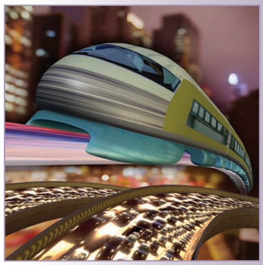

Porting the Session Initiation Protocol to IPv6

A modified Java-based SIP implementation offers one approach to developing multimedia services that run on both IPv4 and IPv6 networks.

Tomás Robles, Ramiro Ortiz, and Joaquín Salvachúa
Technical University of Madrid

The transition from IPv4 to IPv6 will be a slow evolutionary move enabled by the middleware and software development tools to develop new applications, especially for mobile users. An important driver for such new Internet services could come from combining the session initiation protocol (SIP)[1] — used in multimedia sessions — with Java, which is popular for developing portable code and wireless applications. To date, however, few developers have done much work in porting Java programs to IPv6.

In an attempt to produce software that works over both IPv6- and IPv4-enabled computers, we ported the code from a publicly available, Java-based SIP library to IPv6. Like any ongoing development, the original JSIP library contained errors and did not cover all the features that building SIP elements requires. The associated documentation was also scarce.

This article describes our experiences in porting SIP to IPv6. To deploy a service in which IPv6 nodes communicate with IPv4 nodes, we selected an IPv4–IPv6 transition mechanism that combines an application-level gateway (ALG) with network address translation—protocol translation (NAT-PT)[2] for application traffic. Our solution required us to make several modifications to the original JSIP library. Modifications where required for supporting both types of IP addresses, and we had to modify the application payload to translate the IP addresses when required for IPv6 to communicate with. The final code can be found on our Web site (www.dit.upm.es/~robles).

The Session Initiation Protocol

SIP is an application-layer control protocol that is used to establish, modify, and terminate multimedia sessions such as Internet telephony calls. SIP can also be used for inviting participants to existing sessions, such as multicast conferences. SIP transparently supports name mapping and redirection services, which in turn support personal mobility[3] by allowing a user to maintain a single externally visible identifier, regardless of network location.

SIP is used for peer-to-peer communications — that is, both parties in a call are considered equals; there is no master or slave. Similar to HTTP, however, SIP uses a transaction model in which a SIP client generates a request and a SIP server generates a response. During a session, a SIP end point will typically switch between being a client and a server, depending on whether it is initiating or responding to a request.

There are three main elements in a SIP network. *User agents* (UAs) are the end devices that originate SIP requests to establish media sessions and send and receive media. *Servers* are intermediary devices that assist user agents in session establishment and other functions. *Location servers* provide information about a caller's possible location.

There are several types of SIP servers:

- A *SIP proxy* receives SIP requests from a UA or another proxy and forwards the request to another location. A request can traverse several proxies on its way to a UA.
- A *redirect server* receives a request from a UA or proxy, maps the address into zero or more new addresses, and returns a redirect response with these addresses to the client.
- A *register server* receives SIP registration requests and updates the user agent's information in a location server or another database.
- A *user agent server* (UAS) is a logical entity, a server application that contacts the user when a SIP request is received and returns a response on behalf of the user. This role lasts the duration of one transaction.

SIP is part of the overall IETF multimedia data and control architecture, which currently incorporates protocols such as the resource reservation setup protocol (RSVP), the real-time transport protocol (RTP), the real-time streaming protocol (RTSP), the session announcement protocol (SAP), and the session description protocol (SDP). Although, SIP's functionality and operation do not depend on any of these other protocols, SIP is usually used for negotiating and carrying information related to all of them.

The JSIP Project

JSIP is an open-source implementation that provides a library of basic Java classes for implementing SIP.[4] Compared to other alternatives, JSIP provides two key advantages:

- *Availability*. As an open-source library, JSIP offers free access to the completely Java-based code, which is unique because most available SIP implementations are based on C/C++; meanwhile, in line with our work, many emerging SIP implementations are based on Java.
- *Independence*. JSIP is not associated with any proprietary application, and its Java-based code makes it portable to different platforms. Nevertheless, there is no IPv6 support for Java JDKs for Windows, which jeopardizes development of programs for this platform.

From a functional point of view, we can identify two main classes in the JSIP library: those related to analyzing and processing SIP elements and those related to creating UAs. The first group of classes is organized around `SipMessage` and represents the core of the library. Figure 1 shows the main classes related to message processing. This group of classes is related to analysis and processing of syntactic elements of the SIP protocol and deals with IP addresses contained on SIP messages. The `SipMessage`, `SipRequestMessage`, and `SipResponseMessage` deal with requests and responses. `SipRequestMessage` derives a set of classes, each corresponding to a SIP message. Classes surrounded by ovals indicate those we added during our work. Management of IP addresses should be performed by the SIP elements build over the library, not by the library itself. Toward that end, we provided a mechanism for allowing transparent representation, management, and storage of IP addresses independently of their type (IPV4 or IPv6). We reached this objective by defining the abstract class `IpAddress`, which is used for deriving `IpAddress4`, `IpAddress6`, and `UnknownAddress`. Then using Java's object-oriented functions, we can manage IP addresses independently of their specific type. Only when specific processing is required is the exact type investigated.

A SIP message is composed of other structures, mainly headers and identifiers. The `SipUri` class, which we have added to our improved library, represents an address or SIP user identifier, and can be defined with the following simplified structure:

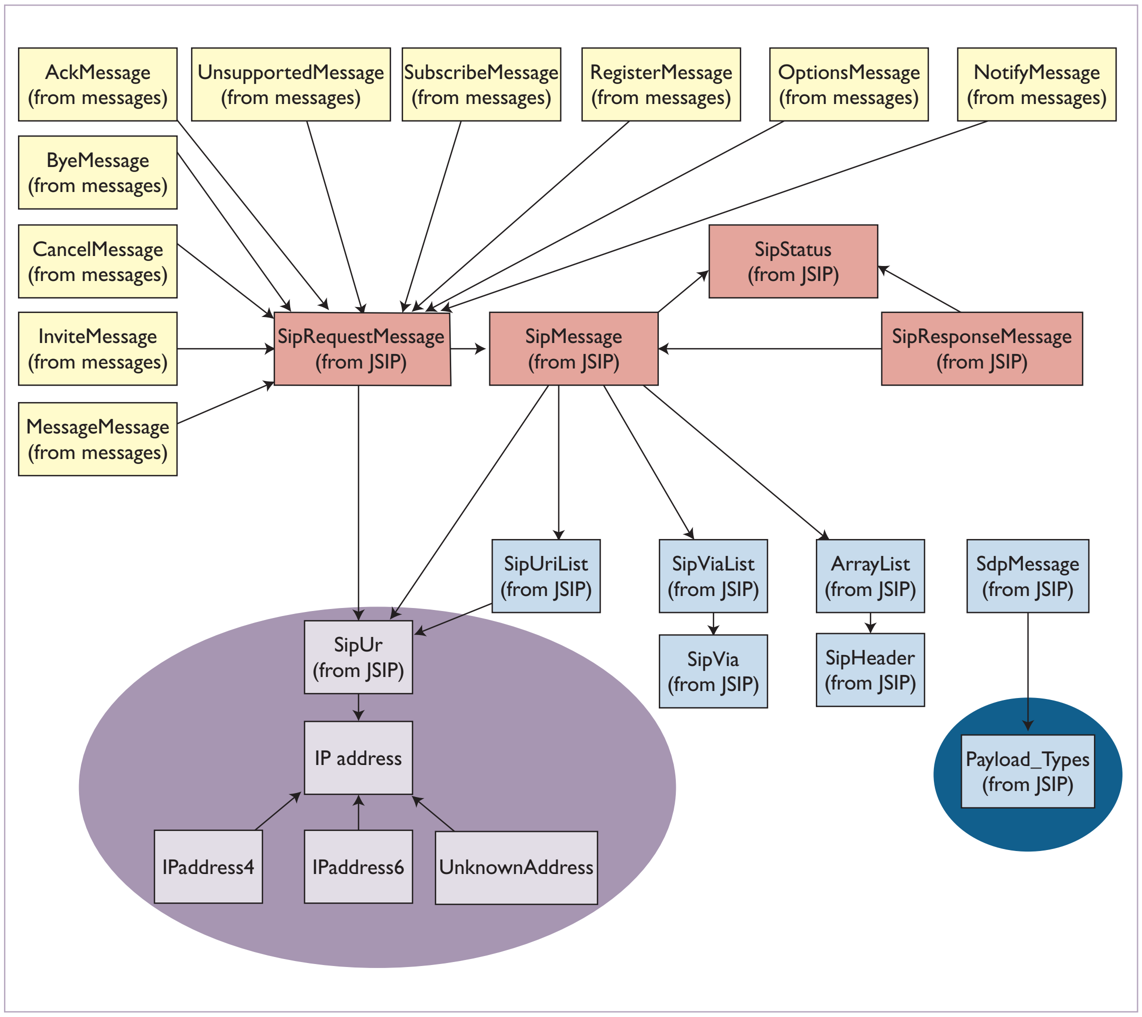
AckMessage (from messages)
UnsupportedMessage (from messages)
SubscribeMessage (from messages)
RegisterMessage (from messages)
OptionsMessage (from messages)
NotifyMessage (from messages)
ByeMessage (from messages)
CancelMessage (from messages)
InviteMessage (from messages)
MessageMessage (from messages)
SipStatus (from JSIP)
SipRequestMessage (from JSIP)
SipMessage (from JSIP)
SipResponseMessage (from JSIP)
SipUriList (from JSIP)
SipViaList (from JSIP)
ArrayList (from JSIP)
SdpMessage (from JSIP)
SipUr (from JSIP)
SipVia (from JSIP)
SipHeader (from JSIP)
IP address
Payload_Types (from JSIP)
IPaddress4
IPaddress6
UnknownAddress

Figure 1. JSIP class message-processing diagram. This figure shows the main classes of the JSIP library organized around `SipMessage`*, and identifies key classes added to produce a generic SIP library that runs on both IPv6 and IPv4 networks.*

```
'user name"
<sip:user_id@hostname;parameters>'
```

in which `hostname` is the name of the user host that a literal IP address or domain name can represent. In the original JSIP library, this information is stored as a string value, which does not offer any high-level facilities for running over different IP networks.

Our experiences show that the original string mechanism can contain IPv6 addresses; other functions fail when trying to manage those addresses in the correct context. Extensive modifications were required for adapting to IPv6, lacking any methodology for supporting them now and in the future. Therefore, we introduced the new class structure instead of the original string value. `SdpMessage` represents the description of a multimedia session using SDP.[5] Jointly with `Payload_Type`, it provides support for managing SDP. The main class for supporting UA creation is `SipClient`, which offers the required characteristics for the UA clients (UAC — a client application that initiates the SIP request), as well as for UASs.

Porting JSIP to IPv6

Java 1.4.0/1 lets the user create connections transparently to both IPv4- and IPv6-enabled nodes. In most usual services development, we do not need to modify legacy applications for classes that deal with IP communications because of the built-in support this version of Java provides. Java 1.4 uses the abstract Factory pattern, which provides a method for creating and managing generic IP addresses, independently of the type of network for dealing with different IP address types. Java 1.4.0/1 extends the generic IP address class InetAddress with two new classes, Inet4Address and Inet6Address, which represent IPv4 and IPv6 addresses, respectively. Thus, we do not need to specify the type of IP address we want to connect with. The variable java.net.preferIPv4Stack lets us set user and application preferences for how to handle addresses when we can reach a host name using multiple IPv4 and IPv6 addresses. The end application developers have to specify their preferences only once; after that, Java automatically and transparently selects a suitable IP address.

Despite Java's flexibility with regard to IPv6 addressing, we had to modify the JSIP library to handle some major issues in order to properly support IPv6. We divided this work into four steps — bug fixing, SIP feature completion, migration to IPv6, and IPv4–IPv6 interoperability.

Table 1 summarizes the components we created for porting SIP to IPv6 and the resources required for completing each component. The first column represents the different elements handled during the porting process:

- original JSIP library (JSIP at http://jsip.mitre.org);
- debugged and enhanced JSIP library (JSIP+ at jsip.mitre.org), which we created by improving the original JSIP library, fixing bugs, and adding some missing key facilities;
- JSIP version enabled for working over IPv4 and IPv6 (JSIP IPv4-6);
- test suite for validating the JSIP code;
- SIP proxy for performing SIP gateway functions;
- UA based on the JSIP library; and
- videoconference application used for validating the global service.

The second column details the packages provided for each element, which shows the relevance of each part of the library (events management, messages management, proxy classes related, and general facilities provided by Util and JSIP). Columns 2 through 5 show the number of classes, methods, and lines for each package; column 6 shows the hours required for designing and coding each element; and the last column provides additional information for understanding previous columns.

After analyzing the problems of porting a JSIP library, we dealt with the deployment of a SIP service over heterogeneous transports based on the library we created.

Bug Fixing and Implementation Enhancement

We began the project by studying the original JSIP library. Our first work was the creation of a test suite with 160 test cases for direct UA-to-UA communication, UA registration on a register server, and tests for evaluating the implementation against incorrect SIP messages. Four major defects (parsing incorrect messages, parsing SIP extension, managing SDP messages, and SipRequestMessage management) caused approximately 30 modifications. In addition to the 10 hours of coding and design for the test suite, debugging the detected errors required another 20 hours, which we included during 60 working hours under the JSIP+ version (see Table 1).

JSIP provides a group of classes for managing SDP bodies embedded in SIP messages, which is organized around the SdpMessage class. We enhanced and adapted this set of classes to provide IPv6 support, as well, by providing code for dealing with IPv4 and IPv6 addresses in SDP messages. We added a Payload_Type class to describe the flow of multimedia in the SDP description body.

Modifications for IPv6

A driving idea of our work from the very beginning was to provide high-level support based on key features of object-oriented software, instead of just performing thousands of isolated modification on the original code using the if statement. We modified four main elements in the JSIP library to adapt them to work with IPv6:

- *SIP URIs.* JSIP originally managed URIs as strings. To simplify IP address management, we substituted both IPv4 and IPv6 addresses for a set of classes organized around the new SipUri class (see Figure 1), which is based on the IpAddress class. SipUri class derives the three main classes we will use: IpAddress4 for IPv4 addresses, IpAddress6 for IPv6 addresses, and UnknownAddress for nodes that IPaddress has not yet resolved. The IpAddress stores IP addresses in SipUri independently of the specific address type. When a UA or SIP proxy needs to discover the specific IP address type, it can use standard Java methods (InstanceOf) to determine if the address is IPv4, IPv6, or still unresolved. Meanwhile, the application logic where the IP addresses are not used or modified can be easily coded, managing the

Table 1. Code structure and effect.

Element	Internal packages	Classes	Methods	Lines of code	Effort (hours)	Comments
JSIP	Event	3	14	262	60	Includes 20 hours for fixing bugs and 40 hours for enhancing the original JSIP library.
	Proxy	5	17	414		
	Util	6	23	554		
	JSIP	29	490	10,736		
		52	606	13,639		
JSIP IPv4-v6	Event	3	14	262	10	Time spent providing IPv6 support.
	Messages	9	45	693		
	Proxy	5	17	414		
	Util	6	23	554		
	JSIP	32	499	11,296		
		55	615	14,211		
	Messages	9	45	681		
	JSIP	28	456	9,080		
		51	557	10,991		
Test suite	Test	3	17	992	10	Time dedicated to designing and coding the test cases; another 10 hours went to learning Junit testing tool.
	User agent	1	39	1,425	40	30 hours for creating graphical interface with 6,878 lines, 185 methods, and 12 classes.
	JSIP proxy	4	30	928	5	

abstract class `IpAddress`.

- *SIP messages*. Socket connections receive SIP messages, which must be parsed to identify each message's corresponding elements. The JSIP library provides several classes (`SipMessage`, `SipUri`, and `SipVia`, for example) that perform such parsing, but we enhanced these classes to allow parsing of strings of any IPv6 addresses included in SIP messages.
- *Enhanced SDP support*. The original `SdpMessage` class can manage IPv4 addresses only when they are included in the SDP body. We modified such classes to allow them to manage IPv6 classes as well. In this case, we reused the `IpAddress` class instead of the original String attribute of the `SdpMessage` class (see Figure 1).
- *UA support*. The classes JSIP provides for supporting UA creation must also manage IP addresses. The same code might work over different IP stacks, so the UA must identify which IP address to use depending on the underlying network; it then fills in the corresponding field with this information. We modified the class `SipClient` by adding code for detecting the stack type and generating the proper IP address for completing the connection.

These modifications concentrate on a few classes (`SipUri`, `IpAddress`, and `SdpMessage`) and the code that deals with the type of IP network inside the JSIP library.

Deploying SIP Over Heterogeneous Networks

To validate our approach with the modified JSIP library's IPv4 and IPv6 capabilities, we created a test scenario (shown in Figure 2) with a simple audio-stream application that sent previously captured audio over both IPv4 and IPv6 networks using RTP packets.

This let us test the feasibility of our proposed approach without dealing with the complexity of full-flagged multimedia applications. Using the JSIP library to build SIP elements facilitated this task because the SIP elements created with this library work in both network types. Nevertheless, when SIP elements are located on different types of networks, we need a suitable gateway mechanism to allow UAs to interconnect that cannot connect directly due to the underlying networks' incompatibility.

Figure 2 illustrates the test scenario, in which the streaming application client located in an IPv6 network connects to a server in an IPv4 network.

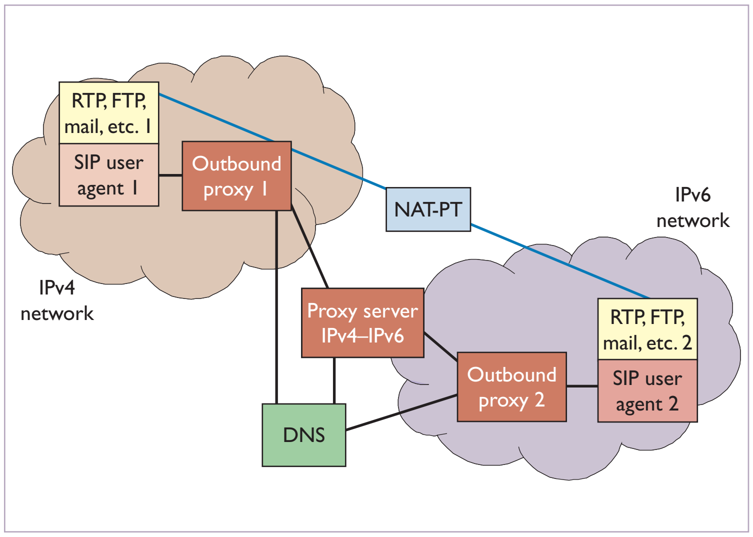

Figure 2. Interworking infrastructure. This figure presents a future scenario in which end users on different network types cannot communicate directly. Our JSIP library aims to facilitate transparent communication using a standard SIP element build without any code modifications.

We used a NAT-PT service located between IPv4 and IPv6 networks to direct the data stream between client and server.

The translator allows IPv6 hosts to communicate with IPv4 hosts. An example of such application is NAT-PT.[7] A solution for a protocol in an IP address embedded in the payload is to use an ALG, which modifies the application payload and performs other necessary functions to make the application work.

In this scenario, we configured the NAT-PT by hand, which let us account for the specific application used for the evaluation. To simulate a realistic scenario, however, we defined outbound proxies at each network and used a third, based on a dual-stack node, to interconnect both outbound proxies and provide SIP gateway functionality. In our scenario, taking advantage of our JSIP library's facilities, configuration files perform proxy and UA configuration, so in this scenario, the code should not be modified to work over the underlying network.

Finally, when the session negotiation ends, the multimedia flows are interchanged using NAT-PT facilities. Other interconnection strategies can be applied at the application layer independently of the SIP proxy.

Implementing a SIP UA

We created a user interface for facilitating UA evaluation and integration into the scenario. We used this user interface for informal evaluation of the whole scenario before integration with the audioconference application. The resulting code has several key characteristics:

- We can configure it using standard SIP mechanisms.
- It can run on IPv4 and IPv6 networks.
- Error sources were reduced because we did not introduce code that was specific to IPv4 or IPv6.

The resulting UA is portable and configurable without working on the code during any specific deployment.

Implementing a SIP Proxy

As mentioned earlier, SIP proxies are elements that route SIP requests to UASs and SIP responses to UACs. A request can traverse several proxies on its way, and each proxy makes routing decisions, modifying the request before forwarding it to the next element. Responses are routed in reverse order through the same set of proxies the request traverses. SIP defines two types of proxies:

- A *statefull* proxy is purely a SIP transaction-processing engine. Its behavior is modeled in terms of the server and client transactions, which are registered for controlling future message interchanges.
- A *stateless* proxy is a logical entity that does not maintain the client or server transaction state machines defined in this specification when it processes requests.

SIP proxies are common elements in SIP services, and administrators can use standard SIP mechanisms to configure them for all UAs in a network.

To provide a SIP gateway, we need a SIP network component that performs a forwarding task between IPv4 and IPv6 without modifying the logic of the end-to-end negotiation. Stateless proxies include such functionality.

Table 2 summarizes statefull and stateless proxy functionalities and those that the JSIP proxy implements, showing which facilities are required (✓) and which are not (✗) for each type of SIP proxy, and for the gateway proxy. As the table shows, a gateway proxy's requirements are a subset of the requirements of the standard stateful proxy. Thus, any properly configured stateful proxy build using our JSIP library will serve as gateway proxy. In our scenario, application-level

Table 2. Statefull and stateless proxies' functionalities. The ✓ indicates which facilities are required, and the ✗ indicates those that are not required.

Function	Stateful	Stateless	Gateway Proxy
Process request (basic functionality)			
1. Validate the request	✓	✓	✓
1.1 Check reasonable syntax	✓	✓	✓
1.2 Check URI scheme	✓	✓	✓
1.3 Process 'Max-forwards' header	✓	✓	✓
1.4 Loop detection	✓	✓	✓
1.5 Process 'Proxy-require' header	✓	✓	✗
1.6 Process 'Proxy-authorization' header	✓	✓	✗
2. Preprocess routing information	✓	✓	✓
2.1 Process 'Route' header	✓	✓	✓
2.2 Process 'Record-Route' header	✓	✓	✓
3. Determining request target	✓	✓	✓
4. Request forwarding	✓	✓	✓
4.1 Update the request-URI	✓	✓	✓
4.2 Postprocess routing information (local routing policy)	✓	✓	✓
4.3 Add a 'Via' header field value	✓	✓	✓
4.4 Add a 'Content-length' header field if necessary	✓	✓	✗
4.5 Forking proxy: parallel or sequential	✓	✗	✗
4.6 Set timer C	✓	✗	✗
Process response (basic functionality)			
1. Find the appropriate response context	✓	✗	✗
2. Update Timer C	✓	✗	✗
3. Remove the topmost Via	✓	✓	✓
4. Choose the best final response	✓	✗	✗
5. Aggregate authorization field values if necessary	✓	✓	✗
6. Optionally rewrite 'Record-route' header field values	✓	✓	✓
7. Forward the response	✓	✓	✓
8. Handle transport errors	✓	✓	✓
9. Generate any necessary 'Cancel' request	✓	✗	✗

interconnection can be performed by applying any of the IPv4-IPv6 transition mechanisms defined by the IETF.

Further work might entail coordinating SIP and applications strategies, taking into account information interchanged by SIP such as the SDP primitives used for Session Description. Adding extra functionalities to the proxy server lets it look into this information interchanged by SIP and coordinate with ALGs.

Conclusions

Despite the problems detected during this work, Java offers many porting facilities that may be used for running code over IPv6 and/or IPv4. Java 1.4.0 allows the user to transparently create connection over both IPv4 and IPv6. The object-oriented paradigm made it easy for us to create the `SipUri` and related classes, which facilitate the management of IP addresses independently of their realm. The use of patterns,[6] as in Java 1.4, simplifies the generation of code that could work over IPv4 and IPv6 networks. This leads to clearer code, in which specific sentences are concentrated on clearly identified classes, and methods avoid the use of `if` statements common to other methodologies.

The use of an existing, open-source, mature, and well-structured library facilitated our work and allowed us to work efficiently, concentrating on our porting problem.

Acknowledgments

The Euro6IX project (IST-2001-32161) partially supported the work described in this article.

References

1. J. Rosenberg et al., "SIP: Session Initiation Protocol," Internet Eng. Task Force RFC 3261, June 2002; www.ietf.org/rfc/rfc3261.txt.
2. G. Tsirtsis and P. Srisuresh, "Network Address Translation—Protocol Translation (NAT-PT)," Internet Eng. Task Force RFC 2766, Feb. 2000; www.ietf.org/rfc/rfc2766.txt.
3. R. Pandya, "Emerging Mobile and Personal Communication Systems," *IEEE Comm. Magazine*, vol. 33, no. 3, 1995, pp. 44–52.
4. M. Handley et al., "SIP: Session Initiation Protocol," Internet Eng. Task Force RFC 2543, Mar. 1999; www.ietf.org/rfc/rfc2543.txt.
5. M. Handley and V. Jacobson, "SDP: Session Description Protocol," Internet Eng. Task Force RFC 2327, Apr. 1998; www.ietf.org/rfc/rfc2327.txt.
6. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.

Tomás Robles is an associate professor of protocol engineering at the Technical University of Madrid. He got his PhD in Telecommunication Engineering at Technical University of Madrid. His research focuses on service provision, wireless networks, and IPv6. He participates in several EU projects, such as BRAIN/MIND (concerning service provision over wireless networks), and Euro6IX in the development of new services using SIP. Contact him at robles@dit.upm.es.

Joaquín Salvachúa is an associated professor of protocol engineering at the Technical University of Madrid. He received his PhD in telecommunication engineering at Technical University of Madrid. His interests are P2P, SIP, HCI, and tele-education. He currently works on new semantic Web applications and is involved in several EU projects. Contact him at jsr@dit.upm.es.

Ramiro Ortiz is studying telecommunications at the Technical University of Madrid. Currently, he is finishing his project work on service provision with SIP and interning with IBM Spain. Contact him at rortiz@dit.upm.es.

A LEADING JOURNAL ON TECHNOLOGY IN **MEDICINE, BIOLOGY,** AND **HEALTH CARE** NOW HAS A NEW SPONSOR!

The IEEE Computer Society has joined the IEEE Engineering in Medicine and Biology Society in delivering *IEEE Transactions on Information Technology in Biomedicine*.

Peer-reviewed articles feature topics such as

- **biomedical engineering**
- **virtual reality applications for surgery**
- **visualization and biomedical imaging**
- **ethical issues in biomedical applications**
- **information infrastructures in health and medicine**
- **high performance biomedical computing**
- **biotechnology**
- **broadband technologies in medicine**

IEEE Computer Society members can subscribe for the low member rate of $25!

Read more about *IEEE Transactions on Information Technology in Biomedicine*
http://www.ieee.org/organizations/pubs/transactions/titb.htm

IEEE EMB IEEE COMPUTER SOCIETY