This paper deals with the design of telecommunications services using the SIP protocol. We give an overview of SIP and an overview of the programming models CGI and Servlets together with their SIP-based extensions SIP-CGI and SIP-Servlets. Then, we show how we have created several SIP-based services using SIP-Servlets.

1 Introduction

With the increasing number of Internet users, Internet telephony (or IPtel) is more and more considered as a competitor to the classical circuit switch telephony. IPtel presents several advantages in comparison to circuit switch telephony, for example end user programmability, which is the possibility for end users to develop services. Therefore, IPtel promotes the development of a large variety of services. One of the main signaling protocols for IPtel is SIP [14, 15] and is inspired from HTTP [8, 3]. SIP is used for establishing and ending a call or media session, and for changing parameters of a current session.

CGI [13] and Servlets [6, 7, 9] are among the main programming models for the creation of HTTP-based services. The similarities between HTTP and SIP have led to the creation of SIP-CGI [11] and SIP-Servlets (or more briefly, Siplets) [5, 10] that are the respective extensions of CGI and Servlets for SIP. Such programming models have been developed to promote the creation of SIP-based services.

To acquire an experience on the creation of SIP-based services, we have designed and implemented several services using the Siplet technology. In this paper, we first give an overview of SIP and an overview of the programming models CGI and Servlets together with their SIP-based extensions SIP-CGI and Siplets. Then, we explain how several SIP-based services have been created using Siplets.

The rest of this article is structured as follows. In Section 2, we give an overview of the main features of SIP. Section 3 gives an overview of CGI and Servlets and of their SIP-based extensions: SIP-CGI and Siplets. In Section 4, we present four services we have created using Siplets and we discuss their realization. Section 5 highlights our contributions and proposes some future work.

2 Main features of SIP

SIP (Session Initiation Protocol) [14, 15] is inspired from HTTP [8, 3] and has been standardized by IETF (Internet Engineering Task Force). SIP is used for initiating and terminating an interactive call or media session, and for changing parameters of a current session. A session can involve voice and other types of media such as video and text. SIP uses SDP (Session Description Protocol) for defining and negotiating session parameters at either end-point of the communication. SIP supports name mapping and redirection functionality, and thus, permits user mobility.

2.1 SIP User Agents and SIP Servers

A typical SIP architecture consists of SIP UAs (User Agents) and SIP Servers. A SIP UA is associated to a user and consists of two parts:

UAC (UA Client) sends request messages to UASs and SIP servers, and receives response messages from UASs and SIP servers. INVITE is the most known request, which is used to initiate a session.

UAS (UA Server) receives request messages coming from UACs and proxies (the latter being defined below), and accepts or rejects them by sending response messages. Let us denote by UAC_X and UAS_X the UAC and UAS associated to a user X. There exist three types of SIP servers:

A registrar handles REGISTER request messages received from UACs. Each REGISTER request contains the current address of the UA that has sent it, and the registrar can then update its database of addresses.

A redirect server receives an INVITE request message from a UAC and replies back with a specific response message containing the right address where the INVITE request must be sent.

A proxy receives a request message and forwards it forth to another proxy or to a UAS. When a proxy receives a response message, it forwards it to the sender of the corresponding request (may be proxy or UAC). Therefore, a SIP message (request or response) can traverse several proxies before to reach its final destination (UAS or UAC).
2.2 SIP response messages

Each SIP response message (in reply to a request message) has a reply code ranging from 100 to 699 and a reason phrase. For brevity, a response message with a reply code RC and a reason phrase RP will be denoted “RC-RP” response message. As a simple example, when a UAS receives an INVITE (request) message, it may reply by sending sequentially the following three responses:

100-TRYING means that the UAS has received the INVITE and is taking some unspecified action on behalf of this request.

180-RINGING means the UAS has received the INVITE and is alerting the user.

200-OK means that the INVITE is accepted (e.g., the called user has picked-up).

A reply-code 1xx (from 100 to 199) indicates a response which will be followed by another response. This type of responses are qualified as provisional. For example, the above 100-TRYING and 180-RINGING are provisional.

A reply-code from 200 to 699 indicates a response which is not followed by another response. This type of responses are qualified as final. Among the final responses:

- a reply-code 2xx corresponds to a so-called positive response, and indicates that the corresponding request was successfully processed and accepted. For example, the above 200-OK is a positive final response.
- a reply-code 3xx corresponds to a response indicating that further action needs to be taken in order to complete the request;
- a reply-code 3xx, 4xx, 5xx or 6xx corresponds to a so-called negative response, and indicates that an error or failure has occurred.

2.3 Example of SIP session

Figure 1 represents an example of a SIP session involving redirect and proxy servers and two users Samy and Mazaya with their corresponding UAs. We assume that each user Samy and Mazaya has registered his current address by sending a REGISTER request to a registrar server. The latter has recorded the address in a location server. The following steps 1-15 describe an example of call process where a communication between Samy and Mazaya is initiated by Samy and terminated by Mazaya.

As Samy dials the address of Mazaya, his UAC sends an INVITE request for Mazaya (step 1) to the redirect server which verifies the address of Mazaya by contacting a location server and finds out that Mazaya has temporarily another address. So, the redirect sends back to UAC_samy a “302 MOVED TEMPORARILY” response message containing the temporary address (step 2). Consequently, UAC_samy acknowledges receipt of the 302 response by an an ACK request (step 3), and sends a new INVITE request for Mazaya (step 4) passing through a proxy server. The latter replies by a TRYING (step 5), adds its address to the via header field, decodes the address of Mazaya by contacting a location server, and forwards the INVITE message to UAS_mazaya (step 6). The latter sends back TRYING and RINGING responses to the proxy (steps 7-8), and the latter forwards the RINGING towards UAC_samy (step 9).

When Mazaya picks up the phone (acceptance), then her UAS sends a 200 OK response message (step 10) which is forwarded by the proxy to UAC_samy (step 11). The latter acknowledges receipt of the OK by sending an ACK request directly to UAS_mazaya (step 12). The ACK does not pass through the proxy, unless the message is forced to pass through the proxy by using a specific header called the record routing header. Samy and Mazaya have established a session and can communicate with each other using some other protocols such as RTP (step 13). When Mazaya decides to hang up and end the call, her UAC sends a BYE request directly to UAS_mazaya (step 14), which in turn sends a 200 OK response to UAC_mazaya (step 15). Then, both Samy and Mazaya get disconnected.

Figure 1. Diagram illustrating a SIP session

3 Programming models for HTTP and SIP

The similarities between HTTP and SIP have led to SIP’s programming models that are inspired from HTTP’s programming models. In this section, we give an overview of the two programming models of HTTP, namely CGI [13] and Servlets [6, 7, 9], together with their respective SIP-extensions, namely SIP-CGI [11] and Siplets [10, 5].

3.1 HTTP’s programming models

Initially, HTTP servers were used with a very simple request-response model as follows: a client browser sends to the HTTP server a request message asking for an HTML file, and then the server replies by sending the HTML file to the browser. Over time, requests have evolved and thus
necessitated more complex treatments than the simple “ask for an HTML file”. More precisely, the request-response model has become: a client browser sends to the HTTP server a request message necessitating a given processing (or more simply, a job), and then the server executes the job and sends back the result of the job to the browser. In the simplest case, the job consists of just selecting an HTML file. But in the general case, the job may consist of any treatment that can be processed by the HTTP server. In order to deal with this evolution, several programming models have been developed, such as CGI and Servlets.

### 3.1.1 CGI for HTTP

The basic principle of CGI (Common Gateway Interface) is illustrated in Figure 2. The HTTP server receives from a HTTP client (e.g., web browser) an HTTP request (step 1) indicating a file (by its location and its name) and possibly containing some parameters. The indicated file is located with the HTTP server and contains a script (e.g., in Perl) or an executable program (e.g., produced from a C source), and the parameters (if any) are input arguments necessary for the execution of the script or program. The HTTP server invokes the script or program and passes the input arguments to it, if any (step 2). When the HTTP server receives the results of execution (step 3), it returns them to the browser in an HTTP response (step 4). In Figure 2, the term CGI Program means a script or an executable program.

![Figure 2. CGI model for HTTP](image)

### 3.1.2 Servlets for HTTP

A generic version of Servlets has been defined, but for simplicity we consider here only the HTTP adaptation of Servlets. A Servlet is a java class that extends the functionality of a HTTP server. It contains a method

```
void service(HTTPRequest,HttpServletResponse)
```

where HttpServletRequest and HttpServletResponse are java interfaces for handling HTTP request and response messages, respectively.

The basic principle of Servlets is illustrated in Figure 3. When the HTTP server receives an HTTP request from a HTTP client (e.g., web browser) (step 1), it invokes a servlet engine (step 2) that provides the runtime environment in which Servlets execute and manage their life-cycle from their creation to their destruction. The servlet engine first loads dynamically a Servlet and constructs two java objects httpReq and httpResp that implement HttpServletRequest and HttpServletResponse, and then invokes service(httpReq,httpResp) of the Servlet (step 3). httpReq contains all the information in the request message (name of HTTP method, ...), and httpResp is an initialization of the response. The java method service() extracts from httpReq the name XXX (e.g., GET, POST) of the HTTP function to be executed and then dispatches the request to the correct doXXX() method (e.g., doGet(), doPost()) that handles XXX. The result of execution is returned via the object httpResp to the HTTP server passing through the servlet engine (steps 4-5). The HTTP response message is then constructed from httpResp by the HTTP server, and sent to the HTTP client (step 6).

![Figure 3. Servlet model for HTTP](image)

### 3.2 SIP’s programming models

In order to understand the extension of CGI and Servlets for SIP, it is worth categorizing the servers of a SIP architecture as follows:

1. UASs, registrar and redirect servers behave in a request-response way, like HTTP servers.
2. Proxy servers behave in a more complex way, because they can forward and broadcast requests and responses.

#### 3.2.1 SIP-CGI

As already said, the similarities between HTTP and SIP have naturally led to the development of SIP-CGI, which is an extension of CGI for SIP. The main difference between CGI and SIP-CGI is that the latter is not restricted to the request-response model. Indeed, SIP-CGI also supports request generation and response reception. This extension is particularly necessary for developing proxy functionalities. The principle of SIP-CGI model is illustrated in Figure 4 where two SIP proxies, a UAC and a UAS are involved. A SIP proxy (with CGI capability) receives from a UAC a SIP request (step 1) indicating a file and possibly containing some parameters. The indicated file is located with the
proxy and contains a script or an executable program, and the parameters (if any) are input arguments necessary for the execution of the script or program. The SIP proxy invokes the script or program and passes the input arguments to it, if any (step 2). The result of execution is returned to the proxy (step 3) and indicates that the SIP request must be forwarded to another SIP proxy, hence step 4. The second proxy (without CGI capability) forwards the SIP request to a UAS (with CGI capability) (step 5), which invokes a script or program and receives the result of execution (steps 6-7). The latter is sent to the UAC in a SIP response passing through the two proxies (steps 8-10). Note that in this example, the leftmost SIP proxy does not invoke a CGI program when it receives the SIP response of step 9. This may for example be the case if the proxy has been configured so that it invokes CGI programs only at the reception of requests.

3.2.2 Siplets

The similarities between HTTP and SIP and the extension “CGI→SIP-CGI” have naturally led to the development of SIP-Servlets (also called Siplets), which is an extension of Servlets for SIP. Like “CGI→SIP-CGI”, the extension “Servlet→Siplet” is particularly motivated by the desire to develop proxy functionalities. The main advantages of Servlets (resp. Siplets) in comparison with CGI (resp. SIP-CGI) are that they are in general much faster and have easy access to all Java APIs. And the main advantage of CGI and SIP-CGI is that they are programming language independent. Since we use Java as a programming language, we have naturally selected Siplets for the development of SIP-based services.

The principle of Siplet model is illustrated in Figure 5. We have the same scenario as in Figure 4 with the difference that each invocation of a CGI program has been replaced by an invocation of a servlet engine. The behavior of a servlet engine is similar to the one of Section 3.1.2 with the difference that the java method service() invokes methods doInvGet(), doAck(), doBye(), . . . instead of doGet(), doPost(), . . .

Two approaches of extension “Servlet→Siplet” have been proposed [10]:

1. Define one Siplet API denoted API₁ that can be used with all SIP servers.
2. Define two APIs, let us call them API₂ and API₂: API₂ is for UAS, redirect, and registrar servers (working in the request-response mode) and is quite similar to the Servlet API of HTTP.

API₂ is for proxy servers and extends the functionalities of API₂.

The first approach presents the advantage that a single programming model is used for all SIP servers. The main advantage of the second approach is that API₂ is simpler than API₁ and is quite similar to the API for HTTP. Hence, it is more advantageous for programming UAS, redirect, and registrar servers, especially by a person with experience in developing web services using Servlets. Since in the sequel we will study only the creation of UAS and redirect servers, API₂ is clearly the most advantageous for us. We plan to study proxy servers in a near future.

4 Services created using Siplets

We have developed several Siplet-based services. For lack of space, rather than present them all, we have selected four services which are sufficient to illustrate important issues in the design of Siplet-based services. Before presenting services, let us first give a brief introduction of the environment used for service implementation.

4.1 Environment for implementation

As already mentioned and justified, we have selected the second approach, that is the API denoted API₂ (see Section 3.2.2) to create services involving UAC and servers functioning in the request-response mode, i.e., UAS, redirect, and registrar servers. The environment used for service implementation is thus based on the second approach. Let us introduce briefly the main software modules that have been used as environment for implementation.

Tomcat-JServ is a combination of two open-source products Tomcat and JServ that constitute an HTTP server with a sophisticated HTTP-Servlet engine.

This product has been reused and extended to obtain a prototype of SIP server with a SIP-Servlet engine. For greater detail about JServ and Tomcat, see [1]

PICO is a SIP User Agent (UAS and UAC), which is written in Java, from Ellemtel’s PICO project. We have used it in our study.
4.2 Approach for triggering Siplets

Every service used by a UA is implemented by a Siplet and is triggered with the help of a Policy Server accessed by the UA using the service. The policy server handles a Policy File containing a set of rules defined in the form (Condition→SipletName). A rule is said active if its Condition is satisfied. A Condition can for example correspond to an event (e.g., user clicks on a button, UAS receives a request) and/or to the status (e.g., busy) of a user.

We have used the following approach: each time a UAS receives a SIP request (e.g., INVITE) or detects an action of the user (e.g., click on a button), it communicates with its policy server that reads a policy file and returns to the UAS the SipletName of every active rule. Then, the UAS triggers the servlet engine and indicates to it the name(s) of the Siplet(s) to be executed. This approach is illustrated by four services presented in the following subsections.

4.3 Service 1: Email Notification with Click-and-Dial Callback (ENCDC)

A user B can activate the service ENCDC so that his UA reacts to incoming calls by sending a busy tone to the caller and an email notification to the callee (i.e., B). The latter can return the call by clicking on an HTTP link contained in the email message. This service can be programmed to be unconditional or subject to one of the following three conditions: callee is busy, callee clicks on a Busy button while the phone is ringing, or callee does not reply.

The diagram of Fig 6 illustrates the service ENCDC with the condition “callee clicks on a Busy button while the phone is ringing” by an example of scenario involving a caller A and a callee B. We assume that the policy file of the UAS_B contains a rule (INVITE and Busy→EmailNotif), meaning that a Siplet named EmailNotif must be executed if the UAS_B has received an INVITE request and is in a busy mode.

When UAS_B receives an INVITE request from UAC_A (step 1), it answers by TRYING and RINGING responses (steps 2-3). We assume that instead of picking up, User B clicks on a Busy button (step 4). UAS_B checks with the policy server if some Siplet has to be executed (step 5). The policy server returns to UAS_B a message indicating that the Siplet named EmailNotif must be executed (step 6). Therefore, UAS_B triggers the servlet engine and asks it to process the INVITE request using the Siplet EmailNotif (step 7). UAS_B also replies to UAC_A by a BUSY response (step 8). As a result of step 7, the servlet engine downloads EmailNotif from a Siplet repository and triggers its execution to process the INVITE (step 9). This results in the invocation of service() which in turn invokes doInvite() that sends an email notification to B using the Mail Server (step 10). We have not represented the fact that B can return the call by clicking on an HTTP link contained in the email notification. This last step was realized by using an API developed by the PARLAY group [2].

4.4 Service 2: Log Call Duration (LCD)

A UA can be programmed so that it records details related to every established incoming call. The main detail consists of the start time and end time of the call, and thus the call duration can be deduced. This information can for example be useful for billing.

The diagram of Fig 7 illustrates the service LCD by an example of scenario involving a caller A and a callee B, where the service is used by UAS_B. We assume that the policy file of the UAS_B contains a rule (INVITE and Accept or BYE→LogCallDur), meaning that a Siplet named LogCallDur must be executed if UAS_B receives an INVITE which is accepted by B or receives a BYE. The accepted INVITE is processed for recording the start time, while BYE is processed for recording the end time.

When UAS_B receives an INVITE request from UAC_A (step 1), it answers by TRYING and RINGING responses (steps 2-3). We assume that User B accepts the call by clicking on an Accept button (step 4). UAS_B checks with the policy server if some Siplet has to be executed (step 5). The policy server returns to UAS_B a message indicating that the Siplet named LogCallDur must be executed (step 6). Therefore, UAS_B triggers the servlet engine and asks it to process the INVITE request using the Siplet LogCallDur (step 7). As a consequence of step 7, the servlet engine downloads LogCallDur from a Siplet repository and triggers its execution to process the INVITE (step 8), which results in the invocation of service() which in turn invokes doInvite() that records the start time in a database (step 9). Another consequence of Step 7 is that the servlet engine replies to UAS_B that the call is accepted (step 10), which results in the establishment of the call (steps 11-12). When A decides to terminate the

Figure 6. Sequence diagram illustrating the Email Notification (ENCDC) service

Figure 7. Sequence diagram illustrating the Log Call Duration (LCD) service
call, his UAC sends a BYE request to UAS\(_B\) (step 14), which checks with the policy server if some Siplet has to be executed (step 15). The policy server returns to UAS\(_B\) a message indicating that the Siplet named \(\text{LogCallDur}\) must be executed (step 16). Therefore, UAS\(_B\) triggers the servlet engine and asks it to process the BYE request using the Siplet \(\text{LogCallDur}\) (step 17). As a consequence of step 17, the servlet engine triggers the execution of \(\text{LogCallDur}\) to process the BYE (step 18), which results in the invocation of \(\text{service}()\) which in turn invokes \(\text{doBye}()\) that records the end time in a database (step 19). Another consequence of Step 17 is that the servlet engine replies to UAS\(_B\) that the call termination is accepted (step 20), which results in the termination of the call (step 21).

![Figure 7. Sequence diagram illustrating the Log Call Duration (LCD) service](image)

**4.5 Service 3: Prepaid Call (PC)**

A UA can be programmed so that it reacts to every incoming call by checking the balance of the caller or callee. If there is not enough balance, the call is terminated; otherwise, the call will proceed. This service can be configured by selecting the party to be checked: caller or callee.

The diagram of Fig 8 illustrates the service PC by an example of scenario involving a caller \(A\) and a callee \(B\), where only the balance of \(B\) is checked. We assume that the policy file of the UAS\(_B\) contains a rule \(\text{INVITE and Accept} \rightarrow \text{PrepaidCall}\), meaning that a Siplet named \(\text{PrepaidCall}\) must be executed if the UAS\(_B\) receives an INVITE request which is accepted by \(B\). We also assume that there is no funds in \(B\)'s account. For conciseness, we do not present here the case where \(B\)'s account has enough funds.

When UAS\(_B\) receives an INVITE request from UAC\(_A\) (step 1), it answers by TRYING and RINGING responses (steps 2-3). We assume that User \(B\) accepts the call by clicking on an Accept button (step 4). UAS\(_B\) checks with the policy server if some Siplet has to be executed (step 5). The policy server returns to UAS\(_B\) a message indicating that the Siplet named \(\text{PrepaidCall}\) must be executed (step 6). Therefore, UAS\(_B\) triggers the servlet engine and asks it to process the INVITE request using the Siplet \(\text{PrepaidCall}\) (step 7). The servlet engine downloads \(\text{PrepaidCall}\) from a Siplet repository and triggers its execution to process the INVITE (step 8). This results in the invocation of \(\text{service}()\) which in turn invokes \(\text{doInvite}()\) that checks the balance of \(B\) in a database (step 9). Since the returned balance is negative (step 10), an indication that the call must be rejected is sent from \(\text{PrepaidCall}\) to UAS\(_B\) passing through the servlet engine (steps 11-12). As a result, UAS\(_B\) sends to UAC\(_A\) a TEMPORARILY UNAVAILABLE response (step 13).

![Figure 8. Sequence diagram illustrating the Prepaid Call (PC) service with no funds](image)

**4.6 Service 4: Internet Voice Mail (IVM)**

A user can activate the service IVM so that his UA reacts to every incoming call by sending a recorded voice message to the caller offering him to leave a voice mail. The caller can then leave his message that will be recorded and saved in a database. The callee can retrieve the voice mail from the web later on.

The diagram of Fig 9 illustrates the service IVM by an example of scenario involving a caller \(A\) and a callee \(B\).
We assume that the policy file of the UAS_B contains a rule (BYE and VoiceMail→IntVoiceMail), meaning that a Siplet named IntVoiceMail must be executed if the UAS_B receives an BYE request and the IVM service is active.

We assume that B has activated the IVM service by clicking on a VoiceMail button (step 1). When UAS_B receives an INVITE request from UAC_A (step 2), it answers automatically by an OK response (step 3) which is acknowledged by UAC_A (step 4). At this stage, a call/media session starts and UAS_B sends immediately a message to A (step 5) and starts recording every message of A (sent from UAC_A to UAS_B) (step 6). When A decides to terminate the session and hangs up, his UAC sends a BYE request to UAS_B (step 7), which checks with the policy server if some Siplet has to be executed (step 8). The policy server returns to UAS_B a message indicating that the Siplet named IntVoiceMail must be executed (step 9). Therefore, UAS_B triggers the servlet engine and asks it to process the BYE request using the Siplet IntVoiceMail (step 10). As a consequence of step 10, the servlet engine downloads IntVoiceMail from a Siplet repository and triggers its execution to process the BYE (step 11), which results in the invocation of service() which in turn invokes doBye() that saves the recorded message (step 12). Another consequence of Step 10 is that the servlet engine replies to UAS_B that the call termination is accepted (step 13), which results in the termination of the call (step 14).

![Figure 9. Sequence diagram illustrating the Internet Voice Mail (IVM) service](http://www.oop-research.com/tomcat_3.1)

5 Conclusion

A first contribution of this paper is that it introduces, in a simple and pedagogical way, the bases of the SIP protocol and of the programming models CGI and Servlets together with their SIP-based extensions SIP-CGI and Siplets. Section 2 shows that the main principles of SIP can be easily learned and that SIP is well adapted for the creation of internet-telephony (or IP) services. Section 3 shows that CGI and Servlets can be easily learned and thus are a good solution for the creation of Web services. Section 3 also shows that SIP-CGI and Siplets can be easily learned and are a good solution for the creation of SIP-based services.

Another contribution of the paper is that in Section 4, we have illustrated by several diagrams how Siplets can be used for the creation of SIP-based services. These diagrams can be adapted and extended by Java developers for the creation of new SIP-based services. By these contributions, we hope that this paper will be encouraging and helpful for computer scientists wanting to start learning the creation of IP services.

In a near future, we intend to apply Siplets for the creation of SIP-based services involving SIP proxies. We also plan to study the integration of several services running in parallel. Another planned study is to investigate the use a standard Java-API for SIP, such as JAIN-SIP [4, 12].

References