ELEKTROTEHNIŠKI VESTNIK 78(5): 293–297, 2011
ENGLISH EDITION

SimpleFSM - a domain-specific language for SIP communication systems - Part II: Application to SIP Servlets

Edin Pjanić, Amer Hasanović

Faculty of Electrical Engineering, University of Tuzla,
Franjevačka 2, Tuzla 75000, Bosnia and Herzegovina
E-mail: {edin.pjanic, amer.hasanovic}@untz.ba

Abstract. This is the second of a two-part paper on the SimpleFSM, a domain-specific language, developed to simplify the application develoment for the SIP communication systems. While Part I describes the development of the SimpleFSM and its syntax, Part II gives some details of the SimpleFSM DSL integration with the Java SIP servlet architecture utilizing JRuby, a Ruby implementation for the Java Virtual Machine. Finally, a more complex converged application is developed and implemented using the developed DSL.

Keywords: Domain-specific languages, Session-Initiation Protocol, finite-state machine, metaprogramming, Ruby, telecom applications.

1 INTRODUCTION

In Part I [1] of this two-part paper, we presented the SimpleFSM, a Ruby domain-specific language (DSL) [2] for finite-state machines (FSM) developed to be used in modeling FSMs in various domains, including complex communication applications based on the SIP protocol [3]. The DSL syntax is simple. It supports state and transition definitions that include definitions of the events the FSM accepts as well as specification of actions executed on certain events.

The DSL is developed as an internal DSL and does not require any parser or other facility in order to be used in Ruby applications.

In Part II we demonstrate an application of the SimpleFSM DSL to SIP communication systems. In particular, we combine the SimpleFSM DSL with SIP servlets in order to simplify the description of the SIP call flows inside the application.

The paper is organized as follows. Section 2 gives a brief overview of SIP-message handling in a standard Java SIP servlet. An approach to SIP servlet development using the SimpleFSM DSL is presented in Section 3. Finally, Section 4 presents an application that utilizes the presented approach.

2 MESSAGE HANDLING IN SIP SERVLETS

The de facto standard for the SIP-application development and deployment is the Java platform utilizing the Sip Servlets API [4]. The SIP application developed in Java is deployed to the Sip Servlets Container which is usually a part of the JEE application server. The Sip Servlet Container is responsible for routing the received SIP messages to appropriate applications, managing servlet lifecycles, sending and receiving messages and providing other services to the servlets.

When a SIP message arrives, the Sip Servlet Container finds a suitable SIP servlet and invokes its `service` method with `ServletRequest` or `ServletResponse` objects as arguments. The default implementation of the `service` method is shown in Listing 1.

The programmer can override this default implementation and customize the message handling in the `service` method. However, the common practice is to dispatch the SIP message from inside the `doRequest` and `doResponse` methods to other message handlers usually named in the form `doXXX`, where `XXX` stands for a SIP request method or a SIP response family, as shown in Fig. 1. Hence, to handle a SIP message, a programmer typically overrides one of the `doXXX()` methods. The

Listing 1 Default implementation of the `service` method
```
public void service(ServletRequest req,
                    ServletResponse resp)
      throws javax.servlet.ServletException,
             java.io.IOException
{
  if (req != null) {
    doRequest((SipServletRequest) req);
  } else {
    doResponse((SipServletResponse) resp);
  }
}
```

Received October 5, 2011
Accepted October 24, 2011

Listing 2 Default implementation of `doResponse` method

```
protected void doResponse(SipServletResponse resp)
  throws javax.servlet.ServletException,
         java.io.IOException
{
  int status = resp.getStatus();
  if (status < 200) {
    doProvisionalResponse(resp);
  } else if (status < 300) {
    doSuccessResponse(resp);
  } else if (status < 400) {
    doRedirectResponse(resp);
  } else {
    doErrorResponse(resp);
  }
  if(resp.isBranchResponse()) {
    doBranchResponse(resp);
  }
}
```

default implementation of the `doResponse` method is implemented with a series of `if` branches as shown in Listing 2. The code of the `doRequest` method is similar.

While handling the SIP call flows, the programmer has to consider different messages originating from different call parties inside a single `doXXX` method. With the more elaborate call flows, the code of the `doXXX` method would be dominated by complex `switch` or `if` statements. Hence, this approach would lead to reduced code readability and tangled control structures.

3 SIP Application Development Using the SimpleFSM DSL

To better organize the code for the SIP application logic, the SimpleFSM DSL can be utilized inside a SIP servlet class implemented in Ruby.

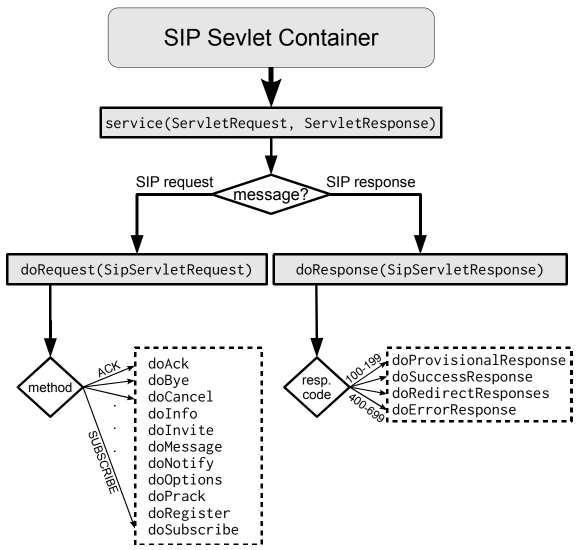

Figure 1. SIP message dispatching in a standard SIP Servlet

For this purpose, the Ruby `SipFSM` class was developed as a subclass of Java `SipServlet` class (`javax.servlet.sip.SipServlet`). It includes the `SimpleFSM` module and some minor modifications in the base class to effectively utilize the `SimpleFSM` module and simplify the SIP servlet development. The idea is to define the concrete SIP application logic using the FSM meta-language inside the Ruby SIP controller class, a subclass of the `SipFSM` class. The FSM inside the controller class is manipulated as a consequence of events generated by the JEE server after receiving SIP messages.

The FSM events that represent the received SIP messages are specified in the format `sipXXX`, where `XXX` represents the SIP request method extracted from the message, in case the received message is a SIP request. Similarly, a `sipRESPONSE_YYY` event, where `YYY` represents a SIP response code or SIP response code class, is used to notify the FSM that a SIP response message was received. Futhermore, the FSM can accept events `sipREQUEST_ANY` and `sipRESPONSE_ANY` if it is required to process any received request and/or response inside a certain application state. Examples of events that can be handled by the FSM inside the Ruby SIP controller are:

- `sipINVITE` - for INVITE request,
- `sipACK` - for ACK request,
- `sipCANCEL` - for CANCEL request,
- `sipRESPONSE_200` - for SIP response with code 200 (OK),
- `sipRESPONSE_2xx` - for any SIP responses with code begining with 2 (success responses),
- `sipREQUEST_ANY` - for any SIP request,
- `sipRESPONSE_ANY` - for any SIP response.

The application's FSM, while in some state, can react, for example, to `sipRESPONSE_1xx` events, which represent all informational responses. While in another state, the FSM can react distinguishing between the events `sipRESPONSE_180`, `sipRESPONSE_183` or `sipRESPONSE_1xx`, as shown in the following listing:

```
transitions_for :state1 do
  event :sipRESPONSE_1xx, :new => :state1,
      :do => :action1
  event :sipRESPONSE_ANY, :new => :state12
end

transitions_for :state2 do
  event :sipRESPONSE_1xx, :new => state2,
      :do => :action2
  event :sipRESPONSE_180, :new => state2,
      :do => :action3
  event :sipRESPONSE_183, :new => state2,
      :do => :action4
  event :sipRESPONSE_ANY, :new => :state22,
      :do => :action5
end
```

The FSM facility considers the current application state and generates an event that is the most specific to the message it receives. If, for example, the FSM receives a response message `180 Ringing` while in `:state1`, then the most specific event to generate is the `sipRESPONSE_1xx` event. Similarly, when the application is in `:state2`, the `180` response message would result in generation of the `sipRESPONSE_180` event, the `182 Queued` message would result in generation of the `sipRESPONSE_1xx` event, while any other received SIP response message would result in generation of the `sipRESPONSE_ANY` event.

The described message hierarchy is accomplished utilizing the Ruby's metaprogramming features when redefining the `doRequest` and `doResponse` methods inside the `SipFSM` Ruby class. As shown in Listing 3, the `doRequest` method first executes the `fsm_prepare_state` method, which reads FSM state data stored in the request's application session, and initializes several instance variables used for bookkeeping, such as the `@state` variable that keeps track of the current state. The current state of the FSM is written in the servlet application session in order to make it available for both HTTP and SIP traffic when developing converged applications. If the FSM data is not found in

Listing 3 `doRequest` and `doResponse` methods of the `SipFSM` Ruby class

```
class SipFSM < Java::javax.servlet.sip.SipServlet
  include SimpleFSM
  #-further code omitted-#

  def doRequest(request)
    run if !fsm_prepare_state([request, nil])
    m = request.get_method
    fsmm = "sip#{m}".to_sym

    if fsm_state_responds_to @state, fsmm
      send(fsmm, request, nil)
    elsif fsm_state_responds_to @state,
           :sipREQUEST_ANY
      send(:sipREQUEST_ANY, request, nil)
    else
      super
    end
  end

  def doResponse(response)
    run if !fsm_prepare_state([nil, response])
    rc = response.get_status.to_s

    exact="sipRESPONSE_#{rs}".to_sym
    group="sipRESPONSE_#{rs[/./].to_s}xx".to_sym

    if fsm_state_responds_to @state, exact
      send(exact, nil, response)
    elsif fsm_state_responds_to @state, group
      send(group, nil, response)
    elsif fsm_state_responds_to @state,
           :sipRESPONSE_ANY
      send(:sipRESPONSE_ANY, nil, response)
    else
      super
    end
  end
  #-further code omitted-#
end
```

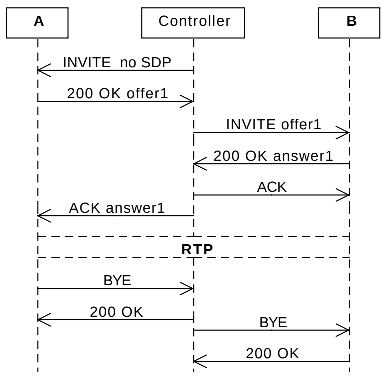

Figure 2. Click to dial call flow with call termination

the application session, the FSM is started and put to the initial state executing the `run` method. In method `doRequest`, the SIP request method is extracted from the SIP message and the corresponding FSM event name is constructed, such as `sipINVITE`, `sipACK`, etc. After that, the corresponding method is called dynamicaly using the `send` method which accepts the method name as the first parameter. The remaining parameters of the `send` method are passed to the called method. Method `fsm_state_responds_to` is a private method defined in the `SimpleFSM` module. It is used to check if the current FSM state responds to a certain event. The event corresponding to the received SIP request is checked first. Then, `:sipREQUEST_ANY` is checked. Finally, the `doRequest` method of the superclass is invoked performing the default SIP Servlet dispatching.

The logic of the `doResponse` method is similar. The only difference is that it deals with responses instead of requests.

4 EXAMPLE - CLICK TO DIAL APPLICATION

In order to demonstrate the SIP application development using the FSM DSL, a click to dial converged application is analyzed in this section. The application uses a web interface, in which the users can sign in by entering their SIP client (VoIP phone) address and see all other currently signed in users. The application uses a database to keep track of the required information and is implemented as a normal Ruby on Rails [5] web application.

After signing in, a user can initiate a VoIP call to another user by clicking the appropriate link on the web page. For this part a Sip Servlet API is used. The application acts as a back to back user agent (B2BUA) and is developed to model the flow I of the RFC 3725 [6]. This call flow is shown in Fig. 2.

Furthermore, the application handles the rejection

```
class C2dSipHandler < SipFSM
  fsm do
    state :idle
    state :calling_leg1
    state :calling_leg2, {:enter => :invite_leg2}
    state :connected
    state :terminating, {:enter => :b2bua_BYE_other}

    transitions_for :idle do
      event :sendREQ, :new => :calling_leg1,
            :guard => :is_INVITE?, :do => :b2b_send_initial_req
    end

    transitions_for :calling_leg1 do
      event :sipRESPONSE_4xx, :new => :idle, :do => :invalidate_session
      event :sipRESPONSE_6xx, :new => :idle, :do => :invalidate_session
      event :sipRESPONSE_200, :new => :calling_leg2
      event :hangUP, :new => :idle, :do => :cancel_req
    end

    transitions_for :calling_leg2 do
      event :sipBYE, :new => :terminating, :do => :send_response_200
      event :sipRESPONSE_4xx, :new => :terminating
      event :sipRESPONSE_6xx, :new => :terminating, :do => :invalidate_session
      event :sipRESPONSE_200, :new => :connected, :do => :send_ACKs
      event :hangUP, :new => :idle, :do => :bye_cancel
    end

    transitions_for :connected do
      event :sipRESPONSE_4xx, :new => :terminating, :do => :invalidate_session
      event :sipBYE, :new => :terminating, :do => :send_response_200
      event :hangUP, :new => :idle, :do => :b2bua_BYE_both
    end

    transitions_for :terminating do
      event :sipRESPONSE_200, :new => :idle
      event :sipRESPONSE_4xx, :new => :idle, :do => :invalidate_session
    end
  end

  private
  #- further code omited -#
end
```

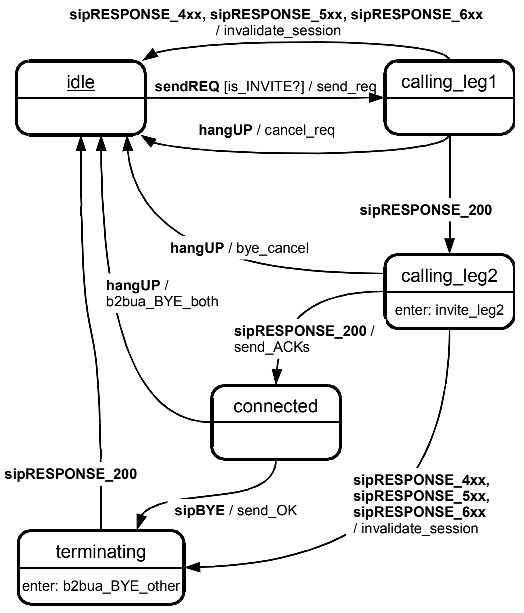

Figure 3. Click to dial Ruby SIP controller implemented using the SimpleFSM DSL and the corresponding state diagram

and call termination. Call termination is initiated after receiving a SIP BYE request from either call party. After that, the controller (SIP servlet) has to terminate the other call leg, as shown in Fig. 2. Additionaly, the user that initiated the call can terminate it at any time from the Web interface.

The state diagram that appropriatelly models the controller behavior described in the call flows and its implementation as the C2dSipHandler class using the developed DSL is shown in Fig. 3. For a detailed explanation of FSM modeling using SimpleFSM DSL and better understanding of the state diagram, we direct the reader to Part I of this two-part paper. The listing in Fig. 3 shows only the fsm block that specifies the FSM controller logic. The rest of the code that contains several short private methods utilizing SipServlet API is omitted.

As an example, Listing 4 shows the two private methods called from within the fsm block of the C2dSipHandler class. Method is_INVITE is used to check if a SIP request is an INVITE request. The second method, b2bua_BYE_other, is used as an :enter method of the :terminating state. It uses B2BUAHelper of the SipServlet API to send the BYE request to the linked session of the received SIP request or response.

The application is initially in the :idle state. When a user clicks on one of the links in the web interface, associated with another user's SIP address, the sendREQ event is invoked with the appropriate INVITE SIP request as an argument.

Listing 4 Selected private methods of the C2dSipHandler class

```
def is_INVITE? msgs
  req, res = msgs
  req.get_method == "INVITE"
end

def b2bua_BYE_other msgs
  req, res = msgs
  req ||= res.get_request
  current_sess = req.get_session
  b2b = req.get_b2bua_helper
  session2 =
     b2b.get_linked_session(current_sess)
  session2.create_request("BYE").send
end
```

The request is then sent to the SIP phone of the user who initiated the call. After sending the initial INVITE request, the Ruby SIP controller expects to receive the 200 OK SIP response. While the servlet is waiting for the 200 OK response to arrive, the application is in the :calling_leg1 state.

When the 200 OK response from the first call leg arrives, the connection is established between the Ruby

SIP controller and the first call party. The controller then has to establish the call dialog to the other user. The controller prepares and sends the `INVITE` request with the Session Description Protocol (SDP) offer received from the first call party, and waits for the response. The SDP contains the data required to initialize the streaming media used for voice communication. The application transitions to `:calling_leg2` state.

When the `200 OK` response with the SDP answer from the second call leg arrives, the call is established between the Ruby SIP controller and the second call party. Now, the SIP controller sends the `ACK` requests to both call parties. The `ACK` request sent to the first call party includes the SDP answer received from the second call party, which is enough to establish the RTP media session between the VoIP clients. The application now goes to the `:connected` state.

When one of the call parties ends the call, the SIP controller receives the `BYE` request that is sent by the VoIP client of that party. The application goes to the `:terminating` state and finally, after receiving the `sipRESPONSE_200` event, transitions back to the `:idle` state.

When the user terminates the call from the Web interface, the Web part of the converged application sends a `:hangUP` event to the FSM by calling the `hangUP` method of the servlet. In this case, the servlet, according to the model depicted in Fig. 3, performs appropriate actions, depending on the current state of the call.

The described Ruby SIP servlet class was developed using the infrastructure based on an embedded Cipango [7] server, presented in [8]. This application was also tested on the JBoss [9] application server with Mobicents SIP Servlets [10] and TorqueBox [11] deployer for Ruby applications.

5 Conclusion

In Part II of our two-part paper, an approach to SIP application development using the SimpleFSM DSL described in Part I is presented.

In this approach, a special Ruby-based SIP Servlet class is used to create a Ruby SIP controller and utilize the SimpleFSM DSL. The developed Ruby SIP Servlet class offers additional flexibility in handling SIP messages compared to the standard Java SIP Servlet.

The presented class and the DSL have enough features to develop complex telecom applications. However, other features can be added using the described metaprogramming techniques. Furthermore, no additional language, other than Ruby, is required to develop SIP applications. Moreover, the Ruby code and Ruby DSLs do not require compilation for the code to be executed, which is an important requirement for rapid application development with short iteration cycles. Based on this implementation of FSM in Ruby and the already established web frameworks, such as Rails, it is possible to develop complex converged SIP and HTTP applications entirely in Ruby, thus cutting down the time and cost of application development.

References

[1] E. Pjanić and A. Hasanović. SimpleFSM - a domain specific language for SIP communication systems - Part I: Language description, *Elektrotehnički vestnik*, (submited for publication).
[2] A. van Deursen, P. Klint and J. Visser. Domain-specific languages: an annotated bibliography, *ACM SIGPLAN Notices*, Vol. 35, pp. 26-36, 2000.
[3] J. Rosenberg, H. Schulzrinne, G. Camarillo A. Johnston, J. Peterson, R. Sparks, M. Handley and E. Schooler. SIP: Session Initiation Protocol, RFC 3261 (Proposed Standard), IETF, updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621
[4] Y. Cosmadopoulos and M. Kulkarni. SIP Servlet v1.1, JSR 289
[5] M. Bachle and P. Kirchberg. Ruby on Rails, *IEEE Software*, Vol. 24, pp. 105-108, 2007.
[6] J. Rosenberg, J. Peterson, H. Schulzrinne and G. Camarillo. Best Current Practices for Third Party Call Control (3pcc) in the Session Initiation Protocol (SIP), RFC 3725 (Best Current Practice)
[7] Cipango - SIP/HTTP Servlets Application Server Website, http://cipango.org (1.9.2011)
[8] E. Pjanić and A. Hasanović. A JRuby Infrastructure for Converged Web and SIP Applications, In: *Digital Information Processing and Communications, ser. Communications in Computer and Information Science*, Vol. 188, pp. 72-84, 2011.
[9] JBoss Website, http://jboss.org (1.9.2011)
[10] Mobicents Sip Servlets, http://www.mobicents.org/products_sip_servlets.html (1.9.2011)
[11] TorqueBox Project Website, http://torquebox.org (1.9.2011)

Edin Pjanić received his M.Sc. degree from the University of Tuzla, Bosnia and Herzegovina, in 2005. He is currently working towards his Ph.D. degree at the same university where he is a teaching assistant. His research interests include rapid web and telecom application development, dynamic programming languages and domain-specific languages.

Amer Hasanović received his B.Sc. degree in electrical engineering from the University of Tuzla, Bosnia and Herzegovina, in 1999 and his M.Sc. and Ph.D. degrees in 2001 and 2004 respectively from the West Virginia University, Morgantown, USA. In 2004 he joined the Faculty of Electrical Engineering, University of Tuzla, where he is now working as an Associate Professor. His interests have been in robust decentralized control and component-oriented software design for large scale systems simulations and currently in telecom and web application development based on dynamic programming languages.