Technical Report SSC/1998/025. Available from http://sscwww.epfl.ch/

An Architecture for the Integration of Internet and Telecommunication Services

Constant Gbaguidi, Jean-Pierre Hubaux,
Institute for computer Communications and Applications (ICA)
Swiss Federal Institute of Technology
1015 Lausanne, Switzerland
{constant.gbaguidi, jean-pierre.hubaux}@epfl.ch

Giovanni Pacifici and Asser N. Tantawi
IBM Research Division
Thomas J. Watson Research Center
Yorktown Heights, NY 10598
{giovanni, tantawi}@us.ibm.com

Abstract

In this paper, we propose an architecture for *hybrid services*, i.e., services that span many network technologies, especially the PSTN and the Internet. These services will play an important role in the future, because they leverage on the existing infrastructures, rather than requiring brand-new and sophisticated mechanisms to be deployed. We explore a few issues related to hybrid services and propose a platform, as well as a set of components, to facilitate their creation and deployment. The existing infrastructure is only required to generate specific events when requests for hybrid services are detected. We present the design of a service layer, based on Java, that handles the treatment of these special requests. Our service layer is provided with a set of generic components realized as Java Beans. Hence, we can provide hybrid services without changing the existing infrastructure. We illustrate this strength of our architecture by discussing the call forwarding service.

1 Introduction

The recent growth in the number of Internet users, coupled with the strong foothold of the Public Switched Telephone Network (PSTN), is creating a demand for a new class of PSTN and Internet integrated services that can take advantage of both technologies simultaneously. We refer to this new class of services as *hybrid services*. Examples of hybrid services are: Click-to-Dial [Fay97a] (which enables the user to request, from a Web browser, a connection to be set up between two telephone sets connected to the PSTN); voice over PSTN and Internet interconnected by H.323 gateways; universal messaging; and access to both electronic-mail and voice-mail either from the Internet or the PSTN.

The demand for hybrid services is further fostered by the anemic market penetration of broadband integrated networks based on Asynchronous Transfer Mode (ATM) technology, as well as by the fact that cellular networks are already well integrated with the PSTN. This makes purely Internet-based solutions, relying on Internet telephony for voice, impractical. Taken separately, the PSTN and Internet are far from being an ideal ground for developing future hybrid services; however, if coupled together they can complement each other effectively.

The PSTN (including its cellular extension) provides an extremely reliable, available and ubiquitous system, with guaranteed Quality of Service (QoS), and a business that commands $600 billion in total annual revenues (two orders of magnitude higher than the one of the Internet). The PSTN is endowed with a powerful service creation and provision platform called the Intelligent Network (IN) [Q.1211][Fay96][MPZ96]. The design of the IN followed a simple principle: separation of service-specific software from basic call processing. Before the advent of the IN, services were incorporated in the network switches in a manner that was specific to each manufacturer. Therefore, introducing new services required the modification of software in each and every switch in the network. Such a process took years to complete and made network operators heavily dependent on their equipment suppliers. The IN reduced a great deal of this dependency by moving service-specific software into specialized nodes called Service Control Points (SCPs). Basic call processing is performed in the switches, named Service Switching Points (SSP) in IN parlance. The communication between SSPs and SCPs is done through a channel called Common Channel Signaling No. 7 (CCS7) [Mod90]. The advent of the IN reduced the time frame of the introduction of a new service to a few months. The success of IN concepts steered their application in the mobile communications environment; examples of applications are Wireless IN (WIN) [Fay97b] and Customized Applications for Mobile network Enhanced Logic (CAMEL) [GSM 02.78].

The Internet is characterized by a devolved service architecture, i.e., there is no global service creation and provision framework. New services can be created by any user that can afford a server. Creating new services implies developing a distributed application that must be installed and executed in the terminals and servers. Internet applications take advantage of intelligent terminals and powerful user interfaces. However, some services (and even basic mechanisms) of the Internet require specialized servers; the most important of such servers is the Domain Name Server (DNS).

Hybrid services are expected to play a very important role in the years to come. This is due to both the desire of users to integrate the ways they communicate (a promise made more than 20 years ago by the Integrated Services Digital Network, ISDN, but not fulfilled yet) and the willingness of service providers (Internet Service Providers, ISP, Internet Telephony Service Providers, ITSP, and Telcos) to differentiate their offers from their competitors'. Last but not least, smart cellular phones (featuring the functions of both a phone and a small laptop) are expected to fuel the integration of services.

In the recent past, new research activities that focus on hybrid services have emerged [TAPI3.0, Fay97a, Geoplex]. So far, these efforts have focused mainly on *inter-working* rather than *integration*. The common approach taken by these activities is to model the PSTN (or the Internet) as a stand-alone system whose services can be accessed through a gateway that acts like a PSTN (or Internet) terminal. While these new activities are leading to the development of new services, such as Click-to-Dial, we are still far from the true integration that will allow service providers to rapidly create and deploy services that can take advantage, simultaneously, of all existing delivery and access systems.

In this paper, we present a new approach to hybrid service creation in a heterogeneous environment that spans the PSTN and the Internet. We propose an architecture, as well as an early set of *service components*, that allows simple and rapid creation and deployment of integrated telecommunication and Internet services. A service component defines a set of pre-developed service elements that can be used to assemble a distributed service. These service components are run on terminals, network elements, as well as servers and peripherals; they therefore provide an openly programmable service environment. The proposed service components are implemented as a set of JavaBeans. Using Java technology allows service developers to write applications in a single language that can be executed without modification on any equipment, while the JavaBeans technology is used to implement the *service components*. In our model, these service components are provided by vendors of network equipment, servers, peripherals and terminals. Therefore, the service developer needs only to customize the service logic and integrate it with the service components.

This paper is organized as follows. In Section 2, we review the state of the art in the Internet and the PSTN. In Section 3, we describe the concepts of hybrid and teleinformation services according to our framework. In Section 4, we present our service platform architecture, as well as a set of generic service components. In Section 5, we discuss an example of hybrid service that can be constructed using the framework and service components described in Section 4. Finally, in Section 6 we summarize the main contributions of this paper and discuss some remaining issues.

2 Internet and PSTN service integration: state of the art

In the recent past, we have witnessed research efforts geared towards the development of services that can span both the PSTN and the Internet. We classify the efforts relevant to this paper into four categories: enabling technologies for interoperability at the signaling and control level, efforts related to the access to telephony services from a computer, works on the access to the IN infrastructure from the Internet, and enablers of distributed computing. Highlight representatives (H.323, Computer-Telephony Integration, PSTN/Internet inter-working, and CORBA, respectively) of the four categories are presented below.

2.1 Interoperability among network technologies with and without QoS guarantees : H.323

Networks over which hybrid services are deployed use differing technologies; specifically, some of them can embed mechanisms for delivering the QoS requested by the user, while others may not. Making such networks interoperate is one important issue involved in providing hybrid services. H.323 [H.323][Thom96] is an ITU-T standard that solves much of this problem. Its primary goal is to enable audiovisual interactions among terminals situated in various environments, i.e., Local Area Networks (LANs) with their many technologies (which may or may not guarantee QoS) and telecommunication networks (PSTN, ISDN). The key H.323 components that are important to the scope of our work are the *gateway* and the *gatekeeper*. An H.323 gateway is in charge of translating call signaling, control messages, and multiplexing techniques across the network technologies involved in the call. While the gateway is an optional element in an H.323 system, its absence, however, undermines interoperability among differing network technologies. An H.323 gatekeeper performs network administration, bandwidth re-negotiation and address translation. H.323 terminals must get permission from the gatekeeper before they can place or accept a call. A gatekeeper is not required in an H.323 system ; however, its presence empowers the network administrator with much control on the network, and provides users with the possibility to define alias addresses that are independent of the network addresses.

2.2 Computer-Telephony Integration (CTI): TAPI and JTAPI Approaches

CTI is the main scheme for the integration of computer networking and Telecommunications. The foundations for this integration were laid by the European Computer Manufacturers Association (ECMA) in the technical report ECMA TR/52 [ECMA] on the provision of Computer-Supported Telecommunications Applications (CSTA) [Ans96][Dho96]. Many implementations were carried out for CSTA : Microsoft’s Telephony Application Programming Interface (TAPI) [TAPI3.0], Novell’s Telephony Services Application Programming Interface (TSAPI) [Cro96] and Sun’s XTL [SunXTL]. In this section, we focus on TAPI 3.0, because it has already incorporated H.323, which enables services to span several network technologies. When using TAPI, an application can generate calls that will actually go through many different network technologies (because of the use of H.323) ; this makes TAPI a major player at the hybrid service provision marketplace. Most of the other telephony APIs consider only the Internet and the telephone network.

Despite its wealth of relevant features, TAPI suffers from an important limitation: its tight relation to Microsoft Windows platforms. Relief is supplied by a solution based on the Java language [Fla96]: Java Telephony API (JTAPI) [JTAPI97]. Java is a platform-neutral language, i.e., a program written in Java is portable, as is, onto any platform that runs a Java Virtual Machine (JVM); the JVM translates the program into platform-specific code [Fla96]. JTAPI shields applications from the specifics of the used platforms (Fig. 1). It can be used to access TAPI functionalities when the host runs Microsoft Windows, or XTL functionalities on a Sun machine. It is made up of a core package which provides the basic framework for modeling telephone calls and rudimentary features (e.g., placing, answering, and dropping a telephone call). This package is composed of objects which define the JTAPI call model : *Provider* (of the telephony service), *Call*, *Address*, *Connection*, *Terminal*, and *TerminalConnection* objects.

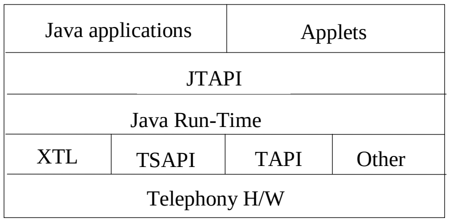

Fig. 1. Typical stack using JTAPI.

More interestingly, JTAPI can be used with dumb terminals such as Network Computers (NCs). In this case, much of the processing is uploaded to a network-based proxy. Therefore, we can use JTAPI for dumb mobile terminals as well. There is, however, another scheme that can be used to perform the uploading: it is a script language based on the Wireless Markup Language (WML) [WML98], which is an adapted version of HTML to the wireless environment. This script language, called WMLScript, is much simpler than JavaScript (script language associated with Java). A major drawback of using WMLScript for hybrid services is that it is too tightly related to the wireless environment and is still less known in the wide communications community. It is based on tools that currently have little acceptance. For now, we promote the use of JTAPI, essentially because of its completeness.

2.3 PINT: PSTN and Internet Inter-working

A working group, called PSTN/Internet Interworking (PINT) group, was created at the IETF in 1997 with the objective of opening up the IN architecture to user requests issued from the Internet [Fay97a][Kris97]. In this scenario, the user makes the call request from the Internet (the Web) and later communicates through the PSTN. The reference model laid out by the PINT group is depicted in Fig. 2. Requests received from the user on the Web are sent to a Web server that processes and forwards them to the gateway between the Internet and the PSTN. This gateway then communicates with legacy IN components such as the Service Node (SN), the SSP (both the mobile switching center and the classical telephone central office), the SCP, and the Service Management System (SMS). An SN can be regarded as a hybrid component that provides, among other activities, the combined functionality of an SSP and an SCP. The SMS is a system that embeds the functions necessary for the management of the IN infrastructure.

There is another trend, toward the interweaving of the IN and the Internet, where nearly the entire IN control system runs on the Internet [Low97]. In this trend, the role of the SCP is reduced to finding a Web server that contains the logic and data to be used for the services, unlike in the conventional IN system where the whole service software is controlled by the SCP. By running much of the IN control system on the Internet, service providers can enable the customers to create their services as easily as create their Web pages. Interoperability, among IN system providers, can also be achieved [CNIS].

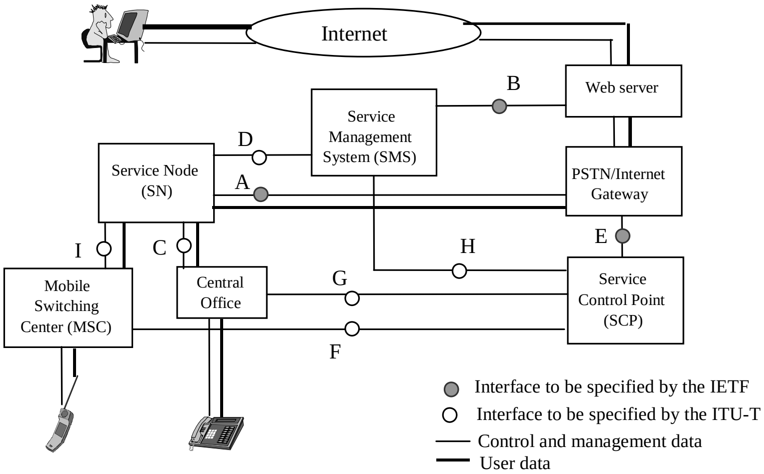

Fig. 2. The PINT reference model.

2.4 Distributed Computing and Telecommunications Services

The part of telecommunication services that is implemented in software has been increasing ever since the advent of the IN, which is one of first architectures that viewed services as interactions among software components. Therefore, the design of telecommunications has been influenced by developments in software engineering. Two main trends have emerged during the past years : object-orientation and distributed computing. These trends are inter-related: Distributed computing can take advantage of relevant properties of object-orientation such as encapsulation, which is highly desirable in a distributed environment. Approaches that integrate the two of the aforementioned trends are Open Distributed Processing (ODP) [X.901] and the Common Object Request Broker Architecture (CORBA) [CORBA]. Their aim is to overcome platform and language discrepancies; an object is only required to define an interface in order to be invoked by other objects; this interface depicts the operations, notifications and streams that the object can handle, as well as its attributes. The interface is declared according to a language called Interface Definition Language (IDL). The choice of the programming languages used for implementing the interface is left to the developer.

ODP and CORBA paved the road to a new way of creating distributed services that span multiple platforms. This new way is exemplified by the Telecommunications Information Networking Architecture (TINA) [TINA] and the extended binding model (XBIND) [Lazar]. The idea behind these approaches is to build an openly programmable environment for service creation. The focus of these activities is on QoS control and resource allocation. Both approaches have found limited applicability because they are both strongly tied to ATM technology which, so far, has had little acceptance as an access network. A discussion of the strengths and weaknesses of TINA can be found in [CNIS].

The CORBA technology is also being considered for Intelligent Networks as well as Wireless access and terminal mobility. In the Intelligent Network, current efforts focus on how CORBA could effectively interwork with SS7. A first approach consists in providing gateways between the SS7 network and CORBA-based systems; a rather simple solution with arguable scalability. Anothe solution would be to make use of the SS7 infrastructure as the interoperability protocol between "islands" of CORBA-based telecommunications equipment [CORBA-IN]. Wireless networking and terminal mobility are a formidable challenge for large object-based middlewares such as CORBA. The problems range from the ability to cope with the characteristics of a radio channel (limited bandwidth, non-permanent connectivity) to the capacity to take mobility mechanisms into account (hand-over, terminal tracking, etc.) [CORBA-MOB]. More generally, there are some concerns about the ability of CORBA-based-products to fulfill the requirements of the telecommunication systems, when it comes to performance and scalability.

It is worthwhile noticing that the many implementations of CORBA currently have limited interoperability. This present-day weakness restricts the ability of CORBA to patch computing platforms together. Moreover, most CORBA implementations are unavailable for free. This situation impacts the development of CORBA, compared to the fast-evolving pace of Java, which is onstantly enriched with new features in an open way.

In this Section, we reviewed the main enabling technologies that we will need in the construction of our platform for hybrid services. These services cover a wide range of the service spectrum. In the following Section, we define the terminology, as well as the categories of services that are of interest to us in the remainder of this paper.

3 Service Concept

To set the stage for the discussions to follow, we briefly review in this section the many meanings of the word *service* in different communities. Then we propose a more rigorous definition. In particular we define *teleinformation service* and *hybrid service*.

So far, the word *service* has received many definitions in the literature. In the area of communication protocols, a service usually designates the set of primitives that a given protocol layer provides to the upper protocol layer. In the narrowband ISDN [I.210], the standardization identifies *bearer services* (which provide the capability for information transfer between two network access points), *teleservices* (which supply full capability for communication between user end-systems) and *supplementary services* (which supplement teleservices by providing additional value to the customer ; call forwarding and call screening are examples of supplementary services). In the Internet, the concept of service is usually related to the *end-to-end quality of service*; consequently the word *service* is ubiquitous in all discussions on reservation strategies (integrated services [Brad94], differentiated services [Nic97], etc.).

We define a *teleinformation service*[1] as a contractual relationship between two entities: the *service provider* and the *service user*. By contract, the provider commits to providing facilities, based on a communication network, with a given level of QoS. A service must involve : a terminal used by the user ; a server or several other terminals ; an appropriate set of applications running on these servers and terminals ; one or several networks providing connectivity between servers and terminals (these networks might not belong to the service provider).

The most traditional contractual relationship is the subscription for a telephone line. It should be noted, however, that a contract is not necessarily explicit, nor does it necessarily require a transfer of money from the client to the operator. For example, the activation of a search engine from a Web browser is considered to be free of charge; there is, however, an implicit contract: The service provider commits to give an ordered list of the best matching hits (based on the accuracy of its Web crawlers); in return, the user accepts to be exposed, for example to some advertising animation. Users also accept that their data be recorded for further use in data-mining or similar applications (frequently for the purpose of selective marketing). Finally, both the provider and the user implicitly agree that they will not take advantage of the connection to undertake harmful operations such as the introduction of a virus in the other party's computing infrastructure.

Hence, a teleinformation service is what the service provider sells to its users. This idea is very intuitive in the area of telecommunications; in fact, both the teleservices and the complementary services defined for

---

[1] We coined the term "teleinformation" in order to express the fact that these services consist in either performing telecommunications with remote users or retrieving information from servers.

the ISDN are subsets of teleinformation services. In the area of the Internet, this concept has hardly been investigated so far, because all use of the network was thought to be free.

A *hybrid service* is a teleinformation service that relies both on the Internet and on the PSTN for its execution and delivery. Examples of hybrid services are teleshopping sessions and hybrid access to email and voicemail. In a teleshopping session, a user, after browsing the Web site of a given retailer, interacts with a salesperson to purchase the selected good. Hybrid access to email and voicemail allows a user to either access Internet email from a PSTN phone by means of voice synthesis, or retrieve and play back voice messages using a computer connected to the Internet.

4 Platform Architecture

The process of implementing a service within a system infrastructure involves a number of steps, including creation, design, development, testing, operation, provision, management, and termination. We call this process *service engineering*. Such a process requires a platform, or an environment that enjoys the following features:

- independence of the service from the underlying operating systems;
- a set of reusable components for the creation of new services;
- efficient mapping of the service components onto the system infrastructure;
- a graphical and easy-to-use interface to create, manage and operate services;
- generality and expandability to allow for the creation of novel services.

Our focus in the remainder of the paper will be on the determination of generic service components which, put together, yield a wealth of hybrid services. In the following sections, we present our methodology toward the construction of a platform that exhibits the aforementioned features. In section 4.1, we describe the service platform and its main underlying principles and concepts. Because it is unrealistic to cover all sorts of hybrid services, we focus on a specific family, i.e., real-time communication services (section 4.2). We describe a generic topology of the system infrastructure for this family of hybrid services. A middleware that enables the programmability of this infrastructure is presented in section 4.3.

4.1 Service Platform

As mentioned in Section 1, the most popular service architecture for the PSTN is the IN. It is provided with a Service Creation Environment (SCE) in which a service is created out of a set of standardized service-independent building blocks (SIBs). Although the concept of SIB has been removed from the standard in the Capability Set 3, it is still used by the vendors. Once a service is built as a chain of SIBs, the service logic is spread out into components that are uploaded into network and control nodes, such as SCPs and SSPs.

In a similar manner, we envision an SCE that uploads service components into terminals and servers; in some cases, components must also be uploaded into network nodes or into specialized elements such as gateways (cf. Section 5). Our proposal goes far beyond the IN SCE. In particular, it supports the upload of specific *protocols* into the elements that require specific communication mechanisms. As depicted in Fig. 3, three services (S1, S2, and S3) are provided on a system consisting of:

- Terminals (e.g., telephones, mobile phones, Personal Digital Assistants, Personal Computers, and embedded devices)
- Network Nodes (e.g., switches, routers, Mobile-services Switching Centers, and satellites)
- Information Servers (e.g., Web servers and mail servers)
- Control Servers (e.g., SCPs).

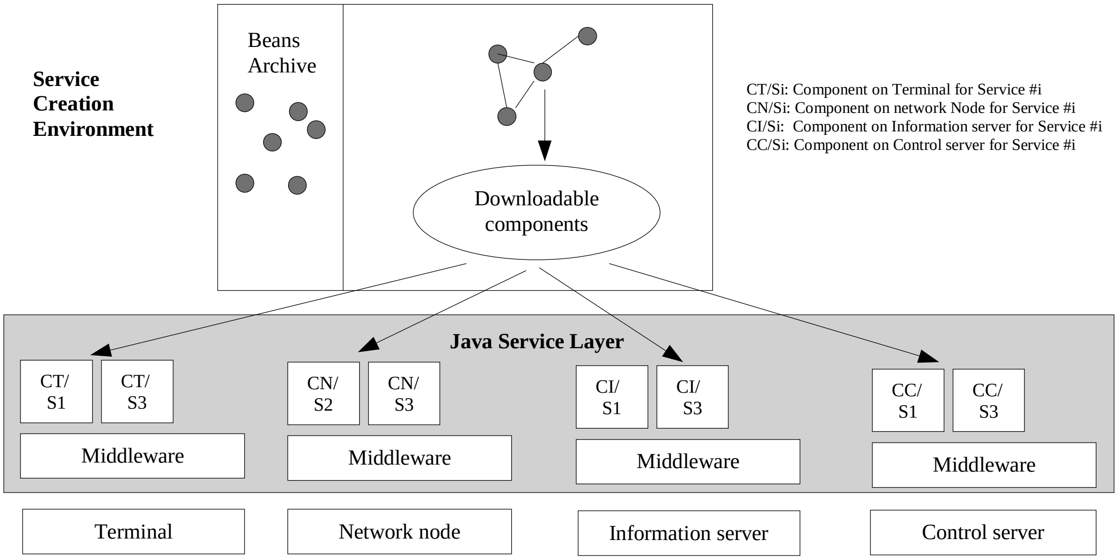

Fig. 3. The proposed service architecture.

For the sake of conciseness, inter-network elements (such as gateways) are not depicted in Fig. 3. The SCE converts a service into components to be uploaded into the system infrastructure. For example, service S1 is broken down into two components: CT/S1 in the terminal and CI/S1 in the information server. The *middleware* layer provides the binding and integration of the various components. It also provides vertical independence in such a way that common components are uploaded into heterogeneous elements. The middleware, together with the uploaded components, defines the *Java Service Layer*.

It has to be stressed that the software installed on the terminal is not fully under the control of the service provider; indeed, the end-user is very often entitled to select and install the application he prefers. For example, he is usually free to choose the Web browser; in this case, the service provider would still have the possibility to impose the downloading of appropriate applets at the beginning of the session.

The system infrastructure in Fig. 3 exhibits a great deal of heterogeneity in terms of both hardware and software interfaces to the system elements. Therefore, we need to define a common, ubiquitous interface for the downloaded components. One candidate for such an interface is the Java language and its extensions. Basically, a computing device that has a Java virtual machine can host a Java application. This observation is valid for both the data networking and Telecommunications environments. This is obvious for the data networking environment. As for the Telecommunications environment, intelligent nodes and terminals are expected to become more open and provide a Java interface, such as the JAIN extension [JAIN] for IN SCPs. We decided to choose the Java interface for our platform for the following reasons:

1. Ubiquity: Most terminals, even smart phones and embedded devices, can now be provided with a Java virtual machine; this trend is also emerging for devices located in the PSTN such as SCPs, SSPs and SNs

2. Power: Applications written in Java can be run on any platform that has a JVM installed; Java actually makes the phrase “Write once, execute everywhere” a reality

3. Extensibility: Many capabilities are being added to the core Java specification. In particular, a component architecture (i.e., JavaBeans [Jub98]) has been developed to allow programmers to re-use previously written Beans. Beans are Java classes written according to a number of rules that especially facilitate their graphical manipulation and inspection. They make extensive use of events to

communicate with one another. The Bean Development Kit (BDK) [BDK] provides a user-friendly environment for the manipulation of Beans.

The envisioned SCE provides a set of Beans that can be put together to create new services. It then converts the Beans into a set of components that are sent to the Java Service Layer. This layer maps the components onto system elements. The mapping depends on the capabilities of the elements; components may reside physically in existing elements, or they may reside in new physical elements. It is worth noting that Beans are used in the SCE, while the Java Service Layer manipulates service components. A Java virtual machine runs within the middleware and is used all over the service lifetime.

4.2 Real-Time Communication Hybrid Service

In order to illustrate the concepts introduced above, we focus in the sequel of this paper on a subset of hybrid services: hybrid services that provide interactive communications. By interactive communication, we mean a communication between two or more end-users at the same time; examples are telephony, videoconference and chat. This subset has been chosen because it is at the convergence point of Telecommunications and Internet services; this focus makes it possible to define precise Beans in the remainder of the paper.

From a platform perspective, the system infrastructure that realizes these services is represented in terms of objects. Such objects are integrated to form components that the service engineer uses to create the service. First, we present the abstraction of the hybrid system infrastructure. Then, in the next sections, we create the object model and the corresponding components.

Fig. 4 illustrates an abstraction of a system that can provide hybrid real-time communication services. We consider three planes: users, end-systems, and networks. The user plane contains user objects, abstracting over real users of the service. Users may be either at fixed locations or mobile (an example of user mobility is Universal Personal Telecommunications, UPT [F.851]). Users interact with the system through terminals that are abstracted as *Terminal* objects in the end-systems plane. Examples of terminals are telephones, portable phones, mobile terminals, PDAs, and PCs. The characteristics of the terminal form the attributes of the *Terminal* object. Terminals are connected to the system through Network Access Points (NAP) as shown in the networks plane. NAPs may belong to different networks, thus forming a hybrid environment. Examples of NAPs are the port on the subscriber board of a local exchange, the wireless network interface at the Base Station, and the port on an access router. A network provides connectivity[2] among the NAPs that belong to it. A gateway connects NAPs from different network technologies; it acts like an end-system to both networks. The gateway device is abstracted in the end-systems plane and provides all necessary conversions between the two networks.

The splitting of the connection between the terminal and the gateway in three parts may look superfluous at first. The rationale behind this is that we consider the general case where the terminal can move *during* an on-going session. In this case, the NAP will have to change (hand-over mechanism). Note that a gateway can be mobile, as well.

---

[2] In the sequel, we will use the term “NAP-to-NAP Connection” to designate the way connectivity is provided between two NAPs; in the case of the Internet, this is motivated by the fact that the kind of services we consider here requires resource reservation strategies to be deployed.

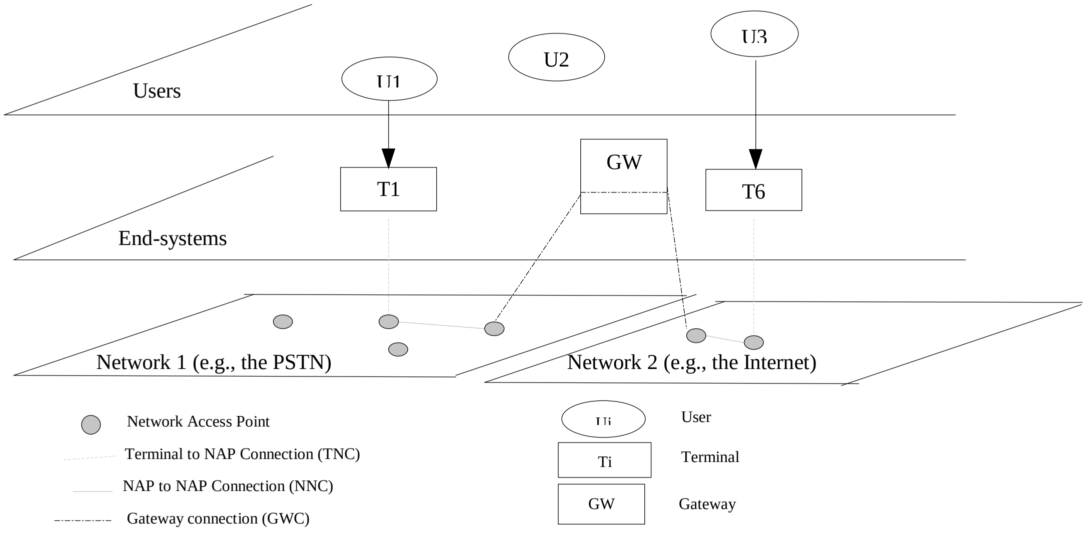

Fig. 4. System abstraction for hybrid real-time communication services.

The object model corresponding to the above hybrid system abstraction is depicted in Fig. 5. The user makes use of a terminal to access services. The terminal accesses the network through a terminal-to-NAP connection. Access points are linked to one another by either a NAP-to-NAP connection (when the two access points relate to homogeneous network elements), or a gateway connection (connection that goes through a gateway). A gateway connection (GWConnection) is made up of two GW-to-NAP connections and one connection binding. The connection binding embodies the interconnection performed by the gateway. In a programmable platform, the gateway can receive instructions from the Java Service Layer to bind two connection segments on both sides of it.

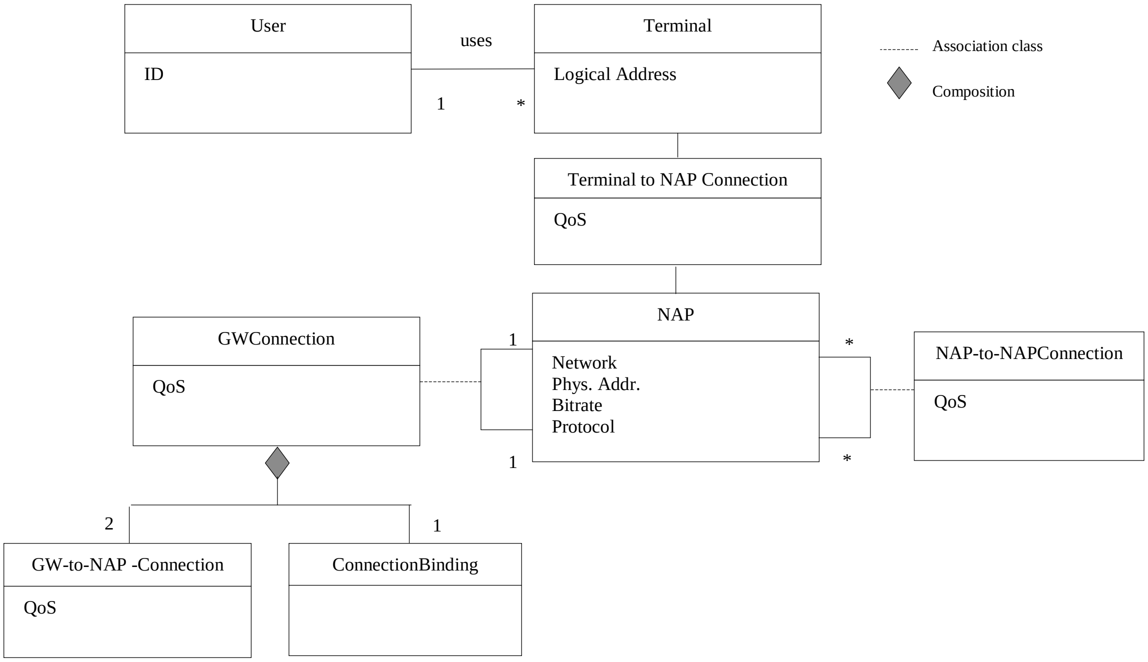

Fig. 5. Object Model.

4.3 Middleware for the Programmability of the Hybrid Service Platform

The components of the architecture described in section 4.2 need to be programmed in order to introduce new services. In this Section, we address the programmability of these components, by leveraging on the existing equipment, i.e., we do not intend to create new equipment; we rather enhance the existing equipment in order to facilitate the programmability of the service platform.

In section 4.3.1, we present a generic information flow diagram that describes the processing of information within and across the main nodes of the platform. In sections 4.3.2 and 4.3.3, we describe relevant events and service components necessary to the provision of hybrid services.

4.3.1 Generic information flow diagram

In the vision that we are developing in this paper, the introduction of new services must be smooth, in order to minimize the upgrade of the existing service platform elements. This principle can be neatly implemented by using triggers. The existing element, after detecting a special trigger, calls the Java Service Layer that embeds the logic according to which service requests must be processed. As depicted in Fig. 6, we suggest implementing a process, in the elements, to detect specific patterns in the service request and pass the treatment on to listeners located in the Java Service Layer. Note that this mode of operation fits well with the way the IN runs : The SSP detects special patterns and then calls the SCP in order to get instructions on how to process the service request. As we will see, the model also nicely fits with Internet-based mechanisms.

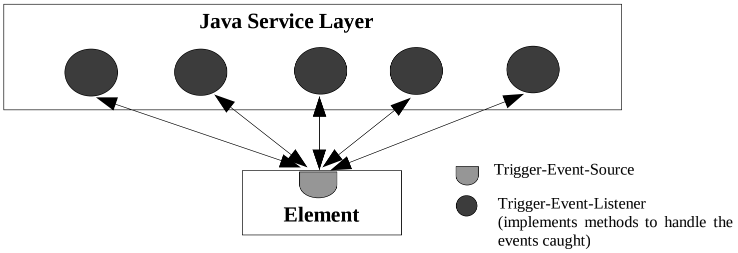

Fig. 6. Trigger-Event-Source and Trigger-Event-Listener model for the platform elements.

We expand further the above simple event source and listener model. A source (`TriggerEventSource`) throws a trigger event which is eventually caught by listeners (Fig. 7). Each trigger is characterized by a pattern (e.g., 1-800, or a full address). When a listener catches a trigger event (`TriggerEvent`), it compares the trigger's pattern with its own attributes. If the pattern matches these attributes, then the listener is authorized to handle the event. This handling includes the service logic and may require a connection to be set up between two endpoints (or more). The listener cannot fire a connection attempt event (`CnxAttemptEvent`) towards any connection object, since the connection that the listener wishes to create does not exist yet. We suggest that the connection attempt event be issued to a connection factory (`ConnectionFactory`), which eventually creates a connection object. The created object fires either `CnxCompleteEvent` (connection complete event) or `CnxFailEvent` (connection fail event), which are propagated down to the trigger event listener and the trigger event source. For the sake of conciseness, this event propagation is not fully depicted in Fig. 7. In the case of a gateway, a binding may be needed between the connection segments on the two networks interconnected by the gateway. The connection factory is then instructed by the trigger event listener to realize the binding.

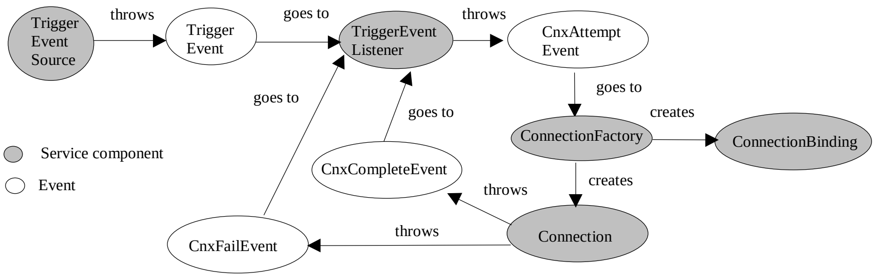

Fig. 7. **A generic information flow diagram for real-time hybrid services.**

The information flow diagram depicts the creation of the topology (Fig. 4 and Fig. 5) used for the provision of hybrid services. Indeed, the figures mentioned essentially reflect the connections among the main components of the architecture. The information flow diagram therefore is an evolution of the work presented in section 4.2 toward concrete programmability. In the next sections, we describe the main events and service components identified in Fig. 7.

4.3.2 Events needed

The main events identified from Fig. 7 are:

- `TriggerEvent`
- `CnxAttemptEvent`
- `CnxCompleteEvent`
- `CnxFailEvent`

Note that all event names have `*Event` as suffix.
We focus on the description of `TriggerEvent`, which is the most illustrative event of the proposed diagram.

`TriggerEvent` is thrown by `TriggerEventSource` when an address with a specific pattern is detected. Its main attributes are (Fig. 8):

- `originatingNetwork` indicates the network (Internet, PSTN, etc.) from where the service request comes
- `trigSrcAddr` indicates the source of the service request (e.g., an SSP address)
- `trigPattern` indicates the pattern whose detection triggered the event.

The `Address` component represents an access point to a resource. Its main attributes are :

- `relatedLayer` tells the layer to which the address relates (e.g., an Internet address relates to layer 3)
- `typeOfNetwork` indicates the type of network that the address relates to
- `addrString` gives the address in string form.

```
public class TriggerEvent extends java.util.EventObject {
        String originatingNetwork ; /* Network where the request comes from */
        Address trigSrcAddress ; /* Address of the place from where the
                                      request comes */
        String trigPattern ; /* Pattern that characterizes the trigger,
                              e.g., 1-800 */

        public TriggerEvent(boolean incoming, Address srcAddr, String pattern) ;
        public ~TriggerEvent() ;
} ;

public class Address{
        int relatedLayer ;
        String typeOfNetwork ;
        String addrString ;

        public Address(int layer, String netType, String addr)
        public ~Address() ;
} ;
```

Fig. 8. Definition of the event `TriggerEvent`.

4.3.3 Service Components needed

The main service component object classes (Fig. 7) are :

- `TriggerEventSource`
- `TriggerEventListener`, with an important subclass called `HybridService`
- `ConnectionFactory`
- `Connection`, with many subclasses, especially `ConnectionBinding`.

The first two components relate to the production and treatment of the event `TriggerEvent`. They are described in Fig. 9. `TriggerEventSource` abstracts over the producer of the trigger event. It implements methods that allow listeners of trigger events to register with it, as well as methods for adding or removing trigger events.

`TriggerEventListener` registers itself with `TriggerEventSource` and listens to `TriggerEvent`. When this event is fired, `TriggerEventListener` catches it ; it then determines whether it is allowed to handle the event by comparing the attributes of the event with its own attributes. Specifically, the `TriggerEvent`'s attribute `trigPattern` is compared with the `TriggerEventListener`'s attribute `patternToCatch`. `TriggerEventListener` implements a method (`trigEventHandler`) to handle the event caught. It can throw an event called `EvtHandlingFailEvent` when it fails to handle a trigger event previously caught. The event `CnxAttemptEvent` can be thrown as well, if the service logic requires a connection to be set up between two network accesses.

`HybridService` is a special subclass of `TriggerEventListener ;` it abstracts over the logic that the processing of the service request must follow. The logic is called after a trigger event has been fired ; it is a listener of a specific trigger event. It can aggregate some other components such as a component for address binding (to be used for address translation, for instance), or groups of terminals (to form a closed group or a Virtual Private Network, VPN). The detail of these components is left for further study. The attributes of `HybridService` are : (1) `servDescr`, a textual description of the service, (2) `customer`, the name of the customer for whom this service was created, and (3) `customerPermanentAddr`, the permanent address of the customer.

The component `HybridService` is to be specialized further (as illustrated in Section 5) in order to take into account the specifics of each service. Hence, there will be specializations for call centers, VPNs, and so forth. In Section 5, this specialization is done for Call Forwarding.

```
public class TriggerEventSource{
       public void addTriggerEventListener(TriggerEventListener refToListener) ;
       public void removeTriggerEventListener(TriggerEventListener refToListener) ;
       public void addTrigger(TriggerEvent trigEvt) ;
       public void removeTrigger(TriggerEvent trigEvt) ;
       public void getListOfListeners(TriggerEventListener listener[]) ;
} ;

public class TriggerEventListener extends java.util.EventListener {
       String eventOrigNet ; /* Origin network from where the event is to come
                                    (e.g., PSTN, Internet) */
       String    patternToCatch ; /* Pattern that the listener intends to catch */
       public      void      triggerEventHandler(TriggerEvent      trigEvt) throws
                    EvtHandlingFailEvent, EvtHandlingSucceedEvent, CnxAttemptEvent;
              /* Handler of the event to be caught */
} ;

public class HybridService extends TriggerEventListener {
       String servDescr ;
       String customer ;
       Address customerPermanentAddr ;

       public HybridService(String descr, String customer, Address custAddr) ;
       public ~HybridService() ;
} ;
```

Fig. 9. Definition of the components `TriggerEventSource`, `TriggerEventListener` **and** `HybridService`.

The second category of components depicted in Fig. 7 concerns the connection control. This category is represented in Fig. 10. `ConnectionFactory` manages the connections set up. It catches connection attempt events (`CnxAttemptEvent`) ; it must therefore implement a listener for these events. Its main attributes are the number of Internet (or PSTN) connections set up (`numberOfIntCnxUp` and `numberOfPstnCnxUp`), and the number of bindings between connections. A special binding is between a PSTN connection segment and an Internet connection segment; such a binding is needed in a gateway, for instance. `ConnectionFactory` implements a method to handle connection attempt events.

`Connection` abstracts over a physical or logical link between two access points that might belong to a network element or a terminal. Its main attributes are:

- `status` tells the current state of the connection
- `typeOfNetwork` tells the type of network in which the connection lies
- `from` indicates the address of the connection source
- `origPort` denotes the port used by the source to send data (e.g., socket port)
- `to` indicates the terminating point of the connection
- `termPort` denotes the port used by the terminating point to receive data
- `directionality` is an integer attribute that evaluates to 1 for one-way connections and 2 for bi-directional ones
- `peakRate` denotes the peak rate of the traffic carried out over the connection
- `avgRate` denotes the average rate of this traffic
- `delay` indicates the tolerable delay.

`Connection` implements a constructor, a destructor and methods to get its attribute values. It can produce the following events : (1) `CnxCompleteEvent`, when the connection set up has been successful, (2) `CnxFailEvent`, when the connection could not be established (the reasons are then given), and (3) `CnxDumpedEvent`, when the connection has been dumped due, e.g., to a user hanging up.

```
public class ConnectionFactory implements CnxAttemptEventListener {
        int numberOfIntCnxUp ; /* number of Internet connections up */
        int numberOfPstnCnxUp ; /* number of PSTN connection segments up */
        int numberOfBindingUp ; /* number of bindings up */

        public void gwCnxAttemptHandler(CnxAttemptEvent attempt) ;
                        /* Handles the event CnxAttemptEvent */
} ;

public class Connection {
        int status ;
        String typeOfNetwork ;
        Address from ;
        String origPort ;
        Address to ;
        String termPort ;
        int directionality ;

        int peakRate ;
        int avgRate ;
        int delay ;

        public Connection(String typeOfNet, Address from, Address to, int peakRate,
                int avgRate, int delay) throws CnxCompleteEvent, CnxFailEvent ;
        public ~Connection() ;
        public int getStatus() ;
        public int getPeakRate() ;
        public int getAvgRate() ;
        private void connectionLife() throws CnxDumpedEvent ;
} ;

public class ConnectionBinding extends Connection implements CnxDumpedListener{
        int status ;
        Connection cnx1 ; /* First connection to bind */
        Connection cnx2 ; /* Second connection  */

        public ConnectionBinding(Connection cnx1, Connection cnx2)
        throws CnxBindingCompleteEvent, CnxBindingFailEvent;
        public ~ConnectionBinding() ;
        public void cnxDumpedHandler(CnxDumpedEvent dumpedEvt)
                        throws CnxBindingDumpedEvent;
        public int getStatus() ;
} ;
```

Fig. 10. Definition of the components `ConnectionFactory`, `Connection` **and** `ConnectionBinding`.

There are several subclasses of the generic component `Connection` described above. Fig. 11 depicts the inheritance tree related to this component in the light of Fig. 5. The main subclasses are `NAP-to-NAPConnection`, `Termination-to-NAPConnection`, and `ConnectionBinding`. `Termination-to-NAPConnection` further has `Terminal-to-NAPConnection` and `GW-to-NAPConnection` as subclasses. Except for `ConnectionBinding`, all the other subclasses are mainly discriminated by their specific implementations; their declaration is similar to that of `Connection`.

`ConnectionBinding` is used to bind a segment of connection to another one. A special case is a binding between a connection segment in the Internet and another one in the PSTN; this happens typically in a gateway. In this case, `ConnectionBinding` abstracts over a special connection that entirely lies within the gateway. Its main attributes are its status and references to the connections bound. `ConnectionBinding` implements methods that allow for its creation and deletion, as well as a method that handles connection breaks. For instance, when the connection segment on the Internet has been dumped, its sibling in the PSTN must be released. An event, `CnxBindingDumpedEvent`, is sent to the connection factory. This management operation is performed by `cnxDumpedHandler`.

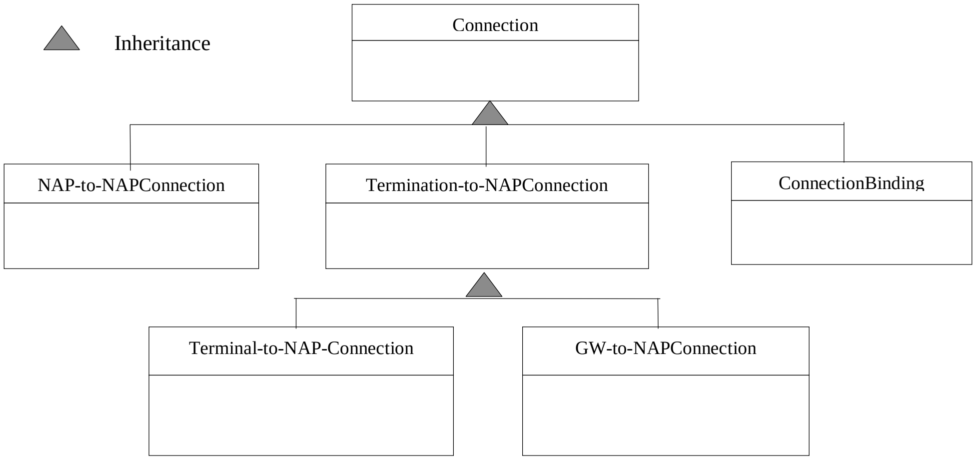

Fig. 11. `Connection` **class hierarchy.**

5 Case Study: Call Forwarding Service across the PSTN and the Internet

In this section we illustrate how the service architecture presented above can be applied to the task of building a service that spans across the PSTN and the Internet. In section 5.1 we present the service principles as well as the specialized component classes necessary to provide the hybrid service. In order to keep our discussion focused we refer to a specific example (i.e., call forwarding) that allows users to forward calls from a PSTN terminal to an Internet-connected H.323 terminal. It should be noted that the call forwarding service presented in this section could be realized in many other ways and, at the same time, we could have based our discussion on other known hybrid services. Our intent is to use this call forwarding service just as an exemplification of the service creation capabilities available with our architecture. Note that the ITU-T has started to standardize some supplementary services for H.323 terminals. We claim that our approach is more powerful ; not only does it support services in which PSTN terminals are also involved, but it makes the whole network *programmable*; this means that a wide range of services can be implemented without having to go through the tedious and slow process of standardization. Clearly, this will be at expense of service interoperability, but this is the price to pay for competition between the various service providers.

5.1 Call Forwarding across the PSTN and the Internet

We consider a service that allows a customer to forward his incoming calls (addressed to his telephone set from any other telephone set) to a computer terminal connected to the Internet. The service will be realized as described in Fig. 12, which is a specialization of Fig. 3. The Java Service Layer runs two distributed components, `scpCFS` and `gwCFS`, that implement the call forwarding service by controlling PSTN SCPs as well as H.323 gateways. The service layer plays the role of a liaison that permits the provision of H.323-based hybrid services across the PSTN and the Internet. The interaction beween the service layer and the PSTN side could recall the model promoted by the PINT group. In our approach we go one step further than the PINT paradigm : We allow the SCP to generate requests toward the service layer. While the PINT model focuses on enabling the PSTN to take service requests from Internet devices, we are concerned with implementing the service logic using Java components. The major functionality that we expect from the SCP is to send an event to the Java Service Layer whenever a special trigger is detected. The logic can later instruct the SCP to open a connection between two endpoints. What we expect from the SCP can therefore be expressed by three words : trigger, service logic and connection.

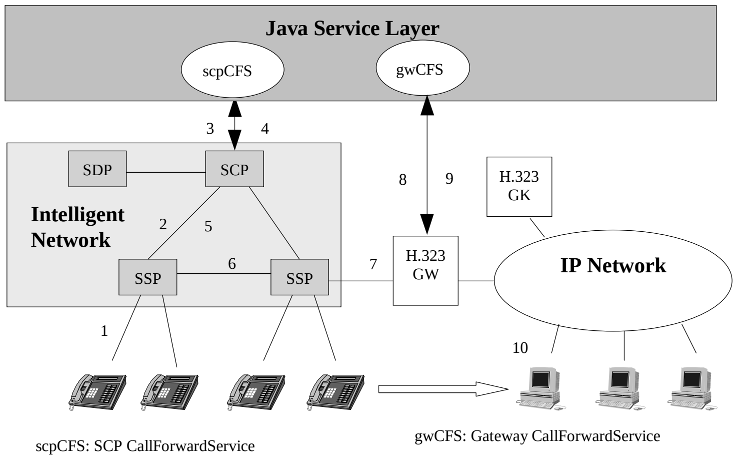

Fig. 12. Provision of call forwarding services across the PSTN and the Internet.

Note that the standard interaction between the SSP and the SCP will still work in our architecture. The SCP normally implements the service logic out of SIBs. We supplement this conventional operation of the SCP with the possibility of implementing the logic by using well-defined Java components. Within such a paradigm, the SCP re-emits the trigger, detected by the SSP, toward the Java Service Layer. This paradigm can be taken even much further, by simply removing the SCP and letting the SSP communicate directly with the Java Service Layer. We do not currently extend our paradigm to this point.

To implement the logic corresponding to Call Forwarding, we use a subclass of `HybridService` called `CallForwardService` (Fig. 13). The main attribute of this subclass is a vector of addresses to which calls should be redirected depending on the time of day. To take this into account, we create a new class called `TimedAddress` which denotes the validity of the redirect address within certain periods of time. `CallForwardService` implements methods to forward a call (`forwardCall`), to terminate the forwarding (`endOfForwarding`), and to get the redirect address (`getForwardAddress`).

```
public class CallForwardService extends HybridService {
        TimedAddress redirectAddr []; /* Vector of addresses, each with their time
                                      availability */

        public void forwardCall(Address permanentAddr, TimedAddress redirectAddr,
                               String customer) ;
        public void endOfForwarding(Address permanentAddr, Address redirectAddr,
                                     String customer);
        public void getForwardAddress(TimedAddress redirectAddr) ;
} ;

public class TimedAddress extends Address {
        Time availabilityStart ; /* Beginning of the availability period. Time is a
                        string of a specific format : day.month.year/hour : :minutes
*/
        Time availabilityEnd ; /* End of availability period */
        int timezone ; /* Time zone related to the availability period ;
                      timezone=-1 means GMT-1 */

        public TimedAddress(Address addr, Time start, Time end , int timezone)
                              throws TimeFormatInvalidEvent, TimeFormatOKEvent ;
        public ~TimedAddress() ;
        public void getAddressAvailability(Time start, Time end, int timezone) ;
} ;
```

Fig. 13. Definition of `CallForwardService` and `TimedAddress`.

5.2 Service Creation and Invocation

The provider decides to realize the service as follows: Any call request, from the PSTN, that is destined to the customer is processed in such a way that the request gets routed through the gateway (Fig. 12). The component that implements the logic of the forwarding for the SCP must return a redirect address that achieves the routing through the gateway. When the gateway receives the request, it alerts the Java Service Layer in order to get the address of the terminal to which the call must be redirected.

Creating a new service results from the specialization of the generic information flow diagram (Fig. 7) for the relevant elements in the service system infrastructure (Fig. 12). In light of the service description given above, the two main elements that need to be tuned in order to provide the Call Forwarding service across the PSTN and the Internet are the SCP and the H.323 gateway. Therefore, the provider must specialize the events and service components outlined in Section 4 for each of these elements. First, he creates `CallForwardService` and `TimedAddress` instances for the SCP. The trigger pattern for the SCP is the customer's permanent address. The trigger event instance associated with the SCP is called `scpTriggerEvent`. The SCP will manage to inform the SSP about the new trigger, so that the SSP can detect this trigger later on. The instance of `CallForwardService` for the SCP is called `scpCallForwardService`. This instance will create a new object of type `TimedAddress`; call it `scpTimedAddress`. The address embodied by this instance must be chosen so that any call to this address gets routed through the gateway.

The provider then creates instances of `TriggerEvent`, `CallForwardService` and `TimedAddress` for the GW; call them `gwTriggerEvent`, `gwCallForwardService` and `gwTimedAddress`, respectively. The attribute `trigPattern` of `gwTriggerEvent` is set to the address corresponding to `scpTimedAddress`. When a request with this address reaches the GW, a trigger will be detected and an event fired by a process within the gateway (this process is of type `EventSource`). Then, `gwCallForwardService` will catch and handle the event fired. `gwTimedAddress` contains the address of the computer to which calls are to be forwarded.

When a PSTN user dials the customer's number, the request is detected by the SSP as a special service. The SCP is then asked to provide the appropriate instructions. It fires a trigger event toward the Java Service

Layer. The `scpCallFowardService` instance created beforehand by the service provider returns a number that enforces the request to be routed through the gateway. The number given to the SCP matches a trigger in the gateway. The gateway detects the special pattern and fires a trigger event towards the listeners that previously registered themselves. The `gwCallForwardService` instance that corresponds to this particular customer is notified and handles the event appropriately, by returning the Internet address where the customer's H.323 terminal can currently be reached. An Internet connection is established from the gateway to this address; this connection is, later on, bound to the connection segment within the PSTN.

6 Conclusion

A few years ago, the ATM technology seemed to be poised to replace all the existing circuit-switched and packet-switched networks; it was expected to become the unique infrastructure for the provision of integrated and advanced services. Clearly, this vision is not valid anymore.

More recently, the progress of real-time services over the Internet led many people to believe that networks based on the Internet protocol would replace the PSTN. This evolution is probably realistic in the long run; however, the recent history of networking has demonstrated that the most important success factor is not so much the target architecture itself, but rather the migration path taken to reach it.

Identifying the appropriate migration path is precisely the purpose of this paper. We believe that the concepts presented will pave the way for the introduction of a new set of services that will take advantage, in an integrated way, of the Internet, PSTN and wireless infrastructures. These services, that we call *hybrid services*, will deliver to the users the integrated communication environment promised more than 20 years ago (but not fulfilled yet). Unlike other service creation architectures, such as TINA, our approach leverages on the **existing** networks and mechanisms to smoothly introduce new services.

We presented a service creation architecture and a set of service components that allow simple, rapid creation and deployment of hybrid services as distributed applications that extend the functionality of existing terminals, servers and network nodes. This is achieved by defining common interfaces and a common execution environment provided by the now mature Java technology. Each element is extended with a Java virtual machine and equipped with a set of pre-developed service components realized by assembling JavaBeans. These service components provide the building blocks that can be used to implement hybrid services.

We illustrated the power of our architecture by constructing an example of a hybrid service. We showed how the service components defined in our architecture could be specialized to construct a call forwarding service that spans both the Internet and PSTN.

While in this paper we laid out the main concepts for an architecture for integrated and seamless provision of hybrid services, there are many other issues that we are currently investigating. First, we are working on the building of the SCE. Second, we are exploring an efficient way to map the service components onto the physical platform elements. Third, we are working on breaking down the service component, `HybridService,` into finer-grained components.

We expect this approach to play a fundamental role in the highly competitive business of service provisioning.

Acknowledgments

The authors are thankful to their colleagues Colin Harrison, Jurij Paraszcsak, and Magda Mourad of the IBM Thomas Research Center, as well as Holly Cogliati, Maher Hamdi, Monika Lundell and Olivier Verscheure of the Swiss Federal Institute of Technology.

References

[Ans96] T. A. Anschutz, A Historical Perspective of CSTA, *IEEE Comm. Mag.*, Vol. 34, No. 4, April 1996, pp. 30-5.

[BDK] http://www.javasoft.com/beans/

[Brad94] R. Braden, D. Clark and S. Shenker, Integrated Services in the Internet Architecture : An Overview, IETF Informational RFC 1633, June 1994.

[CNIS] J. P. Hubaux, C. Gbaguidi, S. Koppenhoefer, and J. Y. Le Boudec, The Impact of the Internet on Telecommunication Architectures, *Submitted to Computer Networks, and ISDN Systems*, Special Issue on Internet Telephony, Oct. 1998.

[Cro96] P. Cronin, An Introduction to TSAPI and Network Telephony, *IEEE Comm. Mag.*, Vol. 34, No. 4, April 1996, pp. 48-54.

[CORBA] OMG, The Common Object Request Broker: Architecture and Specification, Rev. 2.0, July 1995.

[CORBA-IN] OMG Document: telecom/97-12-06, Interworking between CORBA and Intelligent Network Systems, Request for Proposals, Dec. 1997.

[CORBA-MOB] OMG Document: telecom/98-05-xx, Request for Information: Supporting Wireless Access and Terminal Mobility in CORBA.

[Dho96] H. D'Hooge, The Communicating PC, *IEEE Comm. Mag.*, Vol. 34, No. 4, April 1996, pp. 36-42.

[ECMA] European Computer Manufacturers Association (ECMA), Computer-Supported Telecommunications Applications, ECMA Technical Report TR/52, June 1990.

[Fay96] I. Faynberg, L. R. Gabuzda, M. P. Kaplan and N. J. Shah, *The Intelligent Network Standards, their Applications to Services* (McGraw-Hill, 1996).

[Fay97a] I. Faynberg, M. Krishnaswamy and H. Lu, A proposal for Internet and Public Switched Telephone Networks (PSTN) Interworking, Internet Draft, March 1997.

[Fay97b] I. Faynberg et al., The Development of the Wireless Intelligent Network and Its Relation to the International Intelligent Network Standards, *Bell Labs Technical Journal Vol. 2 Number 3, Summer 1997.*

[Fla96] D. Flanagan, *Java in a Nutshell, A Quick Reference for Java Programmers*, (O'Reilly & Associates, Inc., May 1996).

[F.851] ITU-T, *Universal Personal Telecommunications (UPT) – Service Description*, Rec. 851, Feb. 1995.

[Geoplex] http://www.geoplex.attlabs.net/

[GSM 02.78] ETSI, Digital Cellular Telecommunications System (Phase 2+) – Customized Applications for Mobile Network Enhanced Logic (CAMEL) – Service Definition (Stage 1), GSM 02.78, Vers. 5.2.1, July 1997.

[H.323] ITU-T, *Visual Telephone Systems and Equipment for Local Area Networks which provide a Non-guaranteed Qualîty of Service*, Rec. H.323, Nov. 1996.

[Kris97] M. Krishnaswamy, PSTN-Internet Interworking- An Architecture overview, Internet Draft, Nov. 1997.

[I.210] ITU-T, *General Aspects of Services in ISDN*, Rec. I.210, 1988.

[JAIN] J. de Keijzer, Java Advanced Intelligent Networks, 1998.

[JTAPI97] Sun Microsystems, The Java Telephony API- An Overview, White Paper, Vers. 1.1, Jan. 1997.

[Jub98] H. Jubin (ed.), *JavaBeans by Example*, (Prentice Hall, ISBN 0-13-790338-3, 1998).

[Lazar] A. A. Lazar, K. S. Lim and F. Marconcini, Realizing a Foundation for Programmability of ATM Networks with the Binding Architecture, *IEEE Journal on Selected Areas in Communications*, 14(7), Sept. 1996, pp. 1214-27.

[Low97] C. Low, Integrating Communication Services, *IEEE Comm. Mag.*, Vol. 35, No. 6, June 1997, pp. 164-9.

[Mod90] A. R . Modaressi and R. A. Skoog, Signaling System No. 7 : A Tutorial, *IEEE Comm. Mag.*, July 1990, pp. 19-35.

[MPZ96] T. Magedanz and R. Popescu-Zeletin, *Intelligent Networks : Basic Technology, Standards and Evolution*, (Intl. Thomson Computer Press Ed., 1996).

[Nic97] K. Nichols, V. Jacobson and L. Zhang, A Two-bit Differentiated Services Architecture for the Internet, Internet Draft <draft-nichols-diff-svc-arch-00.txt>, Nov. 1997.

[Q.1211] ITU-T, *Introduction to the Intelligent Network Capability Set 1*, Rec. Q.1211, March 1993.

[SunXTL] Sun Microsystems, Inc., The SunXTL Platform, White Paper, Feb. 1995, available as http://www.sun.com/products-n-solutions/sw/SunXTL/whitepaper.ps

[TAPI3.0] Microsoft, Inc., IP Telephony with TAPI 3.0, White Paper, 1997.

[Thom96] G. A. Thom, H.323 : The Multimedia Communications Standard for Local Area Networks, *IEEE Comm. Mag.*, Vol. 34, No. 12, Dec. 1996, pp. 52-6.
[TINA] TINA-C, Service Architecture, Vers. 5.0, June 1997.
[WML98] Wireless Application Protocol (WAP) Forum, Wireless Application Protocol – Wireless Markup Language Specification, Draft, 20 March 1998.
[X.901] ITU-T, *Basic Reference Model of Open Distributed Processing – Part 1: Overview and Guide to Use*, Rec. X.901, 1994.