

Integration of Internet and Telecommunications: An Architecture for Hybrid Services

Constant Gbaguidi, Jean-Pierre Hubaux,
Institute for Computer Communications and Applications
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 article, we propose an architecture for *hybrid services*, i.e., services that span many network technologies, such as the Public Switched Telephone Network (PSTN), cellular networks and networks based on the Internet Protocol (IP). These services will play an important role in the future because they leverage on the existing infrastructures, rather than requiring 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 according to the JavaBeans model. We illustrate the strength of our architecture by discussing two hybrid-service examples: a calendar service and a 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 integrated services that take advantage of both technologies simultaneously. We refer to this new class of services as *hybrid services*. Examples of such services are: Click-to-Dial [12] (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 networks based on the Internet Protocol (IP) through gateways, universal messaging, and access to electronic and voice mail from either an IP network or the PSTN (we consider the PSTN as encompassing its cellular extension).

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 IP-based solutions, which rely on IP telephony for voice, impractical. Taken separately, PSTN and IP technologies are far from being an ideal ground for developing future hybrid services; however, if coupled together they can complement each other effectively.

The PSTN provides an extremely reliable, available and ubiquitous system, with guaranteed Quality of Service (QoS). It is endowed with a powerful service creation and provision platform called the Intelligent Network (IN) [29][11][27]. 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) [26]. 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 such applications are Wireless IN (WIN) [13] and Customized Applications for Mobile network Enhanced Logic (CAMEL) [17].

IP networks are characterized by a devolved service architecture, i.e., there is no global and standardized framework for the provision and creation of services. 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 terminals and servers. IP-based applications take advantage of intelligent terminals and powerful user interfaces. However, some services (and even basic mechanisms) of the IP technology 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 focused on hybrid services have emerged [10][12][16]. So far, these efforts were focused mainly on *inter-working* rather than *integration*. The common approach taken by these activities is to model the PSTN (or IP networks) as a stand-alone system whose services can be accessed through a gateway that acts like a PSTN (or IP) 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 article, we present a new approach to hybrid service creation in a heterogeneous environment that incorporates PSTN and IP technologies. We propose an architecture, as well as an early set of *service components*, that allows simple and rapid creation and deployment of hybrid services. A service component is a pre-developed code that can be used in conjunction with other components to build a distributed service. Service components are run on terminals, network elements, gateways, as well as servers and peripherals; they therefore provide an openly programmable service environment. The proposed service components are implemented in conformance to the component model for the Java language [14], i.e., JavaBeans [2][23]. Components that comply with the JavaBeans specification are also called *beans*. This specification releases the rules to observe when designing a component in the Java language. These rules ensure a high level of re-usability and portability to beans. Using Java allows service developers to write applications that can be executed without modification on any equipment. In our approach, service components can be provided by third parties or vendors of network equipment, servers, peripherals and terminals. Hence, the service creator writes the service logic simply by putting these components together.

This article is organized as follows. In Section 2, we review service integration projects carried out in both Internet and Telecommunications communities. 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 examples of hybrid services that can be constructed using the architecture and the service components described in Section 4. Finally, in Section 6 we summarize the main contributions of this article 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 toward the development of services that can span both the PSTN and IP networks. We classify the efforts relevant to this article into four categories: enabling technologies for interoperability at the signaling and media control level, efforts related to the access to telephony services from a computer, works on the access to the IN infrastructure from IP networks, 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

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 [18][32] 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 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 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; nevertheless, 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 network addresses.

2.2 Computer-Telephony integration

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 [10] on the provision of Computer-Supported Telecommunications Applications (CSTA) [1][9]. Many implementations were carried out for CSTA : Microsoft's Telephony Application Programming Interface (TAPI) [31], Novell's Telephony Services Application Programming Interface (TSAPI) [5] and Sun's XTL [30]. Each of these implementations was targeted for one specific operating system. An integrating implementation, based on the Java language and called Java Telephony API (JTAPI) [22], swept away much of the platform burden. 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. 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. JTAPI is made up of a core package that 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.

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) [34], which is an adapted version of the HyperText Markup Language (HTML) to the wireless environment. This script language, called WMLScript, is much simpler than JavaScript (script language associated with Java). A major limitation of 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. For now, we promote the use of JTAPI, essentially because of its completeness.

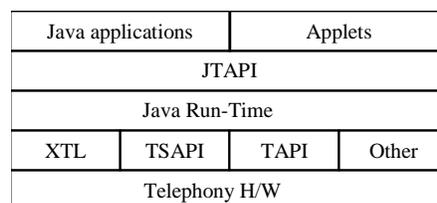


Fig. 1 The JTAPI stack

2.3 PINT

A working group, named 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 IP networks [12][19]. In this scenario, the user makes the call request from an IP network 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 a gateway between the IP network 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 element 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 an IP network [25]. 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 an IP network, service providers can enable the customers to create their services as easily as they create their Web pages [4].

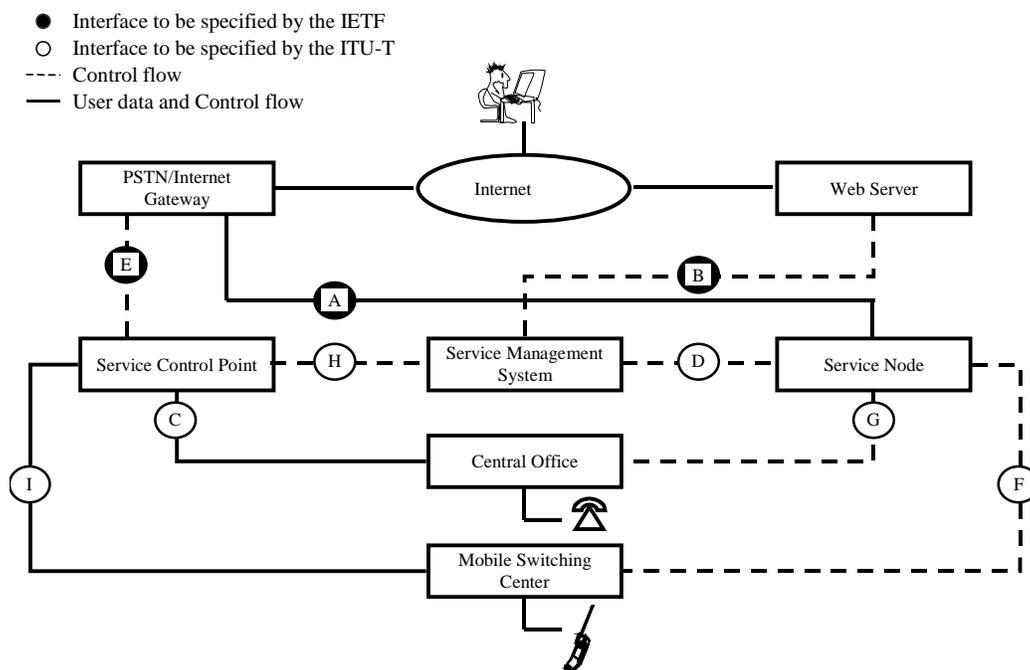


Fig. 2 The PINT model

2.4 Distributed computing and telecommunication 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 aforementioned trends are Open Distributed Processing (ODP) [35] and the Common Object Request Broker Architecture (CORBA) [6]. 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 exposes the operations, notifications and streams that the object can handle, as well as the object's attributes. The interface is declared in 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 the establishment and control of communication sessions, as well as QoS control. 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 Ref. [CNIS].

The CORBA technology is also being considered for the IN, as well as wireless access and terminal mobility. For the IN, current efforts focus on how CORBA could effectively interwork with the Signaling System #7 (SS7) [26] used in the IN. A first approach consists in providing gateways between the SS7 network and CORBA-based systems; a rather simple solution with arguable scalability. Another solution would be to use SS7 as the interoperability protocol between *islands* of CORBA-enabled telecommunications equipment [7]. 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.) [8]. More generally, there are some concerns about the ability of CORBA-based products to fulfill the requirements of telecommunication systems, when it comes to performance and scalability (Java has also generated the same type of concerns).

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 constantly enriched with new features in an open way. Even though some implementations of the Java specification (such as Sun's Java and Microsoft's Java) are incompatible, their free availability minimizes much of the issue. Moreover, while Sun's Java is generally considered as the reference implementation and the *Pure Java* label is issued to the products that fully comply, there is no organization that issues conformance certificates for CORBA products.

In this section, we have 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 considered in this article.

3 Service concept

To set the stage for the discussions to follow, we now briefly review the many meanings of the word *service* in different communities. Then we propose a more rigorous definition. In particular we define *teletinformation 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. For the narrowband ISDN, 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) [20]. In the Internet community, 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 [3], differentiated services [28], etc.).

We define a *teleinformation service* 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. The service consists in either providing connections with remote users or retrieving information from servers. The service must involve a terminal, a server or several other terminals, an appropriate set of applications running on these servers and terminals, and 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 so 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 to advertising animation, for example. 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 use 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 his customers. This idea is very intuitive in the Telecommunications area; in fact, both the teleservices and the complementary services defined for the ISDN are subsets of teleinformation services. In the Internet community, this concept has hardly been investigated so far, because all use of the network has been thought to be free.

A *hybrid service* is a teleinformation service that relies both on IP and on Telecommunication networks 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 goods. 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 an IP network.

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 removal. We call this process *service engineering*. It requires a platform or an environment that has the following features: 1) independence of the service from the underlying operating systems; 2) a set of reusable components for the creation of new services; 3) efficient mapping of the service components onto the system infrastructure; 4) a graphical and easy-to-use interface to create, manage and operate services; 5) generality and expandability to allow for the creation of novel services.

Our focus in the remainder of this article will be on the determination of generic service components that, 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: real-time communication services (Section 4.2). We describe a generic topology of the system infrastructure for this family of hybrid services. Mechanisms that enable the programmability of this infrastructure are 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. The IN is provided with a Service Creation Environment (SCE) in which a service is created from a set of standardized service-independent building blocks (SIBs) (although the concept of SIB has been removed from the standards 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,

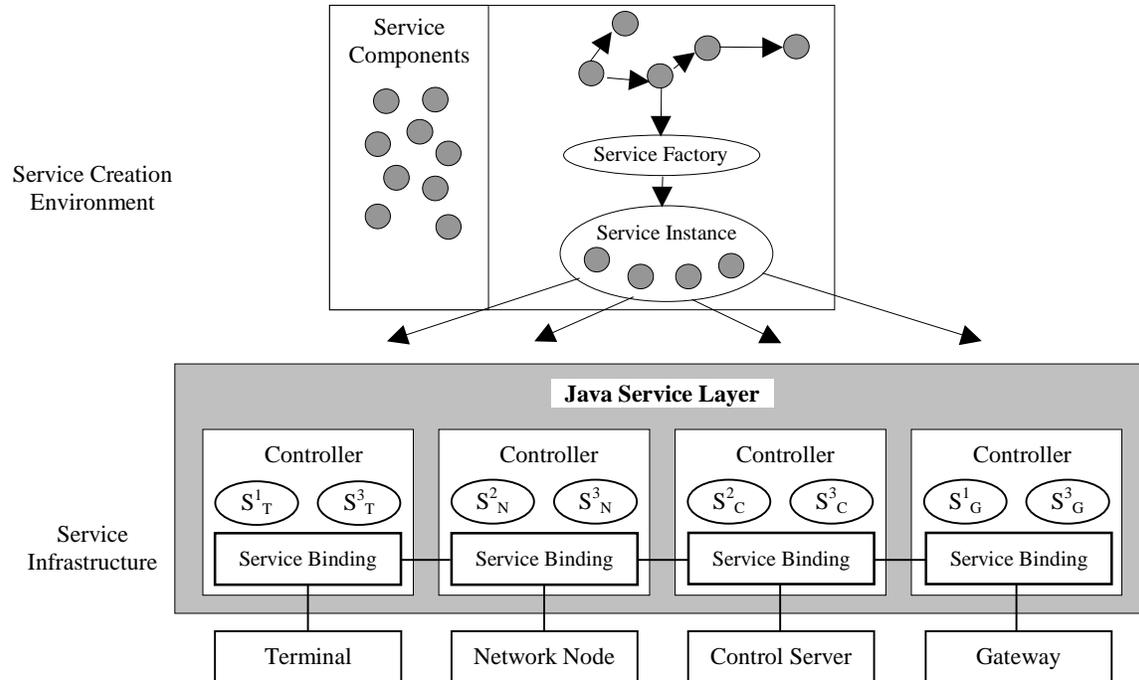


Fig. 3. Service architecture. S^i_T = subsystem of service i running on terminals; S^i_N = subsystem of service i running on network nodes; S^i_C = subsystem of service i running on control servers; S^i_G = subsystem of service i running on gateways

such as SSPs and SCPs. It is worth noting that, even though SIBs are standardized components, the IN SCE is not fully standardized; a service already created within a given IN SCE has to be re-created in another SCE if, for example, the customer wishes to move from one operator to another one. In a manner that is similar to the IN approach, we envision an SCE that uploads service components into terminals and servers, in addition to network nodes, control nodes and specialized elements such as gateways (cf. Section 5).

As shown in Fig. 3, our service architecture is composed of two elements: A *Service Creation Environment (SCE)* and a *Service Infrastructure (SI)*. In the SCE a service logic is put together by the service creator using a set of service components as building blocks. The service logic is then sent to a service factory, which creates a new instance of service. The service instance is organized into service subsystems that are further uploaded into the system infrastructure, more specifically in the controllers of the system elements. For example, service S^1 is broken down into two subsystems: S^1_T on the terminal controller and S^1_G on the gateway controller.

The *service infrastructure* is composed of a collection of controllers and the underlying system elements. The controllers, which together form the *Java Service Layer*, run a Java virtual machine and incorporate mechanisms to interact with the underlying system elements. We classify the system elements into four main categories: 1) terminals, e.g., black telephones, mobile phones, personal digital assistants and personal computers; 2) network nodes, e.g., switches, SSPs, routers, Mobile services Switching Centers, MSCs, and satellites; 3) control servers, e.g., SCPs, DNS servers and H.323 gatekeepers; 4) gateways, e.g., H.323 gateways. Information servers such as file servers, Web servers and email servers can be regarded as powerful terminals. It should be noted that the software installed on terminals is not entirely under the control of the service provider; indeed, the end-users are very often responsible for selecting and installing the application they prefer. However, while users are usually free to choose the Web browser, the service provider may download appropriate applets during the session establishment phase.

The binding among various controllers (horizontal binding) as well as between controllers and system elements (vertical binding) is provided by the *Service Binding Layer* (SBL). As shown in Fig. 4, the SBL is composed of a *binding components library*, a *remote method invocation layer*, an *enterprise components layer* and a *system element interface layer*. The *binding components library* provides entities that ensure the security during the upload of the service subsystems, as well as a Java class library that abstracts the behavior and functionality of the system elements. Since the binding components are written in Java we rely on the Java security package to provide the required security mechanisms. The *remote method invocation* (RMI) layer provides a high-level communications infrastructure between the distributed subsystems of each service, thereby supporting the horizontal binding among components of the same service instance. The *enterprise components* layer provides access to enterprise platforms such as CORBA or databases. Java has an extension called Enterprise JavaBeans which provides several crucial enterprise components, including a bridge between Java RMI and CORBA. The *system element interface* layer embeds drivers to access system element functionality. The implementation of these drivers is, of course, system dependent. For instance, to access the functionality of IN elements the system element interface layer will use the IN Application Protocol (INAP) to implement the functionality required by the service logic. Other implementation of system elements may require the use of the Simple Network Management Protocol (SNMP) and H.323-related protocols. There is a growing trend for providing APIs to drive system elements. In the telecommunication network nodes arena this trend is exemplified by the Java Advanced Intelligent Network (JAIN) initiative, currently under development in a working group led by Sun Microsystems, Inc. [21]. The JAIN initiative is posed to become the standard for writing (and specifying) system element interfaces. While JAIN's focus is on defining interfaces for IN elements, our approach is on building and deployment of a *service logic* that spans multiple environments, including the IN and the Internet. In our architecture the interface to the IN elements can be realized according to the JAIN initiative or other approaches for IN access to programmable architectures such as TINA [36][37][38]. Whereas TINA puts much of the service logic in the concept of session, our aim is to refine this concept, by splitting it into finer-grained components to be assembled to effectively create and control services. Recently, an enhanced version of JTAPI has been proposed as a unified interface to all system elements [39].

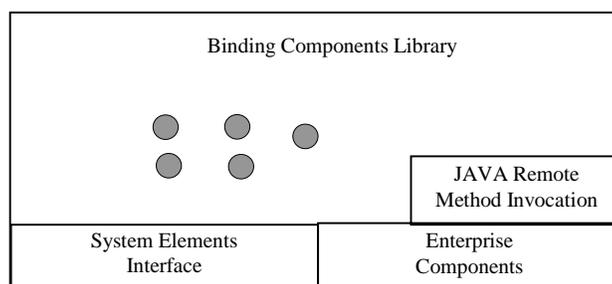


Fig. 4 Service binding layer

The service binding layer allows for the implementation of a service at a high level using Java. At low levels, drivers are provided to access the system elements. We selected Java as the language for service implementation because of the following features: *ubiquity*, *platform-neutrality*, *security* and *extensibility*. Java *ubiquity* is given by the fact that most terminals, from smart phones to embedded devices, can now be equipped with a Java virtual machine. This trend is also spreading to the devices located in the PSTN such as SCPs, SSPs and SNs. *Platform-neutrality* (i.e., write once, execute everywhere) means that applications written in Java can be run on any platform that has a Java virtual machine installed. In the heterogeneous communication environment reflected in Fig. 3, having components that can run everywhere using exactly the same implementation code, is a great benefit. Among the most widely used languages, Java is certainly the one that has the most *security* mechanisms that restrict the actions that a code can perform on a given system. *Extensibility* refers to the many capabilities that are being added to the core Java specification. In particular, a component architecture (i.e., JavaBeans [Jub98]) has been developed to help programmers achieve a high re-usability and portability of their code components (called beans). Beans are Java classes

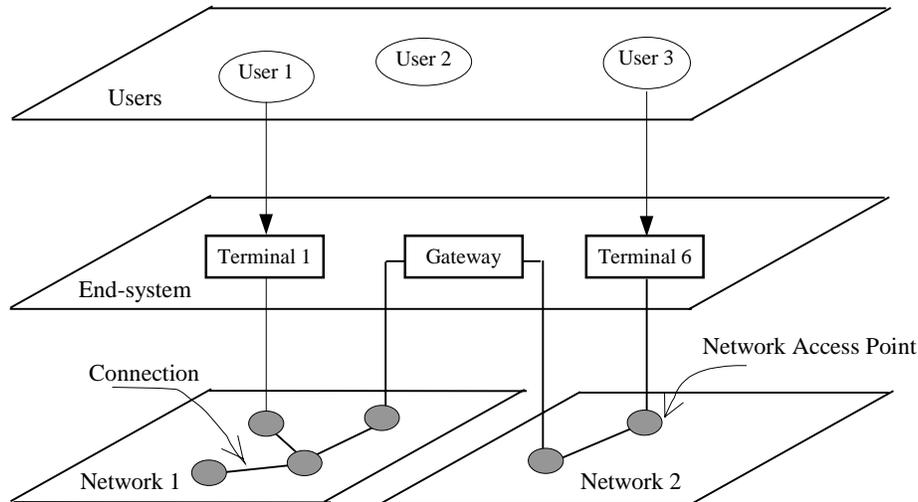


Fig. 5 System abstraction for hybrid real-time communications services

written according to a certain rules that especially facilitate their inspection and, therefore, their correct manipulation and re-use. These rules constitute the main difference between a regular Java class and a bean. We use these rules to design our architecture's components. Beans make extensive use of events to communicate with one another. Other relevant packages are Java RMI, Java security and Enterprise JavaBeans.

By using technologies such as object-orientation, code uploading, distribution and component design our service architecture goes beyond the concepts predicated by the IN service architecture and provides a more suitable ground for the implementation of hybrid services.

4.2 Real-Time Communication Hybrid Service

In order to illustrate the concepts introduced above, we focus, in the remainder of this section, on a subset of hybrid services, i.e., 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 IP-based services, and, it also allows us to refer to more specific service components.

From a platform perspective, the system infrastructure that realizes these services is represented in terms of objects. These 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, we design the object model and the corresponding components.

Fig. 5 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 the real users of the service. Users may be either at fixed locations or mobile (an example of user mobility is the Universal Personal Telecommunications, UPT [15]). Users interact with the system via terminals that are abstracted as *Terminal* objects in the end-systems plane. Examples of terminals are telephones, portable phones, mobile terminals, 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 leading to 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 among the NAPs that belong to it. We use the term "NAP-to-NAP Connection" to designate the way connectivity is provided between two NAPs; in the case of the IP networks, we use the term *Connection* because the kind of services we consider here require resource reservation strategies to be deployed. A

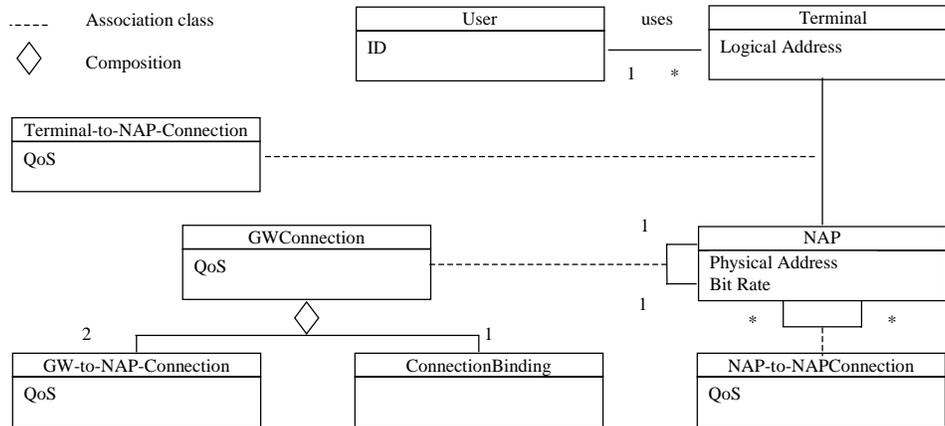


Fig. 6 Object model for hybrid real-time communications services

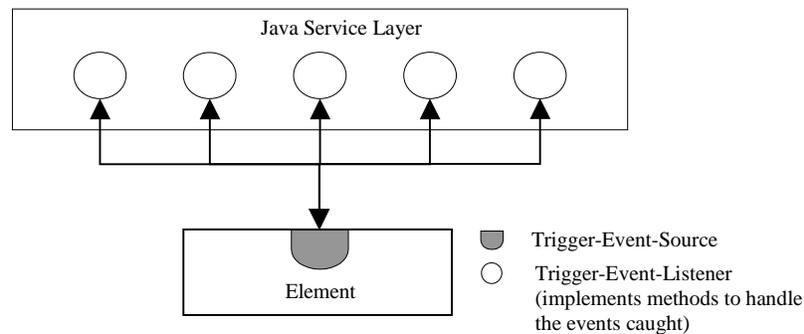


Fig. 7 Trigger-Event-Source and Trigger-Event-Listener model for the platform elements

gateway connects NAPs from different network technologies; it generally 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. In fact, 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.

The object model corresponding to the above hybrid system abstraction is depicted Fig. 6. 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 (GW Connection) 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.

4.3 Programmability of the Hybrid Service Platform

In this section, we address the programmability of the elements of the service architecture introduced in Section 4.2. We present a generic information flow diagram that describes the processing of information within and across the main nodes of the platform. We then discuss the events and service components necessary to the provision of hybrid services.

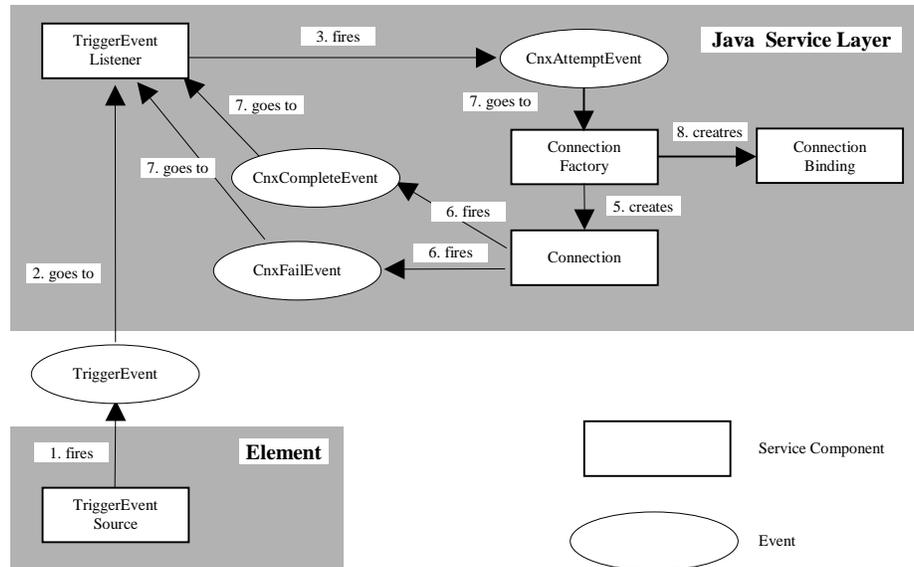


Fig. 8 A generic information flow diagram for real-time hybrid services

To allow for the introduction of new services, while minimizing the upgrade of existing elements we propose the use of triggers as the basic information-flow building-block. As depicted in Fig. 7, we suggest to associate with each system element a *trigger process* that detects specific conditions (*service events*) during the processing of a call request. Once the condition is detected the trigger process delegates the handling of the condition on to *listener processes* located in the Java Service Layer. The listener processes embed the logic according to which service requests must be processed. This mode of operation fits well with the IN functionality: the SSP detects special patterns and then invokes the SCP in order to get instructions on how to process the service request. As we will see, the model also nicely fits with the way H.323 elements (in IP networks) are expected to operate.

Fig. 8 describes the information flow diagram for real-time hybrid services according to our event source and listener model of Fig. 7. A source (*TriggerEventSource*) fires a trigger event (*TriggerEvent*) which is eventually caught by listeners (*TriggerEventListener*). When a listener catches a trigger event, it compares the trigger's attribute (e.g., 1-800 prefix or a full address) with its own trigger-dependent attribute and, if the attributes match, the listener handles the event. The event handling mechanism includes the service logic and, as for the case in Fig. 8, it may require a connection to be set up between two (or more) endpoints. To create the connection the *TriggerEventListener* fires a connection attempt event (*CnxAttemptEvent*) to a connection factory (*ConnectionFactory*), which in turns creates a connection object. The connection factory component is necessary because the *TriggerEventListener* cannot fire a connection attempt event (*CnxAttemptEvent*) directly to connection objects, since the does not exist yet. The connection object fires either a connection complete event (*CnxCompleteEvent*) or a connection fail event (*CnxFailEvent*), 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. 8). In the case of a gateway, a binding may be needed between the connection segments on the two networks interconnected by the gateway. If so, the connection factory realizes the binding.

The information flow diagrams of Fig. 8 includes four events: *TriggerEvent*, *CnxAttemptEvent*, *CnxCompleteEvent*, *CnxFailEvent* and five service components: *TriggerEventSource*, *TriggerEventListener*, *ConnectionFactory*, *Connection* and *ConnectionBinding*.

Fig. 9 describes, in details, the methods and attributes of *TriggerEvent*, which is the most illustrative event of the diagram in Fig. 8. *TriggerEvent* is fired by *TriggerEventSource* when an address with a specific pattern is detected. Its main attributes are: *originatingNetwork*, which indicates the network (Internet, PSTN, etc.) from where the service request comes from; *trigSrcAddr*, which indicates the source of the service request (e.g., an SSP address); *trigPattern*, which indicates the pattern whose

detection triggered the event. The `Address` component detailed in Fig. 9 represents an access point to a resource. `Address` main attributes are: `relatedLayer`, which defines the layer to which the address relates (e.g., an Internet address relates to layer 3); `typeOfNetwork`, which indicates the type of network that the address relates to and `addrString`, which gives the address in string form.

The `TriggerEventSource` and `TriggerEventListener` service components relate to the production and treatment of the event `TriggerEvent` (Fig. 9). `TriggerEventSource` abstracts over the producer of the trigger event, and 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 and then determines whether it is supposed 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 fire an event called `EvtHandlingFailEvent` when it fails to handle a trigger event previously caught. The event `CnxAttemptEvent` can be fired 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 according to which the service request must be processed. The logic is called after a trigger event has been fired, i.e., `HybridService` is a listener of a specific trigger event. `HybridService` can aggregate some other components such as a component for address binding (used for address), or groups of terminals (to create a closed group or a Virtual Private Network). The attributes of `HybridService` are: `servDescr`, a textual description of the service; `customer`, the name of the customer for whom this service was created; `customerPermanentAddr`, the permanent address of the customer. The component `HybridService` must be further specialized in order to take into account the specifics of each service. Hence, there will be specializations for call centers, VPNs, etc. In Section 5 we will show how `HybridService` is specialized for the case of a call forwarding services that spans both PSTN and Internet.

`ConnectionFactory`, `Connection` and `ConnectionBinding` service components are used for connection control (Fig. 8). `ConnectionFactory` manages the connections set up by catching connection attempt events (`CnxAttemptEvent`). To this end, `ConnectionFactory`, implements a listener for these events, as well as a method to handle connection attempt events. `ConnectionFactory`'s 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, for instance, in a gateway.

`Connection` abstracts over a physical or logical link between two access points that belong to a network

```

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

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

```

Fig. 9 Definition of `TriggerEvent`

```

public class TriggerEventSource{
    public 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
                        * coming (e.g., PSTN, Internet)
                        */

    String patternToCatch; /* Pattern that the listener intends to catch */

    public TriggerEventListener(String evtOrigNet, String pattern);
    public void triggerEventHandler(TriggerEvent trigEvt);
                                /* fires 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 void remove();
};

```

Fig. 10 Definition of TriggerEventSource, TriggerEventListener and HybridService

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; *vgRate* denotes the average rate of this traffic; *delay*, indicates the tolerable delay.

Connection implements a constructor, a destructor and methods to get the values of its attributes. It can produce the following events: *CnxCompleteEvent*, when the connection set up has been successful; *CnxFailEvent*, when the connection could not be established (the reasons are then given); *CnxDumpedEvent*, when the connection has been dumped (due to a user hanging up, for example).

We consider several subclasses of the generic component Connection as depicted by the inheritance tree of Fig. 12. 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 and 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 typically happens in a gateway where *ConnectionBinding* abstracts over a special connection that lies entirely within the gateway. *ConnectionBinding* attributes are its status and the 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*.

```

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);
    /* Fires CnxCompleteEvent, CnxFailEvent */

    public void releaseConnection();
    public int getStatus();
    public int getPeakRate();
    public int getAvgRate();
    private void connectionLife(); /* Fires 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);
    /* Fires CnxBindingCompleteEvent, CnxBindingFailEvent */

    public void releaseConnectionBinding();
    public void cnxDumpedHandler(CnxDumpedEvent dumpedEvt);
    /* Fires CnxBindingDumpedEvent */

    public int getStatus();
};

```

Fig. 11 Definition of the components ConnectionFactory, Connection and ConnectionBinding

5 Case Studies: Calendar Access and Call Forwarding 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 the PSTN and an IP network. In order to keep our discussion focused we refer to two specific examples: *hybrid calendar service* and *call forwarding*. The first service allows users to create and access calendar entries from PSTN, wireless, wired, as well as IP, terminals. The second service allows users to forward calls from a PSTN terminal to an H.323 terminal and vice versa.

It should be noted that the services presented in this section could be realized in many other ways. At the same time, we could have based our discussion on other hybrid services. Our intent is to use the hybrid calendar and call forwarding services just as examples of service creation capabilities available with our architecture. As it is impractical to discuss a complex service such as hybrid calendar down to the level of the single service components, we specify such components for the second, less complex example (i.e., Call Forwarding). Call Forwarding is available on the PSTN and is currently under consideration by the ITU-T as a supplementary service for H.323 system elements.

Using the two examples, we are able to show that our approach is more powerful. In fact not only does it support services in which PSTN and IP terminals are involved, but it makes the whole network

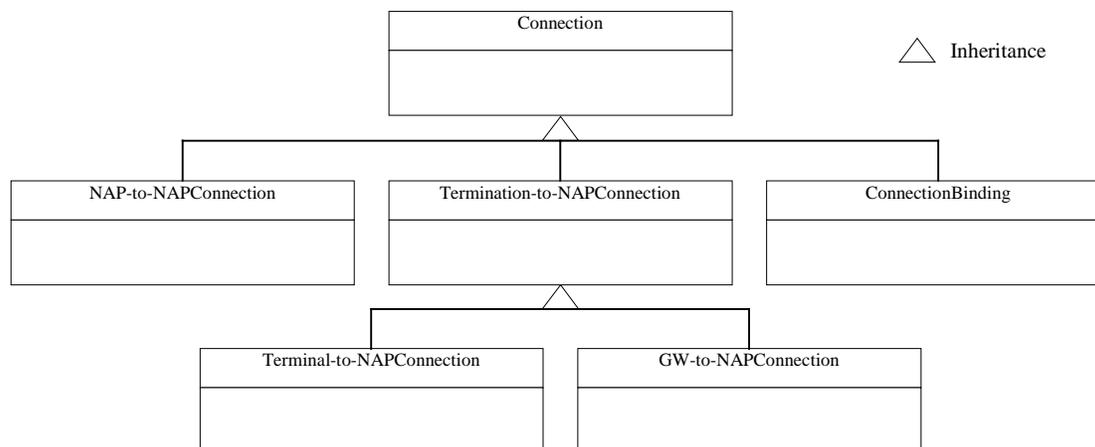


Fig. 12 Connection class hierarchy

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, adopting our approach is at the expense of service interoperability, but this is the price to pay for the competition among ever-hungry service providers

5.1 Hybrid Calendar Service

Today, several calendar applications are organized around a client/server model, where a user can, from an Internet host, create and manage calendar entries that are stored on a server. Such applications allow users to access their calendars from any terminal connected to the Internet, as well as expose and share portions of their calendars with other users. In this section, we consider a hybrid calendar service that extends and enriches this paradigm with the following features:

- Users can access their calendars from a wired or wireless PSTN terminal.
- The calendar system can report events to the users through a PSTN or cellular terminal, thus allowing users to associate alert notifications with calendar entries.
- The hybrid calendar service can in turn deliver an alert to the user via email, telephone or pager messaging. Users can control the time and frequency of the alerts (e.g., a few minutes before an important call, users may wish to have an alert be delivered to their mobile terminal).
- Users can create tele-conference call entries in their calendars.

A tele-conference call entry specifies the attendees and other conference parameters, such as the type of information to be exchanged (voice, video, data), the provider, the call-in number, and pass-codes. The calendar system provides automatic and transparent tele-conference call setup for the user. For example, if the user cuts and pastes a tele-conference call entry to a different day, the calendar system will take care of contacting the service provider, canceling the original reservation and creating a new reservation.

The hybrid calendar service can be realized as described in Fig. 13, which is a specialization of Fig. 3. The Java Service Layer runs a set of distributed components that implement the hybrid calendar service by controlling an IP-based Calendar Server, as well as the SCP, a Conference Bridge and an Interactive Voice Response (IVR) device. In IN parlance, the last two elements are called intelligent peripherals. The service layer plays the role of a liaison that permits the provision of the Calendar service across the PSTN and IP networks. The interaction between the service layer and the PSTN could recall the model promoted by the PINT group. In our approach we go one step further, i.e, we allow the SCP to generate requests toward the service layer. Whereas the PINT model focuses on enabling the PSTN to take service requests from IP 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.

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.

When users wish to access their calendar from a wired or wireless PSTN terminal, they trigger an event through the SCP by dialing a special prefix. The SCP propagates the event to the `scpCalendarService` component after adding the service parameters needed (e.g., calling number and service invoked). The component `scpCalendarService` then instructs `ivrCalendarService` to connect the IVR device with the user terminal. Through an interaction between `ivrCalendarService` and `csCalendarService`, the service logic will authenticate the user, retrieve the calendar records and, through a text-to-speech engine, read out the calendar to the user.

Another feature of this service allows users to automatically be connected to an active conference call specified in their calendar entries. The service subsystem `scpCalendarService` can instruct `csCalendarService` to contact the calendar server and retrieve the conference call parameters (i.e., pass-code and call-in number). Afterwards, `scpCalendarService` contacts the conferencing bridge via `cbCalendarService` and instructs the SCP to transfer the user connection to the conferencing bridge. Through the `scpCalendarService` and `cbCalendarService` interaction, the calling user is authenticated to the bridge, introduced and connected with the other participants in the conference call.

The alert message service can be implemented using the same paradigm. An event is generated by the calendar server at a scheduled time and is propagated to `csCalendarService` which then collects the alert parameters (e.g., the address where the alert must be sent, as well as the alerting channel that might be an email, a telephone call or a paging message). Then, `csCalendarService` contacts `scpCalendarService` and `ivrCalendarService`. The SCP is instructed to establish a connection between the user terminal and the IVR device and have this device play out a voice message to the user, for instance.

To create the Calendar service, the developer has to specialize the generic information flow diagram for the relevant elements in the service system infrastructure. In the light of the description given above, the main elements that need to be tuned for the hybrid calendar service are the SCP, IVR, CB and CS. Therefore, the provider must specialize the events and service components outlined in Section 4 for each of these elements. Describing all these events and service components for such a complex service as the hybrid calendar service will hinder the conciseness and readability of this article. The motivation for discussing the hybrid calendar service is to show the suitability of our approach for the implementation of sophisticated services. Therefore, we have chosen a well-known and simpler example (Call Forwarding) to

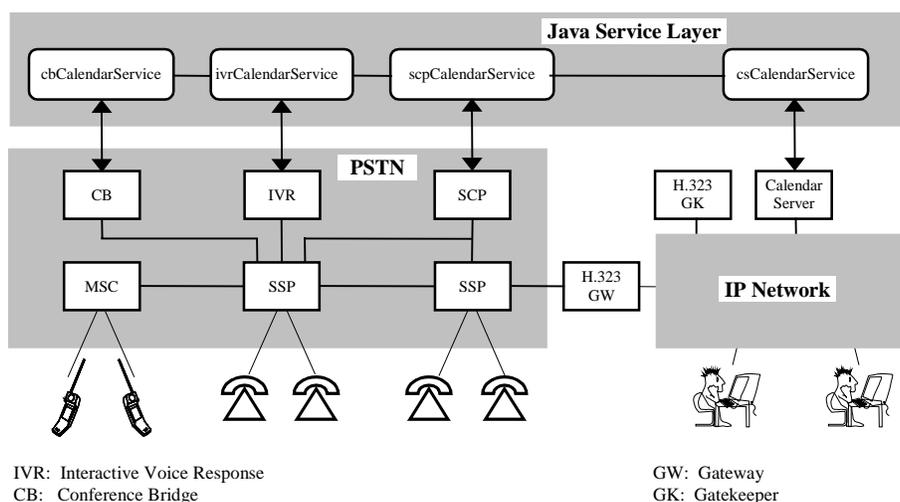


Fig. 13 Provision of hybrid calendar services across the PSTN and the Internet

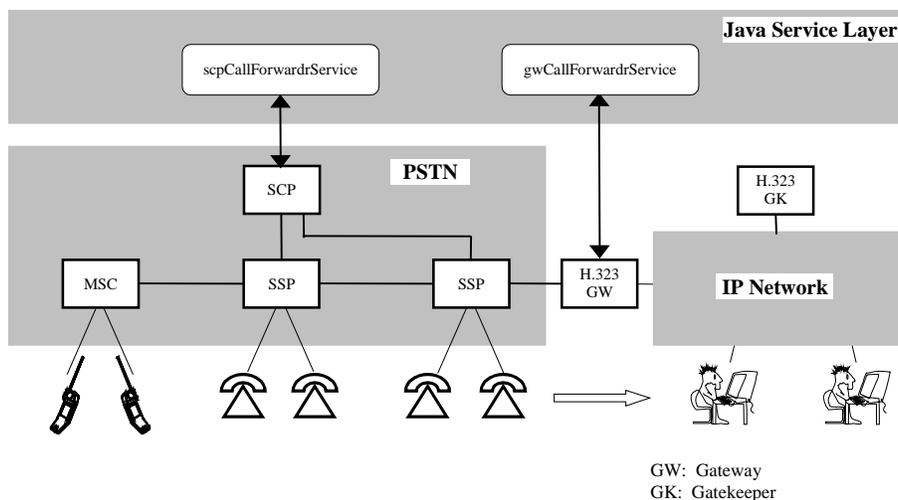


Fig. 14 Provision of Call Forwarding across the PSTN and the Internet

apply all the concepts introduced in Section 4.

5.2 Call Forwarding across the PSTN and the Internet

Hybrid Call Forwarding allows a customer to forward his incoming calls (addressed to his telephone set from any other telephone set) to a computer connected to an IP network. The service is realized as described in Fig. 14, which is a specialization of Fig. 3. The Java Service Layer hosts two distributed components, `scpCallForwardService` and `gwCallForwardService`, which implement the Call Forwarding service by controlling the SCP and the H.323 gateway, respectively.

To implement the logic corresponding to Call Forwarding, we use a subclass of `HybridService` named `CallForwardService` (Fig. 15). 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`).

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 run the corresponding logic and provide back the appropriate instructions. The SCP fires a trigger event toward the Java Service Layer. The `scpCallForwardService` 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 toward 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 IP address where the customer's H.323 terminal can currently be reached. A connection in the IP network is established from the gateway to this address; this connection is, later on, bound to the connection segment within the PSTN.

Creating a new service results from the specialization of the generic information flow diagram (Fig. 8) for the relevant elements in the service system infrastructure (Fig. 14), i.e., the SCP and the H.323 gateway. Therefore, the provider must specialize the events and service components identified 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 informs 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

```

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

    public CallForwardService(TimedAddress redAddr[]);
    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);
    /* fires TimeFormatInvalidEvent
     * and TimeFormatOKEvent
     */

    public deleteTimedAddress();
    public void getAddressAvailability(Time start, Time end, int timezone);
};

```

Fig. 15 Definition of CallForwardService and TimedAddress

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 gateway; 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 gateway, a trigger will be detected and an event fired by a process within the gateway (this process is of type TriggerEventSource). Then, gwCallForwardService will catch and handle the event fired. gwTimedAddress contains the address of the computer to which calls are to be forwarded.

6 Conclusion

A few years ago, the ATM technology was poised to replace all 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 IP networks led many people to believe that these networks 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 an appropriate migration path is precisely the purpose of this article. 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 IP, 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 many service creation architectures, our approach leverages on the existing networks and mechanisms to smoothly introduce new and advanced 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, gateways 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 controller running a Java virtual machine and equipped with a set of pre-developed service components realized according to the JavaBeans model. These service components provide the building blocks that can be used to implement hybrid services.

We illustrated the power of our architecture by constructing two examples of hybrid services: hybrid calendar and call forwarding. We showed how the service components defined in our architecture could be specialized for these services. The components used for the Calendar access service were specified with a coarse granularity, whereas those for Call Forwarding were specified with a higher level of details.

We are currently implementing the concepts proposed in this paper. We started by implementing the Call Forwarding Service on top of simulated hardware elements, using simple mechanisms capable, for example, of detecting special patterns in the numbers dialed by users. We are now in the process of porting the service components software onto a real test-bed composed of a PBX, an H.323 gateway, an H.323 gatekeeper and a LAN. Other services such as Voice-access-to-Email and Voice-access-to-Web as well as the hybrid calendar service are also being deployed on our test-bed.

While in this article 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 component `HybridService` into finer-grained components. We expect the approach presented in this article to play an important role in the highly competitive business of service provisioning [41].

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. Elena Alonso is acknowledged for her contribution to the implementation of some of the concepts presented in this article.

References

- [1] T. A. Anschutz, A Historical Perspective of CSTA, *IEEE Comm. Mag.*, 34(4), April 1996, pp. 30-5.
- [2] <http://www.javasoft.com/beans/>
- [3] R. Braden, D. Clark and S. Shenker, Integrated Services in the Internet Architecture : An Overview, IETF Informational RFC 1633, June 1994.
- [4] J. P. Hubaux, C. Gbaguidi, S. Koppenhoefer and J. Y. Le Boudec, The Impact of the Internet on Telecommunication Architectures, *Computer Networks*, Special Issue on Internet Telephony, 31(3), Feb. 1999, pp. 257-73.
- [5] P. Cronin, An Introduction to TSAPI and Network Telephony, *IEEE Comm. Mag.*, 34(4), April 1996, pp. 48-54.
- [6] OMG, The Common Object Request Broker: Architecture and Specification, Rev. 2.0, July 1995.
- [7] OMG Document: telecom/97-12-06, Interworking between CORBA and Intelligent Network Systems, Request for Proposals, Dec. 1997.
- [8] OMG Document: telecom/98-05-xx, Request for Information: Supporting Wireless Access and Terminal Mobility in CORBA.
- [9] H. D'Hooge, The Communicating PC, *IEEE Comm. Mag.*, 34(4), April 1996, pp. 36-42.
- [10] European Computer Manufacturers Association (ECMA), Computer-Supported Telecommunications Applications, ECMA Technical Report TR/52, June 1990.
- [11] I. Faynberg, L. R. Gabuzda, M. P. Kaplan and N. J. Shah, *The Intelligent Network Standards, their Applications to Services* (McGraw-Hill, 1996).
- [12] I. Faynberg, M. Krishnaswamy and H. Lu, A proposal for Internet and Public Switched Telephone Networks (PSTN) Interworking, Internet Draft, March 1997.

- [13] I. Faynberg et al., The Development of the Wireless Intelligent Network and Its Relation to the International Intelligent Network Standards, *Bell Labs Technical Journal*, 2(3), Summer 1997.
- [14] <http://www.javasoft.com>
- [15] ITU-T, Universal Personal Telecommunications (UPT) – Service Description, Rec. 851, Feb. 1995.
- [16] <http://www.geoplex.attlabs.net/>
- [17] 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.
- [18] ITU-T, *Packet-based Multimedia Communications Systems*, Rec. H.323, 1998, Geneva, Switzerland.
- [19] M. Krishnaswamy, PSTN-Internet Interworking- An Architecture overview, Internet Draft, Nov. 1997.
- [20] ITU-T, General Aspects of Services in ISDN, Rec. I.210, 1988.
- [21] J. de Keijzer, Intelligent Agents and Java Advanced Intelligent Network architecture (JAIN), In Proc. of IATA '98. <http://sokrates.cs.tu-berlin.de/deutsch/news/tagungen/IATA98/main.html>
- [22] Sun Microsystems, The Java Telephony API- An Overview, White Paper, Vers. 1.1, Jan. 1997.
- [23] H. Jubin (ed.), *JavaBeans by Example*, (Prentice Hall, ISBN 0-13-790338-3, 1998).
- [24] 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.
- [25] C. Low, Integrating Communication Services, *IEEE Comm. Mag.*, 35(6), June 1997, pp. 164-9.
- [26] A. R. Modaresi and R. A. Skoog, Signaling System No. 7 : A Tutorial, *IEEE Comm. Mag.*, July 1990, pp. 19-35.
- [27] T. Magedanz and R. Popescu-Zeletin, *Intelligent Networks : Basic Technology, Standards and Evolution*, (Intl. Thomson Computer Press Ed., 1996).
- [28] 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.
- [29] ITU-T, Introduction to the Intelligent Network Capability Set 1, Rec. Q.1211, March 1993.
- [30] Sun Microsystems, Inc., The SunXTL Platform, White Paper, Feb. 1995, available as <http://www.sun.com/products-n-solutions/sw/SunXTL/whitepaper.ps>
- [31] Microsoft, Inc., IP Telephony with TAPI 3.0, White Paper, 1997.
- [32] G. A. Thom, H.323 : The Multimedia Communications Standard for Local Area Networks, *IEEE Comm. Mag.*, 34(12), Dec. 1996, pp. 52-6.
- [33] TINA-C, Service Architecture, Vers. 5.0, June 1997.
- [34] Wireless Application Protocol (WAP) Forum, Wireless Application Protocol – Wireless Markup Language Specification, Draft, 20 March 1998.
- [35] ITU-T, Basic Reference Model of Open Distributed Processing – Part 1: Overview and Guide to Use, Rec. X.901, 1994.
- [36] Deutsche Telekom and France Télécom, IN Access to TINA services and Connection management (IN-TINA Adaptation Unit), Vers. 1, March 1, 1999 (<http://www.tinac.com>).
- [37] Lucent Technologies, Inc., Proposal for IN-TINA interworking, March 5, 1999 (<http://www.tinac.com>).
- [38] Alcatel, Answer to the RFP TINA-IN Adaptation Unit, March 5, 1999 (<http://www.tinac.com>).
- [39] F. Anjum, F. Caruso, R. Jain, P. Missier and A. Zordan, ChaiTime: A System for Rapid Creation of Portable Next-Generation Telephony Services Using Third-Party Software Components, In Proc. of the 2nd IEEE Conference on Open Architectures and Network Programming (OPENARCH), New York, USA, March 1999.
- [40] T. M. Chen and A. Jackson (eds.), *IEEE Network Magazine*, Special Issue on Active and Programmable Networks, 12(3), May/June 1998.
- [41] J. P. Hubaux and D. Nagel (Eds.), *IEEE Comm. Mag.*, Feature Topic Issue on the Provision of Communication Services over Hybrid Networks, July 1999.