
      chat.openSession("user01@imses.local", "Hello", );
    }

    public void processRinging(ChatEvent event) { ... }

    public void openSessionFailed(ChatEvent event) { ... }

    public void openSessionSucceed(ChatEvent event) { ... }
    public void sessionInitiated(final ChatEvent event) {
      // Show and handle invitation dialog
    }

    public void messageReceived(MsgEvent evt) {
      // Display chat message in GUI
    }
```

```
    public void sessionClosed(ChatEvent event) { ... }
}
```

- **Buddy List:** the main features that the Presence API provides are the opportunity to publish user information and to subscribe to presence information of other users. The methods that perform these tasks are provided by Presence class (implemented as a singleton object).

```
void publish(String presInfo)
void unpublish()
void subscribe(String uri)
void unsubscribe(String uri)
```

An example of the methods that must be implemented in order to manage Presence flow of messages follows.

```
public class Foo implements PresenceListener {
   private Presence pres;

   public Foo() throws ConfigurationException, SipException {
         pres = Presence.getPresence();
         pres.addPresenceListener(this);
   }

   public void publish(String xml) throws ConfigurationException,
   PresenceException, SipException {
         pres.publish(xml);
   }

   public void subscribe() {
         subscribe("<sip:user01@imses.local>");
   }

   public void publishSucceed(PresenceEvent e) { ... }

   public void publishFailed(PresenceEvent e) { ... }

   public void subscribeSucceed(PresenceEvent e) { ... }

   public void subscribeFailed(PresenceEvent e) { ... }

   public void notifyReceived(PresenceEvent e) {
         // Parse XML information
         // Update buddy status/information in GUI
   }
}
```

4.2 Plug in Extensibility

The whole environment has been developed with the intent to permit developers (MNOs) to build more blocks on the existing platform. This is certainly a great advantage in building new services, because it is not necessary to start from scratch.
The extensibility is possible on two different levels, High Level API and GUI, and may be done on both of them.
The High Level API permits the developer to register its service plug in through the ProviderDistributor class. The only instance of this class acts mainly as a mux/demux; it recognizes to which service a received message must be passed or it may be used from a service to send a request or a response. When a developer wants a new service to be considered in the distribution of incoming messages, the method registerPlugin must be invoked on the ProviderDistributor class. This action tells the class to send any incoming requests to the service object, before passing it to Jeti for processing. The service checks incoming messages and it responds with a true or false. In case the return value is true, the message is not passed to Jeti

since the message has been processed by the service. In case the return value is false, the message is not sent to any of the available services and Jeti will process it. The only disadvantage of this architecture is that for every new service one more check must be done when a message is received and this may slow down the client with many active services.
The Jeti GUI permits the developer to start new services directly from the existing graphical interface. When the application is entered, on the top right four different buttons are visible. The third from the left is the one that leads to the plug in panel. In this panel, the plug in reported in the plugins.txt file (file that resides in the same directory as the application) are listed. In order to activate the desired plug in, the name of the plug in must be checked and the start button must be clicked. This action invokes the startPlugin() method on the plug in. In order to be recognized as a plug in, the service must implement the GUIPlugin interface.

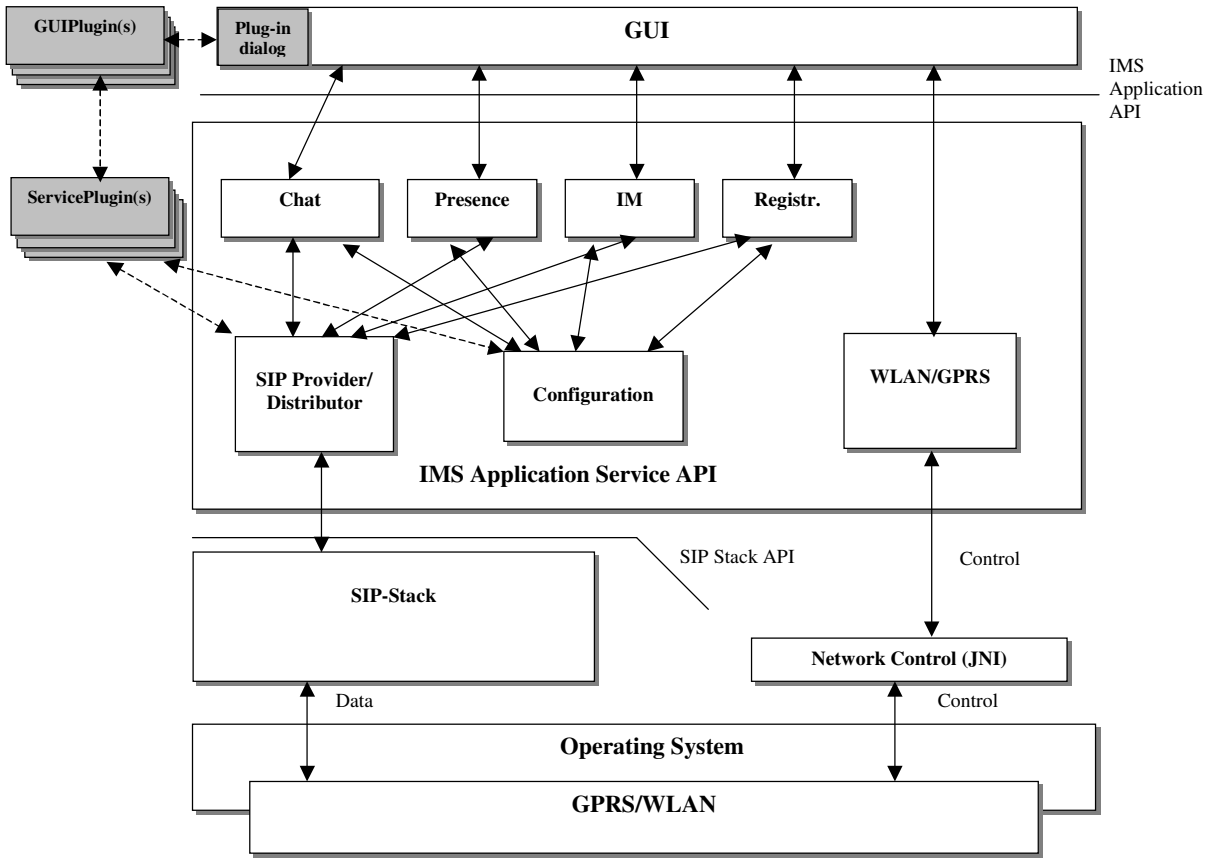

Fig. 4.3 Two level plug in extensibility

4.2.1 Plug in deployment and creation

Each GUI plug in must implement interface class de.siemens.icm.ims.client.testapp.util.GUIPlugin. Each plug in that requires access to the IMS core network must also implement interface class de.siemens.icm.ims.client.testapp.util.GUIPlugin.

In order to deploy the plug in a java archive file (.jar) of the plug in must be created. This file must be copied on the PDA, for example under the directory where the application runs. At this point a new record in the /JETI/plugins.txt file must be inserted. Each record has to start at a new line and consists of three parts divided by colons:

plug in name:main class of the plug in:jar file of the plug in.

Fig. 4.4 Plugin Use Case

4.2.2 Plugin start

JETI plugin needs to be started from the plugins panel. Pressing the start button the method stopPlugin on the class that implements the interface class GUIPlugin is invoked. The sequence diagram that describes what happens when the user starts the plug in follows.

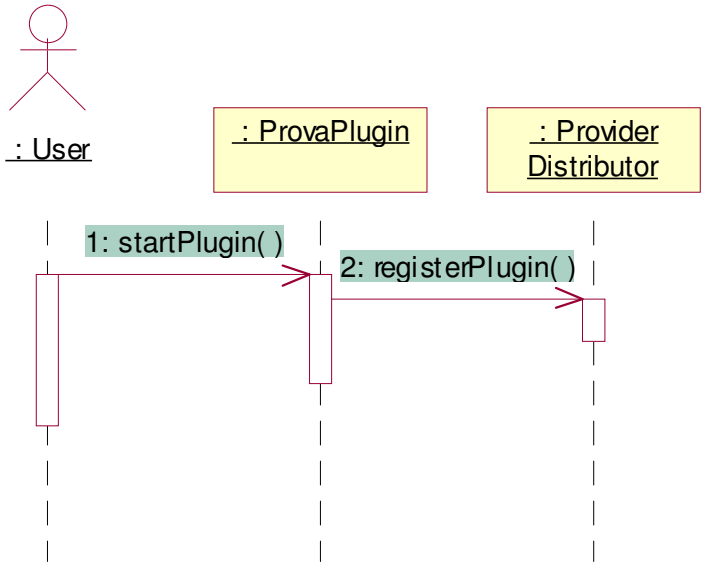

Fig. 4.5 Plugin Start Sequence Diagram

4.2.3 Plugin stop

The user can also stop the plug in. The stop button is next to the start button in the Plug in panel and invokes the method stopPlugin on the desired plug in. In the stopPlugin method a

call on the ProviderDistributor's functions must be made, so that the plug in no longer intercepts incoming messages.

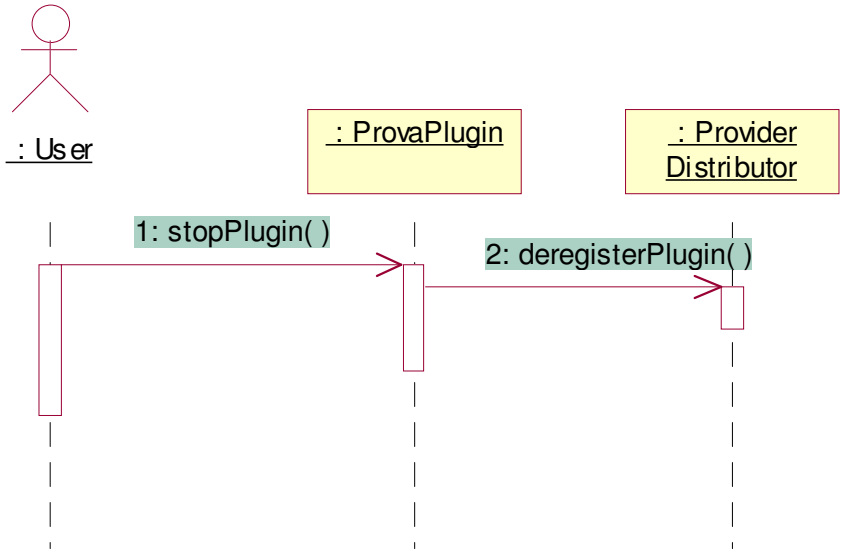

Fig. 4.6 User stops the plugin

4.2.4 Handling of incoming SIP requests

Any requests are first processed by plug-ins. The plug in must verify that the request is for it. This is accomplished through the return type of the plug-in's processRequest() method. If the message is meant for a service managed by the plug in the method returns a true value (boolean), otherwise it returns a false one.

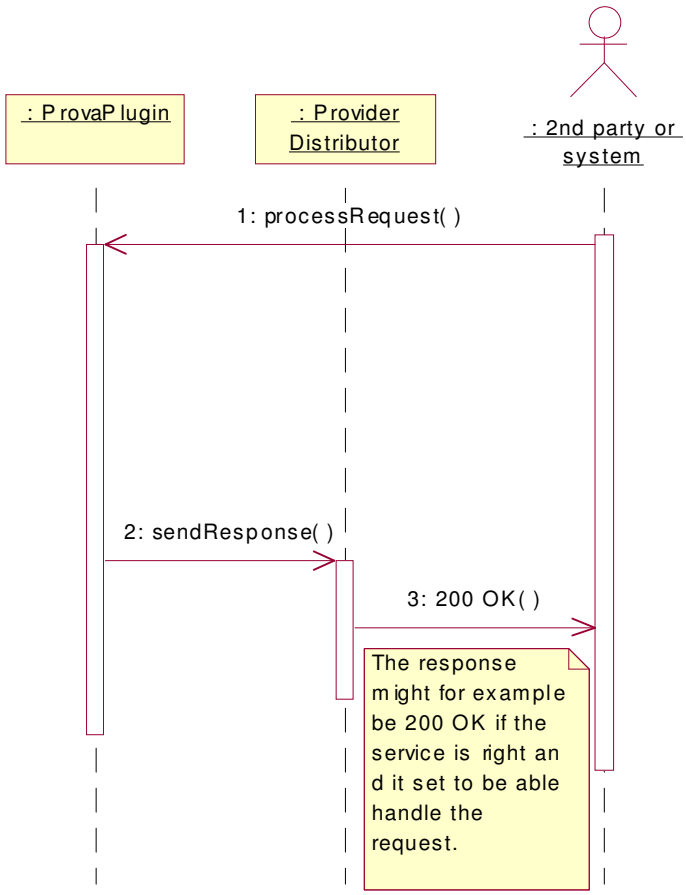

Fig. 4.7 Handling of SIP requests by a plug in

4.2.5 Implementation of a sample plug in class

As already said at the beginning, there are two interface classes that must be implemented in order to gain interoperability with the Jeti GUI and to gain access to the IMS network: GUIPlugin and ServicePlugin. Here are the signatures of the methods that are inherited from these two interfaces and that must be implemented.

```
Public class ProvaPlugin implements GUIPlugin, ServicePlugin {
      ProvaPlugin() {
      }

      public boolean processRequest(SipEvent event) {
            //inherited from the ServicePlugin interface.
            //Handles incoming requests. Must return true when a
            //request is recognized as destinated to one of the
            //services implemented in the plugin.
      }

      public void processResponse(SipEvent event) {
            //inherited from the ServicePlugin interface.
            //Handles incoming responses. No check must be done
            //since the ProviderDistributor class keeps track of
```

```
            //the objects that start a transaction and so delivers
            //it related responses.
        }

        public void processTimeout(SipEvent event) {
            //inherited from the ServicePlugin interface.
            //This method is invoked when a timeout on a request
            //is called from the SIP stack.
        }

        public void startPlugin() {
            //inherited from the GUIPlugin interface.
            //This method permits the user to start the plugin
            //from the Jeti interface. Here the initialization of
            //services and registration of the plugin in the
            //ProviderDistributor class must be done.
        }

        public void stopPlugin() {
            //inherited from the GUIPlugin interface.
            //This method permits the user to close all services
            //related  to this plugin and to deregister the plugin
            //from the ProviderDistributor class.
        }

        public void showPlugin() {
            //inherited from the GUIPlugin interface.
            //This method is intended for the rendering visible
            //the plugin GUI, when the plugin is active in the
            //background.
        }

        public String getStatus() {
            //inherited from the GUIPlugin interface.
            //This method return the status of the plugin. This
            //status is set to "running" when a call on
            //startPlugin() is made and back to "stopped" when a
            //call to stopPlugin() is made.
        }

    private String status  = new String("stopped");

}
```

5. Exploit Client Architecture

The architecture of the Exploit Client is that depicted in the following figure:

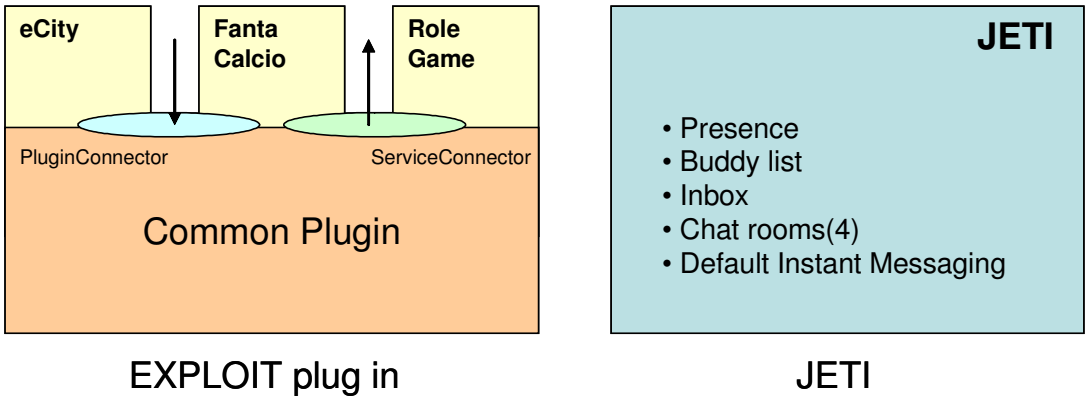

Fig. 5.1 Exploit plug in Architecture and basic services provided by Jeti

Service specific functionalities are implemented as a JETI plug in, named EXPLOIT plug in.

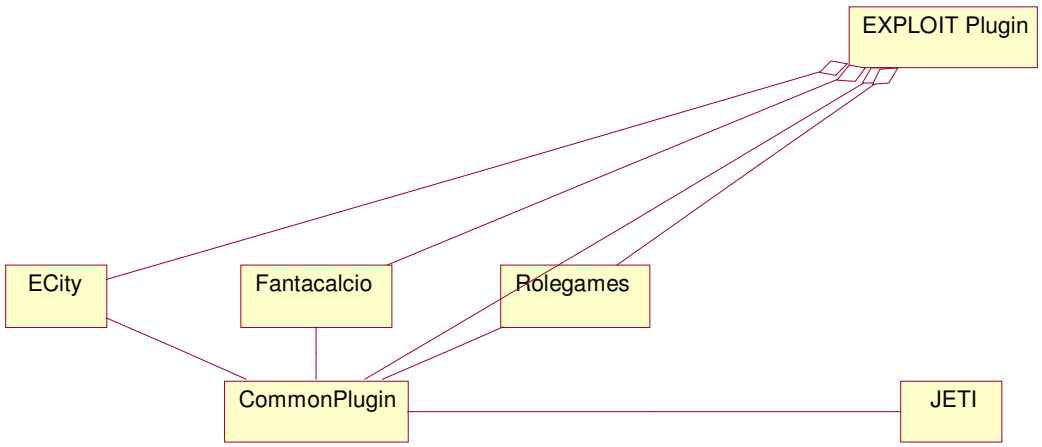

Fig. 5.2 Exploit plug in components and their interaction with Jeti

The EXPLOIT plug in is designed to support all service specific GUIs (ECity, Fantacalcio, Rolegames) and a common GUI on top of a communication infrastructure. The communication infrastructure is common to all services.
The Exploit's client GUI is composed of a top panel where services may be viewed and accessed and a bottom panel that lets the user communicate and switch between services. The Rolegames, ECity and Fantacalcio services manage the top panel while the initiation of an audio call, a chat session or the sending of an IM is possible from the bottom panel.

All communication services have been implemented in a single plug in. This design choice has been made in order to minimize development and integration efforts. Because of this, development of service GUI and logic may be separated from the development of communication functionalities. Moreover, the interface defined between the service GUIs and the CommonPlugin allows a good degree of independence between work on GUIs and on low-level functionalities.
In particular, two interfaces have been identified:

- **ServiceConnector,** which is implemented by each service in order to receive incoming requests and calls (incoming messages from the network);
- **PluginConnector**, which is implemented by CommonPlugin to manage outgoing service requests and calls (outgoing messages to the network).

EXPLOIT plug in comprises the following main modules:

- CommonPlugin (the module that embodies all communication features);
- Service logics and GUIs
    - CommonPlugin GUI (this GUI permits the user to perform conventional communication):
        - ServiceSupport.
    - Service GUIs (these GUIS permit the user to play or use a particular service, the following being those implemented for the project):
        - ECity;
        - Fantacalcio;
        - Rolegames.

Each service requires a specific module containing the local application GUI and logic.
The CommonPlugin module is intended to provide the following capabilities required by EXPLOIT services:

- Buddy List support;
- Push messages management;
- Alias support;
- Chat management;
- Enriched INVITEs (SIP INVITE containing images and ring tones);
- Audio call.

The CommonPlugin module is responsible for the dispatching of notifications about incoming messages that have an impact on service GUI and logic and for the management of actions triggered by GUI events.
Services related messages (e.g. IM for Fantacalcio and for Rolegames) are discriminated by their SIP URLs and other project specific headers (new SIP headers have been defined within the project so to simplify client-service interactions) like FromAlias, ToAlias and ServiceName.
Each service GUI is responsible for:

- Managing service look and feel (graphical appearance and service logic);
- Mapping of GUI events into CommonPlugin actions;
- Mapping of CommonPlugin notifications into GUI events.

EXPLOIT plug in is designed for allowing the management of:

- one buddy list for each service GUI;
- four chat rooms available for the all services (chat rooms are allocated on a FCFS basis).

5.1 Exploit plug-in interfaces

As introduced in the previous section, Exploit plug in can be seen as the aggregation of service specific components and a common component which takes care of "common functions", called CommonPlugin. Low level interactions between Exploit plug in and JETI are handled by CommonPlugin.

In order to permit the interaction between service GUIs and CommonPlugin two interfaces have been defined: ServiceConnector and PluginConnector.

ServiceConnector must be implemented by all service GUIs. PluginConnector is implemented by CommonPlugin and used by all service GUIs.

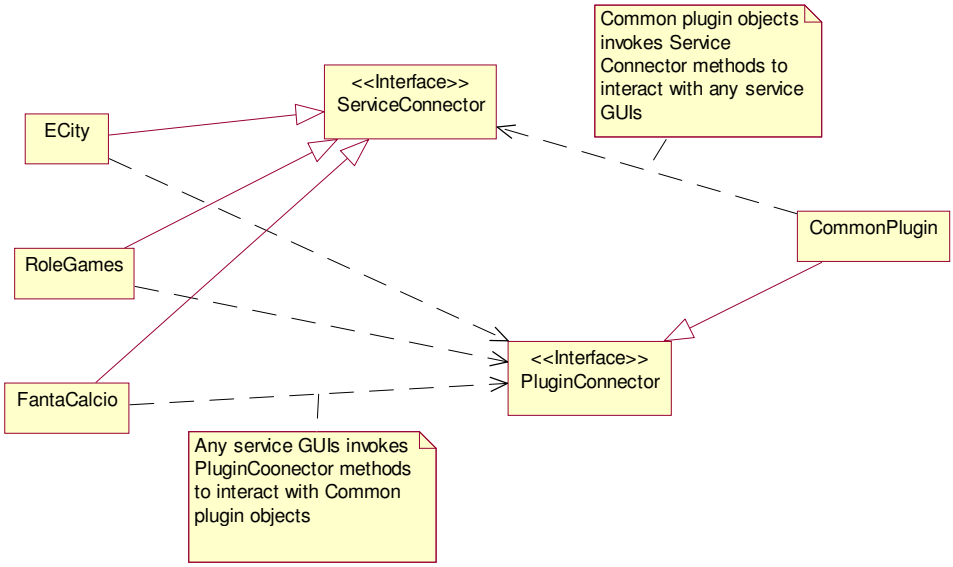

Fig. 5.3 Interaction between the PluginConnector and ServiceConnector interfaces

A more in depth description of the new interfaces defined in the Exploit plug in follows.

5.1.1 The ServiceConnector Interface

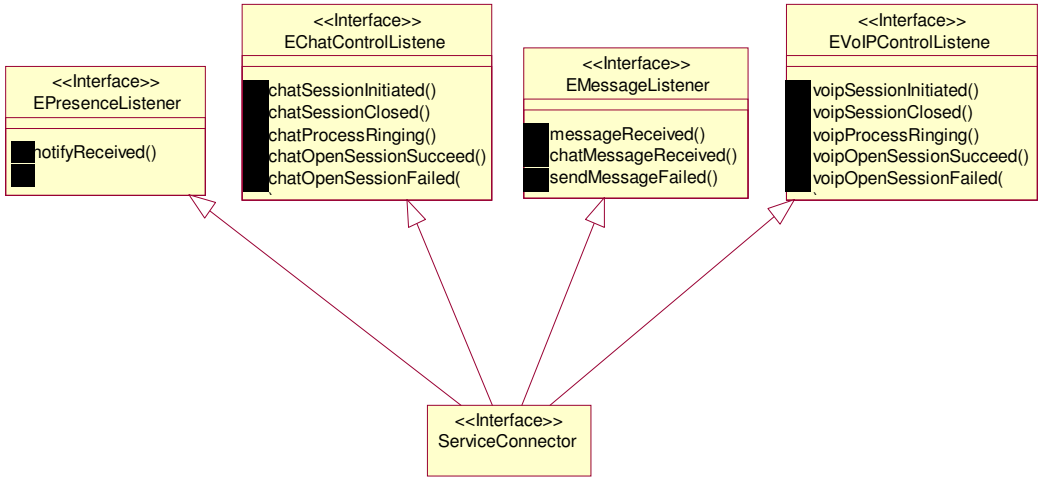

Fig. 5.4 Class Diagram of the ServiceConnector interface

ServiceConnector provides each service with all the methods that have to be implemented in order to reflect incoming notification into GUI events.
This means that the control of application behavior and GUI must be associated with the invocation of all the above-identified methods.
ServiceConnector inherits the methods provided by four abstract classes:

- **EPresenceListener**: deals with presence
  - ***notifyReceived***: notifies that new presence information is available for one of the buddies listed in the service related buddy list.
- **EChatControlListener**: deals with chat session management and setup
  - ***chatSessionInitiated***: notifies that an invitation to join a session has arrived (the attached image and ring tone are returned);
  - ***chatSessionClosed***: notifies that the session has been closed by a remote party;
  - ***chatProcessRinging***: notifies that the called party terminal is ringing;
  - ***chatOpenSessionSucceed***: notifies that called party accepted the call;
  - ***chatOpenSessionFailed***: notifies that called party refused the call.
- **EMessageListener**: deals with incoming messages:
  - ***messageReceived***: notifies that a message has been received;
  - ***chatMessageReceived***: notifies that a message within a chat session has been received;
  - ***sendMessageFailed***: notifies that some problems occurred, this method is used both inside and outside a chat session;
- **EVoIPControlListener**: deals with audio call management and session setup
  - ***voipSessionInitiated***: notifies that an invitation to join a session has arrived (the attached image and ring tone are returned);
  - ***voipSessionClosed***: notifies that the session has been closed by the remote party;
  - ***voipProcessRinging***: notifies that the called party terminal is ringing;
  - ***voipOpenSessionSucceed***: notifies that called party accepted the call;
  - ***voipOpenSessionFailed***: notifies that called party refused the call.

Here are the detailed signature of the above methods:

- **EPresenceListener**
  - `void notifyReceived(String sipURL, String alias, String status) throws UnknownBuddy`. This method is invoked when a SIP NOTIFY sent from the Presence Server arrives and the CommonPlugin identifies it as related to a buddy set in one of the service's buddy list.

- **EChatControlListener:**
  - `void chatSessionInitiated (String fromAlias, String toAlias, String from, String subject, byte[] image, byte[] sound, String text, String callId)`. This method is invoked on a service when a SIP INVITE identified as session setup for a chat communication arrives and when the ServiceName header value is identified as that of an available service.
  - `void chatSessionClosed (String callId) throws UnknownCallId`. This method is invoked when a SIP BYE is sent by the 2nd party who leaves the chat room. Since the chat exchanges are identified on a Call-ID basis, the exception is

thrown if the Call-ID passed as parameter is not recognized by the service logic. The rule remains the same in all the other methods that may throw this exception.

- ¬ `void chatProcessRinging (String callId) throws UnknownCallId.` When provisional responses of type 180 Ringing arrive from the called client, this method is called on the service implementation.
- ¬ `void chatOpenSessionSucceed (String callId) throws UnknownCallId.` When the called party accepts the invitation to enter the chat room a 200 Ok response is sent. This is translated at the service level by the call to this method.
- ¬ `void chatOpenSessionFailed (String callId) throws UnknownCallId.` In case the response is of type 3xx, 4xx, 5xx, 6xx (any kind of failure), this method is invoked.

- **EMessageListener**
  - ¬ `void messageReceived (String fromAlias, String toAlias, String from, String subject, byte[] image, byte[] sound, String text) throws UnknownMessage.`
  - ¬ `void chatMessageReceived (String fromAlias, String toAlias, String from, String subject, byte[] image, byte[] sound, String text, String callId) throws UnknownCallId.` This method is called on the service's implementation when the incoming SIP MESSAGE is recognized as belonging to a chat session.
  - ¬ `void sendMessageFailed (String to, String subject, String errorMessage).` When a failure response is returned to a sent SIP MESSAGE, the invokation of this method alerts the service implementation to this.

- **EVoIPControlListener:** the methods that manage the incoming SIP requests or responses related to the voip service have the same logical meaning as those already described for the chat service.
  - ¬ `void voipSessionInitiated (String fromAlias, String toAlias, String from, String subject, byte[] image, byte[] sound, String text, String callId).`
  - ¬ `void voipSessionClosed (String callId) throws UnknownCallId.`
  - ¬ `void voipProcessRinging (String callId) throws UnknownCallId.`
  - ¬ `void voipOpenSessionSucceed (String callId) throws UnknownCallId.`
  - ¬ `void voipOpenSessionFailed (String callId) throws UnknownCallId.`

5.1.2 The PluginConnector Interface

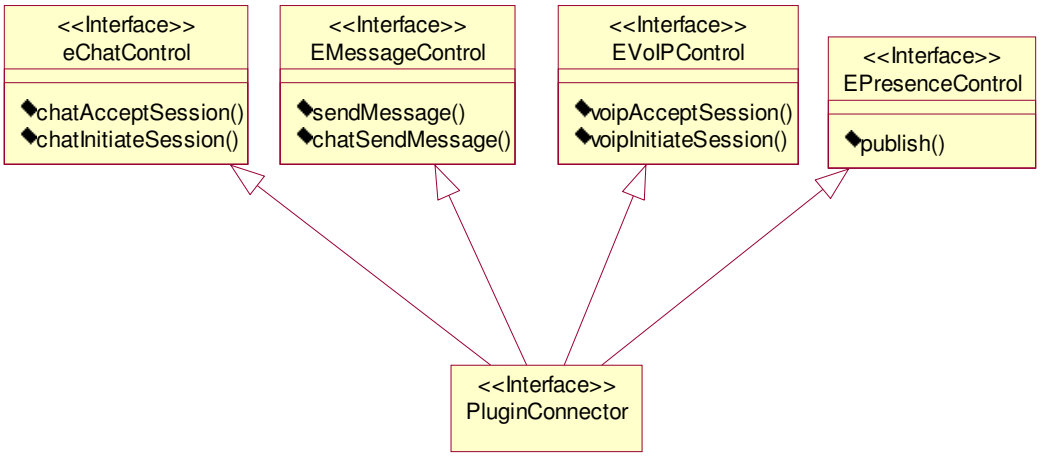

Fig. 5.5 Class Diagram of the PluginConnector interface

The PluginConnector interface provides services with methods that have to be invoked in order to trigger CommonPlugin actions.
CommonPlugin inherits the methods provided by four specific abstract classes:

- **EChatControl**: deals with chat session management:
  - ***chatAcceptSession***: allows acceptance or rejection of a session invitation;
  - ***chatInitiateSession***: allows the sending of a session invitation (an image and ringtone can be specified).
- **EMessageControl**: deals with outgoing messages:
  - ***sendMessage***: allows the sending of a message;
  - ***chatSendMessage***: allows the sending of a message within a chat session.
- **EVoIPControl**: deals with audio session management:
  - ***voipAcceptSession***: allows the acceptance or rejection of a session invitation;
  - ***voipInitiateSession***: allows the sending of a session invitation (an image and ringtone can be specified).
- **EPresenceControl**: deals with the publishing of service specific presence information
  - ***publish***: allows the provisioning of game specific availability information.

These detailed signatures of the above methods:

- **EChatControl**
  - `void chatAcceptSession(String callId, String soundPath, String imagePath, boolean accept) throws UnknownCallId`. This method is invoked when the user agrees to enter a chat session to which he has been invited. At a SIP stack level this information is translated into the sending of a 200 Ok response. The parameters soundPath and imagePath may be set to attach information as an image or a sound clip in the response (i.e. when a Rolegames player is invited to battle, he is able to send the image of his character and his sound clip when he accepts in the response).
  - `void chatAcceptSession(String callId, byte[] sound, byte[] image, boolean accept) throws UnknownCallId.`

- `String chatInitiateSession(String fromAlias, String toAlias, String serviceName, String to, String subject, String soundPath, String imagePath, String text, ServiceConnector serviceImplementation) throws ChatRoomsBusy`: the last parameter informs the PluginConnector implementation about which service (i.e. Fantacalcio, Rolegames) is sending the request. The Call-ID return value must be stored in order to be able to recognize further messages related to the initialized session and to send messages within the session. The other parameters are related to the amount of information that the user wants to convey in the SIP INVITE (i.e. Rolegames uses chat communication to implement battles, the sent image represents the battle Icon of the player and the sent sound his battle yell).
- `String chatInitiateSession(String fromAlias, String toAlias, String serviceName, String to, String subject, byte[] sound, byte[] image, String text, ServiceConnector serviceImplementation) throws ChatRoomsBusy.`

- **EMessageControl:**
  - `void sendMessage (String fromAlias, String toAlias, String serviceName, String to, String subject, String imagePath, String soundPath, String text)`. This method permits the service to send standalone SIP MESSAGEs.
  - `void sendMessage (String fromAlias, String toAlias, String serviceName, String to, String subject, byte[] image, byte[] sound, String text).`
  - `void chatSendMessage (String fromAlias, String toAlias, String serviceName, String to, String subjectString imagePath, String soundPath, String text, String callId) throws UnknownCallId`. It is used within a chat session. It permits a service to send IM within a chat session specifying the Call-ID of the message.
  - `void chatSendMessage (String fromAlias, String toAlias, String serviceName, String to, String subject, byte[] image, byte[] sound, String text, String callId) throws UnknownCallId`: to be used within a chat session.

- **EVoIPControl**:
  - `void voipAcceptSession (String callId, boolean accept) throws UnknownCallId.`
  - `String voipInitiateSession (String fromAlias, String toAlias, String serviceName, String to, String subject, String soundPath, String imagePath, ServiceConnector serviceImplementation) throws SessionBusy`: the String return value is the Call-ID that must be stored in order to be able to recognize further messages related to the initialized session. The SessionBusy exception is thrown when the user attempts to make a new call while still busy on an old one.

- **EPresenceControl**:
  - `void publish (String[] openGames, String alias) throws SyntaxError`: the method permits specifying the game availability (media type) and the alias (note) that user wants to register/publish. The information passed by the publish parameters is stored in the PluginConnector implementation. When a publish message is sent, the Presence Server is triggered to send a SIP NOTIFY to all watchers. The information sent with the SIP NOTIFY is the presence information received in the SIP PUBLISH. The Presence Server fills NOTIFYs with only the last user information it received. If presence information was not

stored in the PluginConnector object and was not all sent in every publish, a service would activate itself sending only its presence information. All the information sent by already active services would be forgotten. This would result in other users' clients understanding that only this last service is activated while the others have been shut down.

5.1.3 Alias management and service discrimination:

The ServiceConnector and PluginConnector methods provide support for alias management by means of two specific parameters:

- String **fromAlias**: it allows specifying the originator's alias;
- String **toAlias**: it allows specifying the alias of the message recipient.

The values set in these parameters are those set in the FromAlias and the ToAlias headers of a SIP message. The use of these headers has been conceived for those within a service who may want to use an alias. Some services are built on aliases. An example is the Fantacalcio game, where the alias represents the name of the user's team. The same may be said for the Rolegames service; the alias represents the name of the character invented by the user. These headers are meant to support services.

The ServiceConnector and PluginConnector methods allow discriminating between supported services by means of the following parameter:

- String **serviceName**: it allows specifying the name of the service (e.g. "fantacalcio") that originated a service specific message (e.g. SIP MESSAGE, SIP INVITE).

The following EXPLOIT specific SIP headers convey the fromAlias, toAlias and serviceName information:

```
…
FromAlias: fromAlias
ToAlias: toAlias
ServiceName: serviceName
…
```

The "ServiceName" SIP header will be used by services running on the Application Engine to discriminate which service is in charge to process the SIP message. Given that SIP messages related to peer2peer communication do not need to be processed, ServiceName should be coded with "peer2peer".

Here follows an example of the messages that may be exchanged between two users in the play of a game. The messages are exchanged within the Fantacalcio service, so the ServiceName header value is set to "fantacalcio". In the Fantacalcio game, the user's alias is the name of his team. In this case, the coding rule says that the values of the FromAlias and the ToAlias headers must be set specifying the game's name (i.e. fantacalcio), the kind of league the player plays in (i.e. private), the name of the league (i.e. H3G) and the team's name (i.e. Roma). The SIP MESSAGE's subject explains that the players are trading players for their teams.

```
SIP MESSAGE sip:user02@tre.it
From: <sip:user01@tre.it>
To: <sip:user02@tre.it>
Call-ID: 483754325@20.20.20.20
CSeq: 456 MESSAGE
Subject: Football player trading
FromAlias: fantacalcio.private.H3G.Roma
ToAlias: fantacalcio.private.H3G.Lazio
ServiceName: fantacalcio
…
```

The following could be a standalone SIP MESSAGE or it may be sent within a chat session. The value set in the FromAlias header is the user's nickname. The user in the setup panel of the Jeti application may set the nickname. The ToAlias header value is set to CHAT_USER, because the alias of the remote user is not known.

```
SIP MESSAGE sip:user02@tre.it
From: <sip:user01@tre.it>
To: <sip:user02@tre.it>
Call-ID: 483754325@20.20.20.20
CSeq: 436 MESSAGE
Subject: How are you?
FromAlias: Gustavo
ToAlias: CHAT_USER
ServiceName: peer2peer
…
```

5.2 Exploit plug-in main classes

The main module that takes care of communication has been called CommonPlugin, in the previous chapters. The CommonPlugin module contains several classes that will be here described. These classes may be grouped on the basis of their functionalities.

In order to gain interoperability with Jeti, two classes have been defined, CommonPlugin and CommonFunctions. CommonPlugin implements the GUIPlugin interface. It is the entry point to the Exploit Client from the Jeti GUI because it is invoked when starting the plug in. CommonFunctions implements the ServicePlugin and the PluginConnector interface. It manages incoming and outgoing messages. It is the heart of the application.

The classes that manage communication services are ChatService, InstantMsgService, PresenceService and Voip (the Voip service is implemented by several classes, as will be seen further on). These classes may be invoked by the CommonFunctions instance and may invoke a ServiceConnector implementation and access the SIP stack. They implement the logic of communication, compliant with the SIP protocol.

More classes have been implemented. They mainly support those that have already been mentioned.

5.2.1 The CommonPlugin class

In order to be instantiated from the Jeti GUI, the CommonPlugin class must implement the GUIPlugin interface. The methods inherited from this interface are here described in more detail:

- **startPlugin()**: The start button on the Jeti GUI invokes the startPlugin method on the plug in. In the body of this method, an instance of the CommonFunctions class is obtained and registered in the ProviderDistributor class. After registering, the start() method is invoked on the CommonFunctions class, which initializes an internal state variable and the service's GUIs.

```
public void startPlugin() {
      try {
            cm = CommonFunctions.getInstance();
      } catch (Exception e) {
      }
      try {
            pd = ProviderDistributor.getProviderDistributor();
            pd.registerPlugin(cm);
      }
      catch (Exception e) {
      }
      status = "started";
      cm.start();
}
```

- **showPlugin()**: The start button initializes the client to listen for incoming requests, but no GUI appears. In order to make the GUI appear, the show button must be clicked. With a click on the show button the showPlugin() method is invoked on the plug-in. The only task of this operation is to invoke show() on CommonFunctions that will let the Exploit GUI appear.

```
public void showPlugin() {
      if (status.equals("stopped")) {
            mw.showMessage("The plugin must be started");
            return;
      }
      else {
            cm.show();
      }
}
```

- **hidePlugin()**: This method is implemented, but is not necessary. When the Exploit Client is active, the hide button is not visible because of the PDA's limited screen. In order to hide the GUI the top left cross button on the window may be used, leaving the client listening for incoming requests.
- **stopPlugin()**: The stop button stops the client listening to incoming requests. This is achieved by deregistering the CommonFunctions instance in the ProviderDistributor class.

```
public void stopPlugin() {
      try {
            pd.deregisterPlugin(cm);
      }
      catch (Exception e) {
            System.out.println(e);
      }
      status = "stopped";
      cm.stop();
      cm = null;
}
```

- **getStatus()**: This method returns the value stored in the "status" global variable. The values it takes may be "started" or "stopped".

5.2.2 The CommonFunctions class

CommonFunctions is the main class of the Exploit Client. This class has the role of managing incoming requests from the SIP stack, requests coming from GUI events and the interactions with the classes that implement IM, Presence, Chat and Voip. This class is registered in the ProviderDistributor class, in order to manage incoming requests. It implements the PluginConnector interface in order to be invoked by GUI events. This is the only class that keeps instances of the ChatService, PresenceService, InstantMsgService and Voip in order to deliver them incoming messages. The singleton pattern has been chosen for this class. The factors that led to this design are the need of a single point of access to incoming messages and services and the need to limit class instances on a poorly performing environment such as a PDA. When the constructor is called (only once: when the object is already built the same instance is always returned), the "servicesconf.txt" file is read. This file tells the object which services are active and which related classes may be instantiated.

The methods of this class that manage incoming messages are the processRequest() and the processResponse() class, both are inherited from the ServicePlugin interface.

The processRequest() method is called every time a new SIP request is received by the stack. The ProviderDistributor class first delivers the request to the CommonFunctions instance. This action is performed by invoking the processRequest() method on CommonFuntions and passing it a parameter of type SipEvent. The SipEvent class is a container for all the information regarding a SIP event. SIP events, for example, are the receipt of a request, a response or a timeout. By performing checks on the attributes of the SipEvent object the CommonFunctions class can understand if the message is directed to the Exploit client or not. In the former case, CommonFunctions invokes one of ChatService, PresenceService, InstanMsgService or Voip returning true to the ProviderDistributor instance, in the latter it simply returns false. The first check that is done in the processRequest() method is on the type of request. The requests that are handled are SIP MESSAGEs, SIP INVITEs, SIP CANCELs, SIP BYEs, SIP ACKs and SIP NOTIFYs.

The CommonFunctions class does not simply hand messages to other classes, it is able to understand which objects must handle the messages it receives. When a chat session is setup the callId is bound to the chat object that handles the new chat room. This is done in order to distinguish a SIP MESSAGE related to the chat. In case the request is a SIP MESSAGE, here follows the pseudo code of the CommonFunctions' processRequest() method,

```
IF the CallID is not recognized as that of an existing session,
      THEN
            IF the message is recognized as a push message (the From header
            user name value is set to a service's name, i.e. From:
            <sip:ecity@h3g.it>)
            ELSE IF the message was sent from a peer service(the
            ServiceName header must be present)
                  IF the ServiceName is of the type servicename, i.e.
                  ServiceName: fantacalcio, the InstantMsgService's
                  messageReceived() method is invoked.
                  ELSE the ServiceName header value is blank, the
                  InstantMsgService's messageReceived() method is invoked
                  and the message will be managed by the ServiceSupport
                  implementation of the ServiceConnector interface.
```

ELSE the CallId is recognized as that of an existing session. The ChatService's messageReceived() method is invoked.

CommonFunctions keeps track of the available chats; in case all chats are busy, a SIP 486 Busy Here response is sent. In case an available chat is available, the callId value of the SIP INVITE header will be bound to the ChatService instance that manages the new chat room. As well as for chats, CommonFunctions stores information on the state of a Voip session. If no call is setup, the callId value of the SIP INVITE header is stored in the CommonFunctions instance. If a call is setup, only a new SIP INVITE with the same callId value will be processed (it could be the case that the second party is willing to change dynamically the media exchanged), otherwise a failure response will be sent. Voip invitations must have a ServiceName header value set to "Voip", otherwise the message will not be understood. There follows a pseudo code example of how SIP INVITEs are processed:

IF the CallID in known as that of an existing VoIP session **OR (**the message is recognized as an invitation to a new VoIP session (the ServiceName header must be set to voip) **AND** the voip session is not already busy**),**
THEN the Voip's inviteRequestReceived method is invoked and the ServiceSupport implementation of the ServiceConnector interface will manage the request.
IF the CallID is known as that of an existing Chat session **OR (**the message is recognized as an invitation to a new session (the ServiceName header is set to one of the available services, i.e. ServiceName: fantacalcio) **AND** a free chat room is available**),**
THEN the ChatService's inviteRequestReceived method is invoked and the service's implementation of the ServiceConnector interface will manage the request.

The pseudo code that describes the behavior of the processRequest() method in the case of the receipt of a SIP ACK, a SIP CANCEL or a SIP BYE request is similar:

IF the CallID of the incoming request is that of an existing chat,
THEN the request is forwarded to the ChatService's object invoking the ackRequestReceived, the cancelRequestReceived() or the byeRequestReceived() method. These methods are then mapped on the ServiceConnector's implementation.

IF the CallID of the incoming request is that of an existing Voip session,
THEN the request is forwarded to the Voip's object invoking the ackRequestReceived, the cancelRequestReceived() or the byeRequestReceived() method. These methods are then mapped on the ServiceSupport class which implements the ServiceConnector interface.

When a SIP NOTIFY is received, the state table is checked and the message is forwarded to the services the SIP URI is bound to:

IF the dblTable (the table that binds SIP URIs to services; it is built when a DBL is received) contains the presentity's SIP URI,
THEN extract the services bound to the presentity's SIP URI from the dblTable and cycle invoking the notifyReceived() method on the PresenceService instance, once for each service.

The processResponse() method manages responses to sent requests. The ProviderDistributor class keeps track of the transaction ids of the objects that have sent a

request; in this way, responses are directly forwarded to the object that sent the request. CommonFunctions has a similar mechanism necessary to know which of ChatService, InstantMsgService, PresenceService and Voip sent the request to which the response is meant. Every time an instance of ChatService, InstantMsgService, PresenceService or Voip sends a new request, it must register the transaction id of the new request in the CommonFunctions instance. The registration binds the transaction id to the instance that sends the request. When a response to the request arrives, the CommonFunctions class will know to whom the response is sent and will be able to cancel the associated entry in the transaction id table.

The CommonFunctions class must implement one more method inherited from the ServicePlugin interface, the processTimeOut() method. This method is not bound to any incoming message. The SIP event that leads the stack to invoke this method is the timeout of a request. When a request is sent, the SIP stack expects a response within a given time. If the response doesn't arrive in the expected time, the processTimeOut() method is invoked.

The CommonFunctions class must implement all the methods defined in the PluginConnector interface. The Exploit services see it as the connection point to the SIP stack. When one of the methods inherited from the PluginConnector interface is invoked, the CommonFunctions instance sends the request to the class that effectively manages that type of communication. Most of the methods have a double signature in their implementation. One signature permits the invoking class to pass an image and a sound clip in an array of bytes format. The other allows setting the path to the file on the filesystem that stores the image or the sound clip. This choice has been dictated by the need, on one hand, to be able to receive media files from the system and send them back to the system or to other players (i.e. in the Rolegames service card decks are downloaded from the system and may be sent when a game is played). On the other hand, the user may want to send media files stored on the filesystem, so this justifies the necessity for the second signature.

5.2.3 The Communication classes

While the CommonFunctions class has a managing role in the module, communication classes are specialized on single tasks. The ChatService class performs signaling within a chat session. The PresenceService class lets services send and receive presence information. The InstanMsgService class takes care of sending IM and of receiving them out of chat sessions. The Voip class performs signaling within a voip session and manages the RTP session start and closure.

The ChatService, PresenceService and Voip classes extend the TimeTask abstract class. This class belongs to the de.siemens.icm.ims.client.protocol.ip.sip.util class (part of the jeti High Level API). This abstract class, together with the Timer class belonging to the same package, permits the implementation of the timeout mechanism. This mechanism is used, for example, when a 200 Ok is sent in response to a SIP INVITE. The timeout is set waiting for the ACK request. In case the acknowledgement does not arrive, the session will be closed and the resources freed (this behavior is achieved invoking the timeout() method inherited from the TimerTask class).

- **ChatService**: The ChatService class manages the signaling related to the setup of a chat session. This is the only communication class (the others are PresenceService, InstantMsgService and Voip) that may have multiple objects instantiated. The reason why this class does not follow the singleton design pattern is that up to four chats may be open at

a time. Each chat is managed by one ChatService object. The CommonFunctions class keeps track of the number of already open sessions and of those that may still be set up.
The methods of the PluginConnector interface related to a chat communication map in methods of the ChatService class. The chatInitiateSession() method invokes the openSession() method on the ChatService class. The same happens with the chatAcceptSession() and the chatCloseSession() methods, that map on the acceptSession() and on the closeSession() methods. The instance of the ProviderDistributor class is called from this class because signaling messages are sent from here over to the SIP stack.
The only method of the PluginConnector interface that does not map onto the ChatService class is the chatSendMessage method. The InstantMsgService class, as we shall see, manages this method.

Incoming requests and responses must be sent to the related ServiceConnector implementation. The ServiceConnector interface extends the EChatControlListener interface. Each chat object has its EChatControlListener. The value of the listener may be set in two cases, when chat setup SIP INVITE is sent or when it is received. The CommonFunctions class has the role of assigning the appropriate listener to the chat object. In case a request is received, this is set checking the ServiceName header value and invoking the addChatServiceControlListener() method on the object. When a request is sent, a reference to the service that is sending the request is passed, so the mechanism is the same as the last described.
The management of incoming SIP MESSAGEs related to a chat session is asymmetric with respect to the management of outgoing ones. When the session is setup successfully, an EMessageListener listener is bound to the chat object. If a SIP MESSAGE recognized as part of a chat exchange is received, the messageReceived() method is invoked on the chat object. The chat object then processes the message and forwards it to its EMessageListener. The EMessageListener and the EChatControlListener are typically the same.

- **InstantMsgService**: The role of this class is that of managing SIP MESSAGEs. The PluginConnector's interface methods chatSendMessage() and sendMessage() are mapped on the InstantMsgService's method sendIM(). This class has been designed following the singleton design pattern, just as for the CommonFunctions class, only one instance may be referenced.

  Unlike a ChatService object, that must be able to forward all messages to the same EMessageListener, consecutive messages may be meant for different services. This is why this class does not offer the possibility of assigning it a message listener. The appropriate listener is assigned on a per message basis. The CommonFunctions instance recognizing that a SIP MESSAGE is not tied to a chat, checks its ServiceName header value and passes it in the messageReceived() method of the InstantMsgService instance. SIP MESSAGEs that are typically processed by this class are push messages. In the Rolegames service, for example, the players' cards are received by this means.

- **PresenceService**: This class manages the receipt of SIP NOTIFYs. At the same time, it is responsible for the publishing of the user's presence information and for the subscription to the buddies' user information.

  A particular service may require or not that a subscription to the buddies' presence information is performed. This action may be triggered or not by the set to true or false of an attribute in the XML document. If the subscription is required, a SIP SUBSCRIBE is sent for

every buddy stored in the buddy list. On the receipt of a SIP SUBSCRIBE, the Presence Server sends SIP NOTIFYs to the watcher. The CommonFunctions instance maps SIP NOTIFYs on the notifyReceived() method of the PresenceService instance. The notifyReceived() method is invoked as many times as the number of services the presentity is registered to. If a buddy is present in the Rolegames' and ECity's buddy lists, the notifyReceived() method will be called twice, once for each service. The notifyReceived() takes a parameter of type ServiceConnector. The notifyReceived() method is then called on the ServiceConnector reference passed as parameter.

The PluginConnector's method publish() is mapped on the PresenceService's publish() method. The XML information is set in the CommonFunctions' method, as already explained in one of the previous chapters. The PresenceService publish() method sends the information in a SIP REGISTER request to the Presence Server. The Presence Server does not understand the SIP PUBLISH method but understands SIP REGISTERs. This behavior is not compliant with the latest RFCs.

5.3 Exploit Plug in Message coding

How messages are coded is explained in this chapter. The messages are compliant with the SIP standard, but in order to support services some new coding rules have been introduced. The capabilities that are supported by these rules are:

- User alias;
- Buddy lists;
- Multiple active applications on the client;
- Messages sent within a session.

The service names are coded as follows (user part of SIP URI: user@domain):

- fantacalcio;
- rolegames;
- ecity.buddyfinder;
- ecity.carsharing.

Buddyfinder and Carsharing need the "ecity" preposition, since they are part of the same suite of services.

5.3.1 SIP INVITE

Here follow two examples. The first shows a SIP INVITE sent in order to setup a call, and the second a SIP INVITE sent to setup a chat. Both of these messages carry a text message, an image and a sound clip. In the set up of a session, the image and the sound clip are meant as an improvement to the mere ringing of a phone. The user that receives an invitation will see the image on the display and hear the sound clip, instead of a regular phone ringing.

The only rule that has been applied to the Voip call SIP INVITE is the set of the ServiceName to voip. Other headers follow the SIP standard.

- Voip session invitation example

From user01 (pippo) to user02 (pluto):

```
SIP INVITE sip:user02@h3g.it SIP/2.0
From: <sip:user01@h3g.it>
```

```
To: <sip:user02@h3g.it>
Subject: whatever
Call-ID: 847yfbwqeu24@20.20.20.20
CSeq: 4 INVITE
FromAlias: pippo
ToAlias: VOIP_USER
ServiceName: voip
ContentType: multipart/mixed;boundary="jksjhaadurgf245rt"
ContentLength: xx

--jksjhaadurgf245rt
Content-type:text/plain
……
--jksjhaadurgf245rt
Content-type:image/jpeg;name="image.jpg"
……
--jksjhaadurgf245rt
Content-type:audio/au;name="yell.au"
……
--jksjhaadurgf245rt
Content-type:application/sdp

v=0
o=user01@h3g.it 4384957385 4384568930 IN 20.20.20.20
s=SIP
c=IN IP4
t=0 0
m=audio 5070 RTP/AVP 3
a=rtpmap:3 GSM/8000
```

This SIP INVITE has been sent to invite a challenge to rolegames player. For this reason the ServiceName's header value must be set to "rolegames". The subject may be significant to identify the kind of game that will be played. In this case, user01 wants to fight user02.
This message carries a SDP body. SDP bodies are necessary to setup chat sessions as well as to setup voip sessions. The most important information carried by the SDP body is its "m" (media) line:

```
m=message 5060 sip:user09@h3g.it?Call-ID=847yfbwqeu24@20.20.20.20
```

The first parameter tells by which means media will be sent. In this case, the "message" value means that the exchange of media in the chat session will be done by SIP MESSAGEs. The second parameter sets the port on which SIP MESSAGEs will be listened to by the sender of the SIP INVITE. The third parameter contains the Call-ID that must be set in the SIP MESSAGEs. The client that sends the invitation knows the Call-ID that must be expected for the SIP MESSAGEs belonging to the chat.

- Chat service related invitation example

From user01(Dragone) to user02(Elfo) of rolegames service:

```
SIP INVITE sip:user02@h3g.it SIP/2.0
From: <sip:user01@h3g.it>
To: <sip:user02@h3g.it>
Call-ID: 847yfbwqeu24@20.20.20.20
CSeq: 8 INVITE
Subject: Battle
FromAlias: rolegames.Dragone
ToAlias: rolegames.Elfo
ServiceName: rolegames
ContentType: multipart/mixed;boundary="jksjhaadurgf245rt"
```

```
ContentLength: xx

--jksjhaadurgf245rt
Content-type:text/plain
……
--jksjhaadurgf245rt
Content-type:image/jpeg;name="image.jpg"
……
--jksjhaadurgf245rt
Content-type:audio/au;name="yell.au"
……
--jksjhaadurgf245rt
Content-type:application/sdp

v=0
o=user01@h3g.it 0 0 IN 20.20.20.20
s=SIP
c=IN IP4
t=0 0
m=message 5060 sip:user09@h3g.it?Call-ID=847yfbwqeu24@20.20.20.20
```

A simple chat between two users will have the ServiceName value "peer2peer". The CommonFunctions knows from this header that the invitation is not meant within a service (Rolegames, Fantacalcio, ECity). The ServiceSupport instance will manage this invitation.

- Peer2peer chat invitation example

From user01(pippo) to user02(pluto):

```
SIP INVITE sip:user02@h3g.it SIP/2.0
From: <sip:user01@h3g.it>
To: <sip:user02@h3g.it>
Call-ID: 847yfbwqeu24@20.20.20.20
CSeq: 12 INVITE
Subject: hello
FromAlias: pippo
ToAlias: pluto
ServiceName:peer2peer
ContentType: multipart/mixed;boundary="jksjhaadurgf245rt"
ContentLength: xx

--jksjhaadurgf245rt
Content-type:text/plain
……
--jksjhaadurgf245rt
Content-type:image/jpeg;name="mypicture.jpg"
……
--jksjhaadurgf245rt
Content-type:audio/au;name="greetingsmessage.au"
……
--jksjhaadurgf245rt
Content-type:application/sdp

v=0
o=user01@h3g.it 0 0 IN 20.20.20.20
s=SIP
c=IN IP4
t=0 0
m=message 5060 sip:user09@h3g.it?Call-ID=847yfbwqeu24@20.20.20.20
```

5.3.2 SIP MESSAGE

The SIP MESSAGE has been used in several ways in the project. A SIP MESSAGE may be sent as a standalone message from a user to another. It may be sent within a chat session. It is used to convey information from the system to the terminal and the other way round. The different meanings that a SIP MESSAGE may assume led to differente coding conventions.

Here follows an example of a SIP MESSAGE, sent from the Fantacalcio service to a user. This kind of message is a push message that stores general information about the service. The information may be of any kind; in this case, it tells the user when the championship will begin.

- Service server message example

From fantacalcio service to a user02(fantacalcio.private.H3G.Milan):

```
SIP MESSAGE sip:user02@h3g.it SIP/2.0
From: <sip:fantacalcio@h3g.it>
To: <sip:user02@h3g.it>
Call-ID: 48375432S@20.20.20.20
CSeq: 456 MESSAGE
Subject: News
ToAlias: fantacalcio.private.H3G.Milan
ServiceName: fantacalcio
ContentType: text/plain
ContentLength: xx

The new championship will begin next Monday.
```

The following IM has been exchanged by two Fantacalcio players. Since the IM is sent within the game, the ServiceName is set to "fantacalcio". The FromAlias and the ToAlias headers are set to the player's teams.

- Service related instant messaging example

From user01(fantacalcio.private.H3G.Inter) to user02(fantacalcio.private.H3G.Milan) for fantacalcio service:

```
SIP MESSAGE sip:user02@h3g.it SIP/2.0
From: <sip:user01@h3g.it>
To: <sip:user02@h3g.it>
Subject: Team
FromAlias: fantacalcio.private.H3G.Inter
ToAlias: fantacalcio.private.H3G.Milan
ServiceName: fantacalcio
ContentType: text/plain
ContentLength: xx
<team info>
…
</team info>
```

Exploit users may communicate out of a service. They may use the basic features offered by a chat room. This message has sent within a chat session. The ServiceName header must be set to "peer2peer" in this case.

- Peer2peer instant message example

From user01(pippo) to user02(pluto) for a normal instant messaging service:

```
SIP MESSAGE sip:user02@h3g.it SIP/2.0
From: <sip:user01@h3g.it>
To: <sip:user02@h3g.it>
Subject: weather
FromAlias: pippo
ToAlias: pluto
ServiceName: peer2peer
ContentType: text/plain
ContentLength: xx

What a nice day!
```

5.3.3 SIP NOTIFY

The PresenceServer sends SIP NOTIFYs. The PresenceServer listens to SIP PUBLISHs sent by presentities (in fact, presence information is sent in SIP REGISTERs; the Presence Server is not compliant with the latest SIP standards) and forwards the information they contain to the user's watchers. SIP NOTIFYs may be meant for the Exploit client or for the Jeti client as well. This may happen because a buddy may be set in both an Exploit's buddy list and in the Jeti's buddylist. This is the reason why SIP NOTIFYs are always managed by both the applications. Moreover, the CommonFunctions class must remember the Exploit's buddies SIP URIs, in order to deliver SIP NOTIFYs to the Exploit services. This mechanism is necessary because it is not possible to modify the behavior of the Presence Server, which must be taken as it is. No changes that might simplify the processing of SIP NOTIFYs in the client have been possible.

5.3.4 SIP REGISTER

SIP REGISTERs are sent for two purposes: in order to register with the Registrar and in order to send presence information. The registration is performed when accessing to the IMS. Presence information is sent in the body of the SIP REGISTER message, in an XML format. The first value in the XML document tells where its namespace is defined. The "basic value= open" line means that the user is registered in the IMS. Media types tell which services the user is registered to. In this case, the user is registered only to the Rolegames service, among the Exploit services. The name of the character he is registered with is "Elfo".

```
SIP REGISTER sip:user02@h3g.it SIP/2.0
From: <sip:user01@h3g.it>
To: <sip:user01@h3g.it>
Call-ID: 483754325@20.20.20.20
CSeq: 456 REGISTER
Contact: <sip:user01@h3g.it:5060>
Accept: application/sdp, text/plain, multipart/mixed
Expires: 3000
Content-Type: application/xpidf+xml
Content-Length: …

<presence xmlns=http://www.ietf.org/ns/cpim-pidf-xml-1.0>
<tuple id = "device1">
<status>
<basic value="open"/>
<media type="voice">open</media>
<media type="instant message">open</media>
<media type="chat">open</media>
<media type="games/rolegames">open</media>
<mood type="online"/>
</status>
<note>rolegames.Elfo+</note>
<timestamp>Tuesday Jul 22 16:21:34 2003<timestamp/>
</tuple>
</presence>
```

5.4 CommonPlugin interaction with the GUI

The CommonPlugin module must interact with the services' graphical user interfaces. The ServiceConnector and the PluginConnector interfaces support communication among the GUIs and the CommonPlugin module.

A service must implement the ServiceConnector interface. Four classes that implement this interface have been defined: Rolegames, Fantacalcio, ECity and ServiceSupport.

The ServiceSupport class follows the singleton design pattern. The ServiceSupport class is always created, even if no other service is open. The existence of this object doesn't depend on the "servicesconf.txt" configuration file. The main tasks of this class are twi:

- Manages advanced peer2peer communication (i.e. audio call, chat and IMs) characterized by INVITEs with images and ring tones and alias support;
- Service GUIs control (activation of service GUIs and switching between services).

The other ServiceConnector implementations (i.e. Rolegames, Fantacalcio, ECity) manage the GUI and logic of their specific services.

The Exploit Client start and show is commanded by the Jeti GUI and executed by the CommonPlugin module in the CommonPlugin class. When the user clicks on the show button, the Exploit Welcome Page appears. The CommonPluginGUI class has been defined; its instance is in charge of the common graphical components and their behavior. These components are the Exploit Main Window, the Peer-to-Peer windows and the Bottom Panel. The Exploit Main Window is the first window shown when accessing the Exploit client (an example is depicted in Fig. 6.3). Peer-to-Peer windows are in charge of the user interfaces of the IM, Chat and Voip communication services (this is performed translating graphical events into PluginConnector's methods and SIP events into graphical events). The bottom panel is always visible and does not change when using the Exploit client. Its behavior is very similar to that of the menu bars on the bottom of the GUIs of some popular OSs, such as Windows or Linux. The bottom panel lets the user switch between the services' GUIs. The CommonPluginGUI listens to the graphical events originated by the Bottom Panel.

Fig. 5.6 makes clear how the CommonPluginGUI and the ServiceSupport class interact with the CommonPlugin module. Services are instantiated in the CommonPlugin module (in the CommonFunctions instance after the servicesconf.txt file is read). The same happens for the ServiceSupport instance. The ServiceSupport instance controls the CommonPluginGUI object, which listens for graphical user events from the Bottom Panel and from the Peer-To-Peer windows. The interaction between the CommonPluginGUI events and the CommonPlugin module is managed by the ServiceSupport instance, as far as the opening of the application and peer-to-peer communications are concerned. When the GUI of Rolegames, ECity or Fantacalcio is accessed or when SIP events are related to one of the services the control is shifted to the services' implementation. The interaction between the service and the CommonPlugin module is no more managed by ServiceSupport. The Rolegames, Fantacalcio or ECity GUIs are accessed clicking on one of the buttons of the bottom panel. A SIP event related to one of the services may be the arrival of a push message, for example.

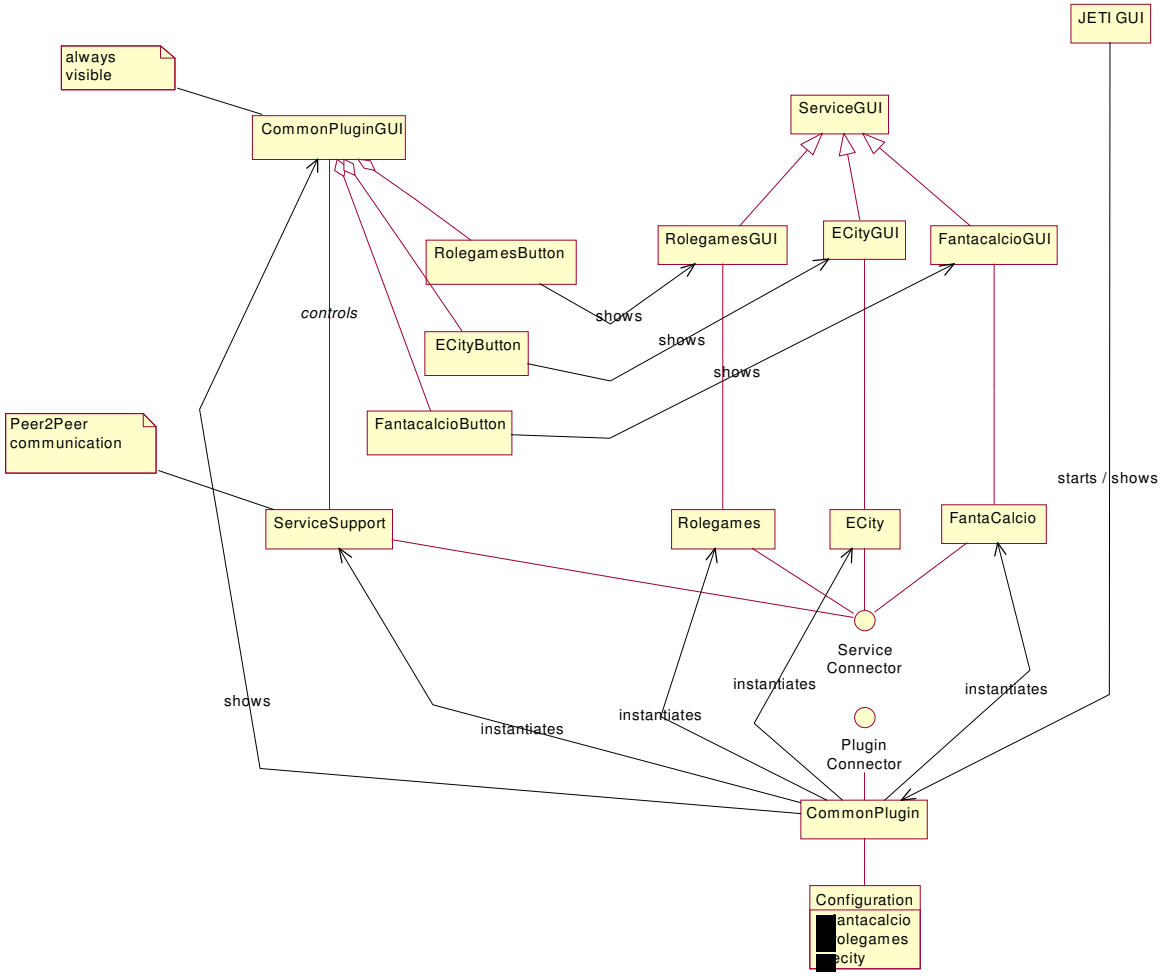

Fig. 5.6 CommonPlugin interaction with the GUI components

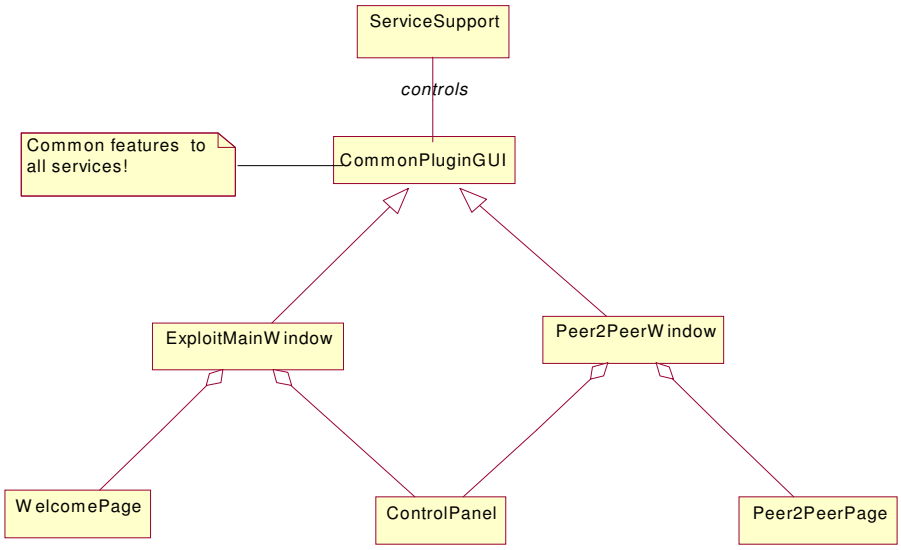

Fig. 5.7 CommonPluginGUI components

The relationships between panels and pages are illustrated in the following diagram. The diagram also includes the class Mind which is responsible for controlling the loading and unloading of pages on the Top panel.

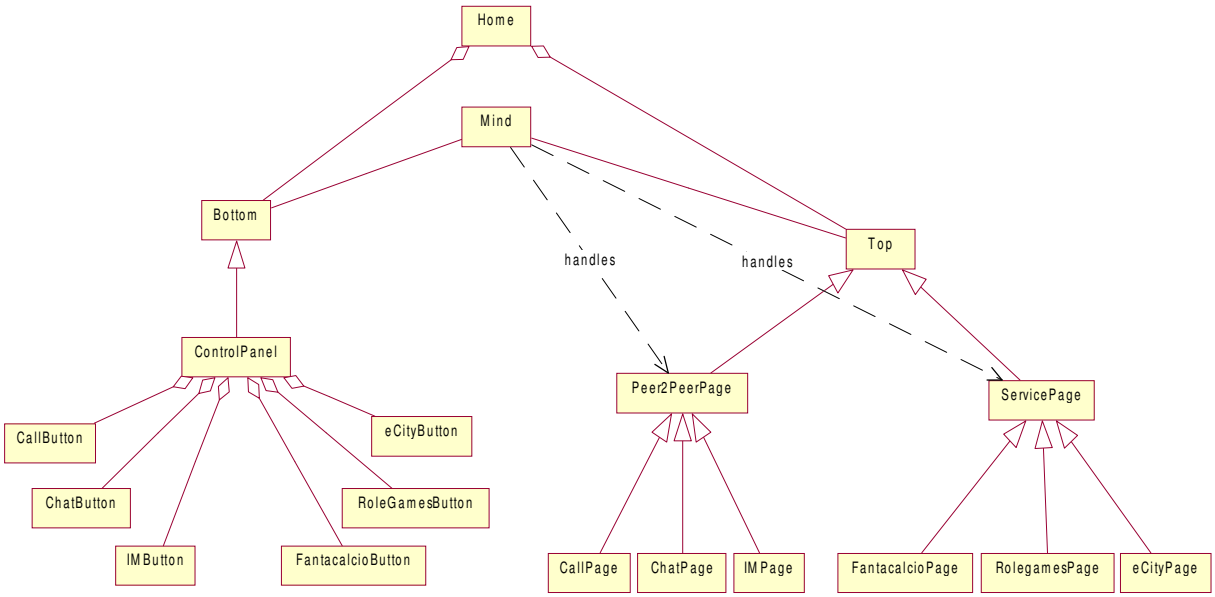

Fig. 5.8 The Mind class

5.4.1 The Mind class

The Exploit client window has been divided into a Top Panel and a Bottom Panel. The bottom panel is fixed; it never changes while using the client. The Top Panel view changes, depending on which service is used. Each service (Fantacalcio, Rolegames and ECity) can have ten pages that compose its GUI. If a user visits every single page of a service for each service,

and these pages are never destroyed this would lead to performance issues. The Mind class sets a criterion by which pages are loaded and unloaded into memory. The Mind rules are:

- No more than two pages per service can be loaded (current page and previous seen page);
- When the user moves from service1 to service2, the last page of service1 is saved;
- The loading of new pages forces the unloading of older pages.

A unique number, an index, identifies all Exploit pages. The index of each page belongs to a service specific range that is communicated to the Mind instance during service registration. The following addressing spaces have been defined:

- 1-10 for CommonPluginGUI;
- 11-20 for Rolegames;
- 21-30 for Fantacalcio;
- 31-40 for ECity.

The Mind class through the ServiceInfo class stores information regarding the GUIs. This information is:

- The object reference of the class which implements ServiceConnector interface (e.g. Fantacalcio, ServiceSupport);
- The name of the service;
- The last page shown for that service;
- The lower boundary of service page range;
- The upper boundary of service page range;
- The user's alias within that service.

The Peer class stores the information about the last buddy selected by the user for Peer2Peer communication. This information is managed by Mind. The data is:

- The SIP URL of the buddy;
- The alias of the buddy.

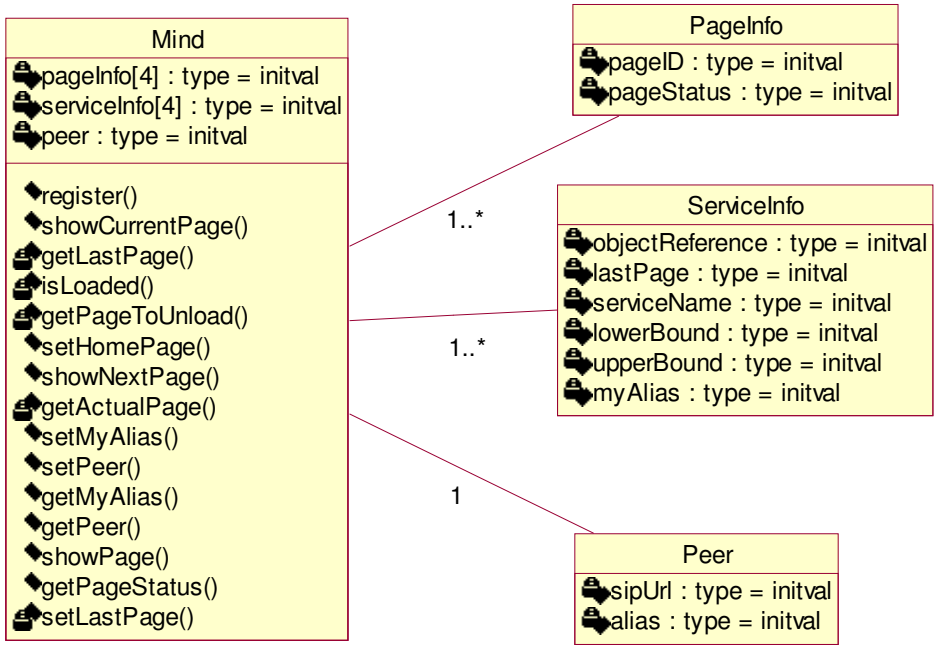

Fig. 5.9 The Mind class diagram

The following public methods of the Mind instance can be invoked by the GUIs for page navigation:

- `public void showCurrentPage (String serviceName) throws UnknownService`: this method allows moving between services through graphical user events on the ControlPanel;
- `public void showNextPage (int pageNumber) throws UnknownPage`: this method permits switching between the pages of a service;
- `public void showPage (int pageNumber) throws UnknownPage`: this method shows a service page when a SIP message arrives.

Any updates of the top panel, i.e. any change on the active page, must be controlled by Mind. A new page is always requested to Mind using one of three methods defined above.
Other public methods are:

- `public void register (String serviceName, Service serviceReference, int minPage, int maxPage)`: this method provides Mind with service information;
- `public void setHomePage (String serviceName, int pageNumber)`: this method sets the Exploit Plugin home page (i.e. ExploitMainWindow);
- `public int getPageStatus (int pageNumber) throws UnknownPage`: this method returns the status of a page. The values that may be returned are PageInfo.VISIBLE (2), PageInfo.HIDDEN (3), or PageInfo.UNLOADED (0);
- `public void setMyAlias (String serviceName, String alias) throws UnknownService`;
- `public String getMyAlias (String serviceName) throws NoAlias`;
- `public void setPeer (String sipUrl, String alias)`;
- `public Peer getPeer () throws NoPeer`.

5.4.2 The GUI class

The GUI class defines an interface between Mind and the service implementation; it so defines how Mind and a service must interact. All services must extend this class.
For example, Fantacalcio is defined as follows:

```
public class Fantacalcio implements ServiceConnector extends GUI {
      private Fantacalcio (PluginConnector pluginConnector, Mind mind);

      public static Fantacalcio getInstance(PluginConnector
      pluginConnector, Mind mind);

      // Service Connector methods
      // GUI methods
}
```

The GUI class allows storing:

- The service name (serviceName);
- The current Buddy List;
- The presence information about buddies.

The following public methods have to be implemented by each service:

- `public void showPage (int pageNumber);`
- `public void loadHomePage ();`
- `public void loadPage (int pageNumber);`
- `public void unloadPage (int pageNumber);`
- `public void hidePage (int pageNumber);`
- `public void setCallID (String callID)`: the callID generated from a service page is stored;

- `public boolean checkCallID (String callID): to verify if messages refer to the same SIP dialogue;`
- `public void setBL (List list);`
- `public void setPresence (String sipUrl, String alias, String status);`
- `public List getBL (List list);`
- `public PresenceInfo getPresence (String sipUrl, String alias, String status).`

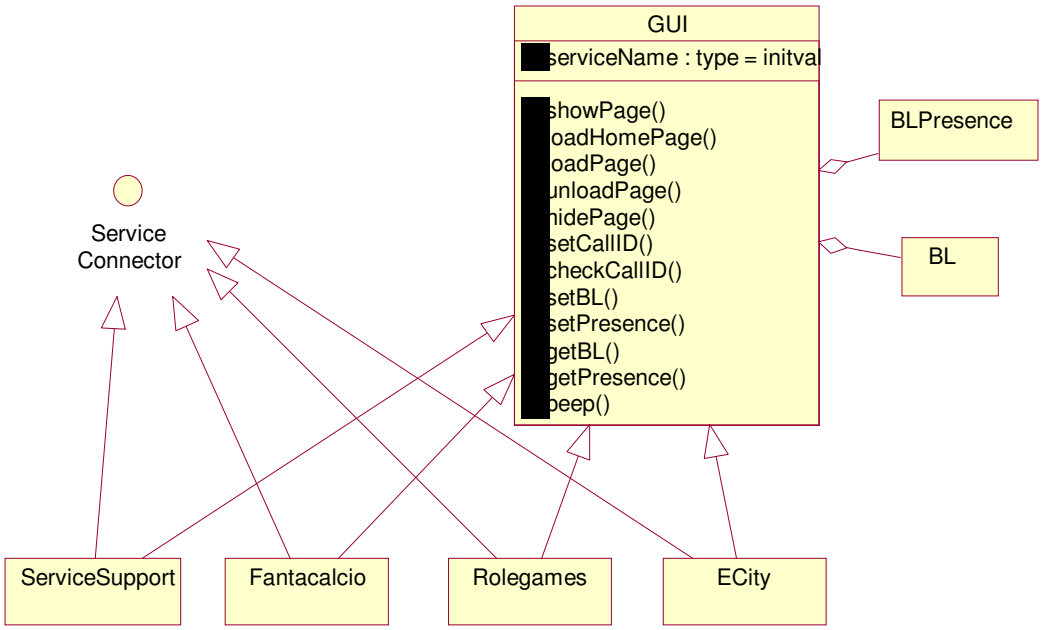

Fig. 5.10 The GUI class diagram

5.5 Exploit Service example

The following paragraphs show how the CommonPlugin module works. The Rolegames service has been chosen for the examples.

5.5.1 Rolegames challenge request and succeed

The main classes that take part in this scenario are:

- **RolegamesGUI**: this class must be a graphical event listener;
- **CommonFunctions**: this class implements the PluginConnector interface.

When the user sends a challenge to a peer through a button present on the GUI, this will trigger the specific service's class (in this case RoleGamesGUI) to invoke the chatInitiateSession on the PluginConnector object. The chatInitiateSession maps upper level related actions on the INVITE SIP message of the protocol stack and must therefore contain all the information necessary to send the INVITE (this information is passed through the method's parameters and is represented by the message's recipient, its subject and body).
The 180 Provisional Response is mapped on the chatProcessRinging method of the Rolegames instance. The 200 Final Response is mapped on the chatOpenSessionSucceed method of the

Rolegames instance. If the final response were not a successful one, the chatOpenSessionFailed would be invoked. Once the session is established, the exchange of cards may start.

.

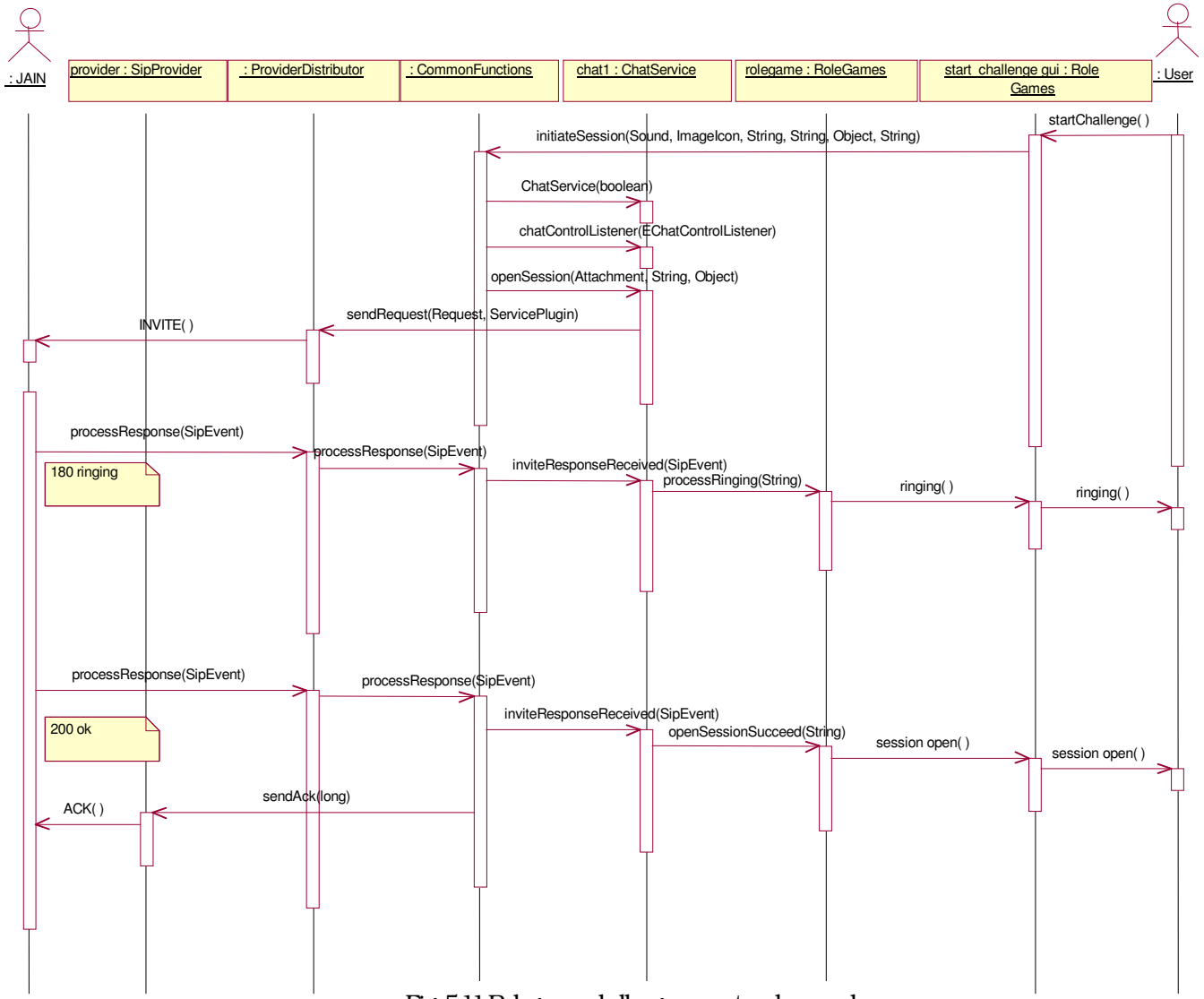

Fig. 5.11 Rolegames challenge request and succeed

5.5.2 Rolegames challenge invitation and acceptance

The CommonFunctions object filters incoming requests and decides whether they must be dispatched to its services or not. If the request must be delivered to the Rolegame service the sessionInitiated() method is invoked on the Rolegames's instance. The activation of a window and the playing of the audio clip carried in the INVITE will tell the user about the invitation receipt. The user will be able to refuse or accept the invitation. If the accept button is clicked, the acceptSession method is invoked on the CommonFuntions object acting in this case as PluginConnector.

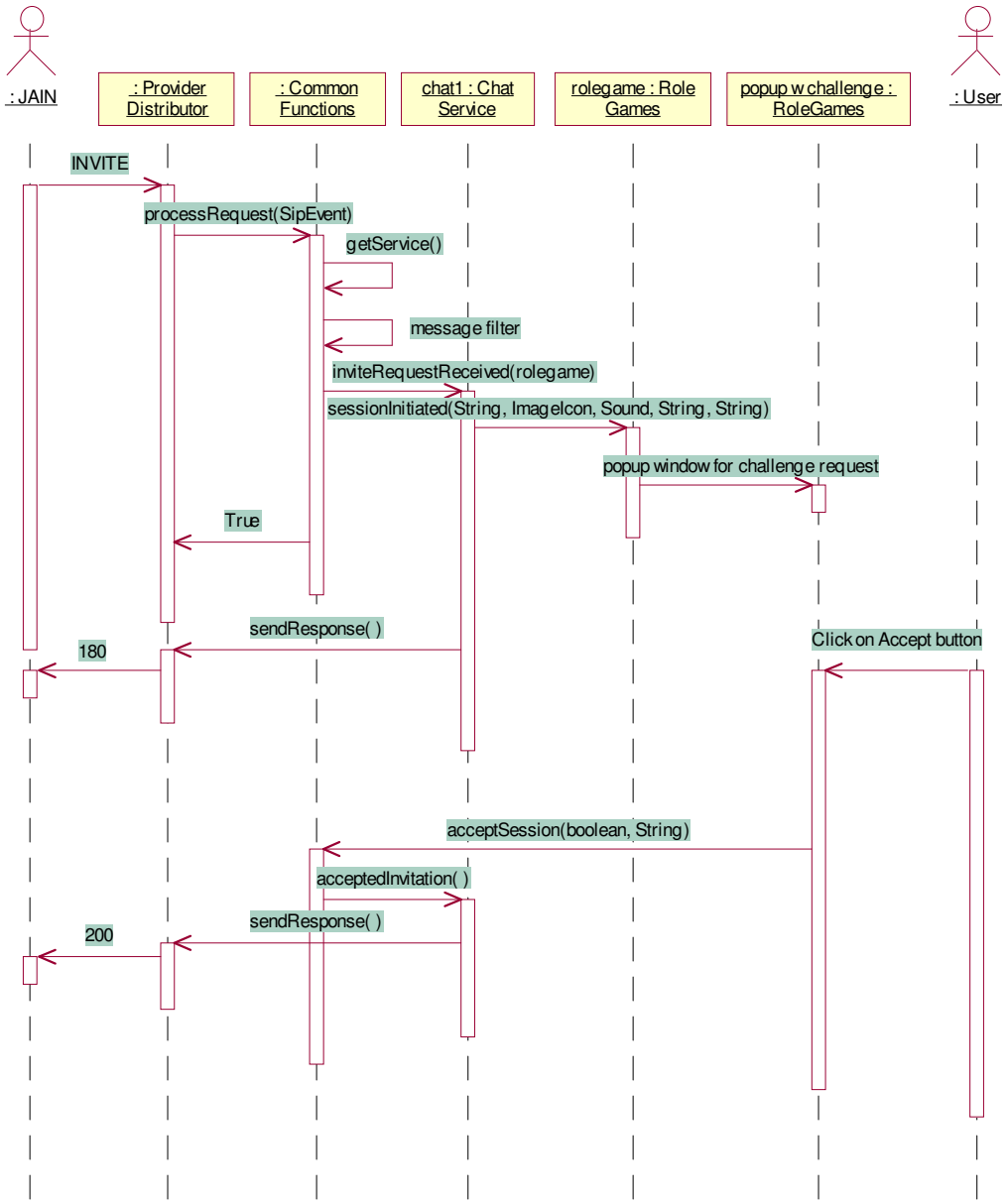

Fig. 5.12 Rolegames challenge invitation and acceptance

5.6 GUI related sequence diagrams

The following sequence diagrams are intended to explain through some examples how service GUIs and CommonPlugin interact each other.

5.6.1 ExploitPlugin start

The ExploitPlugin can be started by selecting "Exploit Client" among the available plug-ins listed in JETI and clicking on the "Start" button.
When CommonPlugin module is started, it performs the following tasks:

- It initializes internal object and connects to the protocol stack;
- It instantiates the Mind class;
- It creates the ExploitMainWindow which is composed of two containers:
  - Top level container that can be used by service GUIs with the restriction that only one service page can be visible at a time: at the beginning a welcome page is shown;
  - Low level container which shows the ControlPanel; it allows:
    - Performance of advanced peer to peer communication independently from service GUIs: audio call, multimedia chat and instant messaging;
    - Activation and movement between service GUIs.
- It instantiates, based on the settings read from a configuration file, the classes (i.e. Rolegames, ECity, Fantacalcio) which implement the service GUI and logic. A reference to the CommonFunctions instance is passed as parameter, this object implements the PluginConnector interface that is necessary to start a communication from the service. The reference to the Mind object is also passed as input parameter in order to allow services to interact with Mind. When a service is instantiated (service constructor method), it registers to Mind and it passes it its reference and the range of its pages;
- It registers the plugin in the Jeti environment;
- It creates and keeps tables for communication and dispatching purposes.

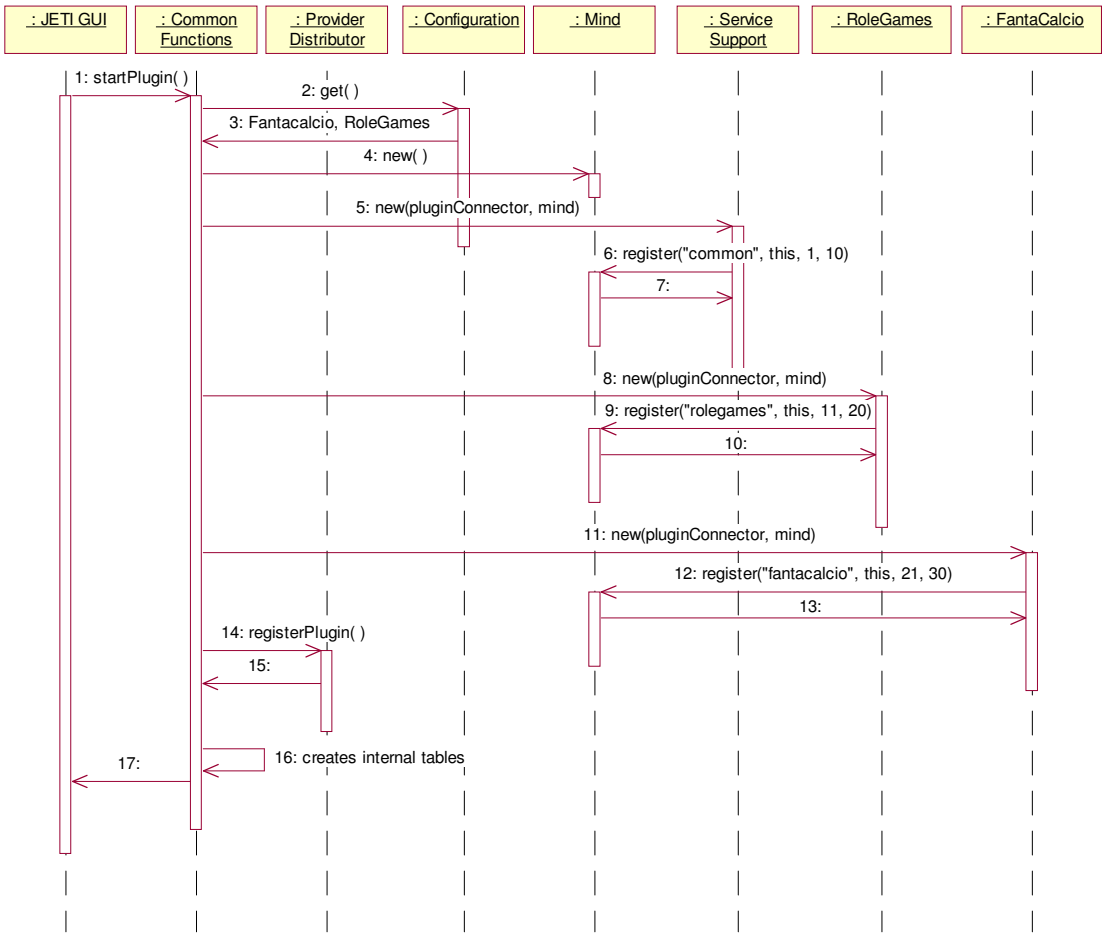

Fig. 5.13 Start of the Exploit application

5.6.2 ExploitPlugin show

When the user requests by clicking the "Show" button of JETI GUI, the show method is invoked on CommonPlugin that in turn invokes loadHomePage () method on ServiceSupport.
A new window is created which shows the Exploit Welcome Page on the Top panel and the ControlPanel on the Bottom.
When ControlPanel is instantiated the reference to PluginConnector and to Mind are provided.

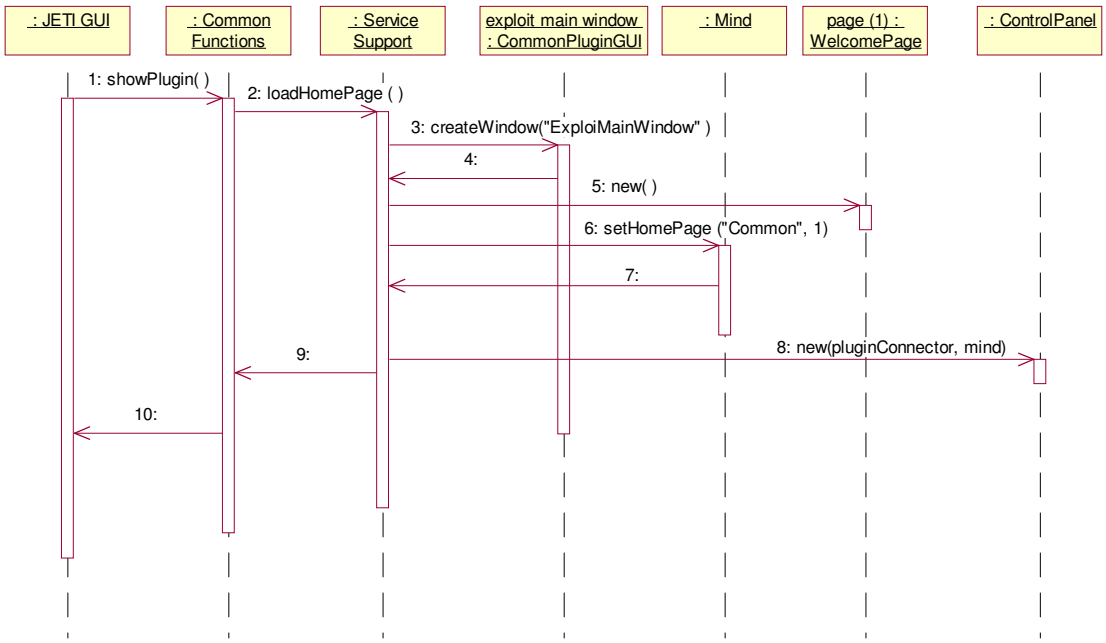

Fig. 5.14 Show of the GUI of the Exploit application

5.6.3 Service activation from the Control Panel

The following diagrams illustrate two examples of activation of a service by clicking the associated icon/button on the ControlPanel.

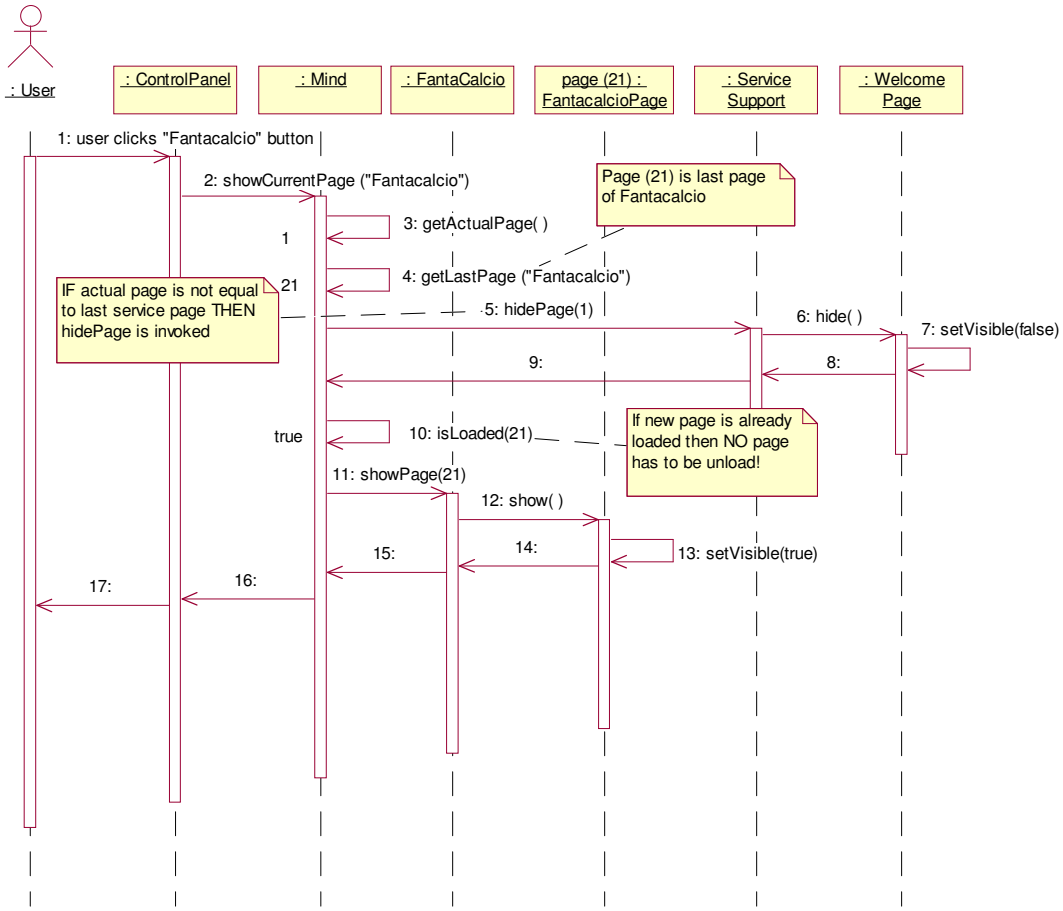

Fig. 5.15 Activation of the Fantacalcio GUI

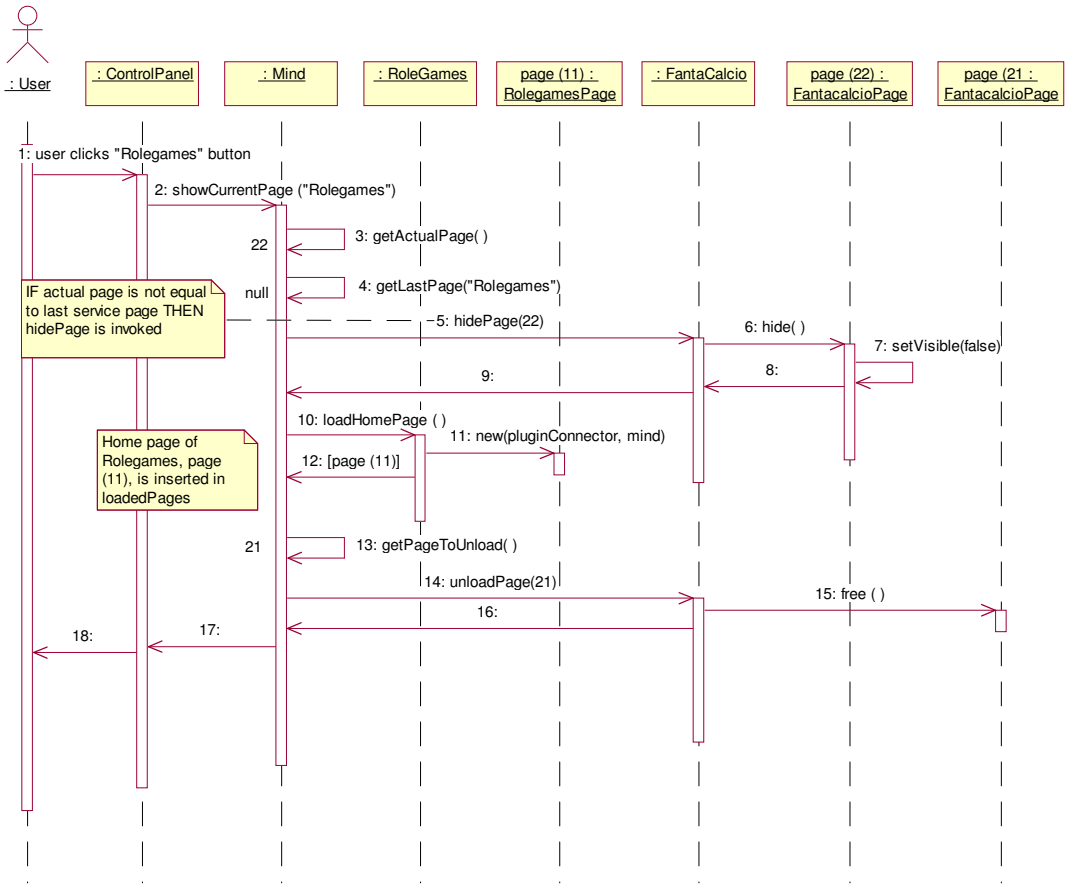

Fig. 5.16 Activation of the Rolegames GUI

5.6.4 Navigation through the service GUI pages

The following diagrams provide two example of navigation between the pages of a specific service.

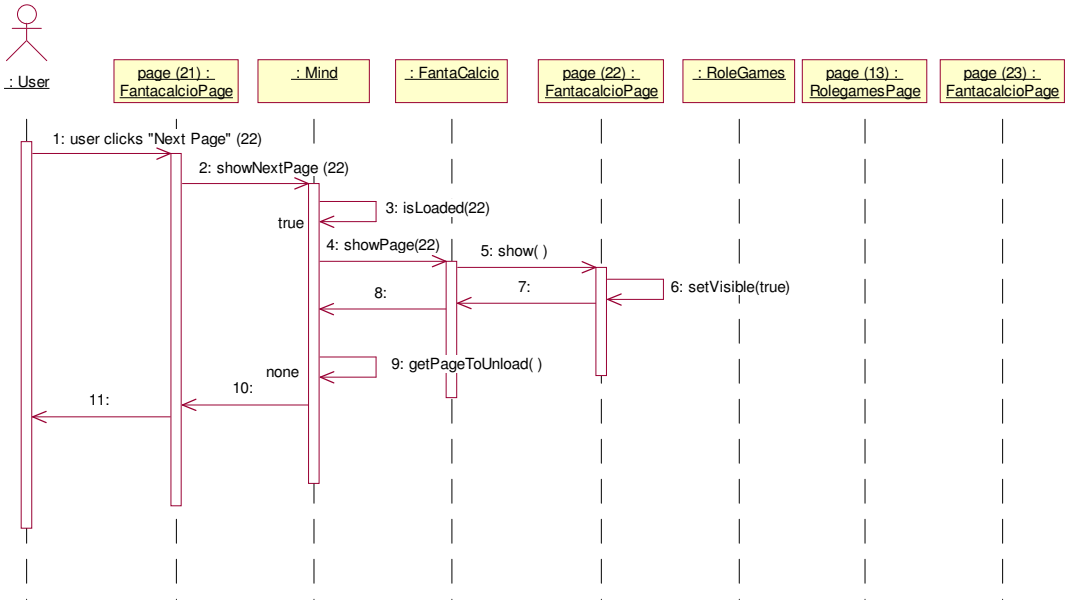

Fig. 5.17 Example of navigation through pages

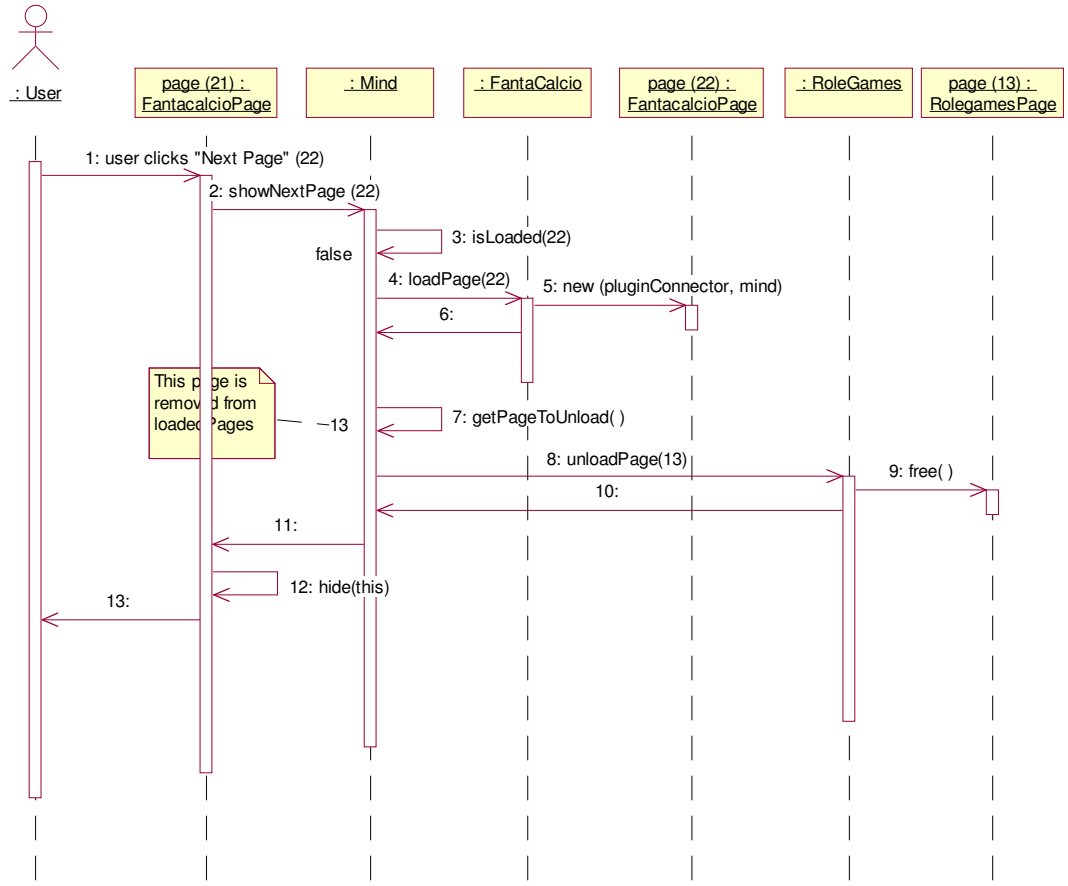

Fig. 5.18 Example of navigation through pages

5.6.5 User registration

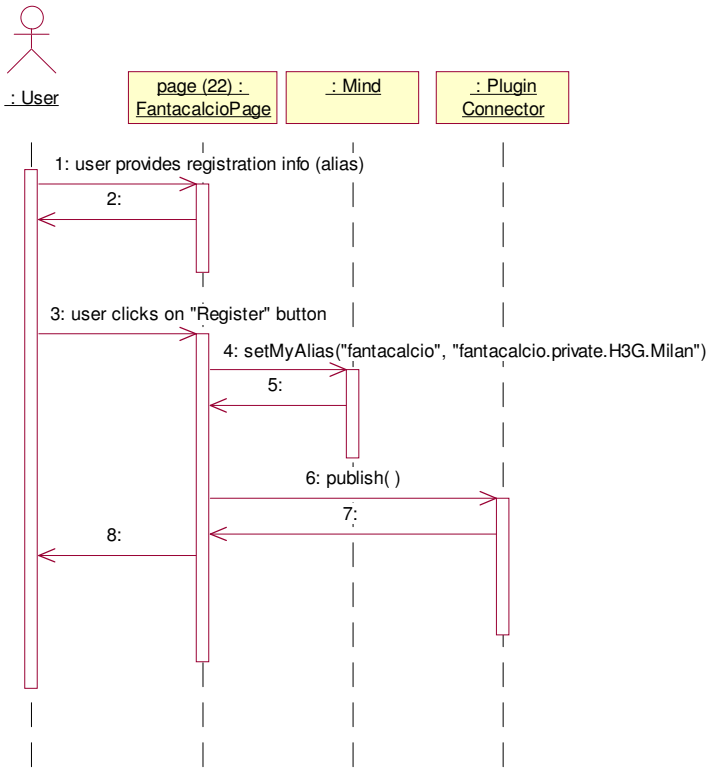

Fig. 5.19 Registration to the Rolegames service

5.6.6 Chat with a buddy

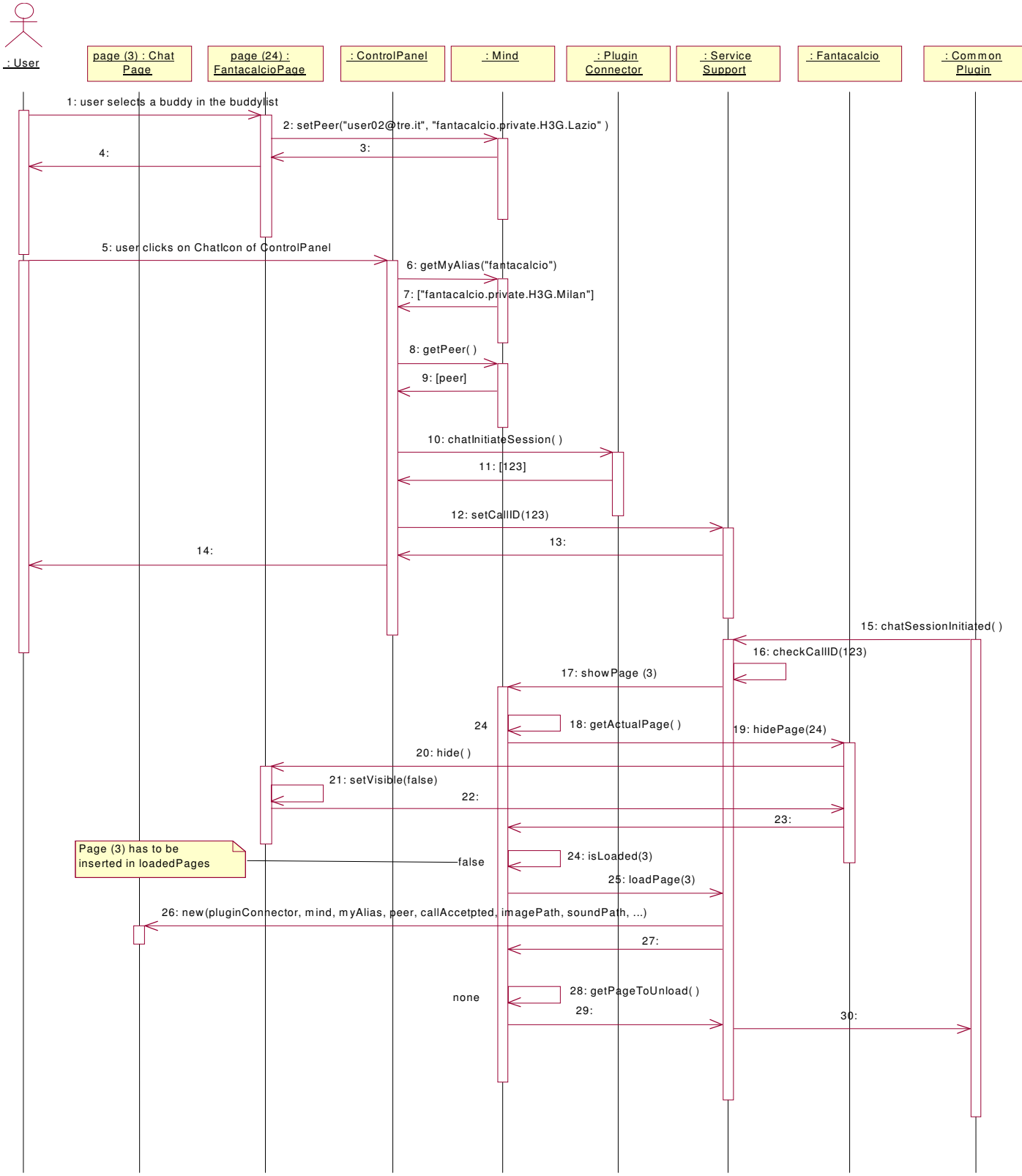

Fig. 5.20 Chat with a buddy

The diagram shows the entire step from peer selection, to chat invitation and acceptance. Please, note that the peer is always the last selected buddy!

5.7 Java packages

The following packages contain Exploit client classes:

Package	Content
exploit.client	Classes of CommonPlugin
exploit.client.gui	Classes of CommonPluginGUI and of support for GUI
exploit.client.pluginconnector	Classes of interface PluginConnector
exploit.client.serviceconnector	Classes of interface ServiceConnector
exploit.client.rolegames	Classes for RoleGames service logic and GUI
exploit.client.fantacalcio	Classes for Fantacalcio service logic and GUI
exploit.client.ecity.buddyfinder	Classes for BuddyFinder service logic and GUI
exploit.client.ecity.carsharing	Classes for CarSharing service logic and GUI

serviceconnector package

```
package exploit.client.serviceconnector

public interface EPresenceListener {
     void notifyReceived (String sipURL, String alias,
         String status) throws UnknownBuddy;
}

public interface EChatControlListener {
     void chatSessionInitiated (String fromAlias, String
         toAlias, String from, String subject, Image image,
         Sound sound, String text, String callId);
     void chatSessionClosed (String callId) throws
         UnknownCallId;
     void chatProcessRinging (String callId) throws
         UnknownCallId;
     void chatOpenSessionSucceed (String callId) throws
         UnknownCallId;
     void chatOpenSessionFailed (String callId) throws
         UnknownCallId;
}

public interface EMessageListener {
```

```
     void messageReceived (String fromAlias, String
         toAlias, String from, String subject, Image image,
         Sound sound, String text) throws UnknownMessage;
     void chatMessageReceived (String fromAlias, String
         toAlias, String from, String subject, Image image,
         Sound sound, String text, String callId) throws
         UnknownCallId;
     void sendMessageFailed (String to, String subject,
         String errorMessage);
}

public interface EVoIPControlListener {
     void voipSessionInitiated (String fromAlias, String
         toAlias, String from, String subject, Image image,
         Sound sound, String text, String callId);
     void voipSessionClosed (String callId) throws
         UnknownCallId;
     void voipProcessRinging (String callId) throws
         UnknownCallId;
     void voiOpenSessionSucceed (String callId) throws
         UnknownCallId;
     void voipOpenSessionFailed (String callId) throws
         UnknownCallId;
}

public interface ServiceConnector extends
                      EPresenceListener,
                      EChatControlListener,
                      EMessageListener,
                      EVoIPControlListener;
pluginconnector package
package exploit.client.pluginconnector

interface EChatControl {
     void chatAcceptSession (String callId, boolean accept)
         throws UnknownCallId;
     String chatInitiateSession (String fromAlias, String
         toAlias, String serviceName, String to, String
         subject, String soundPath, String imagePath, String
         text, ServiceConnector serviceImplementation)
         throws ChatRoomsBusy;
}

public interface EMessageControl {
     void sendMessage (String fromAlias, String toAlias,
         String serviceName, String to, String subject,
         String imagePath, String soundPath, String text);
```

```
     void chatSendMessage (String fromAlias, String
         toAlias, String serviceName, String to, String
         subject, String imagePath, String soundPath, String
         text, String callId) throws UnknownCallId;
}

public interface EVoIPControl {
     void voipAcceptSession (String callId, boolean accept)
         throws UnknownCallId;
     public String voipInitiateSession (String fromAlias,
         String toAlias, String serviceName, String to,
         String subject, String soundPath,  String
         imagePath, String text, ServiceConnector
         serviceImplementation) throws SessionBusy;
}

interface EPresenceControl {
     void publish (ArrayList openGames, String alias) throws
SyntaxError;
}

public interface PluginConnector extends
                         EChatControl,
                         EmessageControl,
                         EVoipControl,
                         EPresenceControl;
```

6. Audio Call Solution

The Voice over IP call is one of the requirements for the Exploit client. The codec that has been chosen for the exchange of media is the GSM codec. This choice has been made because of its low bit-rate (13 kb/s), compared to the other available codecs. The audio call is full-duplex; calls may be performed in a traditional phone call style.

I worked on the design and implementation of the signaling and of the exchange of media. In the implementation, the Siemens' SIP APIs and the Simple Media Framework (SMF) have been used. The SIP APIs are the same that are found in the implementation of the rest of the client. The SMF APIs are a JMF based solution for PocketPC 2002 powered PDAs. The SMF and the JMF offer the same interfaces at a high level. The difference between JMF and SMF is that SMF maps some java methods into DLLs that have been designed specifically for the PDA (this result makes RTP transmission and voice capturing possible).

6.1 Signaling

The signaling design may be seen from two points of view, the design of the state machine that implements the call logic and the design of a flexible software pattern.

The state machine that has been used in the implementation of the call logic may be seen in the following figure:

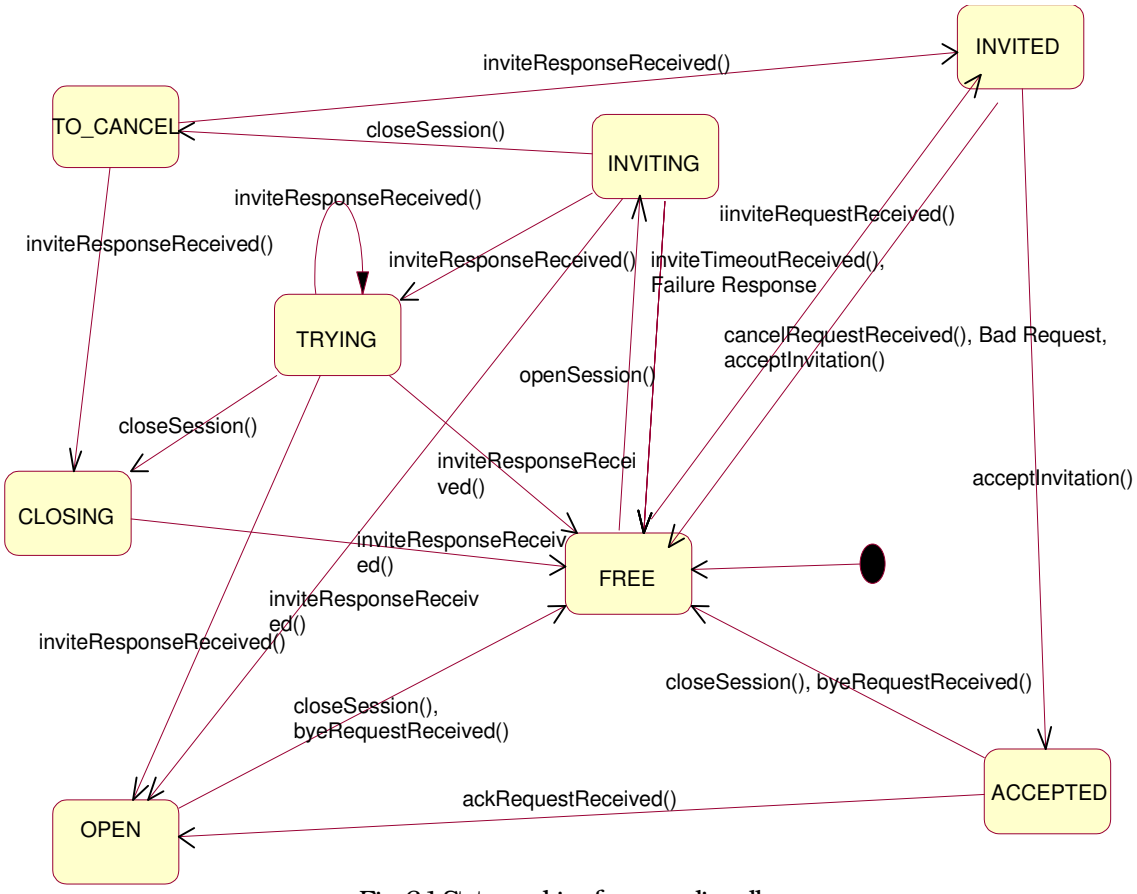

Fig. 6.1 State machine for an audio call

This state machine is the same that has been used for the chat session setup. The states are:

- **Free**: This is the starting point for the Voip machine. When the user starts the client no phone call has been possible before, so the call setup begins in the Free state. This is also the final state for all phone calls. It is possible to get back to the Free state in the following cases:
  - From the Inviting state:
    - A Failure Response (3xx, 4xx, 5xx, 6xx) is received;
    - No response is received before the timeout;
  - From the Invited state:
    - A SIP CANCEL request is received before having sent a Final Response;
    - The received SIP INVITE is not compliant with the message coding expected by the client;
    - The user refuses the invitation. The acceptInvitation() method is called, passing it a FALSE value as parameter.
  - From the Open state:
    - The user hangs up the communication;
    - The second party hangs up the phone (a SIP BYE is received).
  - From the Closing state:

- ♣ If a success Final Response is received in reply to the SIP INVITE that has been sent, a SIP BYE is sent and the transition to the Free state is performed. If a Failure Response is received in reply to the SIP INVITE, the transition is performed directly. The first case happens when the callee receives the SIP CANCEL after having sent the success response. The second case happens when the callee receives the SIP CANCEL before sending the success response.

¬ From the Trying state:

- ♣ If a Failure Response (3xx, 4xx, 5xx, 6xx) is received.

- **Inviting**: When in the Free state, the user is able to start a phone call. This triggers the send of a SIP INVITE to the called party. The Voip machine will be in the Inviting state as long as no response has been received.
- **Invited**: Beginning from the Free state, this state is entered when an invitation is received. Until a response is sent, the Voip machine will be in this state (unless no response is sent and a timeout is triggered; this would result in going back to the free state). It is possible to get to the Invited state from the Cancel state, but this transition is much less common. This may happen if a user starts a phone call and hangs up immediately. If the hang up happens when only Provisional Responses have arrived, the client sends a SIP CANCEL, assuming that the callee's client has not had the time to send a Final Response. If in the meanwhile a success Final Response arrives in reply to the sent SIP INVITE (this response is generated before the SIP CANCEL arrives), the SIP CANCEL has no effect in canceling the session. The Voip machine moves to the Invited state.
- **Accepted**: The Accepted state may be accessed only from the Invited state. The willingness to talk of the user triggers this transition. When the users click on the Accept button of the Voip GUI, the acceptInvitation is invoked on the Voip machine. The True value for the passed parameter states that the invitation is accepted (from a SIP point of view, this is translated by sending a 200 Ok Response).
- **Trying**: The Trying state may be from the Inviting state. The arrival of SIP Provisional Responses triggers this transition. The arrival of multiple Provisional Responses keeps the Voip machine in this state. A typical case is the arrival of a 100 Trying and of a 180 Ringing in sequence. The 100 Trying is the response sent by the proxy who forwarded the request. The 180 Ringing is the provisional response sent by the callee's client.
- **To Cancel:** The ToCancel state represents the state into which the Voip machine transitions when the user tries to set a call and closes it before any response is received. The closeSession() method is called before the arrival of a response. For this reason, this state may be accessed only from the Inviting state.
- **Closing:** The Closing state may be accessed from the ToCancel state and the Trying state. The transition from the ToCancel state happens when a provisional response to the invitation request arrives. The transition from the Trying state happens if the user is willing to close the session and so the closeSession() method is invoked.

- **Open:** The transition to the Open state may happen starting the following states:

¬ Accepted state: This transition happens when the SIP ACK is received from the caller.

- ¬ Inviting state: This transition is performed when a success response (2xx) is received.
- ¬ Trying state: This transition is performed when a success response (2xx) is received.

The software pattern that has been chosen in the design of the Voip service is the State pattern. The reason for this choice is the demand for keeping states and state-specific behavior separated. Keeping states separated from their behavior means that improvements may be made in a very easy way. If the behavior of one state changes, only one particular class must be changed. This pattern has been chosen because the SIP protocol is still an evolving protocol.

A Voip class has been defined. The singleton pattern has been chosen for this class, as for the other classes involved in the Voip implementation. The reasons are two: the target platform for this solution is a PDA, and only one Voip session may be open at time. Since only one Voip session may be open at time, there is no reason to keep multiple instances of this class. The Voip instance stores the instance of the VoipState class. The VoipState class defines an interface common to all classes that represent different operational states. The subclasses of VoipState implement state-specific behavior. The signatures of the VoipState class methods are:

- `public void acceptInvitation(boolean accept);`
- `public void ackRequestReceived(SipEvent event);`
- `public void byeRequestReceived(SipEvent event);`
- `public void cancelRequestReceived(SipEvent event);`
- `public void closeSession();`
- `public void inviteRequestReceived(SipEvent event);`
- `public void inviteResponseReceived(SipEvent event);`
- `public void inviteTimeoutReceived(SipEvent event);`
- `public String openSession(String fromAlias, String toAlias, String serviceName, String to, String subject, String soundPath, String imagePath, String text);`
- `protected final void timeout();`
- `protected void extractPeerIPAddress(String s);`
- `protected void extractPort(String s);`
- `protected void receivedMultiInvite(SipEvent sipevent, ContentTypeHeader contenttypeheader);`
- `public void changeState(Voip voip, VoipState voipstate);`
- `protected void cleanUp().`

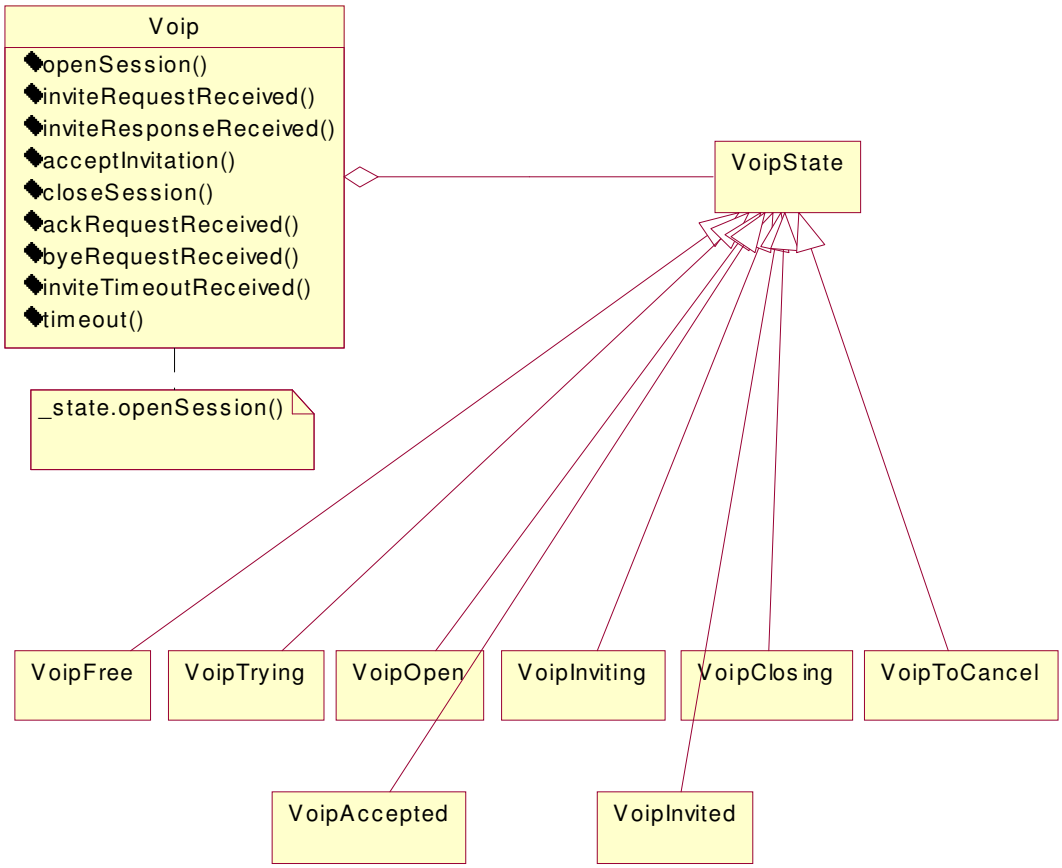

Fig. 6.2 Voip class diagram

All the above methods, except the last four, are implemented in the Voip class too. The Voip class' attribute _state, of type VoipState, has the role of storing the current state of the Voip machine. When Voip class is instantiated for the first time, the _state attribute is initialized with the instance of the VoipFree class. Depending on the methods that are invoked on the Voip instance the _state attribute changes its value. The result is that the same method but in different classes is invoked. This behavior may be understood by the implementation of the Voip class, that is here shown.

The Voip class extends the TimerTask class, as mentioned in the previous chapter.

```
public class Voip extends TimerTask
{
```

The constructor has been declared private, in order to achieve the singleton behavior. Here all possible states are initialized and the _state attribute is set to Free.

```
private Voip()
{
      VoipAccepted.getInstance();
      VoipClosing.getInstance();
      VoipInvited.getInstance();
      VoipInviting.getInstance();
      VoipOpen.getInstance();
      VoipToCancel.getInstance();
      VoipTrying.getInstance();
      _state = VoipFree.getInstance();
}

public static synchronized Voip getInstance() {
      if (_instance == null)
            _instance = new Voip();
      return _instance;
}
```

The openSession() method is not implemented here. The implementation that is invoked depends on the state of the Voip machine. We have seen above which are the allowed transitions in the Voip machine. If a not allowed transition is invoked, an exception is handled. The only allowed transition starts from the Free state. In all other cases a `new Exception("Session is in use")` will be thrown.

```
public String openSession(String fromAlias, String toAlias,
String serviceName, String to, String subject, String
soundPath, String imagePath, String text) throws Exception
{
      return      _state.openSession(fromAlias,      toAlias,
      serviceName, to, subject, soundPath, imagePath, text);
}
```

The closeSession() method launches no exceptions if called on an illegal transition, it simply does nothing in that case. This happens when the _state variable stores one of the instances of the following classes: VoipFree, VoipClosing, VoipToCancel.

```
public void closeSession()
{
      synchronized(this)
      {
            _state.closeSession();
      }

}
```

The acceptInvitation() throws a `new Exception("Object not invited")` in all states, minus the VoiInvited state.

```
public void acceptInvitation(boolean flag)
      throws Exception
{
      if(VoipState.inviteEvent != null) //INVITED
      {
            _state.acceptInvitation(flag);
```

```
        }
    }
```

Nothing is done if the inviteResponseReceived() method is invoked on the VoipInvited, VoipAccepted and VoipOpen instances. All other cases are managed following the machine state logic.

```
    public void inviteResponseReceived(SipEvent sipevent)
    {
        _state.inviteResponseReceived(sipevent);
    }
```

If the byeRequestReceived() method is invoked out of one of the VoipAccepted and the VoipOpen states, a 481 Transaction Does Not Exist failure response is returned to the sender of the SIP BYE.

```
    public void byeRequestReceived(SipEvent sipevent)
    {
        _state.byeRequestReceived(sipevent);
    }
```

The inviteTimeoutReceived() method does nothing in all states, with the exception of the VoipInviting state. The SIP stack invokes this method, when no response is received for the sent request.

```
    public void inviteTimeoutReceived(SipEvent sipevent)
    {
        _state.inviteTimeoutReceived(sipevent);

    }
```

The only state that manages an inviteRequestReceived() without replying a 486 Busy Here response is the VoipFree state.

```
    public void inviteRequestReceived(SipEvent sipevent)
    {
        _state.inviteRequestReceived(sipevent);
    }
```

The cancelRequestReceived() method is managed only by the VoipInvited state. In all other cases a 200 response is returned.

```
    public void cancelRequestReceived(SipEvent sipevent)
    {
        _state.cancelRequestReceived(sipevent);
    }
```

Nothing happens invoking the following method, with the exception of the VoipAccepted state.

```
    public void ackRequestReceived(SipEvent sipevent)
    {
        _state.ackRequestReceived(sipevent);
    }
```

The timeout() method is invoked when a SIP ACK does not arrive in the given time. The invocation of this method takes the Voip machine in the VoipFree state.

```
public final void timeout()
{
      _state.timeout();
}


public boolean isFree()
{
      if (_state instanceof VoipFree)
            return true;
      return false;
}
```

The classes that implement states invoke the following method on the Voip instance in order to control its state transitions.

```
protected void changeState(VoipState vs) {
      _state = vs;
}
```

This method is inherited from the TimerTask class. It is invoked when setting the timeout for the ACK arrival.

```
protected void setScheduleTime(long l) {
      scheduleTime = l;
}
```

The Voip class stores two variables, its unique instance and its state.

```
private static Voip _instance = null;
private VoipState _state;

}
```

The sequence diagram shows how the state machine is implemented with method invocations. Some classes that take part in this process are omitted in order to focus on the Voip calls.

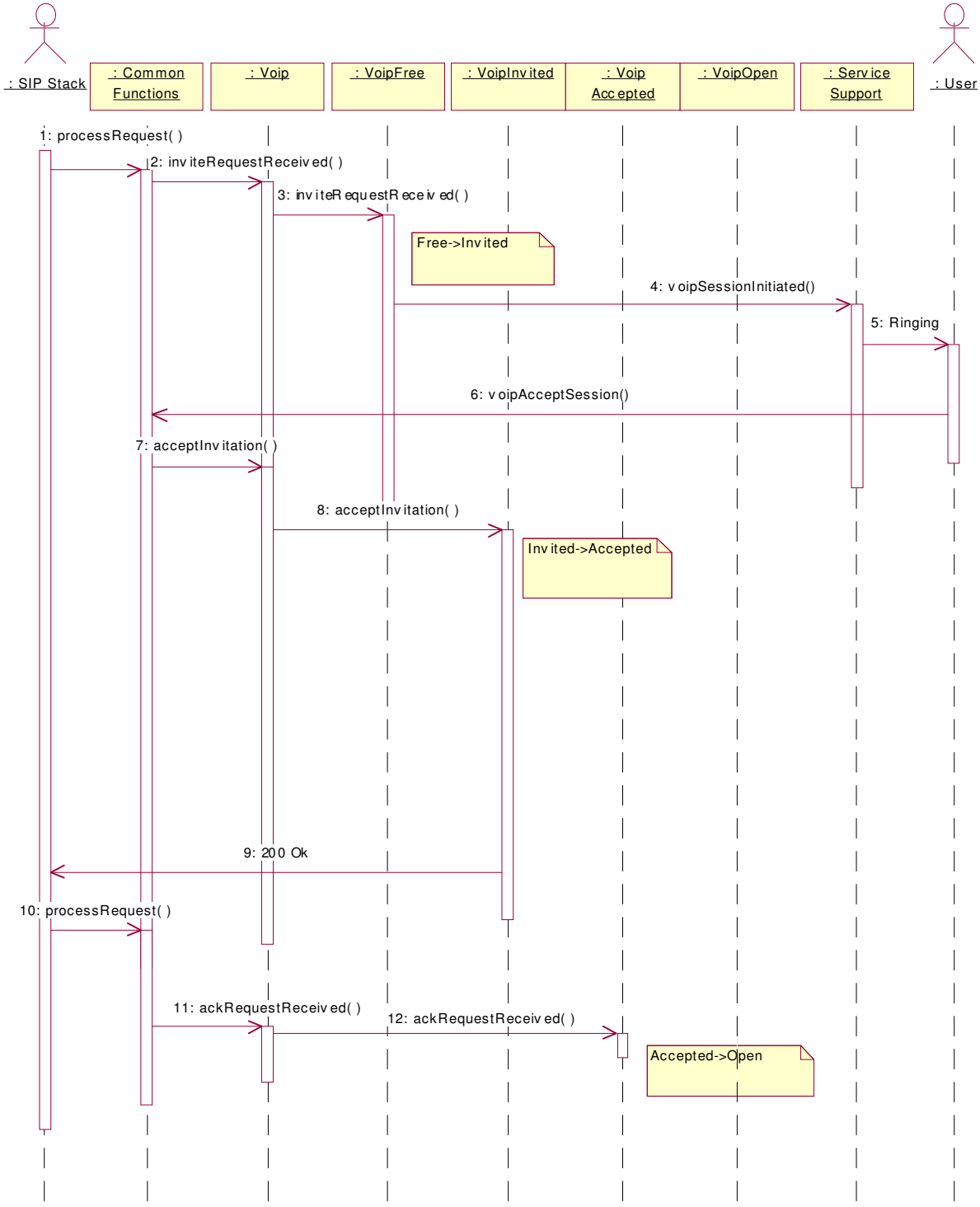

Fig. 6.3 Voip sequence diagram for an incoming call

6.2 Media

The Voip class implements the logic of a call setup, but the management of media is done by the MediaControl class. The MediaControl class is implemented following the singleton pattern. This class controls two classes, Transmit and Receive. The Transmit and Receive classes follow the singleton pattern and are implemented using the SMF APIs.

The MediaControl class is first instantiated when opening the Voip GUI. The resources that are necessary for transmission are reserved in the MediaControl constructor. This is done instantiating the Transmit instance and calling the create() method on it. The reason for reserving resources in advance, before starting a call, is that this process is time consuming. The association of this process to the Voip GUI slows the GUI, but lets the flow of media start faster once the call is setup.

The quality of received and transmitted media is acceptable. A silence codec has been integrated in the audio solution and experimented in transmission. This codec, while processes voice, suppresses silence and any sound under a threshold volume in transmission. This limits the number of sent packets and of generated traffic. Moreover, a smaller delay is experienced in the reception of voice. The quality of an audio call, performed on a PC, is comparable to that of a commercial product such as NetMeeting. The quality of the audio call on the PDA is not as good, because of processing limits, but it is still acceptable.

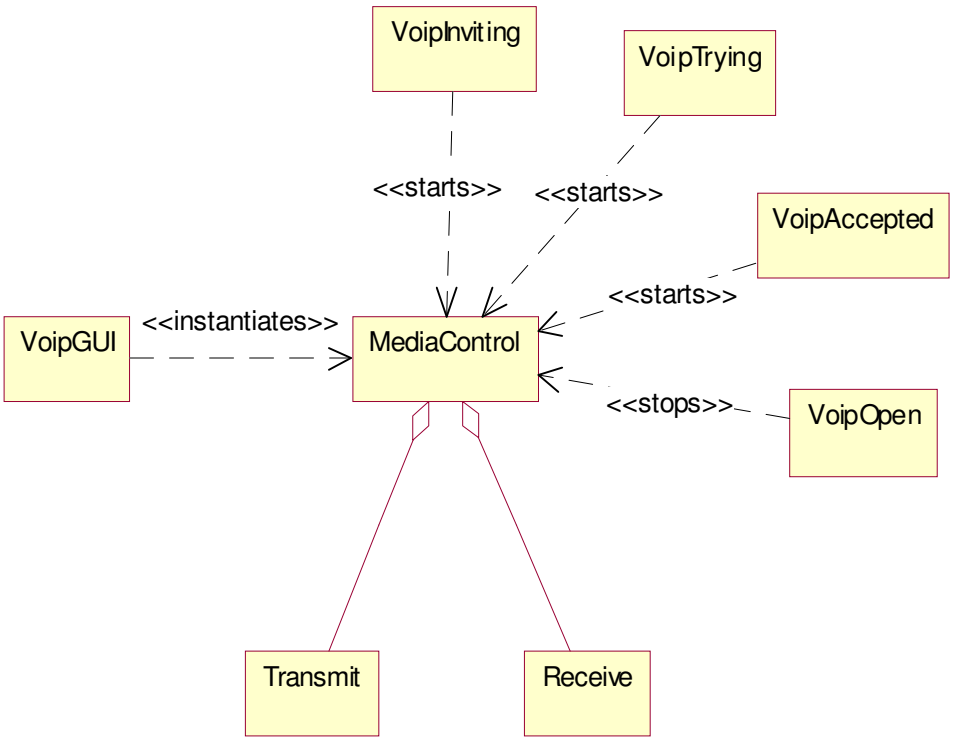

Fig. 6.4 MediaControl class diagram

The signaling states where media flow is commanded to start and stop are those directly connected to the VoipOpen state.

7. Functional Evaluation

The CommonPlugin module has been tested and integrated with the rest of the client modules. The steps that have been taken, once the implementation was finished, have been those of testing the CommonPlugin as a stand-alone application and then testing it when the services were integrated. The objectives were of testing basic functionalities and of discovering whice are the limits of the CommonPlugin module. I also provided support to all other tests, which mainly regarded client-server interactions and service logic.

Four kinds of tests have been done, on the CommonPlugin:

- Tests of Instant Messaging;
- Test of a Chat session;
- Test of a VoIP call;
- Test of Presence;
- Integration tests.

7.1 Instant Messaging Evaluation

The test regarding Instant Messaging was performed integrating a simple dummy service. This dummy service, which implements the ServiceConnector interface, invoked the PluginConnector's method sendMessage(). The message was sent to a peer dummy service that would signal when the message was received and what type of response was returned. A check on the message flow was done from the network as well, tracing messages from the IMS' OAM Network Monitor. The sending and receipt of an IM with any kind of attachment have been tested successfully. The second test, done on the IM service, was sending multiple IM, one after the other. The sending was performed successfully. The reception of multiple IM in rapid sequence has been critical. The receipt of one IM after another makes the CommonPlugin module lose the first IM. More tests were done on this aspect; these led to the conclusion that the problem arises when two IM arrive with less than one second gap between them. This restriction has been considered acceptable for the project.

7.2 Chat Evaluation

The test regarding the setup of a Chat session was done in a similar way. The tests involved basic tasks of the setup, exchange of messages and closure in a chat. This test was performed successfully; all transitions in the state machine were tested.

7.3 VoIP Evaluation

The test of the Voip call was also performed similarly. The test was conducted successfully, with the performing of multiple calls in sequence.

7.4 Presence Evaluation

The test on Presence was done with the implementation of two dummy services. Tests on the send of SIP SUBSCRIBEs were performed successfully. The test that involved the two dummy services was that of publishing the presence information of both, in sequence. This test had to demonstrate that the CommonFunctions implementation stored the presence information, regarding a service, correctly. The test was perfomed having service A registering, then service B registering, service A deregistering and service B deregistering. In the above case the register of service B did not deregister service A and the deregistering of service A did not deregister service B. This test was performed, with the addition of further tests on the same functionality. The Presence service works correctly.

7.5 Integration Tests

The integration of services was performed gradually. The first service that was successfully integrated is the ECity service that embodies the BuddyFinder and CarSharing services. The Fantacalcio service has been integrated in the second step and the Rolegames in the last. The problems that have arisen in the integration are of two types: integration problems (regarding interactions among single modules) and problems regarding the deployment on the PDA of the application. Both of these problems have been solved.
The Exploit client has been successfully tested and demonstrated in a trial demo.

8. Conclusions and Future Developements

The Exploit client, developed within the Exploit project, has been integrated in the Siemens Mobile's Mobilab IMS Experimental System and will soon be integrated in the IMS Experimental System of a MNO. The Exploit client shows the advantages the convergence to an all-IP UMTS network will bring. MNOs will benefit from these advantages, being able to deploy new services rapidly as well as users who will benefit of a wider offer of services. The evolution in IP terms needs updates in the network and in user terminals. Traditional cell phones are being replaced by smartphones; mobile phones with advanced processing capabilities and able to support multimedia applications.

The Exploit client is the example of what a UMTS client could look like in the near future. This client is not only able to perform IM, Chat, Presence and Voip, with multimedia content, but offers an open and flexible interface that permits the developer to build new services on it.

Further testing and development must be made in order to introduce this client on the market, supposing an IMS platform was available in a MNO's core network.

The testing should focus on the reliability of the available services and interfaces, stressing those that may be their weakness. In the testing chapter we met one of the weakness, the handling of close consecutive messages in reception. Further tests must be done, in order to make the client fault tolerant and be sure that all exceptions are handled correctly.

The development should focus on the portability to those platforms that will probably dominate the future terminal market. The portability on the Symbian platform should certainly be considered. In this port issues may come from the availability of the JVM on terminals, lack of memory and GUI handling.

Not all terminals are JVM enabled. Moreover, it is not still clear which type of JVM will dominate the smartphone market. The J2ME offers two configurations, the CDC (Connected Device Configuration) and the CLDC (Connected Limited Device Configuration), and multiple Profiles. The Exploit client is compliant with PersonalJava. Its port should be possible, without major changes, on the CDC configuration and Personal Profile.

Available smartphones still lack in memory and are not able to compete with a PDAs. This problem should be soon solved with the introduction of new models.

PersonalJava supports java AWT APIs. The available Symbian terminals offer the J2ME runtime environment, with the MIDP Profile. This would result in changing the GUI from a Window-like GUI to a GUI closer to that of cell phone. Issues may come in the port of the Siemens SIP API's on a J2ME powered device. Further tests and verifications must be done.

More work must be done on the audio quality, it can be improved with an ad-hoc solution. The introduction of video communication is a must on a UMTS terminal.